Publication Series of the John von Neumann Institute for Computing (NIC)
NIC Series Volume 38

John von Neumann Institute for Computing (NIC)

Parallel Computing: Architectures, Algorithms and Applications

edited by

Christian Bischof
RWTH Aachen University, Germany

Martin Bücker
RWTH Aachen University, Germany

Paul Gibbon
Forschungszentrum Jülich, Germany

Gerhard Joubert
TU Clausthal, Germany

Thomas Lippert
Forschungszentrum Jülich, Germany

Bernd Mohr
Forschungszentrum Jülich, Germany

Frans Peters
Philips Research, The Netherlands

NIC Series Volume 38

ISBN 978-3-9810843-4-4

Die Deutsche Bibliothek – CIP-Cataloguing-in-Publication-Data
A catalogue record for this publication is available from Die Deutsche Bibliothek.

Publisher: NIC-Directors

Distributor: NIC-Secretariat
Research Centre Jülich
52425 Jülich
Germany

www.fz-juelich.de/nic

Co-publisher and international distributor:

IOS Press
Nieuwe Hemweg 6b
1013 BG Amsterdam
The Netherlands

www.iospress.nl
info@iospress.nl

Printer: Graphische Betriebe, Forschungszentrum Jülich

NIC Series
Volume 38
ISBN 978-3-9810843-4-4

Advances in Parallel Computing
Volume 15, ISSN 0927-5452
ISBN 978-1-58603-796-3 (IOS Press)

Preface

Parallel processing technologies have become omnipresent in the majority of new processors for a wide spectrum of computing equipment from game computers and standard PC's to workstations and supercomputers. The main reason for this trend is that parallelism theoretically enables a substantial increase in processing power using standard technologies. This results in a substantial reduction in cost compared to that of developing specialised high-performance hardware. Today the processing capacity of a desktop PC with a multi-core processor supersedes the compute power of a supercomputer of two decades ago at a fraction of the cost.

The utilisation of such powerful equipment requires suitable software. In practice it appears that the construction of appropriate parallel algorithms and the development of system and application software that can exploit the advantages of parallel hardware is not a simple matter. These problems have been studied for nearly five decades and, although much progress was made in the areas of parallel architectures, algorithm and software design, major problems remain to be addressed. The increasing replication of processing elements on chips and the use of standard components (COTS) for the relatively easy assembly of parallel systems comprising of a large number of processors (MPP) to achieve hitherto unachievable processing capacities, highlight the problems associated with the utilisation of these. Combined with the fast growth in the number of multi-core processors for PC's there is an increasing need for methods and tools to support the development of software to effectively and efficiently utilise parallel structures.

The international Parallel Computing conference series (***ParCo***) reported on progress and stimulated research in the high speed computing field over the past quarter century. New research results and techniques associated with the development and use of parallel systems were discussed at ***ParCo2007***. This international event brought together a number of the top researchers in the field of parallel computing. Their research interests covered all aspects from architectures and networks to software engineering and application development. The use of FPGA's (Free Programmable Gate Arrays) was discussed in the same vein as the development of software for multi-core processors. Papers on a wide variety of application areas using high performance computers were presented. In contrast to software for specialised high speed computing applications, where specialists spend considerable time to optimise a particular piece of code, the challenge for the future is to make software development tools available that allow non-specialists to develop 'good' parallel software with minimum effort. All of these areas are in dire need of fundamentally new ideas to overcome the limitations imposed by existing paradigms.

In addition to the contributed papers a total of five mini-symposia on special topics and an industrial session formed part of the program. Prior to the conference two well attended tutorials were offered.

As with all previous conferences in this series the emphasis with the publication of the proceedings of ***ParCo2007*** was on quality rather than quantity. Thus all contributions were reviewed prior to and again during the conference. Organisers of mini-symposia were given the option to publish reviewed papers presented at the conference in these proceedings. In total two invited papers, 63 contributed papers and 26 papers from mini-symposia are included in this book.

The Editors are greatly indebted to the members of the International Program Committee as well as the organisers of the mini-symposia for their support in selecting and reviewing the large number of papers. The organisers of the conference are also greatly indebted to the members of the various committees for the time they spent in making this conference such a successful event. Special thanks are due to the staff of the Jülich Supercomputing Centre at Research Centre Jülich and the Institute for Scientific Computing, RWTH Aachen University for their enthusiastic support.

Christian Bischof, RWTH Aachen University, Germany
Martin Bücker, RWTH Aachen University, Germany
Paul Gibbon, Forschungszentrum Jülich, Germany
Gerhard Joubert, TU Clausthal, Germany
Thomas Lippert, Forschungszentrum Jülich, Germany
Bernd Mohr, Forschungszentrum Jülich, Germany
Frans Peters, Philips Research, The Netherlands

November 2007

Conference Committee

Gerhard Joubert (Germany/Netherlands) (Conference Chair)
Thomas Lippert (Germany)
Christian Bischof (Germany)
Frans Peters (Netherlands) (Finance Chair)

Minisymposium Committee

Gerhard Joubert (Germany/Netherlands)
Thomas Lippert (Germany)
Frans Peters (Netherlands)

Organising Committee

Thomas Lippert (Germany) (Chair)
Bernd Mohr (Germany) (Co-Chair)
Rüdiger Esser (Germany)
Erika Wittig (Germany)
Bettina Scheid (Germany)
Martin Bücker (Germany)
Tanja Wittpoth (Germany)
Andreas Wolf (Germany)

Finance Committee

Frans Peters (Netherlands) (Finance Chair)

Sponsors

Forschungszentrum Jülich
IBM Germany
ParTec Cluster Competence Center
RWTH Aachen University

Conference Program Committee

Christian Bischof (Germany) (Chair)
Martin Bücker (Germany) (Co-Chair Algorithms)
Paul Gibbon (Germany) (Co-Chair Applications)
Bernd Mohr (Germany) (Co-Chair Software and Architectures)

Dieter an Mey (Germany)
David Bader (USA)
Henry Bal (Netherlands)
Dirk Bartz (Germany)
Thomas Bemmerl (Germany)
Petter Bjørstad (Norway)
Arndt Bode (Germany)
Marian Bubak (Poland)
Hans-Joachim Bungartz (Germany)
Andrea Clematis (Italy)
Pasqua D'Ambra (Italy)
Luisa D'Amore (Italy)
Erik H. D'Hollander (Belgium)
Koen De Bosschere (Belgium)
Frank Dehne (Canada)
Luiz DeRose (USA)
Anne Elster (Norway)
Efstratios Gallopoulos (Greece)
Michael Gerndt (Germany)
Bill Gropp (USA)
Rolf Hempel (Germany)
Friedel Hoßfeld (Germany)
Hai Jin (China)
Frank Hülsemann (France)
Odej Kao (Germany)
Christoph Kessler (Sweden)
Erricos Kontogiorges (Cyprus)
Dieter Kranzlmüller (Austria)
Herbert Kuchen (Germany)
Hans Petter Langtangen (Norway)
Bodgan Lesyng (Poland)
Thomas Ludwig (Germany)
Emilio Luque (Spain)
Allen D. Malony (USA)
Federico Massaioli (Italy)
Wolfgang Nagel (Germany)
Kengo Nakajima (Japan)
Nicolai Petkov (Netherlands)
Oscar Plata (Spain)
Rolf Rabenseifner (Germany)
Ullrich Rüde (Germany)
Gudula Rünger (Germany)
Marie-Chr. Sawley (Switzerland)
Henk Sips (Netherlands)
Erich Strohmaier (USA)
Paco Tirado (Spain)
Denis Trystram (France)
Marco Vanneschi (Italy)
Albert Zomaya (Australia)

Program Committee OpenMP Mini-Symposium

Barbara Chapman (USA)
Dieter an Mey (Germany)

Program Committee Scaling Mini-Symposium

Bill D. Gropp (USA) (Chair)
Wolfgang Frings (Germany)
Marc-André Hermanns (Germany)
Ed Jedlicka (USA)
Kirk E. Jordan (USA)
Fred Mintzer (USA)
Boris Orth (Germany)

Program Committee HPC Tools Mini-Symposium

Felix Wolf (Germany) (Chair)
Daniel Becker (Germany)
Bettina Krammer (Germany)
Dieter an Mey (Germany)
Shirley Moore (USA)
Matthias S. Müller (Germany)

Program Committee DEISA Mini-Symposium

Hermann Lederer (Germany) (Chair)
Marc-André Hermanns (Germany)
Giovanni Erbacci (Italy)
Gavin J. Pringle (UK)
Denis Girou (France)

Program Committee FPGA Mini-Symposium

Erik H. D'Hollander (Belgium) (Chair)
Dirk Stroobandt (Belgium) (Co-Chair)
Abdellah Touhafi (Belgium) (Co-Chair)

Abbes Amira (UK)
Peter Y.K. Cheung (UK)
Georgi Gaydadjiev (Netherlands)
Maya Gokhale (USA)
Mike Hutton (USA)
Dominique Lavenier (France)
Tsutomu Maruyama (Japan)
Viktor Prasanna (USA)
Mazen A. R. Saghir (Libanon)
Theo Ungerer (Germany)
Wolfgang Karl (Germany)
Steve Wilton (Canada)
Sotirios G. Ziavras (USA)

Contents

Invited Talks

Enhancing OpenMP and Its Implementation for Programming Multicore Systems
Barbara Chapman, Lei Huang 3

Efficient Parallel Simulations in Support of Medical Device Design
Marek Behr, Mike Nicolai, Markus Probst 19

Particle and Atomistic Simulation

Domain Decomposition for Electronic Structure Computations
Maxime Barrault, Guy Bencteux, Eric Cancès, William W. Hager, Claude Le Bris 29

Load Balanced Parallel Simulation of Particle-Fluid DEM-SPH Systems with Moving Boundaries
Florian Fleissner, Peter Eberhard 37

Communication and Load Balancing of Force-Decomposition Algorithms for Parallel Molecular Dynamics
Godehard Sutmann, Florian Janoschek 45

Aspects of a Parallel Molecular Dynamics Software for Nano-Fluidics
Martin Bernreuther, Martin Buchholz, Hans-Joachim Bungartz 53

Massively Parallel Quantum Computer Simulations: Towards Realistic Systems
Marcus Richter, Guido Arnold, Binh Trieu, Thomas Lippert 61

Image Processing and Visualization

Lessons Learned Using a Camera Cluster to Detect and Locate Objects
Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, Otto J. Anshus 71

Hybrid Parallelization for Interactive Exploration in Virtual Environments
Marc Wolter, Marc Schirski, Torsten Kuhlen 79

Performance Modeling and Tools

Analysis of the Weather Research and Forecasting (WRF) Model on Large-Scale Systems
Darren J. Kerbyson, Kevin J. Barker, Kei Davis **89**

Analytical Performance Models of Parallel Programs in Clusters
Diego R. Martínez, Vicente Blanco, Marcos Boullón, José Carlos Cabaleiro, Tomás F. Pena **99**

Computational Force: A Unifying Concept for Scalability Analysis
Robert W. Numrich **107**

Distribution of Periscope Analysis Agents on ALTIX 4700
Michael Gerndt, Sebastian Strohhäcker **113**

Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer
Jost Berthold and Rita Loogen **121**

Automatic Phase Detection of MPI Applications
Marc Casas, Rosa M. Badia, Jesús Labarta **129**

Biomedical Applications

Experimenting Grid Protocols to Improve Privacy Preservation in Efficient Distributed Image Processing
Antonella Galizia, Federica Viti, Daniele D'Agostino, Ivan Merelli, Luciano Milanesi, Andrea Clematis **139**

A Parallel Workflow for the Reconstruction of Molecular Surfaces
Daniele D'Agostino, Ivan Merelli, Andrea Clematis, Luciano Milanesi, Alessandro Orro **147**

HPC Simulation of Magnetic Resonance Imaging
Tony Stöcker, Kaveh Vahedipour, N. Jon Shah **155**

A Load Balancing Framework in Multithreaded Tomographic Reconstruction
José Antonio Álvarez, Javier Roca Piera, José Jesús Fernández **165**

Parallel Algorithms

Parallelisation of Block-Recursive Matrix Multiplication in Prefix Computations
Michael Bader, Sebastian Hanigk, Thomas Huckle **175**

Parallel Exact Inference
Yinglong Xia, Viktor K. Prasanna **185**

Efficient Parallel String Comparison
Peter Krusche and Alexander Tiskin **193**

Parallel Programming Models

Implementing Data-Parallel Patterns for Shared Memory with OpenMP
Michael Suess, Claudia Leopold **203**

Generic Locking and Deadlock-Prevention with C++
Michael Suess, Claudia Leopold **211**

Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP
Bjoern Knafla, Claudia Leopold **219**

A Framework for Performance-Aware Composition of Explicitly Parallel Components
Christoph W. Kessler, Welf Löwe **227**

A Framework for Prototyping and Reasoning about Distributed Systems
Marco Aldinucci, Marco Danelutto, Peter Kilpatrick **235**

Formal Semantics Applied to the Implementation of a Skeleton-Based Parallel Programming Library
Joel Falcou, Jocelyn Sérot **243**

Numerical Algorithms and Automatic Differentiation

Strategies for Parallelizing the Solution of Rational Matrix Equations
José M. Badía, Peter Benner, Maribel Castillo, Heike Faßbender, Rafael Mayo, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí **255**

A Heterogeneous Pipelined Parallel Algorithm for Minimum Mean Squared Error Estimation with Ordered Successive Interference Cancellation
Francisco-Jose Martínez-Zaldívar, Antonio. M. Vidal-Maciá, Alberto González **263**

OpenMP Implementation of the Householder Reduction for Large Complex Hermitian Eigenvalue Problems
Andreas Honecker, Josef Schüle **271**

Multigrid Smoothers on Multicore Architectures
Carlos García, Manuel Prieto, Fransisco Tirado **279**

Parallelization of Multilevel Preconditioners Constructed from Inverse-Based ILUs on Shared-Memory Multiprocessors
José I. Aliaga, Matthias Bollhöfer, Alberto F. Martín, Enrique S. Quintana-Ortí **287**

Parallelism in Structured Newton Computations
Thomas F. Coleman, Wei Xu **295**

Automatic Computation of Sensitivities for a Parallel Aerodynamic Simulation
Arno Rasch, H. Martin Bücker, Christian H. Bischof **303**

Parallel Jacobian Accumulation
Ebadollah Varnik, Uwe Naumann **311**

Scheduling

Layer-Based Scheduling Algorithms for Multiprocessor-Tasks with Precedence Constraints
Jörg Dümmler, Raphael Kunis, Gudula Rünger **321**

Unified Scheduling of I/O- and Computation-Jobs for Climate Research Environments
N. Peter Drakenberg, Sven Trautmann **329**

Fault Tolerance

Towards Fault Resilient Global Arrays
Vinod Tipparaju, Manoj Krishnan, Bruce Palmer, Fabrizio Petrini, Jarek Nieplocha **339**

Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA–*LC* Component Model
Diego Sevilla, José M. García, Antonio Gómez **347**

VirtuaLinux: Virtualized High-Density Clusters with no Single Point of Failure
Marco Aldinucci, Marco Danelutto, Massimo Torquati, Francesco Polzella, Gianmarco Spinatelli, Marco Vanneschi, Alessandro Gervaso, Manuel Cacitti, Pierfrancesco Zuccato **355**

Performance Analysis

Analyzing Cache Bandwidth on the Intel Core 2 Architecture
Robert Schöne, Wolfgang E. Nagel, Stefan Pflüger **365**

Analyzing Mutual Influences of High Performance Computing Programs on SGI Altix 3700 and 4700 Systems with PARbench
Rick Janda, Matthias S. Müller, Wolfgang E. Nagel, Bernd Trenkler **373**

Low-level Benchmarking of a New Cluster Architecture
Norbert Eicker, Thomas Lippert **381**

Comparative Study of Concurrency Control on Bulk-Synchronous Parallel Search Engines
Carolina Bonacic, Mauricio Marin **389**

Gb Ethernet Protocols for Clusters: An OpenMPI, TIPC, GAMMA Case Study
Stylianos Bounanos, Martin Fleury **397**

Performance Measurements and Analysis of the BlueGene/L MPI Implementation
Michael Hofmann, Gudula Rünger **405**

Potential Performance Improvement of Collective Operations in UPC
Rafik A. Salama, Ahmed Sameh **413**

Parallel Data Distribution and I/O

Optimization Strategies for Data Distribution Schemes in a Parallel File System
Jan Seidel, Rudolf Berrendorf, Ace Crngarov, Marc-André Hermanns **425**

Parallel Redistribution of Multidimensional Data
Tore Birkeland, Tor Sørevik **433**

Parallel I/O Aspects in $\mathrm{PIMA(GE)}^2$ **Lib**
Andrea Clematis, Daniele D'Agostino, Antonella Galizia **441**

Fluid and Magnetohydrodynamics Simulation

Parallelisation of a Geothermal Simulation Package: A Case Study on Four Multicore Architectures
Andreas Wolf, Volker Rath, H. Martin Bücker **451**

A Lattice Gas Cellular Automata Simulator on the Cell Broadband Engine^{TM}
Yusuke Arai, Ryo Sawai, Yoshiki Yamaguchi Tsutomu Maruyama, Moritoshi Yasunaga **459**

Massively Parallel Simulations of Solar Flares and Plasma Turbulence
Lukas Arnold, Christoph Beetz, Jürgen Dreher, Holger Homann, Christian Schwarz, Rainer Grauer **467**

Object-Oriented Programming and Parallel Computing in Radiative Magnetohydrodynamics Simulations
Vladimir Gasilov, Sergei D'yachenko, Olga Olkhovskaya, Alexei Boldarev, Elena Kartasheva, Sergei Boldyrev **475**

Parallel Simulation of Turbulent Magneto-hydrodynamic Flows
Axelle Viré, Dmitry Krasnov, Bernard Knaepen, Thomas Boeck **483**

Pseudo-Spectral Modeling in Geodynamo
Maxim Reshetnyak, Bernhard Steffen **491**

Parallel Tools and Middleware

Design and Implementation of a General-Purpose API of Progress and Performance Indicators
Ivan Rodero, Francesc Guim, Julita Corbalan, Jesús Labarta **501**

Efficient Object Placement including Node Selection in a Distributed Virtual Machine
Jose M. Velasco, David Atienza, Katzalin Olcoz, Francisco Tirado **509**

Memory Debugging of MPI-Parallel Applications in Open MPI
Rainer Keller, Shiqing Fan, Michael Resch **517**

Hyperscalable Applications

Massively Parallel All Atom Protein Folding in a Single Day
Abhinav Verma, Srinivasa M. Gopal, Alexander Schug, Jung S. Oh, Konstantin V. Klenin, Kyu H. Lee, Wolfgang Wenzel **527**

Simulations of QCD in the Era of Sustained Tflop/s Computing
Thomas Streuer, Hinnerk Stüben **535**

Optimizing Lattice QCD Simulations on BlueGene/L
Stefan Krieg **543**

Parallel Computing with FPGAs

IANUS: Scientific Computing on an FPGA-Based Architecture
Francesco Belletti, Maria Cotallo, Andres Cruz, Luis Antonio Fernández, Antonio Gordillo, Andrea Maiorano, Filippo Mantovani, Enzo Marinari, Victor Martín-Mayor, Antonio Muñoz-Sudupe, Denis Navarro, Sergio Pérez-Gaviro, Mauro Rossi, Juan Jesus Ruiz-Lorenzo, Sebastiano Fabio Schifano, Daniele Sciretti, Alfonso Tarancón, Raffaele Tripiccione, Jose Luis Velasco **553**

Optimizing Matrix Multiplication on Heterogeneous Reconfigurable Systems
Ling Zhuo, Viktor K. Prasanna **561**

Mini-Symposium
"The Future of OpenMP in the Multi-Core Era"

The Future of OpenMP in the Multi-Core Era
Barbara Chapman, Dieter an Mey **571**

Towards an Implementation of the OpenMP Collector API
Van Bui, Oscar Hernandez, Barbara Chapman, Rick Kufrin, Danesh Tafti, Pradeep Gopalkrishnan **573**

Mini-Symposium
"Scaling Science Applications on Blue Gene"

Scaling Science Applications on Blue Gene
William D. Gropp, Wolfgang Frings, Marc-André Hermanns, Ed Jedlicka, Kirk E. Jordan, Fred Mintzer, Boris Orth **583**

Turbulence in Laterally Extended Systems
Jörg Schumacher, Matthias Pütz **585**

Large Simulations of Shear Flow in Mixtures via the Lattice Boltzmann Equation
Kevin Stratford, Jean Christophe Desplat **593**

Simulating Materials with Strong Correlations on BlueGene/L
Andreas Dolfen, Yuan Lung Luo, Erik Koch **601**

Massively Parallel Simulation of Cardiac Electrical Wave Propagation on Blue Gene
Jeffrey J. Fox, Gregery T. Buzzard, Robert Miller, Fernando Siso-Nadal **609**

Mini-Symposium "Scalability and Usability of HPC Programming Tools"

Scalability and Usability of HPC Programming Tools
Felix Wolf, Daniel Becker, Bettina Krammer, Dieter an Mey, Shirley Moore, Matthias S. Müller **619**

Benchmarking the Stack Trace Analysis Tool for BlueGene/L
Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Barton P. Miller, Martin Schulz **621**

Scalable, Automated Performance Analysis with TAU and PerfExplorer
Kevin A. Huck, Allen D. Malony, Sameer Shende and Alan Morris **629**

Developing Scalable Applications with Vampir, VampirServer and VampirTrace
Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, Wolfgang E. Nagel **637**

Scalable Collation and Presentation of Call-Path Profile Data with CUBE
Markus Geimer, Björn Kuhlmann, Farzona Pulatova, Felix Wolf, Brian J. N. Wylie **645**

Coupling DDT and Marmot for Debugging of MPI Applications
Bettina Krammer, Valentin Himmler, David Lecomber **653**

Compiler Support for Efficient Instrumentation
Oscar Hernandez, Haoqiang Jin, Barbara Chapman **661**

Comparing Intel Thread Checker and Sun Thread Analyzer
Christian Terboven **669**

Continuous Runtime Profiling of OpenMP Applications
Karl Fürlinger, Shirley Moore **677**

Mini-Symposium "DEISA: Extreme Computing in an Advanced Supercomputing Environment"

DEISA: Extreme Computing in an Advanced Supercomputing Environment
Hermann Lederer, Gavin J. Pringle, Denis Girou, Marc-André Hermanns, Giovanni Erbacci **687**

DEISA: Enabling Cooperative Extreme Computing in Europe
Hermann Lederer, Victor Alessandrini **689**

Development Strategies for Modern Predictive Simulation Codes
Alice. E. Koniges, Brian T.N. Gunney, Robert W. Anderson, Aaron C. Fisher, Nathan D. Masters **697**

Submission Scripts for Scientific Simulations on DEISA
Gavin J. Pringle, Terence M. Sloan, Elena Breitmoser, Odysseas Bournas, Arthur S. Trew **705**

Application Enabling in DEISA: Petascaling of Plasma Turbulence Codes
Hermann Lederer, Reinhard Tisma, Roman Hatzky, Alberto Bottino, Frank Jenko **713**

HEAVY: A High Resolution Numerical Experiment in Lagrangian Turbulence
Alessandra S. Lanotte, Federico Toschi **721**

Atomistic Modeling of the Membrane-Embedded Synaptic Fusion Complex: a Grand Challenge Project on the DEISA HPC Infrastructure
Elmar Krieger, Laurent Leger, Marie-Pierre Durrieu, Nada Taib, Peter Bond, Michel Laguerre, Richard Lavery, Mark S. P. Sansom, Marc Baaden **729**

Mini-Symposium “Parallel Computing with FPGAs”

Parallel Computing with FPGAs - Concepts and Applications
Erik H. D’Hollander, Dirk Stroobandt, Abdellah Touhafi **739**

Parallel Computing with Low-Cost FPGAs: A Framework for COPACOBANA
Tim Güneysu, Christof Paar, Jan Pelzl, Gerd Pfeiffer, Manfred Schimmler, Christian Schleiffer **741**

Accelerating the Cube Cut Problem with an FPGA-Augmented Compute Cluster
Tobias Schumacher, Enno Lübbers, Paul Kaufmann, Marco Platzner **749**

A Run-time Reconfigurable Cache Architecture
Fabian Nowak, Rainer Buchty, Wolfgang Karl **757**

Novel Brain-Derived Algorithms Scale Linearly with Number of Processing Elements
Jeff Furlong, Andrew Felch, Jayram Moorkanikara Nageswaran, Nikil Dutt, Alex Nicolau, Alex Veidenbaum, Ashok Chandrashekar, Richard Granger **767**

Programmable Architectures for Realtime Music Decompression
Martin Botteck, Holger Blume, Jörg von Livonius, Martin Neuenhahn, Tobias G. Noll **777**

The HARWEST High Level Synthesis Flow to Design a Special-Purpose Architecture to Simulate the 3D Ising Model
Alessandro Marongiu, Paolo Palazzari **785**

Towards an FPGA Solver for the PageRank Eigenvector Problem
Séamas McGettrick, Dermot Geraghty, Ciarán McElroy **793**

Invited Talks

Enhancing OpenMP and Its Implementation for Programming Multicore Systems

Barbara Chapman and Lei Huang

Department of Computer Science
University of Houston
Houston, TX 77089
E-mail: {*chapman,leihuang*}*@cs.uh.edu*

A large body of existing, sequential applications must be migrated to multicore platforms with maximum transparency. To achieve this, techniques for automatic parallelization must be pushed to their limits, and current shared memory programming paradigms must be reconsidered with respect to their ease of use, performance characteristics and ability to support a wide range of programming needs. Despite advances in automated and dynamic exposure of exploitable parallelism, the application developer will frequently need to support the process of code migration by making the parallelism in a program explicit. One of the most straightforward ways in which this can be done is by inserting the directives or pragmas of the portable OpenMP shared memory programming interface, a de facto standard.

In this paper we discuss the programming implications of this radical change in the approach to building mainstream processors. We briefly consider the suitability of existing and recently proposed programming models to satisfy multicore application development needs, arguing that OpenMP is the most broadly appropriate of them for higher level multicore programming. Yet OpenMP was designed before the multicore "revolution". We discuss some ways in which the implementation of OpenMP might be improved to meet scalability demands posed by emerging platforms and introduce a potential extension that might simplify application development in some cases. We briefly discuss our own OpenMP implementation. Finally, we offer our conclusions on the ability of the state of the art in programming models and their implementations to satisfy the needs of multicore programming.

1 Introduction

All major computer vendors are aggressively introducing a new generation of hardware that incorporates multiple cores on a chip, sometimes with additional simultaneous multithreading capabilities and potentially along with other kinds of hardware, such as accelerator boards. The number of cores configured is expected to grow rapidly, so that even in the relatively short term we can expect a single chip to support the execution of a few hundred threads. Thus almost all future platforms will directly support parallel applications. There is great variety in chip design, especially in the amount and nature of resource sharing between threads on a given system. With more on-chip memory and potentially shared cache and data paths, resource sharing may benefit, and resource contention among threads may adversely affect, performance. With their high computation density, program performance and energy utilization must be balanced on chip multithreading hardware. The implications for software design, development, deployment and support are enormous.

The first generation of multicore platforms may not require parallel programs; future generations will. As the growth in single-threaded performance stagnates, the drive to higher levels of performance will inevitable involve the exploitation of some amount of application parallelism. In the medium term, the power of these platforms will undoubtedly

lead to considerable software innovation. We expect to see a growth in the deployment of modularized applications, where different concurrently active components may interact to achieve specified goals. The ability to process vast amounts of data more quickly will most likely lead to a strong reliance on visualization of the results for human processing. Other kinds of interfaces may become highly sophisticated.

A large body of existing sequential applications must also be adapted to execute on these new computing platforms. One or more high-level programming models are urgently needed if existing applications in business, techology and science are to be reasonably smoothly migrated to them. In addition to the fairly well-known typical needs of the HPC community, which is experienced in parallel program application development and migration, such requirements as time to market, cost of application development and maintenance, and software reliability, must be considered. In some situations, more modest levels of parallelism, including partial program parallelization, may be sufficient. This generation of platforms also inherently requires power consumption to be taken into account and this may influence the application migration strategy. The launching of heterogeneous systems into the marketplace introduces unprecedented levels of architectural complexity at the very time when simplified approaches to program parallelization are most urgently desired. It is not yet clear whether a single programming model may meet all needs or whether, for example, a programming model might need to be combined with prepackaged or customized libraries for the exploitation of certain kinds of hardware. Finally, there appears to be at least a partial convergence between the needs of the embedded market and the general-purpose computing market and it is worthwhile to explore whether the same, or essentially similar, programming models may be applied in both.

Today, programs are manually parallelized via an appropriate API. For shared memory systems, PThreads[1] and OpenMP[2] are the most widely available. Whereas PThreads often requires major reorganization of a program's structure, the insertion of OpenMP directives is often straightforward. However, current OpenMP does not yet provide features for the expression of locality and modularity that may be needed for multicore applications.

Adaptation of existing techniques to implement programming models on multicore platforms will require some non-trivial modification on the part of the compiler and its execution environment too. A trend to simplify aspects of the hardware will place a greater burden on the compiler to provide a suitable instruction schedule. Strategies for exploiting on-chip accelerators and general purpose graphics processors (GPGPUs) will need to be developed if they are to be transparently exploited where suitable. The potentially reduced amount of cache per thread will require careful evaluation of existing techniques with respect to their memory consumption. Loop nest optimization strategies in parallel code will need to take sharing in the memory hierarchy into account. For systems where resources are available for it, runtime monitoring and optimization may provide the best means of adapting a program to its environment.

In this paper, we first discuss the recent architectural innovations and some potential implications for the programmer. Then in Section 3 we review current parallel programming approaches in the light of these developments. We next introduce OpenMP, a widely available shared memory application programming interface (API) that is one of the candidates for multicore programming. We describe some possible ways in which implementations may be improved to satisfy the needs of emerging platforms. In Section 6 we introduce a potential enhancement to OpenMP to support locality and modularity. Next,

we give an overview of our implementation, the OpenUH compiler. Finally, we draw some conclusions on the current state of technology with respect to its ability to support parallel programming on multicores.

2 The Chip Multithreading Revolution

Chip MultiThreading (CMT)[3] is being deployed in servers, desktops, embedded systems and in the components of large-scale computing platforms. It may be implemented through several physical processor cores in one chip (multicore or Chip MultiProcessing, CMP). Each core may be configured with features that enable it to maintain the state of multiple threads simultaneously (Simultaneous multithreading, SMT). SMT allows several independent threads to issue instructions to a superscalar's multiple function units each cycle and can greatly increase the number of threads that may execute concurrently on a given platform. SMT threads executing within a core share hardware resources. While a processor core has its own resources (e.g. pipeline, L1 cache), it will share some of them with other cores. Existing CMT server processors include IBM's POWER4 and POWER5, Sun Microsystem's UltraSPARC IV, UltraSPARC T1 and UltraSPARC T2, the AMD Opteron, and the Itanium 2 dual-core. The competition for the desktop market is intense, with offerings including AMD's dual-core Athlon 64X2 and Intel's dual-core Xeon, Pentium D, Core Duo and Core 2 Duo. Current multicore/multithreaded processors differ greatly from each other in terms of the extent of sharing of processor resources, the interconnect, supported logical threads per core and power consumption. For example, the two cores of the UltraSPARC IV are almost completely independent except for the shared off-chip data paths, while the Power4 processor has two cores with shared L2 cache to facilitate fast inter-chip communication between threads. The effects of sharing and contention, the great variety and potentially hierarchical nature of sharing between threads, constitutes one of the greatest challenges of multicore application development. Together with a reduction in the amount of on-chip memory per thread, and the sharing of bandwidth to main memory, the need to conserve memory and to access data in ways that make good use of the memory hierarchy is one of the most pressing challenges of the multicore revolution from the application developer's perspective.

In the near future, more cores will be integrated into one processor with even more functionality, greater flexibility, and advanced power management. Products incorporating 4 and 8 cores are already on the market. In the coming years we are likely to see tens to hundreds of cores per processor. Processor cores will be linked by fast on-core interconnects with new topologies. Caches may become reconfigurable, enabling global sharing or exclusive access by a group of cores. There will need to be more memory on chip and close to the cores. Cache coherency will become increasingly expensive as the number of cores grow, so that machines with many cores may not provide it. Moreover, new designs emphasize the projected need for heterogeneity of the computing elements. Some promote heterogeneous architectures as a way to provide support for sequential parts of programs, which may run on a powerful core, while many lightweight cores are used to run parallel parts of a code. Others emphasize the power-saving characteristics of accelerators and yet others consider such components as graphics programming units to be essential for the anticipated workload. The different hardware units configured on a board - general-purpose multicore processors, FPGAs, general purpose GPUs, SIMD co-processors, and more -

may have different instruction sets and vastly different characteristics. It may be necessary to transfer data explicitly between different parts of the system.

There are more changes in store. With their high computational density, CMT platforms will need to balance high performance and energy saving needs. Almost all current processor designs incorporate energy saving features to enhance performance-per-watt via dynamical frequency and voltage adjusting and this will need to be taken into account in the programming model. A single multicore machine may be virtualized to several unified machines to hide the underlying complexity and diversity, and to relieve security concerns. Hardware-assisted transactional memory might be available as a standard feature. Individual cores are expected to become simpler in their structure. Hardware support for branch prediction may become a thing of the past, for example, since it typically requires a buffer to record which path was actually taken when a given branch was previously encountered. When (on-chip) memory is at a premium, this may be jettisoned. Similarly, hardware support for out-of-order execution may no longer be available.

The implications of these changes are nothing short of dramatic. Programs must be multithreaded to take advantage of these systems, yet the overwhelming majority of non-HPC codes in Fortran, C, C++ and other languages is sequential. There will be a new set of rules for optimizing programs, with a premium placed on making good use of the memory hierarchy. Application developers may need to learn how to exploit hardware features such as graphics units for general-purpose programming. These changes also place substantial additional burdens upon the compiler and its runtime environment. They require careful reconsideration of parallel programming models and their implementations. Compiler translation strategies for shared memory models must be rethought to take resource sharing and contention characteristics of these new systems into account, and to deal with the implications of energy saving and virtualization. Support for heterogeneity and program translation for some kinds of accelerators poses a great challenge to implementors.

3 Potential Programming Models

Given the potentially prohibitive cost of manual parallelization using a low-level programming model, it is imperative that programming models and environments be provided that offer a reasonably straightforward means of adapting existing code and creating future parallel programs. From the application developer's perspective, the ideal model would, in many cases, just be the sequential one. Automatic parallelization, especially for loop nests, has been pursued over several decades. Yet despite advances in the required analyses, static compilers still only perform this successfully for simple, regular codes. The problems are well known: unknown variable values, especially in array subscript expressions, lead to conservative assumptions that prevent the parallelization of important code portions. Recent research has successfully used dynamic compilation to improve this process[4,5,6]. This may increase the viability of automatic parallelization, but it is not likely to unleash the levels of parallelism required for a successful exploitation of multicore hardware.

Parallel programming models designed for High Performance Computing (HPC) such as MPI, UPC[7], Co-Array Fortran[8], and Titanium[9], may be implemented on a multicore platform but do not seem to be an appropriate vehicle for the migration of most mainstream applications. They provide the advantage that they permit, or require, the programmer to create code that makes the locality of data explicitly. Unfortunately, some of these

models require major code reorganization: in particular, those that offer a "local" view of computation, such as MPI and Co-Array Fortran, expect the entire program to be rewritten to construct the local code and hence do not permit incremental parallelization. Perhaps worse, they are likely to be wasteful of memory. The drawbacks of memory hierarchies may be experienced without the ability to exploit any possible benefits of cache sharing.

The "HPCS" programming languages Fortress[10], Chapel[11], and X10[12] were designed to support highly productive programming of emerging petascale computers. They provide a wealth of new ideas related to correctness, locality and efficiency of shared memory updates that may eventually influence programming at the multicore level. The different languages offer these features at different levels of expressivity and abstraction, potentially providing a choice for the programmer. They also have much in common, including the assumption of a structured architecture with a shared address space, and features for exploiting atomic updates to regions of code. Designed for high-end computing, the HPCS languages may not be intuitive enough for widespread use, and there are many different features among them that have yet to be tested in any realistic scenario. Nevertheless, features proposed here and experiences implementing them might point the way to innovative approaches to simplifying the task of creating efficient code for multicore machines.

The recently introduced Intel Threading Building Blocks (TBB)[13] utilizes generic programming to provide parallel template classes and functions for C++ programs. However, its use typically requires many changes to the sequential application. The mainstream thread-based programming models, PThreads and OpenMP, are a natural fit for CMT platforms. PThreads provides an extremely flexible shared memory programming model. But the high amount of code reorganization it requires, and the difficulty of determining correctness of PThreads codes (some developers found that its use tripled software development costs[14]), make it suited only for those who are willing to make this investment. Moreover, a library-based approach has inherent problems due to the lack of a memory model. Directive-based OpenMP programs largely avoid these drawbacks (although the incorrect usage of its features may introduce data races) but performance gains under existing translation strategies are sometimes modest unless the program's cache behaviour is analyzed and improved.

Transactions[15], borrowed from the database arena where there may be a large number of distinct, concurrent updates with little danger of conflict, could be used to provide a relatively simple programming model that overcomes drawbacks of lock-based synchronization. Transactions are optimistically performed; a subsequent check is required to ensure that no other concurrently executing transactions have read or written any data that it has updated. If there is such a conflict, then the transaction must be nullified, either by "undoing" it (a rollback) or by not committing the changes until after the check. Small transactional units that keep/extend object-oriented properties may be useful for many applications, especially those in which there is much concurrency with low expectation for conflict. But transactions are not without their downside: if the likelihood of rollbacks is high, the system may perform poorly, their use requires significant changes to a sequential application, and debugging may be extremely difficult. Transactional memory can be implemented using hardware, software or their combination. The provision of hardware support for transactions means that this might be an appealing part of a future programming model. On its own, the notion of a transaction is insufficient to allow for the expression of the many kinds of parallelism that need to be exploited in a general-purpose setting.

A suitable thread-based programming model for general-purpose multicore programming must provide performance and portability along with comparative ease of programming. The last of these implies that the model will not just facilitate application development, as well as the migration of existing code, but also help reduce the effort involved in code maintenance, extension, and testing. Such a model has to support the expression of multiple granularities of parallelism, should provide reasonable scalability in general and must be appropriate for object-oriented programming paradigms. It needs to permit the exploitation of fine-grained parallelism and the hardware's shared memory hierarchy (at least indirectly), yet must attempt to avoid the shared memory programming pitfalls of traditional lock-based synchronization and data race conditions.

While there are likely to be several successful approaches to multicore programming, we believe that a relatively modest set of interfaces will be widely supported. At the high level, OpenMP is a portable, shared memory parallel programming that has been successfully deployed on small-to-medium shared memory systems (SMPs) and large-scale distributed shared memory platforms (DSMs) as well as several existing multicore machines. Its current version 2.5[2] is, at the time of writing, about to be replaced by OpenMP 3.0, which includes features that greatly extend its range of applicability. Indeed, one of the biggest advantages of OpenMP is the fact that it is maintained by a large group of vendors, and several research organizations, who are members of the OpenMP Architecture Review Board. Collectively these institutions are committed to supporting and further evolving OpenMP to meet a wide variety of programming needs, especially with respect to multicore and other multithreading platforms.

4 OpenMP

OpenMP is a collection of directives, library routines and environment variables that may be used to parallelize Fortran, C and C++ programs for execution on shared memory platforms. There is also at least one commercial implementation of OpenMP on clusters[16]. The idea behind OpenMP is that the user specifies the parallelization strategy for a program at a high level by annotating the program code; the implementation must then work out the detailed mapping of the computation it contains to the machine. Note that it is also the user's responsibility to perform any code modifications needed prior to the insertion of OpenMP constructs. The difficulty of this effort will depend on the amount of parallelism inherent in the algorithm employed and the details of the coding techniques and style. In particular, it requires that dependences that might inhibit parallelization are detected and where possible, removed from the code.

Among the potential benefits of this programming interface are the fact that, with a little care, only one version of a program is required: if the program is compiled without the compiler's OpenMP option enabled, sequential machine code will be generated. Moreover, it is possible to incrementally create a parallel program version, parallelizing and testing one part of the code and then working on another portion if additional speedup is desired.

The language features in OpenMP enable the user to indicate the regions of the program that should be executed in parallel, and to state how the work in those regions should be shared among the available resources. Threads are started (or woken up) at the beginning of a parallel region, which is the portion of code that is dynamically contained within the structured region marked by a parallel directive (and corresponding end directive in

Fortran). The threads terminate or are put to sleep at the end of a parallel region. The work inside such a region may be executed by all threads, by just one thread, by distributing the iterations of loops to the executing threads, by distributing different sections of code to threads, and any combination of these. OpenMP also allows for nested parallelism, as threads working on a parallel region may encounter another parallel directive and start up new threads to handle it. Nested parallelism can dynamically create and exploit teams of threads and is well-suited to codes that may change their workloads over time. Further, OpenMP provides routines with which a thread may obtain its unique (within a region) identifier, and the total number of threads executing a region. This enables the application programmer to write code that explicitly assigns work to threads. As a result, OpenMP permits both "high level" compiler-directed parallelization and "low level", explicitly parallel programming.

OpenMP has strong roots in efforts of the late 1980s[17, 18] to create a standard for shared memory programming. Not surprisingly, therefore, it is particularly suitable for scientific programming, where there are often many loops that can be parallelized. However, OpenMP 2.5 expects loops to be countable; many typical loops in C codes, where lists and other structures may be traversed via a pointer, cannot be dealt with. OpenMP 3.0 is designed to overcome this limitation and to enhance several existing features of the standard. In particular, it extends support for nested parallelism so that if multiple parallel regions are active at a given time, they may have different defaults, and may be clearly distinguished from one another if the algorithm requires it. Thus the range of applications that are amenable to OpenMP parallelization is significantly expanded, and the expression of multiple levels of parallelism is becoming more straightforward.

5 Improving The OpenMP Implementation

Most OpenMP compilers translate OpenMP into multithreaded code with calls to a custom runtime library in a straightforward manner, either via outlining[19], inlining[20] or an SPMD scheme[21] for clusters. Since many details of execution, such as the number of iterations in a loop nest that is to be distributed to the threads, and the number of threads that will participate in the work of a parallel region, are often not known in advance, much of the actual work of assigning computations must be performed dynamically. Part of the implementation complexity lies in ensuring that the presence of OpenMP constructs does not impede sequential optimization in the compiler. An efficient runtime library to manage program execution is essential.

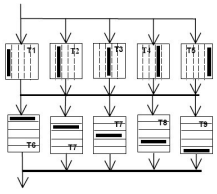
Implementations generally strictly follow the OpenMP semantics. For example, an OpenMP parallel loop is executed in parallel even if it is not profitable. Alternative approaches might fruitfully use OpenMP constructs as a simple means for the application developer to convey information to the compiler and its runtime system. The implementation then has the freedom to translate the code to best match the system. A drawback of such an interpretation of OpenMP is that the performance of the resulting code may be less transparent. However, there are many benefits, including a potential reduction in synchronization costs, and the exploitation of a variety of optimization techniques. For example,the NANOS compiler[22] has translated an extended OpenMP to a Hierarchical Task Graph to exploit multi-level parallelism. Our own work has shown the usefulness of compile-time analyses to improve data locality (including performing a wide-area data pri-

vatization[23]) as well as to reduce synchronization overheads statically. An OpenMP-aware parallel data flow analysis would enable the application of a variety of standard (sequential) optimizations to an OpenMP code.

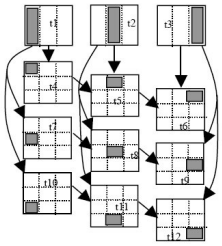
Other translation strategies may be beneficial. In earlier work, we defined and implemented the translation of OpenMP to a data flow execution model[24] that relied on the SMARTS[25] runtime system. Under this approach, the compiler transforms an OpenMP code to a collection of sequential tasks and a task graph that indicates their execution constraints. The mapping of tasks to threads may be static or dynamic. Eliminating constraints in the task graph at run time, and assigning tasks to threads to support data reuse, are the main optimizations in this work. The strategy, which could form the basis for an implementation of the tasking features of OpenMP 3.0, demonstrated the potential of such an approach for reducing synchronization overhead. Performance was enhanced when compile-time information was exploited to improve over a fully dynamic allocation of tasks to threads.

```
!$OMP PARALLEL
!$OMP DO
   do j = 1, N
     do i = 2, N
       A(i,j)=A(i,j)-B(i,j)*A(i-1,j)
     end do
   end do
!$OMP END DO
!$OMP DO
   do i = 1, N
     do j = 2, N
       A(i,j)=A(i,j)-B(i,j) * A(i,j-1)
     end do
   end do
!$OMP END DO
```

(a) The ADI OpenMP Kernel



(b) Standard OpenMPExecution



(c) Macro-pipelinedExecution

Figure 1. The ADI Kernel code using OpenMP.

Other work has pointed out the usefulness of more flexible implementation strategies. For instance, consider the ADI kernel written in Fortran and OpenMP shown in Fig. 1(a). The outer loop of the first loop nest is parallelized so that each thread accesses a contiguous set of columns of the array A. The outer loop of the second loop nest is also parallelizable, but as a result each thread accesses a contiguous set of rows of A. The static OpenMP standard execution model is shown in Fig. 1(b), and it can be easily seen that it has poor data locality. Even though we can interchange the second loop so that it also accesses A in columns, speedup is limited as a result of the overheads of the DO loop and the lack of data locality; the effects of this can be easily observed. An improved implementation strategy would create a version in which threads access the same data and the second loop is executed in a macro-pipelined manner. The task graph for this transformed code is shown in Fig. 1(c); at run time, this results in a macro-loop pipeline wavefront computation that enables data locality reuse.

The state of the art in static and dynamic optimization must be adapted in a variety of ways to meet the needs of multicore platforms. Compilers often rely on analytical models of both applications and platforms to guide program transformations, especially loop transformations. Numerous cost models have been proposed for superscalar architectures

[26,27], including for cache modelling [28,29,30]. This work needs to be extended to multicore systems. Parallel program performance models may be used to offer qualitative insights [31], or estimate the performance of parallel programs[32,33]. However, only very lightweight approaches have been employed in compilation. For example, to decide if parallelization is profitable, a simple heuristic based on the size of the loop body and iteration space is used in SUIF. A simple fork-join model[34] has been used to derive a threshold for dynamic serialization of OpenMP. Jin et al.[35] present an analytical model to optimize reductions on SMPs. Improved models that take the features of OpenMP fully into account might help the OpenMP implementation choose the number of threads to execute a parallel region or determine the most appropriate schedule for execution of a loop by multiple threads. It might also support the development of dynamic optimization strategies. Forms of empirical tuning must be experimented with in order to provide the best possible translation of long-running programs.

6 Thread Subteams: An Extension to OpenMP

OpenMP provides the benefits of incremental parallelism and direct support for shared memory, but it is hard to write a program that scales to high thread counts. Sometimes there may be insufficient parallelism in individual loop nests; at other times, a good deal of effort may be required to reorganize loops to make good use of the memory hierarchy. The need for careful placement of work and data across a multicore system has been demonstrated by a number of publications. The OpenMP approach is to allow the user to influence this implicitly, or to exploit the OS to do so. Yet existing methods do not work well with nested parallelism. Although many OSs and compilers have provided support for thread affinity at a single level, there is no support for thread team reuse at inner nesting levels, and no guarantee that the same OS threads will be used at each invocation of parallel regions. Static mappings of OpenMP threads to hardware at the start of execution cannot address the placement of threads executing inner parallel regions. Yet inappropriate mappings clearly have a negative effect on the overall application performance. The impact may be profound on multi-core, multi-chip architectures.

In order to facilitate the use of a single level of nesting in situations where the dynamic nature of nesting is not essential, and thereby to overcome the thread mapping problem, as well as to enhance modularity by providing for the straightforward specification of multiple concurrent activities, we have proposed the notion of subteams[36]. This might also form the basis for statically identifying activities that are to be performed on specific hardware components. The motivation for this OpenMP extension came from our experiences parallelizing different applications with OpenMP. Our inability to control the assignment of work to subsets of threads in the current thread team and to orchestrate the work of different threads artificially limited the performance that we could achieve. This relatively simple language feature allows us to bind the execution of a worksharing or barrier construct to a subset of threads in the current team. We introduced an *onthreads* clause that may be applied to a work-sharing directive so that only the threads in the specified subteam participate in its work, including any barrier operations encountered. Conceptually, a subteam is similar to a process subgroup in the MPI context. The user may control how threads are subdivided to suit application needs. To synchronize the actions of multiple subteams, we may use existing OpenMP constructs and take advantage of the shared memory. The

subteam feature is also useful in developing efficient hybrid MPI+OpenMP programs. The organization of threads into subteams will allow developers to use OpenMP worksharing constructs to assign different threads to work on communication and computation so as to overlap them. We have implemented this subteam concept in the OpenUH compiler[20].

6.1 Thread Subteam Syntax

We have proposed to add one clause *ONTHREADS* [36] to the existing OpenMP worksharing directives, which include *omp for/do*, *omp section*, *omp single*, and *omp workshare*, in order to enhance OpenMP's ability to support modularity and scalability. The clause specifies the subteam of threads from the current team that share the workload; it does not imply a thread-to-hardware mapping, but the mapping may be achieved implicitly e.g. via OS support, as we may assign work flexibly to threads without nested parallel regions. The syntax of the clause is as follows:

```
onthreadsclause :  onthreads ( threadset )

threadset : scalar integer expression
          | subscript-triplet
          | threadset , threadset

subscript : scalar integer expression
subscript-triplet : [subscript]:[subscript][:stride]
stride    : scalar integer expression
```

There are many potential extensions of this idea to facilitate the creation and modification of subteams and their reuse in multiple parallel regions. We could define a name for a subteam and reuse it later or allow the execution of an arbitrary piece of code within a parallel region to be restricted to a subteam. Barriers could be restricted by the *onthreads* clause.

6.1.1 Experiments with Thread Subteams

Multi-zone application benchmarks were developed as an extension to the original NAS Parallel Benchmarks[37]. They involve solving the BT, SP, and LU benchmarks on collections of loosely coupled discretization meshes (or zones). Together with colleagues at NASA Ames Research Center, we have compared the performance of four versions of these codes using OpenMP nested parallelism (2 versions), OpenMP with the subteam implementation, and hybrid MPI+OpenMP, respectively[38]. The hybrid MPI+OpenMP implementation exploits two levels of parallelism in the multi-zone benchmark in which OpenMP is applied for fine grained intra-zone parallelization and MPI is used for coarse grained inter-zone parallelization. The nested OpenMP implementation is based on the two-level approach of the hybrid version except that OpenMP is used for both levels of parallelism. The subteam version was derived from the nested OpenMP version, where we added the "onthread" clause to specify the subteam assignment on the inner level parallel regions for orphaned "omp do" constructs.

The experiments were conducted on an SGI Altix 3700BX2 system that is one of the 20 nodes in the Columbia supercomputer installed at NASA Ames Research Center. The Altix BX2 node has 512 Intel Itanium 2 processors with ccNUMA architecture, each clocked at 1.6 GHz and containing 9 MB on-chip L3 data cache. Although it is not a multicore system, it enabled us to experiment with high thread counts. Moreover, the non-uniform memory access feature approximates the memory hierarchy of multicores. Fig. 2 presents the results. The notation "M X N" denotes the number of zones formed for the first level of parallelism (M) and the number of threads (N) for the second level of parallelism within each group. The subteam and hybrid versions scaled very well up to 256 CPUs, but the nested parallelism versions performed poorly when we used more than 32 threads. Moreover, the code for the subteam version is much simpler than the other versions. The reason why the subteam and hybrid versions are close in performance is that both of them enable a similar data layout and reuse the data efficiently. A further study based on hardware counters showed that the versions based on nested parallelism have a high number of stalled cycles and L3 cache misses, indicating poor data reuse.

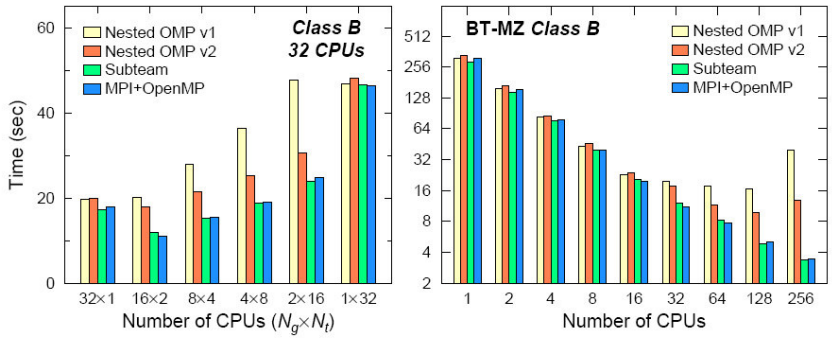


Figure 2. Comparison of subteams with equivalent versions of BT multizone benchmark.

7 The OpenUH Compiler

Our implementation efforts are based upon our OpenUH[20] compiler, a branch of the open source Open64 compiler suite for C, C++, and Fortran 95, to which we have added OpenMP 2.5 support. It generates native code for the IA-64, IA-32 and Opteron Linux ABI standards. For other platforms, an optimized version of the source code is produced in which OpenMP constructs have been replaced by calls to the compiler's portable runtime library. Our efforts form part of the official source code tree of Open64 (http://open64.sourceforge.net), which makes results available to the entire community. It is also directly and freely available via the web (`http://www.cs.uh.edu/$\sim$openuh`).

The major functional parts of the compiler are the front ends, the inter-language interprocedural analyzer (IPA) and the middle-end/back end, which is further subdivided into

the loop nest optimizer, OpenMP translation module, global optimizer, and code generator. A variety of state-of-the-art analyses and transformations are performed, sometimes at multiple levels. Open64's interprocedural array region analysis uses the linear constraint-based technique proposed by Triolet to create a DEF/USE region for each array access; these are merged and summarized at the statement level, at basic block level, and for an entire procedure. Dependence analysis uses the array regions and procedure summarization to eliminate false or assumed dependencies in the loop nest dependence graph. Both array region and dependence analysis use the symbolic analyzer, based on the Omega integer test. We have enhanced the original interprocedural analysis module[24], designing and implementing a new call graph algorithm that provides exact call chain information, and have created Dragon[39] to export analysis details for browsing. We have integrated the compiler with several performance tools to enable selective instrumentation as well as the exploitation of performance data for enhanced optimization[40].

OpenMP is lowered relatively early in the translation process to enable optimization of the explicitly parallel code. To also support high-level program optimizations prior to the replacement of OpenMP constructs by suitably modified program code and runtime library calls, we have worked on the specification and implementation of a parallel data flow analysis[41]. We have also implemented the subteam extensions described in this paper[36] and are in the process of implementing the tasking features of OpenMP 3.0 as described in Section 5. The output makes calls to our runtime library, which is based on PThreads.

7.1 Cost Modelling for OpenMP on Multicore

Cost modelling plays an important role in several phases of OpenUH. The original Open64 static cost model[42] evaluates different combinations of loop optimizations, using constraints to avoid an exhaustive search. It has an explicit processor model, a cache model and a parallel overhead model. The processor model includes instruction scheduling and register pressure, and is based on the processor's computational resources, latencies and registers. The cache model helps to predict cache misses and the cycles required to start up inner loops. It creates instruction footprints, which represent the amount of data referenced in cache for a given instruction. The parallel model was designed to support automatic parallelization by evaluating the cost involved in parallelizing a loop, and to decide which loop level to parallelize. It mainly calculates loop body execution cycles plus parallel and loop iteration overheads. These models are fairly complex but also make simplified assumptions (e.g. fixed 3 cycle loop overhead per loop iteration and 2600 cycles for any thread startup).

This pre-existing Open64 cost model has been significantly revised, in particular to reflect the properties of OpenMP code, and overheads of OpenMP constructs, at the high level[43]. We are further extending it to reflect multicore features, including traditional information about each core such as the instruction set associated with both cycle and energy cost, available function units, latencies, and registers. These are grouped together using models for their connectivity and topology. The cache model will be extended to reflect sharing properties, and used to predict both cache usage as well as the associated energy consumption. New penalties will be added to reflect contention delay. Since we need to be able to optimize for both performance and energy, we require a parameterized cost model to enhance performance-per-watt of OpenMP applications. We plan to add existing energy models for embedded processors into our cost modelling and extend them to accommodate

modern multicores. The model is expected to detect inter-thread resource contention and test different optimizations and scheduling approaches considering performance-per-watt.

We expect the resulting cost model to have the power and flexibility needed to permit both static and dynamic compilation phases to evaluate a much greater number of possible transformations than in previous empirical approaches, and to be able to focus on specific aspects of an application's execution cost. We envision its use in the analysis and prediction of differently-behaving thread (sub)teams, for giving advice on whether SMT threads should be used, and for choosing optimal OpenMP execution parameters such as the number of threads for executing a construct, the task-thread-core mapping, and selecting a loop scheduling policy.

8 Conclusions

The new generation of computer architectures introduces many challenges for application developer and compiler writer alike. The need to provide a simple programming model for multicore platforms that enables the successful exploitation of heterogeneous, hierarchically parallel processors with good levels of energy efficiency is daunting. OpenMP is a widely deployed shared-memory programming API that is currently being enhanced to broaden the scope of its applicability while retaining its ease of use. It seems possible that the judicious addition of language features to increase its power of expressivity, along with an improved OpenMP implementation strategy, might permit this to become the high level programming model of choice for general-purpose multicore programming. In this paper, we have introduced one idea for enhancement of this model and have described several implementation techniques that may improve its performance. But there is much to explore, including the need to consider its use on architectural components with very limited memory, as well as strategies for making good decisions on the number of threads used and for translating loop nests in a manner that takes complex memory sharing relationships between threads into account. We believe that by carefully considering new features that support locality, and enhancing the existing static and dynamic implementations, OpenMP may provide an efficient vehicle for creating and maintaining a variety of programs on the diverse range of future multicore architectures.

9 Acknowledgements

This work is sponsored by the National Science Foundation under contract CCF-0702775 and the Department of Energy project "Center for Programming Models for Scalable Parallel Computing". We would like to thank our colleagues at the University of Houston and at NASA Ames Research Center, especially Chunhua Liao, Deepak Eachempati and Henry Jin, for fruitful discussion of these topics.

References

1. D. Buttlar, B. Nichols and J. P. Farrell, *Pthreads Programming*, (O'Reilly & Associates, Inc., 1996).

2. *OpenMP: Simple, portable, scalable SMP programming*, (2006). http://www.openmp.org
3. L. Spracklen and S. G. Abraham, *Chip multithreading: opportunities and challenges*, in: 11th International Conference on High-Performance Computer Architecture, pp. 248–252, (2005).
4. S. Rus and L. Rauchwerger, *Compiler technology for migrating sequential code to multi-threaded Architectures*, Tech. Rep., Texas A & M University, College Station, TX, USA, (2006).
5. M. Arenaz, J. Tourino and R. Doallo, *A GSA-based compiler infrastructure to extract parallelism from complex loops*, in: ICS '03: Proceedings of the 17th annual international conference on Supercomputing, pp. 193–204, ACM Press, New York, NY, USA, (2003).
6. M. J. Martin, D. E. Singh, J. Tourino and F. F. Rivera, *Exploiting locality in the run-time parallelization of irregular loops*, in: ICPP '02: Proc. 2002 International Conference on Parallel Processing (ICPP'02), p. 27, IEEE Computer Society, Washington, DC, USA, (2002).
7. W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks and K. Warren, *Introduction to UPC and language specification*, Tech. Rep., Center for Computing Sciences, (1999).
8. R. W. Numrich and J. K Reid, *Co-Array Fortran for parallel programming*, ACM Fortran Forum, **17**, no. 2, 1–31, (1998).
9. K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella and A. Aiken, *Titanium: A high performance Java dialect*, Concurrency: Practice and Experience, **10**, 825–836, (1998).
10. E. Allen, D. Chase, V. Luchangco, J-W. Maessen, S. Ryu, G.L. Steele Jr. and S. Tobin-Hochstadt., *The Fortress language specification, version 0.785*, (2005).
11. Cray Inc., *Chapel specification 0.4*, (2005). http://chapel.cs.washington.edu/specification.pdf
12. P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, V. Saraswat, V. Sarkar and C. Von Praun, *X10: An object-oriented approach to non-uniform cluster computing*, in: Proc. 20th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, pp. 519–538, ACM SIGPLAN, (2005).
13. *Intel threading building blocks 1.0 for Windows, Linux, and Mac OS*, (2006). http://www.intel.com/cd/software/products/asmo-na/eng/threading/294797.htm
14. J. De Gelas, *The quest for more processing power, part two: multi-core and multi-threaded gaming*, (2005). http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377
15. M. Herlihy, J. Eliot and B. Moss, *Transactional memory: architectural support for lock-free data structures*, in: ISCA '93: Proc. 20th Annual International Symposium on Computer Architecture, pp. 289–300, ACM Press, New York, NY, USA, (1993).
16. Intel Inc., *Cluster OpenMP for Intel compilers for Linux*. http://www.intel.com/cd/software/products/asmo-na/eng/329023.htm.
17. Parallel Computing Forum, *PCF Parallel Fortran Extensions, V5.0*, ACM Sigplan Fortran Forum, **10**, no. 3, 1–57, (1991).
18. *Parallel processing model for high level programming languages*, Draft Proposed

American National Standard for Information Processing Systems, (1994).
19. C. Brunschen and M. Brorsson, *OdinMP/CCp - a portable implementation of OpenMP for C*, Concurrency - Practice and Experience, **12**, 1193–1203, (2000).
20. C. Liao, O. Hernandez, B. Chapman, W. Chen and W. Zheng, *OpenUH: An optimizing, portable OpenMP compiler*, in: 12th Workshop on Compilers for Parallel Computers, (2006).
21. L. Huang, B. Chapman and Z. Liu, *Towards a more efficient implementation of OpenMP for clusters via translation to Global Arrays*, Parallel Computing, **31**, no. 10-12, (2005).
22. E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez and N .Navarro, *Exploiting multiple levels of parallelism in OpenMP: a case study*, in: International Conference on Parallel Processing, pp. 172–180, (1999).
23. Z. Liu, B. Chapman, Yi Wen, L. Huang and O. Hernandez, *Analyses for the translation of OpenMP codes into SPMD style with array privatization*, in: OpenMP Shared Memory Parallel Programming, Proc. International Workshop on OpenMP Applications and Tools, WOMPAT 2003, June 26-27, 2003, vol. **2716** of Lecture Notes in Computer Science, pp. 26–41, (Springer-Verlag, Heidelberg, 2003).
24. T.H. Weng, *Translation of OpenMP to Dataflow Execution Model for Data locality and Efficient Parallel Execution*, PhD thesis, Department of Computer Science, University of Houston, (2003).
25. S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft and S. Smith, *SMARTS: Exploiting temporal locality and parallelism through vertical execution*, in: Int. Conf. on Supercomputing, (1999).
26. K. Wang, *Precise compile-time performance prediction for superscalar-based computers*, in: PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, pp. 73–84, ACM Press, New York, NY, USA, (1994).
27. R. H. Saavedra-Barrera, A. J. Smith and E. Miya, *Machine Characterization Based on an Abstract High-Level Language Machine*, IEEE Trans. Comput., **38**, 1659–1679, (1989).
28. A. K. Porterfield, *Software methods for improvement of cache performance on supercomputer applications*, PhD thesis, Rice University, (1989).
29. K. S. McKinley, *A compiler optimization algorithm for shared-memory multiprocessors*, IEEE Trans. Parallel Distrib. Syst., **9**, 769–787, (1998).
30. S. Ghosh, M. Martonosi and S. Malik, *Precise miss analysis for program transformations with caches of arbitrary associativity*, SIGOPS Oper. Syst. Rev., **32**, 228–239, (1998).
31. M. I. Frank, A. Agarwal and M. K. Vernon, *LoPC: modeling contention in parallel algorithms*, in: PPOPP '97: Proc. 6th ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 276–287, ACM Press, New York, NY, USA, (1997).
32. V. S. Adve and M. K. Vernon, *Parallel program performance prediction using deterministic task graph analysis*, ACM Trans. Comput. Syst., **22**, 94–136, (2004).
33. V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller and M. K. Vernon, *POEMS: end-to-end performance design of large parallel adaptive computational sys-*

tems, IEEE Trans. Softw. Eng., **26**, 1027–1048, (2000).
34. M. Voss and R. Eigenmann, *Reducing parallel overheads through dynamic serialization*, in: IPPS '99/SPDP '99: Proc. 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing, pp. 88–92, IEEE Computer Society, Washington, DC, USA, (1999).
35. R. Jin and G. Agrawal, *A methodology for detailed performance modeling of reduction computations on SMP machines*, Perform. Eval., **60**, 73–105, (2005).
36. B. Chapman, L. Huang, H. Jin, G. Jost and B. R. de Supinski, *Toward enhancing OpenMP's work-sharing directives*, in: Europar 2006, pp. 645–654, (2006).
37. H. Jin, M. Frumkin, and J. Yan, *The OpenMP implementation of NAS parallel benchmarks and its performance*, Tech. Rep. NAS-99-011, NASA Ames Research Center, (1999).
38. H. Jin, B. Chapman and L. Huang, *Performance evaluation of a multi-zone application in different openmp approaches*, in: Proceedings of IWOMP 2007, (2007).
39. O. Hernandez, C. Liao and B. Chapman, *Dragon: A static and dynamic tool for OpenMP*, in: Workshop on OpenMP Applications and Tools (WOMPAT 2004), University of Houston, Houston, TX, (2004).
40. O. Hernandez and B. Chapman, *Compiler Support for Efficient Profiling and Tracing*, in: Parallel Computing 2007 (ParCo 2007), (2007).
41. L. Huang, G. Sethuraman and B. Chapman, *Parallel Data Flow Analysis for OpenMP Programs*, in: Proc. IWOMP 2007, (2007).
42. M. E. Wolf, D. E. Maydan and D.-K. Chen, *Combining loop transformations considering caches and scheduling*, in: MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, pp. 274–286, IEEE Computer Society, Washington, DC, USA, (1996).
43. C. Liao and B. Chapman, *Invited Paper: A compile-time cost model for OpenMP*, in: 12th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), (2007).

Efficient Parallel Simulations in Support of Medical Device Design

Marek Behr, Mike Nicolai, and Markus Probst

Chair for Computational Analysis of Technical Systems (CATS)
Center for Computational Engineering Science (CCES)
RWTH Aachen University
52056 Aachen, Germany
E-mail: {*behr, nicolai, probst*}*@cats.rwth-aachen.de*

A parallel solver for incompressible fluid flow simulation, used in biomedical device design among other applications, is discussed. The major compute- and communication-intensive portions of the code are described. Using unsteady flow in a complex implantable axial blood pump as a model problem, scalability characteristics of the solver are briefly examined. The code that exhibited so far good scalability on typical PC clusters is shown to suffer from a specific bottleneck when thousands of processors of a Blue Gene are employed. After resolution of the problem, satisfactory scalability on up to four thousand processors is attained.

1 Introduction

Parallel computing is enabling computational engineering analyses of unprecedented complexity to be performed. In this article, experiences with parallel finite element flow simulations supporting the development of implantable ventricular assist devices in the form of continuous-flow axial pumps are reported. As an alternative to pulsatile-type devices, these pumps offer simplicity and reliability needed in long-term clinical applications. Their design however poses continuing challenges[1].

The challenges can be itemized as high shear stress levels, flow stagnation and onset of clotting, and loss of pump efficiency. Elevated shear stress levels are particularly evident in mechanical biomedical devices. Biorheological studies[2] point out a number of platelet responses to shear: adhesion, secretion, aggregation (i.e. activation), and finally, hemolysis (i.e. damage). These responses are dependent on both the level and duration of elevated shear at the platelet surface, not unlike the response to chemical factors. The primary response of the red blood cells is hemolysis[3], again dependent on dose and time.

The distribution of the shear stress levels in a complex flow field in a rotary blood pump chamber as well as the duration of the blood cells' exposure to these pathological conditions are largely unknown. Device designers are often compelled to make decisions about the details of pump configuration guided only by the global, time- and space-averaged, indicators of the shear stress inside the pump, such as the hemolysis observations made on the exiting blood stream. This challenge of detailed analysis and reduction of shear stress levels while maintaining pump efficiency as well as the need to pinpoint any persistent stagnation areas in the flow field motivates the current computational work.

In the following sections, the governing equations and their discretization are briefly discussed in Section 2, the parallel implementation described in some detail in Section 3, and parallel simulations of a complex blood pump configuration, with emphasis on scalable performance on the Blue Gene platform, are reported in Section 4.

2 Model and Discretization

2.1 Governing Equations

The blood is treated here as a viscous incompressible fluid, governed by the 3D unsteady Navier-Stokes equations. The primary variables are the velocity and pressure. No-slip Dirichlet boundary conditions are applied on most device surfaces, and stress-free parallel flow boundary conditions are imposed on the inflow and outflow surfaces.

2.2 Finite Element Formulation

The governing equations are discretized using a stabilized space-time Galerkin/Least-Squares (GLS) formulation[4]. This GLS stabilization counteracts the natural instabilities of the Galerkin method in the case of advection-dominated flow fields, and it bypasses the Babuska-Brezzi compatibility condition on the function spaces for the velocity and pressure fields, allowing both fields to use e.g. equal order basis functions. The Deforming Spatial Domain/Stabilized Space-Time (DSD/SST)[5] formulation is utilized, with equal-order linear interpolation for both fields. The motion of the computational domain boundaries is accommodated by a deforming boundary-fitted mesh. The space-time formulation naturally accounts for the deformation of the mesh, giving rise to a scheme comparable to the Arbitrary Lagrangian-Eulerian (ALE)[6] methods.

2.3 Mesh Update

The simulation of fluid flows in the presence of large but regular deformations, such as straight-line translations or rotations can be accomplished in the context of space-time finite element techniques used here by using the Shear-Slip Mesh Update Method (SSMUM) proposed earlier[7]. This method allows the elements in a thin zone of the mesh to undergo 'shear' deformation that accommodates the desired displacement of one part of the domain with respect to another. This zone is re-meshed frequently via regeneration of element connectivity. The idea behind the special-purpose mesh design is the matching of the node positions for the old (deformed) and new (good-quality) meshes, such that projection is never necessary. Since only a small part of the overall connectivity is being regenerated, the computational cost is greatly reduced.

3 Parallel Implementation

The structure of the XNS code—eXperimental Navier-Stokes finite element solver used in authors' group—resembles the one previously described in the context of the Connection Machine[8] and PC clusters[9]; the most significant difference has been replacement of element-by-element matrix storage by block-sparse-row matrix storage. The major computation stages are recalled in the Box 1.

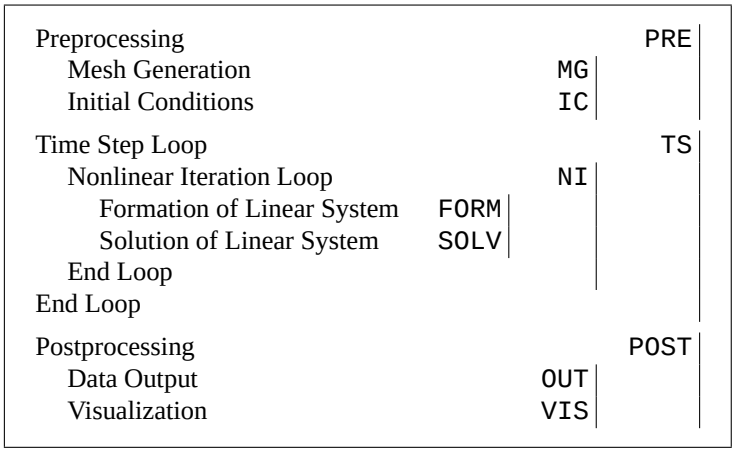


Figure 1. Computational stages of the simulation code.

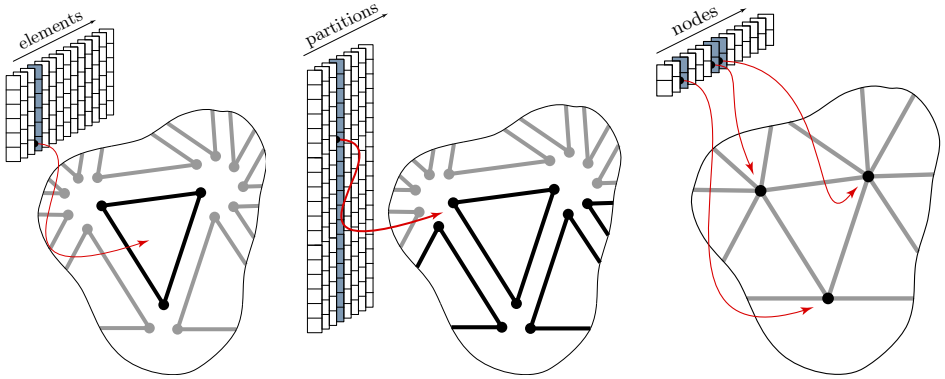
For large-scale unsteady simulations that are the focus here, the operations confined to the nonlinear iteration loop and the time-step loop are contributing most to the overall computational cost. Therefore, only the FORM and SOLV stages will be discussed.

3.1 Data Distribution

Most XNS data arrays fall into two simple categories: partition-based and node-based. In the partition-based arrays, also referred to as PART-type arrays, the last[a] dimension spans the number of nodes in a partition, which corresponds to a contiguous physical subdomain. In the node-based arrays, also referred to as NODE-type arrays, the last dimension spans the number of mesh nodes. Since some nodes are shared between partitions, the overall number of nodes in PART arrays is greater than the overall number of nodes in NODE arrays. A third storage model is useful conceptually, although not necessarily used in its entirety: an ELEM-type array is assumed to store data for all the elements belonging to a single partition. The remaining array dimensions are typically small, and correspond to number of nodes per element, number of degrees of freedom per node, etc. These categories are illustrated in Figs. 2(a), 2(b) and 2(c).

In the distributed-memory implementation, the mesh is partitioned into contiguous subdomains with the aid of the METIS graph partitioning package. Each subdomain and its set of mesh elements is then assigned to a single processing element (PE). The mesh nodes are then distributed in such a way that most nodes which are interior to a subdomain, are assigned to the PE which holds elements of the same subdomain. Nodes at a subdomain boundary are randomly assigned to PEs that store data for elements sharing that boundary. Consequently, contiguous subsets of either the ELEM-type or NODE-type arrays correspond to contiguous sets of elements and nodes, and are stored in local PE memory. The inter-PE communication is thus limited to subdomain boundaries.

[a] Fortran storage sequence is assumed here; a C implementation would use first dimension instead.



(a) element-based (b) partition-based (c) node-based

Figure 2. Storage modes for finite element data.

3.2 Linear System Formation

In the FORM stage, the element-level stiffness matrices and residual vectors are computed through numerical integration element-by-element, with input from various PART-type arrays, such as nodal coordinates, previous flow-field values, and fluid properties which are localized as needed into ELEM-type structure, and the result is assembled immediately into a PART-type matrix and residual at the partition level. This computation does not involve *any* data dependences between different partitions and is in general quite simple to parallelize or vectorize. For the partition-level matrices, Block Sparse Row (BSR) storage[10] is used, with square blocks corresponding to multiple degrees of freedom associated with each node. Such simple storage structure is made possible by equal-order interpolation for the velocity degrees of freedom *and* the pressure.

3.3 Linear System Solution

It is the SOLV phase that presents most complex data-dependence issues and obstacles to parallelization. For most time-dependent problems, the restarted GMRES iterative solver with diagonal or block-diagonal preconditioning is employed. The GMRES stages can be classified into three categories:

- Operations involving complex data dependencies, but carrying insignificant computational cost. The solution of the reduced system falls into that category; such operations can be performed serially on a parallel machine, or even redundantly repeated on all available PEs.
- Operations involving only NODE-type data. Most stages of GMRES, including the Gramm-Schmidt orthogonalization and the preconditioning step fall into this category. These portions are again trivial to parallelize or vectorize, with the only data dependence arising from the need to compute dot products or norms of the NODE-type arrays. These reduction operations are typically provided by the run-time system, e.g., by MPI's MPI_ALLREDUCE call.

- Operations involving both PART and NODE-type data. Barring the use of more complex preconditioners, the only example of such an operation is the matrix-vector product, described below.

In the matrix-vector product, NODE-type arrays represent vectors that must be multiplied by a matrix stored in a PART-type array. In order to avoid the assembly of the global stiffness matrix, the matrix-vector product is performed in the following steps:

1. GMRES:GATHER: localization, or *gather*, of the NODE-type vector into PART-type vector,

2. GMRES:MATVEC: matrix-vector product of that vector and the partition-level contributions to the global stiffness matrix, producing another PART-type vector,

3. GMRES:SCATTER: assembly, or *scatter*, of that vector into NODE-type result.

This is illustrated in Fig. 3.

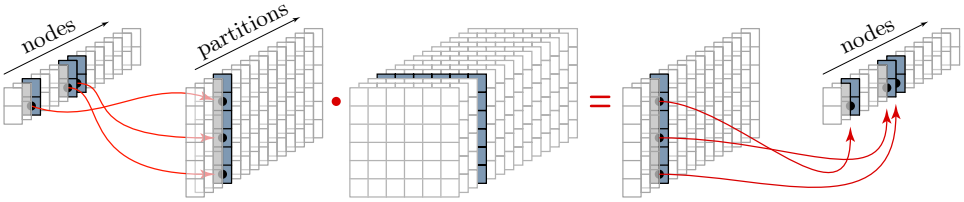


Figure 3. Outline of the matrix-vector product.

The matrix-vector product involving the partition-level stiffness matrix clearly does not involve any data dependencies. In addition, this task is trivially split between multiple PEs in a distributed-memory implementation. On the other hand, the gather and scatter operations involve many-to-one and one-to-many data transfers, and involve communication at the subdomain boundaries in a distributed-memory implementation.

4 Results

The methodology described above is applied to the problem of analysis of blood flow in an axial ventricular assist device—the MicroMed DeBakey LVAD. This pump consists of the flow straightener, a six-bladed impeller, and a six-bladed diffuser inside a cylindrical housing (2.4 cm diameter). The simulations must explore a range of impeller speed, from 7500 to 12000 rpm, and various pressure conditions. The blood viscosity is taken as 0.04 Poise, currently neglecting viscoelastic effects.

An example finite element mesh for this pump configuration has 5,098,349 tetrahedral elements and 1,726,416 space-time nodes, shown in Fig. 4. Two disc-like SSMUM layers, with a total of 30,600 elements, separate the rotating impeller mesh from the stationary straightener and diffuser meshes. The simulations must typically cover 1000 time steps

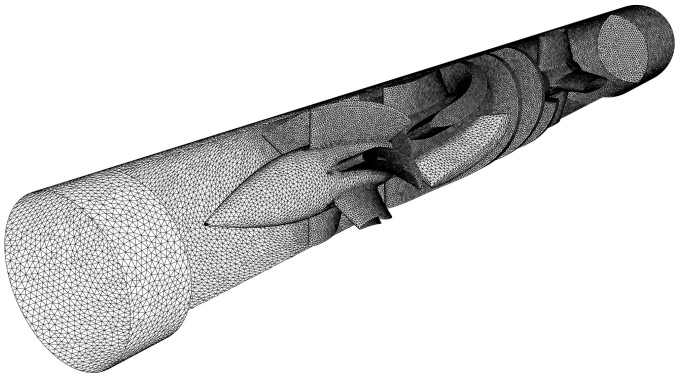


Figure 4. Axial blood pump simulation: computational mesh

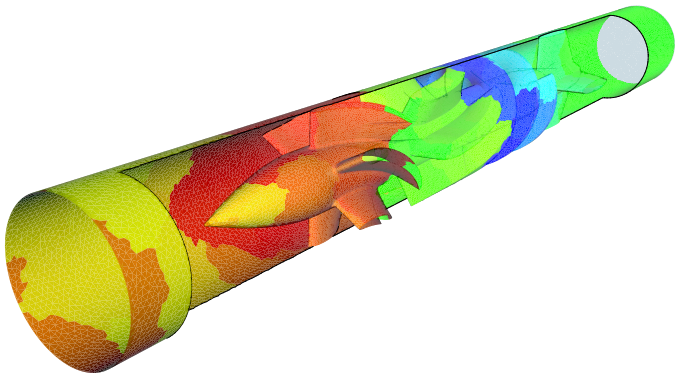


Figure 5. Axial blood pump simulation: partitioning for 1024 processors

with a step size of 0.00005–0.00008 s, representing at least ten full rotations of the impeller. At each time step, four Newton-Raphson iterations are used to solve the nonlinear discretized system, and a GMRES[11] solver with a maximum number of 200 iterations is used to solve the resulting linear system of equations. The computations are performed on an IBM Blue Gene, with up to 4096 CPUs; a sample partitioning of the domain for 1024 processors is shown in Fig. 5. A sample pressure distribution on the pump surfaces is shown in Fig. 6.

The scalability of the XNS, although satisfactory on partitions smaller than 256 PEs, was found seriously lacking on the IBM Blue Gene platform with over 1024 PEs. Subsequent performance analysis showed that the `MPI_Sendrecv` calls used throughout the XNS all-to-all gather and scatter operations are becoming dominated by zero-sized messages as the number of PEs increases. Such a bottleneck could only be detected when employing very large number of PEs, complicating the task of pinpointing the problem which was also specific to the Blue Gene architecture. After introducing conditional and separate `MPI_Send` and `MPI_Recv` calls the scalability improved dramatically, with the

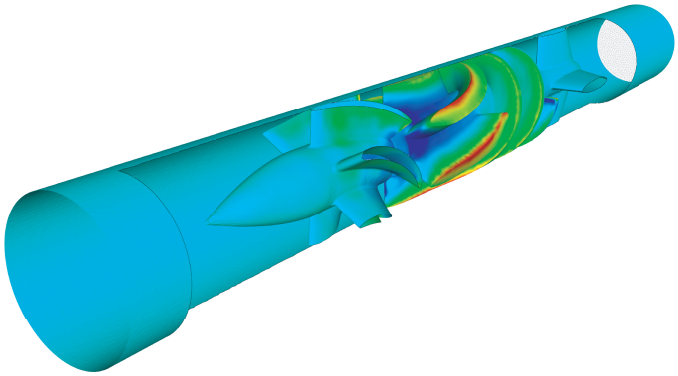


Figure 6. Axial blood pump simulation: sample pressure distribution

performance increasing almost sixfold on 4096 PEs, as shown in Fig. 7.

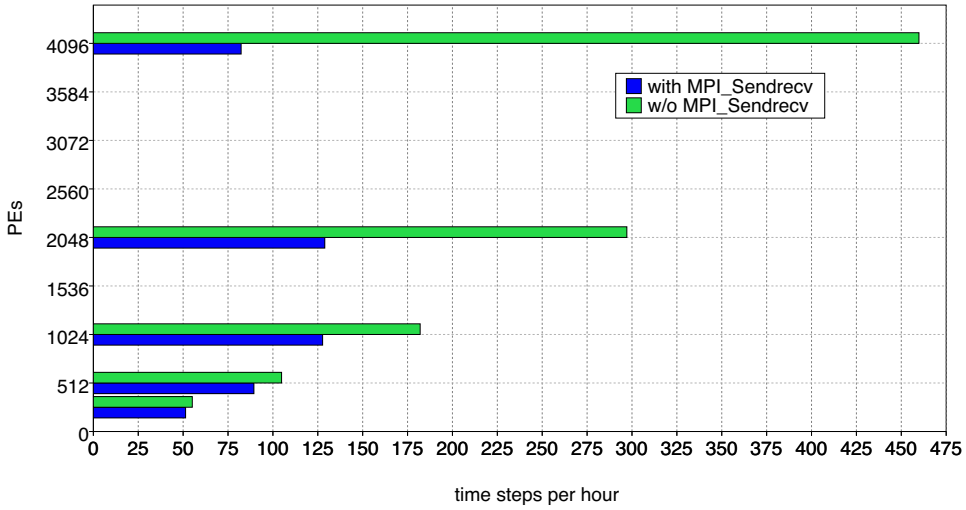


Figure 7. Scalability of the XNS code with and without `MPI_Sendrecv`

In view of the need for design optimization, where unsteady flow fields as well as their sensitivities with respect to the design parameters must be computed repeatedly while seeking a minimum of an flow-dependent objective function, the use of thousands of CPUs is a critical factor that makes such optimization practical. Authors' experience shows that the use of 4096 PEs results in computational speeds in excess of 450 time steps per hour for the model problem, which means that a complete 10-revolution computation can be accomplished in less than 3 hours of wall-clock time.

Acknowledgments

The authors are indebted to the Helmholtz Young Investigator Group "Performance Analysis of Parallel Programs" at the Research Centre Jülich, in particular to Prof. Felix Wolf and Dr. Brian Wylie, for the help in identifying the performance bottlenecks limiting the scalability on the Blue Gene and for the use of the Scalasca (`http://www.scalasca.org`) tool. The benchmark computations were performed with a grant of computer time provided by the Research Centre Jülich.

References

1. M. Yoshikawa, K. Nonaka, J. Linneweber, S. Kawahito, G. Ohtsuka, K. Nakata, T. Takano, S. Schulte-Eistrup, J. Glueck, H. Schima, E. Wolner, and Y. Nosé, *Development of the NEDO implantable ventricular assist device with gyro centrifugal pump*, Artificial Organs, **24**, 459–467, (2000).
2. M. H. Kroll, J. D. Hellums, L. V. McIntire, A. I. Schafer, and J. L. Moake, *Platelets and shear stress*, Blood, **85**, 1525–1541, (1996).
3. J. D. Hellums, *1993 Whitaker Lecture: Biorheology in thrombosis research*, Annals of Biomedical Engineering, **22**, 445–455, (1994).
4. M. Behr and T. E. Tezduyar, *Finite element solution strategies for large-scale flow simulations*, Computer Methods in Applied Mechanics and Engineering, **112**, 3–24, (1994).
5. T.E. Tezduyar, M. Behr, and J. Liou, *A new strategy for finite element computations involving moving boundaries and interfaces – the deforming-spatial-domain/space-time procedure: I. The concept and the preliminary tests*, Computer Methods in Applied Mechanics and Engineering, **94**, 339–351, (1992).
6. T. J. R. Hughes, W. K. Liu, and T. K. Zimmermann, *Lagrangian-Eulerian finite element formulation for incompressible viscous flows*, Computer Methods in Applied Mechanics and Engineering, **29**, 329–349, (1981).
7. M. Behr and T. E. Tezduyar, *Shear-Slip Mesh Update Method*, Computer Methods in Applied Mechanics and Engineering, **174**, 261–274, (1999).
8. J. G. Kennedy, M. Behr, V. Kalro, and T. E. Tezduyar, *Implementation of implicit finite element methods for incompressible flows on the CM-5*, Computer Methods in Applied Mechanics and Engineering, **119**, 95–111, (1994).
9. M. Behr, D. Arora, N. A. Benedict, and J. J. O'Neill, *Intel compilers on Linux clusters*, Technical report, Intel Corporation, Hillsboro, Oregon, (2002).
10. S. Carney, M. A. Heroux, G. Li, R. Pozo, K. A. Remington, and K. Wu, *A revised proposal for a sparse BLAS toolkit*, Preprint 94-034, Army High Performance Computing Research Center, Minneapolis, Minnesota, (1994).
11. Y. Saad and M. Schultz, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal of Scientific and Statistical Computing, **7**, 856–869, (1986).

Particle and Atomistic Simulation

Domain Decomposition for Electronic Structure Computations

Maxime Barrault[1], **Guy Bencteux**[1,4], **Eric Cancès**[2,4], **William W. Hager**[3], **and Claude Le Bris**[2,4]

[1] EDF R&D,
1 Avenue du Général de Gaulle, 92141 Clamart Cedex, France
E-mail: {*guy.bencteux,maxime.barrault*}*@edf.fr*

[2] CERMICS , École Nationale des Ponts et Chaussées
6 & 8 Avenue Blaise Pascal, Cité Descartes, 77455 Marne-La-Vallée Cedex 2, France
E-mail: {*cances,lebris*}*@cermics.enpc.fr*

[3] Department of Mathematics, University of Florida
Gainesville FL 32611-8105, USA
E-mail: hager@math.ufl.edu

[4] INRIA Rocquencourt, MICMAC Project
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay Cedex, France

We describe a domain decomposition approach applied to the specific context of electronic structure calculations. The approach has been introduced in Ref 1. We briefly present the algorithm and its parallel implementation. Simulation results on linear hydrocarbon chains with up to 2 millions carbon atoms are reported. Linear scaling with respect to the number of atoms is observed. High parallel scalability has been obtained on a Blue Gene/L machine with up to 1024 processors.

1 Introduction and Motivation

Numerical simulation is nowadays an ubiquitous tool in materials science and biology. Design of new materials, irradiation induced damage, drug design, protein folding are instances of applications of numerical simulation. When electronic structure plays a role, an explicit quantum modelling of the electronic wavefunctions is required. Solving the Schrödinger equation is out of reach, but in many cases, sufficient precision is obtained with approximated models like those based on the Density Functional or Hartree-Fock theories (See Ref 4 or Ref 5 for a detailed, mathematically-oriented, presentation).

One of the main computational bottleneck of those simulations is the solution to a (generalized) linear eigenvalue problem of the following form:

$$\begin{cases} Hc_i = \epsilon_i S c_i, & \epsilon_1 \leq \ldots \leq \epsilon_N \leq \epsilon_{N+1} \leq \ldots \leq \epsilon_{N_b}, \\ c_i^t S c_j = \delta_{ij}, & \\ D_\star = \displaystyle\sum_{i=1}^{N} c_i c_i^t. & \end{cases} \tag{1}$$

The matrix H is a $N_b \times N_b$ symmetric matrix, called the *Hamiltonian matrix*. It is the matrix of a linear operator in a Galerkin basis set. The matrix S is a $N_b \times N_b$ symmetric positive definite matrix, called the *overlap* matrix, which depends only on the basis set used (it corresponds to the mass matrix in the language of finite element methods).

Systems of equations similar to (1) also arise with semi-empirical models, such as Extended Hückel Theory based and tight-binding models, which contain additional approximations with respect to the above Density Functional or Hartree-Fock type models.

Most of the computational approaches consist in solving the system via the computation of each individual vector c_i, using a direct diagonalization procedure. The $O(N^3)$ scaling of diagonalization algorithms and their poor scalability on multiprocessor machines make prohibitive the simulation of large sets of atoms.

The matrix $D_\star$ defined by equations (1) in fact corresponds to the orthogonal projector (in S dot product) onto the subspace spanned by the eigenvectors associated with the lowest N generalized eigenvalues of H. This elementary remark is the bottom line for the development of the alternative to diagonalization methods, also often called *linear scaling*[6,7] methods because their claimed purpose is to reach a linear complexity of the solution procedure (either in terms of N the number of electrons, or N_b the dimension of the basis set).

2 A New Domain Decomposition Approach

Our purpose is now to expose a method, based on the *domain decomposition* paradigm, which we have recently introduced in Ref 1. In the following, we expose and make use of the method on quasi one-dimensional systems, typically nano-tubes or linear hydrocarbons. Generalizations to three-dimensional (bulk) systems do not really bring up new methodological issues. They are however much more difficult in terms of implementation. Some ideas supporting this claim will be given below, in the conclusion section.

For simplicity, we now present our method assuming that $S = I_{N_b}$. The extension of the method to the case when $S \neq I_{N_b}$ is straightforward. The space $\mathcal{M}^{k,l}$ denotes the vector space of the $k \times l$ real matrices. Let us first notice that a solution $D_\star$ of (1) reads

$$D_\star = C_\star C_\star^t \tag{2}$$

where $C_\star$ is a solution to the minimization problem

$$\inf\left\{ \mathrm{Tr}\Big(HCC^t\Big), \quad C \in \mathcal{M}^{N_b,N}(\mathbb{R}),\ C^tSC = I_N \right\}. \tag{3}$$

Our approach consists in solving an *approximation* of problem (3). The latter is obtained by minimizing the exact energy $\mathrm{Tr}\Big(HCC^t\Big)$ on the set of the matrices C that have the block structure displayed on Fig. 1 and satisfy the constraint $C^tC = I_N$.

A detailed justification of the choice of this structure is given in Ref 1. Let us only mention here that the decomposition is suggested from the localization of electrons and the use of a localized basis set. Note that each block overlaps only with its two nearest neighbors. We will comment further on this assumption in Section 6. Again for simplicity, we expose the method in the case where overlapping is exactly $n/2$, n being chosen even, but it could be any integer smaller than $n/2$.

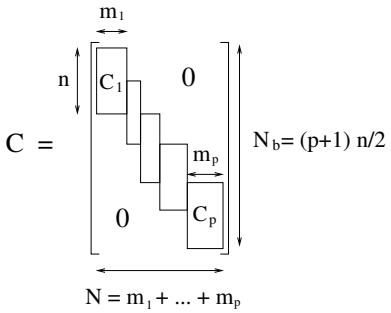


Figure 1. Block structure of the matrices C.

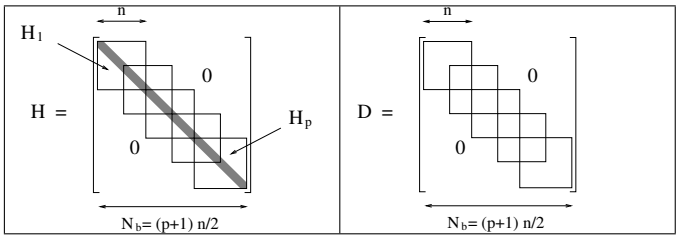


Figure 2. Block structure of the matrices H and D

The resulting minimization problem can be recast as

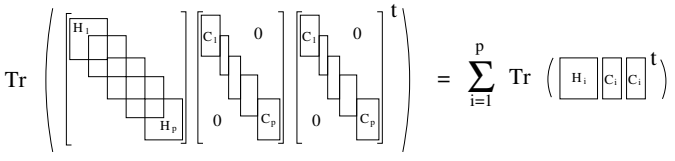
$$\inf\Bigg\{ \sum_{i=1}^{p} \mathrm{Tr}\left(H_i C_i C_i^t\right),\ C_i \in \mathcal{M}^{n,m_i}(\mathbb{R}),\quad m_i \in \mathbb{N},\quad C_i^t C_i = I_{m_i}\quad \forall\, 1 \le i \le p,$$

$$C_i^t T C_{i+1} = 0\quad \forall\, 1 \le i \le p-1,\quad \sum_{i=1}^{p} m_i = N \Bigg\}. \tag{4}$$

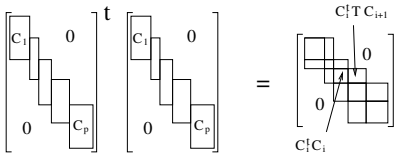
In the above formula, $T \in \mathcal{M}^{n,n}(\mathbb{R})$ is the matrix defined by

$$T_{kl} = \begin{cases} 1 & \text{if } k-l = \frac{n}{2} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

and $H_i \in \mathcal{M}^{n,n}(\mathbb{R})$ is a symmetric submatrix of H (see Fig. 2).



In this way, we replace the $\frac{N(N+1)}{2}$ *global* scalar constraints $C^t C = I_N$ involving vectors of size N_b, by the $\sum_{i=1}^{p} \frac{m_i(m_i+1)}{2}$ *local* scalar constraints $C_i^t C_i = I_{m_i}$ and the



$\sum_{i=1}^{p-1} m_i m_{i+1}$ *local* scalar constraints $C_i^t T C_{i+1} = 0$, involving vectors of size n. We would like to emphasize that we can only obtain in this way a basis of the vector space generated by the lowest N eigenvectors of H. This is the very nature of the method, which consequently cannot be applied for the search for the eigenvectors themselves.

3 The Multilevel Domain Decomposition (MDD) Algorithm

We present here an oversimplified version of the algorithm. A complete version is given in Ref 1. A numerical analysis of the algorithm, in a simplified setting, has been done in Ref 3. The minimization problem (4) is solved iteratively and each iteration is made up of two steps. At iteration k, we have at hand a set of matrices $(C_i^k)_{1\leq i\leq p}$ such that $C_i^k \in \mathcal{M}^{n,m_i}(\mathbb{R})$, $[C_i^k]^t C_i^k = I_{m_i}$, $[C_i^k]^t T C_{i+1}^k = 0$. We now explain how to compute the new iterate $(C_i^{k+1})_{1\leq i\leq p}$:

- **Step 1: Local fine solver**. For each i, find :

$$\inf\Big\{ \mathrm{Tr}\left(H_i C_i C_i^t\right),\ C_i \in \mathcal{M}^{n,m_i}(\mathbb{R}),\quad C_i^t C_i = I_{m_i},$$
$$[C_{i-1}^k]^t T C_i = 0,\quad C_i^t T C_{i+1}^k = 0 \Big\}.$$

  It provides the $n \times m_i$ matrix $\widetilde{C}_i^k$, for each i.

- **Step 2: global coarse solver**. Solve

$$\mathcal{U}^* = \mathrm{arginf}\Big\{ f(\mathcal{U}),\ \mathcal{U} = (U_i)_i,\ \forall 1 \leq i \leq p-1\ \ U_i \in \mathcal{M}^{m_{i+1},m_i}(\mathbb{R}) \Big\}, \qquad (6)$$

  where

$$f(\mathcal{U}) = \sum_{i=1}^{p} \mathrm{Tr}\Big(H_i C_i(\mathcal{U})\ C_i(\mathcal{U})^t \Big) \qquad (7)$$

  and $C_i(\mathcal{U})$ is a function of $\widetilde{C}_{i-1}^k$, $\widetilde{C}_i^k$, $\widetilde{C}_{i+1}^k$, U_{i-1} and U_i that preserves the orthonormality relations with $C_{i-1}(\mathcal{U})$ and $C_{i+1}(\mathcal{U})$.

Notice that in step 1, the computation of each odd block is independent from the other odd blocks, and obviously the same for even blocks. Thus, we use here a red/black strategy.

In equations (6) and (7), all the blocks are coupled. In practice, we reduce the computational cost of the global step, by using again a domain decomposition method. The blocks $(C_i)_{1\leq i\leq p}$ are collected in r overlapping groups $(G_l)_{1\leq l\leq r}$, as shown in Fig. 3. As each group only overlaps with its two nearest neighbors, problem (6) can be solved first for the groups (G_{2l+1}), next for the groups (G_{2l}). We have observed that the number of iterations

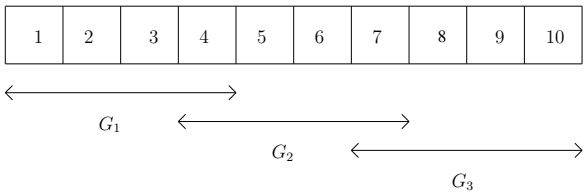


Figure 3. Collection of $p = 10$ blocks into $r = 3$ groups.

Repeat until convergence :
1a) Local step on blocks : 1, 3, ..., $(2i + 1)$, ...
1b) Local step on blocks : 2, 4, ..., $(2i)$, ...
2a) Global step on groups : $\{1, 2\}$, $\{3, 4\}$, ..., $\{2i - 1, 2i\}$, ...
2b) Global step on groups : $\{2, 3\}$, $\{4, 5\}$, ..., $\{2i, 2i + 1\}$, ...

Figure 4. Schematic view of the algorithm in the case of 2-block groups ($r = 2$) : tasks appearing on the same line are independent from one another. Order between steps 1a) and 1b) is reversed from one iteration to the other. The same holds for steps 2a) and 2b).

of the outer loop (local step + global step) does not significantly increase when the 'exact' global step (6) is replaced by the approximate global step consisting in optimizing first the odd groups, then the even groups. The numerical results performed so far (see Section 5) tend to show that the resulting algorithm scales linearly with the system size. A schematic view of the algorithm is provided in Fig. 4.

4 Parallel Implementation

For parallel implementation, the single-program, multi-data (SPMD) model is used, with message passing techniques using the MPI library. Each processor executes a single instance of the algorithm presented in Section 3, applied to a contiguous subset of blocks. Compared to the sequential version, additional data structures are introduced: each processor needs to access the matrices C_i and F_i corresponding to the last block of the processor located on its left and to the first block of the processor located on its right. These frontier blocks play the role of ghost nodes in classical domain decomposition without overlapping. For this reason, we will call them the ghost blocks.

The major part of the communications is performed between neighboring processors at the end of each step of the algorithm (i.e. of each line in the scheme displayed on Fig. (4)), in order to update the ghost blocks. This occurs only four times per iteration and, as we will see in the next section, the size of the passed message is moderate.

Collective communications are needed to compute the current value of the function f and to check that the maximum deviation from orthogonality remains acceptable. They are also needed to sort the eigenvalues of the different blocks in the local step, in the version of the algorithm allowing variable block sizes (see Ref 1). The important point is that the amount of data involved in the collective communications is small.

Apart from the very small part of collective communications, the communication volume associated with each single processor remains constant, irrespective of the number of blocks per processor and the number of processors. We can thus expect a very good scalability.

5 Numerical Tests

This section is devoted to the presentation of the performances of the MDD algorithm on matrices arising in electronic structure computations of real molecules. The set of tested matrices are those used and described in Ref 1.

Let us recall here that they were built from Hartree-Fock simulations of linear polymeric molecules, typically polyethylene chains CH_3-$(CH_2)_{n_m}$-CH_3 with physically relevant carbon-carbon distance. For moderate values of n_m, reference solutions are computed by means of diagonalization, whereas for very large n_m we exploit the bulk periodicity in the density matrix. The localization parameters of the matrices are deduced from the observed (or expected in real future applications) bands of the density matrix and of the overlapping matrix. Let q be the overlapping between two adjacent blocks. It represents also the size of the band where all the terms of the density matrix are computed by the MDD algorithm. The larger the density band, the higher q and n have to be chosen. In the polyethylene case, we have : $n = 308$, $q = 126$, the bandwith of H is 255 and the one of S is 111.

In all computations presented below, the global step is performed with groups made of two blocks ($r = 2$). The algorithm is stopped when the relative error (with respect to the reference solution) on the energy $\mathrm{Tr}\left(HCC^t\right)$ is lower than 10^{-6} and relative error on each significant (greater than 10^{-10} magnitude) coefficient in the density matrix is lower than 10^{-2}.

5.1 Sequential Computations

In this section, we use a processor DELL Precision 450 (Intel(R) Xeon(TM) CPU 2.40 GHz), with 2 GB Memory and a 512 KB cache. Computation of the density matrix has been done for a series of matrices H and S associated with the chemical systems described above. The MDD algorithm is compared to diagonalization, for which we used the Divide and Conquer algorithm (*dsbgvd.f* routine from the LAPACK library). For both computations, the version 3.6.0 of the ATLAS[8] implementation of the BLAS routines has been used. Detailed presentation of the comparison between the methods is given in Ref 1 and Ref 2. We observe that the MDD algorithm exhibits a linear scaling both in CPU and in memory (Fig 5). Apart from an improvement of the MDD performances, due to an algorithmic refinement in the global step (see Ref 2), the complete analysis of results presented in Ref 1 remains unaffected.

5.2 Parallel Computations

Blue Gene/L (IBM) is a supercomputer which provides a high number of processors with modest power (700 MHz). Nodes, each consisting of a pair of processors, are interconnected through multiple complementary high-speed low-latency networks.

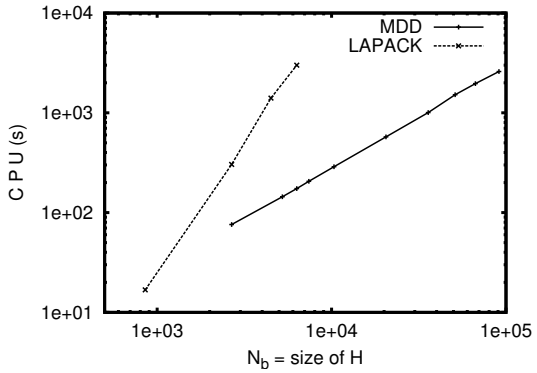


Figure 5. Evolution of the CPU used for computing the density matrix of a polyethylene molecule CH_3-$(CH_2)_{n_m}$-CH_3 with respect to the size of matrices H and S.

# Processors	32	64	128	256	512	1024
Wall clock time (s)	7907	3973	1981	1028	513	300
Speed-up (*ideal*)	1.	2.(*2.*)	4. (*4.*)	7.7 (*8.*)	15.4 (*16.*)	26.4 (*32.*)

Table 1. Evolution of CPU time with the number of processors, to run 8 iterations of the MDD algorithm to solve the problem (1) with matrices associated with polyethylene chains with $213,000$ monomers. The size of the matrices H and S is 1.5×10^6.

The computations presented in Table 1 have been made on polyethylene chains. With the localization parameters chosen for this type of molecule, the size of the ghost blocks is about 150 KB. Lastly, we have successfully applied the MDD algorithm to solve problem (1) for matrices corresponding to polyethylene chains with more than 5 million atoms (10 million atomic orbital basis functions) in less than 1 h Wall Clock Time, with the use of 1024 processors (512 nodes).

6 Conclusions and Perspectives

The MDD algorithm needs further developments and studies to be of real use in computational chemistry. In increasing order of difficulty, we can list the following perspectives:

- interaction of the MDD algorithm with the nonlinear *SCF loop* : at each iteration of the SCF loop, the problem (1) to be solved is a small perturbation of the previous one. Thus, the previous solution can be used as a very good initial guess, and this can lead to different algorithmic choices to solve the local step on the one hand, and the global step on the other hand,
- extension of the software to 2D and 3D molecular systems : all the arguments used to build the method remain valid for 2D and 3D systems, except the assumption that each block overlaps only with two neighbors. Overlapping will increase to 8 neighbors in 2D and 26 neighbors in 3D. Allowing blocks to overlap to more than two neighbors

already makes sense in the quasi-1D systems, since it gives more flexibility to the algorithm.

Generalizing the local step to more than two overlapping neighbors does not present any difficulty. Concerning the global step, the strategy of treating groups of blocks will be maintained and the only difference will be that there will be a bigger number of overlapping groups,

- extension to conducting materials : this is by far the most challenging issue to overcome.

Acknowledgements

This work has been supported by "Agence Nationale pour la Recherche-Calcul Intensif", through project No. ANR-06-CIS6-006.

References

1. M. Barrault, E. Cancès, W. W. Hager, C. Le Bris, *Multilevel domain decomposition for electronic structure calculations*, J. Comp. Phys., **222**, 86–109, (2007).
2. G. Bencteux, E. Cancès, C. Le Bris, W. W. Hager, *Domain decomposition and electronic structure computations: a promising approach*, in Numerical Analysis and Scientific Computing for PDEs and their Challenging Applications, (Springer, 2008, in press).
3. W. Hager, G. Bencteux, E. Cancès, C. Le Bris. *Analysis of a Quadratic Programming Decomposition Algorithm*. INRIA Report No RR-6288, https://hal.inria.fr/inria-00169080
4. C. Le Bris. *Computational chemistry from the perspective of numerical analysis*, Acta Numerica, **14**, 363–444, (2005).
5. E. Cancès, M. Defranceschi, W. Kutzelnigg, C. Le Bris, and Y. Maday. *Computational Quantum Chemistry: a Primer*. In C. Le Bris, editor, *Handbook of Numerical Analysis, Special volume, Computational Chemistry, volume X*, (North-Holland 2003).
6. D. Bowler, T. Miyazaki and M. Gillan, *Recent progress in linear scaling ab initio electronic structure theories*, J. Phys. Condens. Matter, **14**, 2781–2798, (2002).
7. S. Goedecker, *Linear scaling electronic structure methods*, Rev. Mod. Phys., **71**, 1085–1123, (1999).
8. Automatically Tuned Linear Algebra Software http://math-atlas.sourceforge.net/
9. R. Clint Whaley and A. Petitet, *Minimizing development and maintenance costs in supporting persistently optimized BLAS*, Software: Practice and Experience, **35**, 101–121, (2005).

Load Balanced Parallel Simulation of Particle-Fluid DEM-SPH Systems with Moving Boundaries

Florian Fleissner and Peter Eberhard

Institute of Engineering and Computational Mechanics
University of Stuttgart, 70550 Stuttgart, Germany
E-mail: {fleissner, eberhard}@itm.uni-stuttgart.de

We propose a new pure Lagrangian method for the parallel load balanced simulation of particle-fluid systems with moving boundaries or free surfaces. Our method is completely meshless and models solid objects as well as the fluid as particles. By an Orthogonal Recursive Bisection we obtain a domain decomposition that is well suited for a controller based load balancing. This controller approach is designed for being used on clusters of workstations as it can cope with load imbalances not only as emerging from the simulation dynamics but also from competitive processes of other users. In this paper we present the most important concepts such as neighbourhood search, particle interactions, domain decomposition and controller based load balancing.

1 Introduction

Various fields of engineering require the simulation of particle-fluid systems, such as dispersions with dynamically moving boundaries or free surfaces. Modelling these systems with a mixed Eulerian-Lagrangian approach requires a complicated coupling of a grid based and a meshless method, which is especially difficult if computations are to be carried out in parallel. The consideration of boundary conditions, the distribution of the workload to the parallel nodes and the a priori prediction of the simulation domain is a difficult task. Therefore, we propose a new meshless pure Lagrangian approach, that overcomes all of the problems mentioned above.

We start in Section 2 with a description of our simulation model. Therefore, we introduce the two methods for the simulation of rigid particles and the fluid and explain how the complexity of the detection of particle interactions can be reduced by an efficient neighbourhood search. We also describe how the two particle types are coupled in terms of particle interactions. The implementation of simulation boundaries is explained in Section 3. The most important aspects of our parallel simulation approach, such as process synchronization, domain decomposition and load balancing are finally introduced in Section 4.

2 Pure Lagrangian Particle-Fluid Simulation

For our pure Lagrangian approach we employ Smoothed Particle Hydrodynamics[1] (SPH) as a particle based fluid simulation method and the Discrete Element Method (DEM)[2,3] for the simulation of solid particles. The main challenges of the implementation of a pure Lagrangian approach are the coupling of the different particle types, fluid and dry particles as well as the consideration of domain boundaries. Where for grid based fluid simulation methods boundary conditions have to be imposed on the underlying PDE, for particle

boundary conditions can be considered as penalty forces that prevent particle-particle or particle-wall penetrations. These penalty forces are simply added as extra terms to the right hand sides of Newton's or Euler's equations of motion.

2.1 Discrete Element Method

The Discrete Element Method[2,3] models particles as rigid bodies whose dynamics is described by Newton's and Euler's equations of motion. Forces resulting from particle interactions are accumulated, serving as right hand sides of the dynamic equations of motion. Typical types of interactions are e.g. contact forces, modelled by linear or non-linear contact springs, or potentials such as Lennard-Jones potentials[4] that provide attractive or repelling forces depending on the distance of two particles.

2.2 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics[1] calculates field variables such as pressure and fluid velocities only at freely moving discrete points in space. As to every point a constant mass is assigned, it can be considered as a particle. Field variables can be evaluated by evaluating the kernel functions of the particles in the close vicinity and superimposing the results. These interpolation kernel functions, such as the Gaussian kernel, are required to have a finite cutoff radius that restricts their domain. By introducing this approximation in the strong form of the Navier-Stokes equations, the space- and time-dependent PDEs can be transformed into only time-dependent ODEs that describe the motion of the individual particles, yet considering for the effects imposed by pressure and viscosity of adjacent particles.

2.3 Neighbourhood Search

Both particle approaches require the determination of adjacent, potentially interacting particle pairs. To avoid a costly $O(n^2)$ search for interacting particle pairs out of n particles, there exists a large variety of different neighbourhood search algorithms that reduce the complexity to $O(n)$. The existing approaches come with different advantages and disadvantages for different types of interacting geometries[5,6,7,8,9]. To allow for general particle shapes and polydisperse particle systems, we combine two methods based on axis-aligned bounding boxes (AABB)[10,11] to gain a multi-purpose method that is well suited for a hierarchically structured application, an important prerequisite for our parallelization approach[12]. All geometrical entities, including boundaries, are defined as particles surrounded by bounding boxes. An interaction between two particles can only occur if their bounding boxes overlap. A necessary condition for a bounding box overlap is an overlap of the projections of the boxes on the three spatial axes, see Fig. 1.

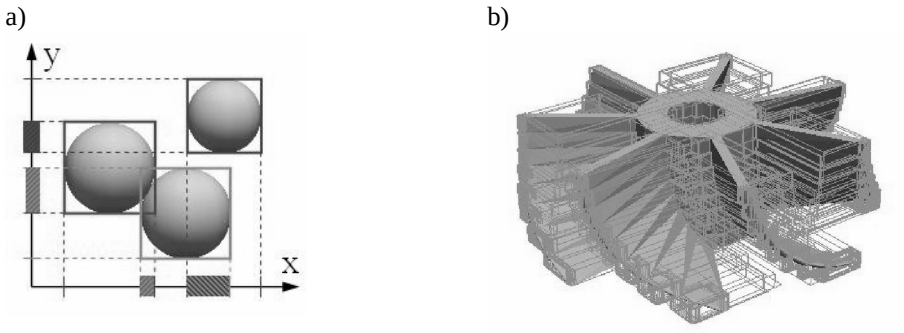


Figure 1. a) Neighbourhood search with axis aligned bounding boxes, depicted in 2D. Particle interactions only occur if the projections of two particles' bounding boxes on all three (3D) spatial axis overlap. b) Triangulated geometrical model of a turbine wheel with bounding boxes for neighbourhood search.

2.4 DEM-SPH Coupling

To couple a solid particle i and a SPH particle j, we employ Lennard-Jones potentials

$$d_{ij} = |\boldsymbol{r}_{ij}|, \tag{1}$$

$$\boldsymbol{n}_{ij} = \frac{\boldsymbol{r}_{ij}}{d_{ij}}, \tag{2}$$

$$\boldsymbol{F}_{ij}^{n} = \frac{\epsilon}{d_0}\left(\left(\frac{d_0}{d_{ij}}\right)^{7} - \left(\frac{d_0}{d_{ij}}\right)^{13}\right)\boldsymbol{n}_{ij}, \tag{3}$$

that may be interpreted as stiff nonlinear penalty forces. Their forces depend on the particle distance vector $\boldsymbol{r}_{ij}$ and a zero force distance d_0. To incorporate no-slip boundary conditions, we add the viscous tangential forces

$$\boldsymbol{v}_{ij}^{t} = \boldsymbol{v}_{ij} - (\boldsymbol{v}_{ij} \cdot \boldsymbol{n}_{ij})\,\boldsymbol{n}_{ij}, \tag{4}$$

$$\boldsymbol{F}_{ij}^{t} = -k\boldsymbol{v}_{ij}^{t} \tag{5}$$

to gain the total force

$$\boldsymbol{F}_{ij} = \boldsymbol{F}_{ij}^{n} + \boldsymbol{F}_{ij}^{t}. \tag{6}$$

A nearly incompressible fluid can thus be simulated by choosing the parameter ϵ of the Lennard-Jones potential in a way that yields a stiff repelling behaviour.

3 Simulation Boundaries

Simulations of particle-fluid systems in engineering applications often involve boundaries with complex geometry, e.g. provided as CAD data. We propose an approach that is based on a surface triangulation of the boundary geometry. There exists a large variety of open source tools for the triangulation of CAD data. All surface triangles are treated as individual particles in terms of neighbourhood search and their distribution to parallel processor nodes. Particle-triangle interactions are computed as described in[13] with Lennard-Jones

a) b)

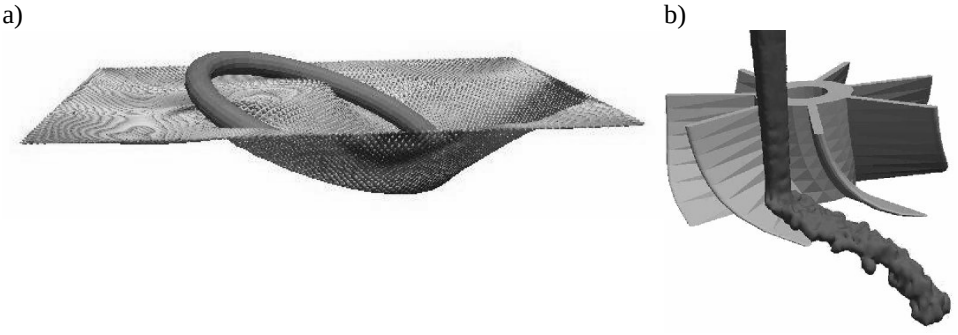


Figure 2. a) Simulation of a rigid torus falling on a membrane. The torus consists of surface triangles which are treated individually during neighbourhood search and interaction computation. b) Simulation of a turbine wheel driven by a liquid jet. The surface of the jet is reconstructed and rendered.

contact forces instead of penalty springs. Visco-elastic continua such as strings, membranes or solids are modelled by bonding particles with visco-elastic rods. For the simulation of dynamically moving rigid bodies, the triangulated surface geometry is defined relative to a moving frame of reference that possesses mass and inertia properties, such as the torus and the turbine wheel depicted in Fig. 2.

4 Parallel Simulation

For parallel simulations we apply a spatial domain decomposition in order to distribute particles to computation nodes of the parallel processor. Our approach is based on the point-to-point communication paradigm as featured by the Message Passing Interface (MPI) or the Parallel Virtual Machine (PVM). We employ a manager-worker modell where the manager process coordinates the work distribution and maintains the load balancing. Communication between the worker processes is initialized dynamically by the manager process, see Fig. 3.

4.1 Process Synchronization

Several substeps of the particle simulation loop require different communication patterns (m: manager, w: worker) and have to be partially synchronized:

- Neighbourhood search and interaction computation (w↔w),
- integration (w→m→w),
- particle inter-node migration (w→m→w, w↔w),
- post-processing data output (w→m),
- load-balancing (w→m→w).

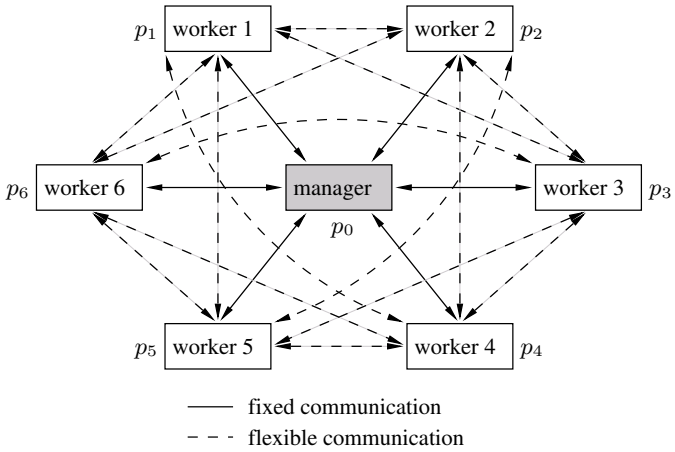


Figure 3. The communication between processes is based on a fixed-flexible communication pattern that is realized via point-to-point communication in MPI. Any communication between the worker processes that is based on the system state (flexible) is initialized by the manager process.

There are substeps which require a great amount of worker to worker communication (w↔w), such as neighbourhood search, and others that require no worker to worker communication, such as post-processing data output, which are handled by the manager process. As we apply point-to-point communication, there is no need to use semaphores to ensure the synchronization of all nodes in all substeps. The worker processes can be allowed to work on until the next synchronization point occurs.

4.2 Domain Decomposition

As a domain decomposition scheme that can be dynamically adapted, we chose Orthogonal Recursive Bisection (ORB)[14,15]. Processors are assigned to subdomains that are logically corresponding to the leaves of the binary tree that emerges from the ORB decomposition. To setup the decomposition, subdomains are consecutively divided in half along one of the three spatial axis. The requirement that the particles in the two emerging subgroups match the accumulated computational power of the nodes in the two corresponding subtrees[12] serves as a constraint for the placement of the division boundary, see Fig. 4. For the placement of the subdivision boundaries, we divide a particle cloud in two subclouds whose cardinalities match the same ratio as two given accumulated computational power values. This task is accomplished by a sampling of the particle positions on a regular grid. The grid based distribution function is used to compute normalized cumulative particle density functions for the three spatial dimensions which can in turn be evaluated to find the appropriate boundary position.

4.3 Load Balancing

A balanced distribution of the simulation work is crucial to gain optimal performance in parallel computations. Overworked nodes can slow down a parallel computation and can

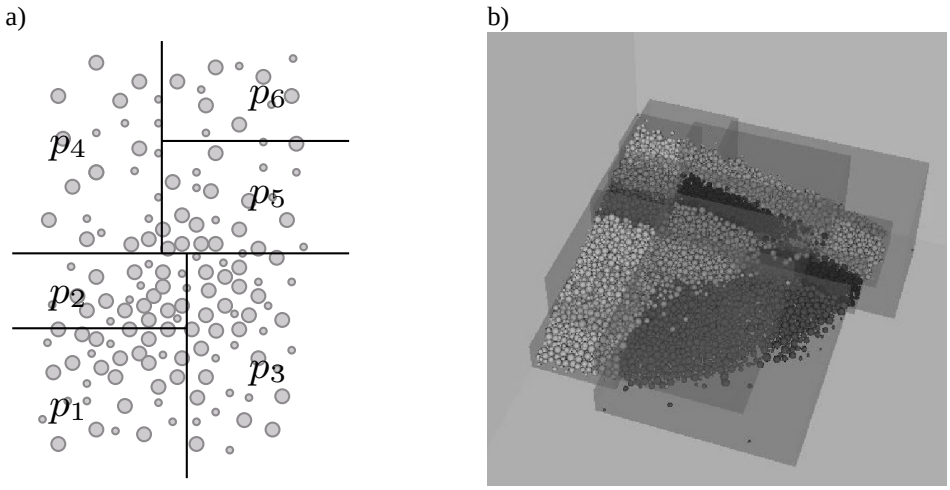


Figure 4. Orthogonal Recursive Bisection: a) The simulation domain is recursively bisected to obtain as many subdomains as nodes. b) Assignment of particles to 16 nodes (coloured) for a 3D simulation.

thus lead to execution times that are in the worst case even longer than that of equivalent sequential computations. To avoid imbalances, the workload needs to be continuously redistributed between the worker nodes. There are two main reasons for an imbalance of workload. Firstly, particle motion requires migration of particles between processors and can thus cause an imbalance of the number of particles assigned to the nodes. Secondly, other users can run competitive jobs that may influence the current performance of the nodes. The latter, however, only occurs in clusters of workstations where nodes are not exclusively reserved for a particular job.

Our load balancing approach works independent from the reason for the imbalance. The basic idea is to shift subdomains boundaries on every level of the ORB-tree. Shifting boundaries causes particles to be migrated to neighboring domains and thus leads to an improved distribution of workload. As imbalances are non deterministic, a control-approach is required. We employ a hierarchical proportional-integral (PI)-controller[16] with the differences of computation times of two twin nodes as controller input. The same approach is applied on all levels of the ORB-tree with groups of interior nodes instead of leafs. Therefore, the computation times of the nodes in a group are accumulated and the differences between the accumulated computation times of two twin groups are used as controller input[12]. See Fig. 5 for an example of the control process employing six worker nodes.

4.4 Performance

As a benchmark for the performance of our simulation approach, we adopted the collapsing block example as depicted in Fig. 4. Simulation series were performed on a cluster of Pentium 4 workstations with ten thousand, hundred thousand and one million particles. The results are depicted in Fig. 6. As expected, the scaling behaviour improves with increasing problem size. However, performance is significantly affected by communication.

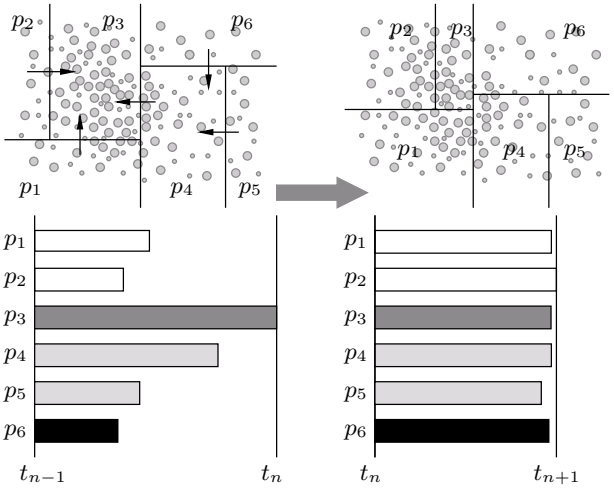


Figure 5. Controller approaches for a six node scheme. During a discrete controller step, all subdivision boundaries of the ORB decomposition are shifted. After the controller step the nodes' per step wall clock computation times have adapted and the total per time step wall time has been reduced.

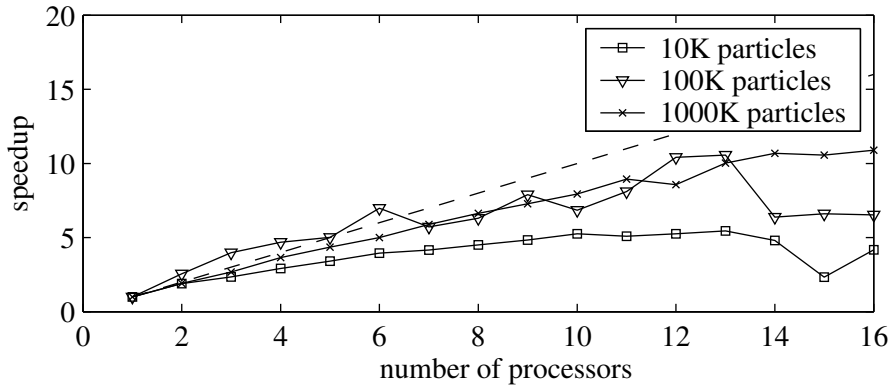


Figure 6. Parallel speedup of simulations of the collapsing particles example, see Fig. 4, for simulation series with ten thousand, hundred thousand and one million particles. The number of processors includes the manager and the worker processes.

5 Summary

We presented a new parallel method for the simulation of particle-fluid systems in a pure Lagrangian way that does not involve any grid based methods. Our method is thus well suited for applications where the simulation domain is not a priory known, such as free surface flows or flows with moving, e.g. elastic boundaries. The method combines Smoothed Particle Hydrodynamics and the Discrete Element Method, both meshless particle meth-

ods. The load balancing approach is based on Orthogonal Recursive Bisection. It is capable of maintaining load balance even on clusters of workstations with competitive jobs. Its PI-controller, controlls the amount of particles assigned to the different nodes based on the differences of their per step computation times.

References

1. J. J. Monaghan, *Smoothed particle hydrodynamics*, Annual Reviews in Astronomy and Astrophysics, **30**, 543–574, (1992).
2. P. A. Cundall, *A computer model for simulation of progressive, large-scale movements in blocky rock systems*, in: Proc. Symposium of the International Society of Rock Mechanics, Nancy, (1971).
3. P. A. Cundall and O. D. L. Strack, *A discrete numerical model for granular assemblies*, Geotechnique, **29**, 47–56, (1979).
4. L. Verlet, *Computer experiments on classical fluids*, Physics Reviews, **159** ,98–103, (1967).
5. G. v.d. Bergen, *Efficient collision detection of complex deformable models using aabb trees*, J. Graphics Tools Archive, **2**, 1–13, (1998).
6. D. James and D. Pai, *Bd-tree: Output-sensitive collision detection for reduced deformable models*, in: ACM Trans. Graphics (SIGGRAPH 2004), (2004).
7. D. S. Coming and O. G. Staadt, *Kinetic sweep and prune for multi-body continuous motions*, in: Second Workshop in Virtual Reality Interactions and Physical Simulations, (2005).
8. M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, and W. Strasser, *Collision detection for deformable objects*, in: Eurographics State-of-the-Art Report (EG-STAR), pp. 119–139, Eurographics Association, (2004).
9. M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets and M. Gross, *Optimized spatial hashing for collision detection of deformable objects*, in: Proc. Vision, Modeling, Visualization VMV'03, pp. 47–54, (2003).
10. A. Schinner, *Fast algorithms for the simulation of polygonal particles*, Granular Matter, **2**, 35–43, (1999).
11. E. Perkins and J. R. Williams, *A fast contact detection algorithm insensitive to object sizes*, Engineering Computations, **18**, 48–61, (2001).
12. F. Fleißner and P. Eberhard, *Parallel load balanced simulation for short range interaction particle methods with hierarchical particle grouping based on orthogonal recursive bisection*, International Journal for Numerical Methods in Engineering, (2007, in press).
13. F. Fleißner, T. Gaugele, and P. Eberhard, *Applications of the discrete element method in mechanical engineering*, Multibody System Dynamics, **18**, 81–94, (2007).
14. J. Salmon, *Parallel hierarchical N-body methods*, PhD thesis, Caltech University, (1990).
15. M. S.Warren and J. K. Salmon, *A parallel hashed oct-tree n-body algorithm*, in: Proc. Supercomputing '93, pp. 12–21, (1993).
16. R. C. Dorf and R. H. Bishop, *Modern control systems*, (Prentice Hall, New Jersey, 2001).

Communication and Load Balancing of Force-Decomposition Algorithms for Parallel Molecular Dynamics

Godehard Sutmann[1] **and Florian Janoschek**[2]

[1] Jülich Supercomputing Centre (JSC)
Research Centre Jülich (FZJ)
D-52425 Jülich, Germany
E-mail: g.sutmann@fz-juelich.de

[2] Stuttgart University
Institute for Computational Physics, Pfaffenwaldring 27
D - 70569 Stuttgart, Germany
E-mail: FJanoschek@gmx.net

A new implementation of a force decomposition method for parallel molecular dynamics simulations is presented. It is based on a geometrical decomposition of the influence matrix where sections are dynamically reorganized during the simulation in order to maintain a good load balance. Furthermore space filling curves are used to sort particles in space, which makes memory access more efficient and furthermore reduces communication between processors due to better locality. Benchmark runs are presented which shows in improvement in scalability due to well balanced work on the processors.

1 Introduction

Classical molecular dynamics simulations are often considered as the *method par excellence* to be ported to parallel computers, promising a good scaling behaviour. On the one hand parallel algorithms exist which enable good scaling. On the other hand the complexity of the problem at hand, often scales like $\mathcal{O}(N)$, enabling a linear increase of problem size with memory. However, this point of view applies only to a limited class of problems which can be tackled by molecular dynamics. E.g. in the case of homogeneous periodic systems, where particles interact via short range interactions, the most efficient algorithm is a domain decomposition scheme, guaranteeing local communication between processors and therefore allowing good parallel scaling. In combination with linked-cell lists, the problem scales like $\mathcal{O}(N)$ both in computational complexity and memory demand, so that an ideal behaviour in both strong and weak scaling might be expected.

On the other hand, this ideal behaviour breaks down if different problem classes are considered, e.g. the case of long range interactions, where not only local communications between processors are required. Another class of counter examples is the case of inhomogeneous systems, which occur e.g. in open systems, where the particle density is considerably larger in the centre of the system than in the diffuse halo or e.g. in systems consisting of different thermodynamic phases as is the case for the coexistence of liquid/-gas or solid/gas phases. In this case, domain decomposition algorithms often fail. Due to a more or less regular geometric decomposition of space, processors are responsible for different numbers of particles, often introducing a strong load-imbalance, which leads to inefficient CPU usage on some processors and therefore to a bad parallel scaling.

This fact requires some sort of load-balancing strategy, which is able to rearrange domains dynamically in order to achieve equal work load through a simulation, which might pass a non-homogenous dynamical particle flow across processors.

In literature, there exist several different approaches to solve the problem of load-imbalance. A general diffusion scheme, which uses near neighbour information was introduced in Ref.[1]. For two-dimensional MD simulations a one-dimensional Cellular Automaton Diffusion scheme was proposed in Refs.[2,3], which was further generalized to three-dimensional MD in Ref.[4] via the concept of *permanent cell* to minimize interprocessor communication. For one- and two-dimensional MD, based on a parallel link-cell domain decomposition method, the Multilevel Averaging Weight was introduced in Ref.[5]. For fully three dimensional MD simulations, based on a linked-cell domain decomposition strategy a load-balancer was proposed in Ref.[6]. For a force-decomposition method, based on Ref.[7], a method based on the generalized dimension exchange was presented in Ref.[8].

In the following a new decomposition scheme, based on an a distributed data model with MPI communication protocol, is presented. It combines elements from force- and domain-decomposition approaches. First, the basic concept is introduced. This is extended to reduce communication overhead and combined with a load balancing strategy. Finally, benchmark results are presented for different kinds of model systems.

2 Force-Domain-Decomposition Scheme

2.1 The Connectivity Matrix

The connectivity matrix $\mathbf{C} \in \mathbb{N}^{N\times N}$ contains information of the system, which particle pairs (i,j) interact with each other, i.e. $C_{ij} = \{1 : \|\mathbf{r}_i - \mathbf{r}_j\| \in \Omega_I\}$, where $\mathbf{r}_k$ is the geometric coordinate of particle k and Ω_I is the range of interaction. For short range interactions, this corresponds to a cutoff radius R_c, beyond of which interactions are switched off. If $\mathbf{C}$ is dense, this corresponds to long range interactions. On the other hand, short range interactions will result in a sparse structure. Since mutual interactions are symmetric, it is clear that also $\mathbf{C}$ is symmetric. Furthermore it is traceless, because there are no self interactions of particles considered. Therefore, taking $\mathbf{C}$ as the basic building block for a parallel algorithm, it is sufficient to consider the upper triangular matrix $\mathbf{C}^u$. Considering on average the system to be homogeneously distributed, a parallel strategy consists in distributing domains of equal area of $\mathbf{C}^u$ to the processors. Various methods were discussed in Ref.[9]. The one, which forms the basis for the following implementation is the stripped-row method. Subsequent rows are combined to blocks, which have nearly similar size, A. The recursive relation

$$X_k = \frac{1}{2}\left(Q_k - \sqrt{Q_k^2 - \frac{4N(N-1)}{P}}\right) \tag{1}$$

with

$$Q_k = 2N - 1 - 2\sum_{j=0}^{k-1} X_j \tag{2}$$

defines a fractional number of rows in $\mathbf{C}$. Taking the integer part $L_k = \lfloor X_k \rfloor$, the area, which is assigned to each processor is calculated as

$$A(L_k) = \left(N - \sum_{j=0}^{k-1} L_j - \frac{L_k + 1}{2} \right) L_k \tag{3}$$

Here, $k \in [0, P-1]$ and consistency has to give $\sum_{k=0}^{P-1} L_k = N$. This scheme defines a load balanced distribution, if the matrix is dense or if the probability of $P(j > i|i)$ for finding a particle $j > i$ for a given particle i in Ω_I is constant. In addition, this scheme will also only work satisfactorily, if memory access is uniform all over the system.

2.2 Communication

Considering a system, where particles are free to move in the simulation, it is not clear *a priori*, which particle pairs $(i, j) \in \Omega_I$. Therefore, traditional schemes of force decomposition algorithms require a replication of the whole system[9] or replication of full subsets of coordinates[7]. In both cases positions are usually distributed by an `mpi_allgatherv` command while forces are summed up by an `mpi_allreduce` procedure. This is certainly the most simple way to proceed. Since most MPI implementations internally make use of a tree-wise communication protocol, the global communication will scale like $\mathcal{O}(\log_2(P))$, if P is the number of processors.

In general there are two steps where communication between processors is required: (i) Exchange of coordinates. Due to the partitioning of the connectivity matrix in the present algorithm, coordinates are transferred only to processors with lower rank than the local one, i.e. the processor with rank $P-1$ only calculates local forces, but it sends coordinates to remote processors. (ii) Exchange of forces. Due to the skew-symmetric nature of forces between particles i, j, forces are sent analogously, but in the back direction of the coordinate flow.

However, for big systems and large number of processors, there will be a lot of redundant data transferred between processors. I.e. global communication operations do not take into account whether transferred data are really needed on remote processors. Therefore an alternative method is considered here.

The basic principle is to combine a domain decomposition with a force decomposition scheme. Domain decomposition is achieved by sorting the particles according to their positions along a space filling Hilbert curve[10]. This ensures that most particles which are local in space are also local in memory. For short range interactions this implies that most interaction partners are stored already on the same processor. According to this organisation, the force matrix becomes dominant around the diagonal and off-diagonal areas are sparse. The next step is to calculate interaction lists, which are distributed onto all processors. In the present implementation, interactions are calculated via Verlet lists, which store indices of particle pairs $(i, j) \in \Omega_{I+R}$, where Ω_{I+R} is the actual interaction range plus a reservoir of potentially interacting particles j. Therefore, during construction of these lists, exchange lists, $\mathcal{L}_X(p\{I\} \rightarrow p'\{J\})$, can be created, which store information, which index set $\{I\}$ from the local processor p has to be transferred to the remote processor p' to calculate interactions with particles from index set $\{J\}$. On the other hand this immediately gives a list $\mathcal{L}_F(p'\{J\} \rightarrow p\{I\})$ which contains information, which forces have to be sent

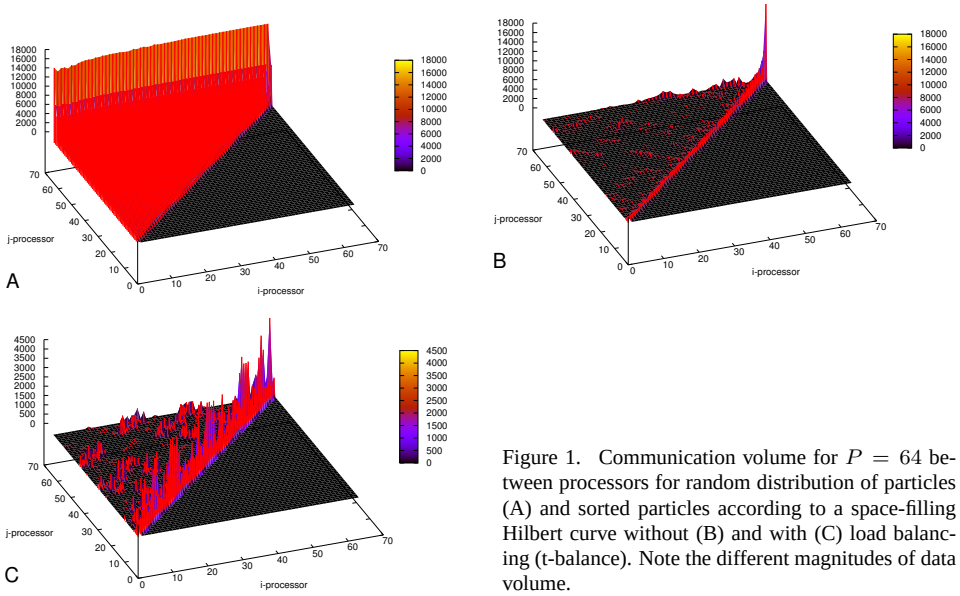


Figure 1. Communication volume for $P = 64$ between processors for random distribution of particles (A) and sorted particles according to a space-filling Hilbert curve without (B) and with (C) load balancing (t-balance). Note the different magnitudes of data volume.

from processor p to p'. It is the latter list, which is communicated to processors with lower rank than the local one. Verlet lists only have to be created from time to time (this may be controlled with an automatic update criterion[11]). Therefore, neighbour lists, as well as exchange lists only have to be created every one of these time intervals (usually neighbour lists are updated after ≈ 20 timesteps, depending on list parameters[11]).

Since the amount of data is very small with respect to global communications of positions and forces, the communication overhead of this method is strongly reduced, enabling a very much better parallel scaling. Since randomisation of particle positions occur on a diffusive time scale, the space filling curve has to be updated only one or two orders of magnitude less than the interaction lists, thus introducing only a small overhead. Therefore, it is not a main bottleneck that at the moment, sorting is done with a sequential algorithm. However, parallel algorithms exist, which will be built into the algorithm in near future.

2.3 Load-Balancing

Having set up the particle distribution according to Eq. 3, does not guarantee a load balance among processors. Considering e.g. inhomogeneous systems, where particles cluster together, this may lead to different work load on processors. If particles on one processor mainly belong to a certain cluster, in which they are densely packed, they will create a large number of interactions on this processor. This might be different on another one, which stores particles belonging to a diffusive halo, and therefore generating less interactions.

Therefore, for inhomogeneous systems, this approach might become quite inefficient. Therefore, two different work measures are defined which lead to two different load balancing strategies. The first is based on the number of interactions (n-balance), which are

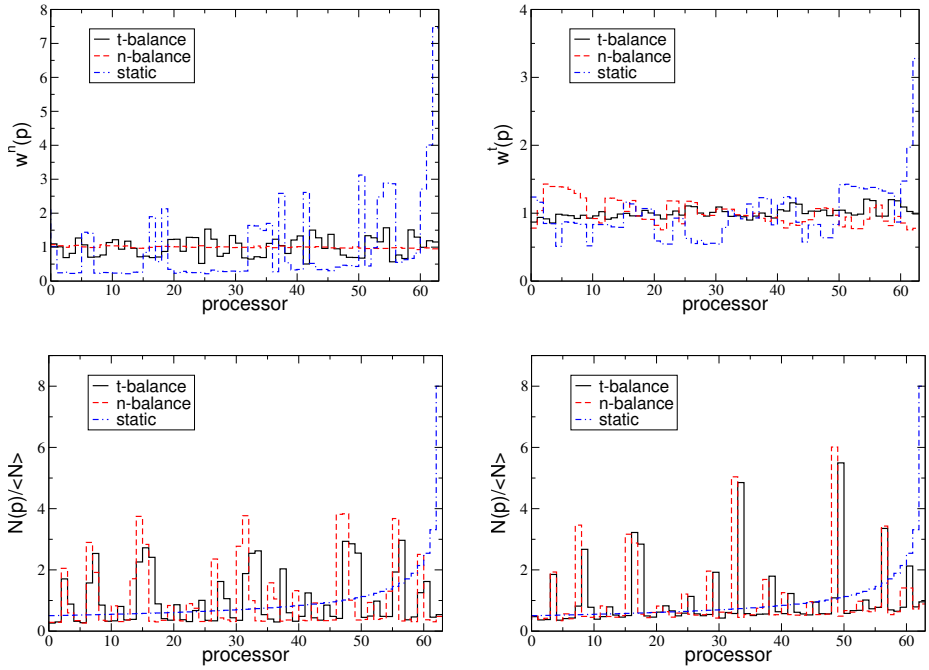


Figure 2. Work load of the static-, t-balance- and n-balance schemes. Upper left: number of interactions, $w^n(p)$; upper right: CPU time, $w^t(p)$, lower left: relative number of particles/processor. All cases for inhomogeneous system. Lower right: relative number of particles/processor for the homogenous system.

calculated on every processor, $n(p_i)$, $i \in [0, P-1]$. Here, one may define the ratio

$$w^n(p_i) = \frac{n(p_i)}{\langle n \rangle_P} \quad , \quad \langle n \rangle_P = \frac{1}{P} \sum_i n(p_i) \tag{4}$$

The second method is similarly defined, but uses the time $t(p_i)$, consumed on every processor (t-balance)

$$w^t(p_i) = \frac{t(p_i)}{\langle t \rangle_P} \quad , \quad \langle t \rangle_P = \frac{1}{P} \sum_i t(p_i) \tag{5}$$

In both cases, if $w = 1 \; \forall \; p_i$, a perfect balance for the force routine is achieved.

Practically, the load balance routine is called, before the interaction lists are being constructed. Within the time interval between list updates, each processor gathers information about the number of interactions or the consumed time [a]. In the load-balancing step, the average quantities per row is calculated. For n-balance, an `mpi_allgatherv` operation collects the number of local interactions on every processor. This vector is then analysed and it is decided, how many rows, i.e. particle information, are transferred or obtained from $p_i \rightarrow p_{i-1}$ and $p_i \leftarrow p_{i+1}$. Similarly it is proceeded in the case of t-balance. Here,

[a] At present, only the CPU time which is consumed in the force routine is considered. Since this is both the most time consuming routine and the only one in which loop structures are likely to develop an imbalance, an even distribution of this time leads to an overall balance of the processes.

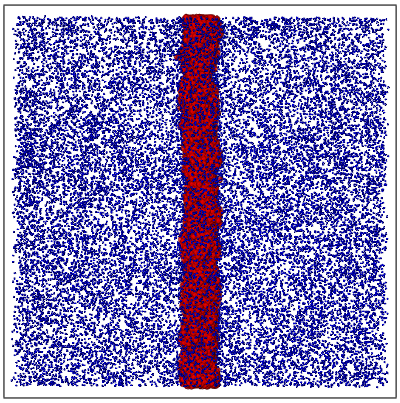


Figure 3. Snapshot of the inhomogeneous system after several thousand time steps. The heavy, immobile particles form a wire structure in a gaseous environment.

`mpi_allgatherv` operation collects the total time, spent on a processor. In a second step, this is translated to an average time t_i per particle, i.e. per row. It was found out that this is more precise and efficient than the measurement of time for each individual particle. According to the time t_i a repartitioning of rows can be performed across processor boundaries.

Since the load imbalance develops with the movement of particles across geometrical domain regions, it is the diffusive time scale which governs this process. Calling the load balance routines with a similar frequency than interaction lists, this ensures a rapid convergence of the load. Also, developing inhomogeneities may be easily compensated by this scheme. Nevertheless, it should be noted that the balancing routine always uses information from the past in order to construct the particle distribution for the next future. Therefore, load balancing is most likely to be not perfect.

3 Benchmark Results

In order to test the algorithms, two types of systems were considered: (i) a homogenous one and (ii) an inhomogeneous one. In both cases, particles interact via a modified Lennard-Jones interaction, which avoids the potential divergence at small particle separations

$$u_{ij}(r_{ij}) = \begin{cases} 48\epsilon \dfrac{1}{\pi} \cos\left(\dfrac{\pi}{2}\dfrac{r_{ij}}{\sigma_{ij}}\right) & r_{ij} < \sigma_{ij} \\ 4\epsilon_{ij}\left(\left(\dfrac{\sigma_{ij}}{r_{ij}}\right)^{12} - \left(\dfrac{\sigma_{ij}}{r_{ij}}\right)^{6}\right) & r_{ij} \geq \sigma_{ij} \\ 0 & r_{ij} > R_c \end{cases} \tag{6}$$

The homogenous system consists of a binary mixture with parameters $\epsilon_{11} = 120\ K$, $\sigma_{11} = 3$Å and $\epsilon_{22} = 160\ K$, $\sigma_{22} = 3$Å. Combinations are taken into account via usual Berthelot mixing rules. The number density was $\rho = 0.02$ Å^{-3}, the temperature of the system was set to $T = 90\ K$ and masses were taken from Argon and Xenon ($m_1 = 39.9\ amu$, $m_2 = 131.8\ amu$). The cutoff-radius for particle interactions was set to $R_c = 2.5\ \sigma_{ij}$.

In the case of inhomogeneous system, one particle species was assigned an artificially large mass ($m_1 = 10000\ amu$), in order to slow down dynamics and to cluster in the

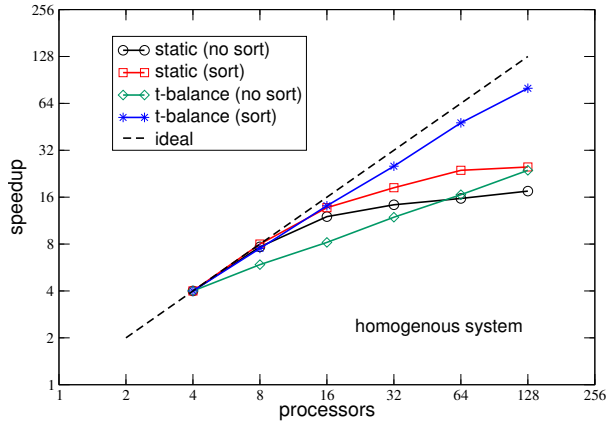


Figure 4. Parallel speedup for the force-domain decomposition method for the case of $N = 10^5$ particles and two different cutoff radii for interparticle interactions.

centre of the simulation box as a wire structure (cmp. Fig. 3). The density was reduced to $\rho = 0.002$ Å^{-3}. Other parameters were $\epsilon_{11} = 50\ K$, $\sigma_{11} = 3$Å, $\epsilon_{22} = 350\ K$, $\sigma_{22} = 3$Å, $T = 150\ K$. Fig. 1 shows results for the load balancing on 64 processors, which was obtained after about 1000 simulation steps. The initial particle distribution onto processors was always chosen according to Eq. 3. Shown is the case for the inhomogeneous system where the load of every processor is measured in terms of number of interactions, CPU time and number of particles, resident on a processor. As it is shown, the static partitioning gives in all cases the worst result. Without load balancing, the processors were out of equal load by a factor of 15. Applying the n-balance scheme, clearly equilibrate the number of interactions within a band of about 10%. However, as is seen from the workload in terms of CPU time, there is still a clear imbalance of about 40%. This is because (i) of different speed of memory access (cache effects are not accounted for in the model) and (ii) because of different waiting times in the communication procedure. If one applies t-balance, a very much better load is observed for the CPU times, while now the number of interactions per processor shows a larger variance ($\approx 50\%$). Looking at the distribution of particle numbers, resident on each processor, a rather non-trivial distribution is found. This is partly due to the distribution of heavy wire particles, which are located in the centre of the simulation box. Using $P = 64$ processors and Hilbert curve sorting of particles, means that approximately half of the processors share heavy particles, while the other half is responsible only for light atoms. For a balanced simulation, processors with only light particles will store more particles, since they exhibit less number of interactions per particle (less dense packed).

Finally a speedup curve for the interaction routine including communication routines is presented for the algorithm. The measurement was performed on the JULI cluster at ZAM/FZJ, consisting of Power 5 processors[12,13]. Shown are results for a homogenous system with $N = 10^5$ particles, where a interaction range of $R_c = 8.5$Å was applied. Cases for static distributions with and without particle sorting are presented as well as those for t-balance. Static distributions saturate already at ≈ 32 processors. Best scaling is

achieved for t-balance with particle sorting. If no sorting is applied, scaling is not linear, because the larger amount of transferred data.

It is noteworthy to state that at present a bottleneck of the method is the construction of interaction lists. At the moment, Verlet lists, from where interaction lists are derived, are constructed by collecting all particles on each processor via an `mpi_allgatherv` operation and then testing mutual distances between particles. Since this is done not frequently, it does not dominate the communication. Nevertheless it limits the overall speedup of the algorithm. To solve this problem, it is planned to use link-cell-lists, in stead of Verlet-lists, in future and to reduce coordinate exchange between processors in the load-balance step to the next smaller and larger ranks. Only in extreme cases, larger rearrangements in index space would be necessary.

References

1. M. H. Wellebeek-LeMair and A. P. Reeves, *Strategies for dynamic load balancing on highly parallel computers*, IEEE Trans. Parall. Distr. Sys., **4**, 979–993, (1993).
2. F. Bruge and S. L. Fornili, *Concurrent molecular dynamics simulation of spinodal phase transition on transputer arrays*, Comp. Phys. Comm., **60**, 31–38, (1990).
3. G. A. Koring, *Dynamic load balancing for parallelized particle simulations on MIMD computers*, Parallel Computing, **21**, 683, (1995).
4. R. Hayashi and S. Horiguchi, *Relationships between efficiency of dynamic load balancing and particle concentration for parallel molecular dynamics simulation*, in: Proc. of HPC Asia '99, pp. 976–983, (1999).
5. M. Zeyao, Z. Jinglin, and C. Qingdong, *Dynamic load balancing for short-range molecular dynamics simulations*, Intern. J. Comp. Math., **79**, 165–177, (2002).
6. N. Sato and J. M. Jezequel, *Implementing and evaluating an efficient dynamic load-balancer for distributed molecular dynamics simulation*, in: International workshop on Parallel Processing, pp. 277–283, (2000).
7. S. Plimpton and B. Hendrickson, *A new parallel method for molecular dynamics simulation of macromolecules*, J. Comp. Chem., **17**, 326, (1996).
8. A. Di Serio and M.B. Ibanez, *Distributed load balancing for molecular dynamics simulations*, in: 16th Annual International Symposium on High Performance Computing Systems and Applications, pp. 284–289, (2002).
9. R. Murty and D. Okunbor, *Efficient parallel algorithms for molecular dynamics simulations*, Parallel Comp., **25**, 217–230 (1999).
10. M. Griebel, S. Knapek, G. Zumbusch, and A. Caglar, *Numerische Simulationen in der Moleküldynamik*, (Springer, Berlin, 2004).
11. G. Sutmann and V. Stegailov, *Optimization of neighbor list techniques in liquid matter simulations*, J. Mol. Liq., **125**, 197–203, (2006).
12. *JULI Project – Final Report*, Eds. U. Detert, A. Thomasch, N. Eicker, J. Broughton, FZJ-ZAM-IB-2007-05, (2007).
13. http://www.fz-juelich.de/zam/JULI

Aspects of a Parallel Molecular Dynamics Software for Nano-Fluidics

Martin Bernreuther[1], **Martin Buchholz**[2], **and Hans-Joachim Bungartz**[2]

[1] Höchstleistungsrechenzentrum Stuttgart
70550 Stuttgart, Germany
E-mail: bernreuther@hlrs.de

[2] Institut für Informatik, Technische Universität München
85748 Garching, Germany
E-mail: {buchholm, bungartz}@in.tum.de

The simulation of fluid flow on the nano-scale in the field of process engineering involves a large number of relatively small molecules. The systems we are simulating can be modelled using rigid molecular models assembled from sites with non-bonded short-range pair potentials Each of the sites is described by a set of parameters which are required for the calculation of interactions between sites of the same type. For the interaction of unequal sites, mixed parameter sets have to be calculated. This has to be done for each possible pair of sites. We describe an approach to precalculate and store those mixed parameter sets in a stream, which allows efficient access and gives the flexibility to add new site types easily.

Another focus of our work has been on software engineering techniques. Using the adapter design pattern, we achieved a complete decoupling of the physical parts of the simulation (e.g. molecule models and interactions) from the data structures and the parallelisation. This eases the further concurrent development of the software and reduces the complexity of the different modules. It also gives us the opportunity to swap modules in a plug-in like fashion.

Finally, we demonstrate the advantages of a "pair ownership" of processes for the parallelisation which allows the joint calculation of macroscopic values and the forces on molecules.

1 Introduction

Molecular dynamics generally is about solving the molecules' equations of motion[3,1]. To be able to do that, the forces which act upon each molecule have to be calculated in each time step. The calculation of the forces is based on pair potentials. For each combination of different molecules, a different set of parameters is needed for the force calculation. Section 3 describes a method we have developed to handle those parameter sets efficiently.

As we are using only short-range potentials[5], all necessary pairs can be found by iterating for each molecule over all neighbouring molecules (e.g. using Verlet neighbour lists[3] or a linked-cell data structure[4]). Doing this, one has to take care not to calculate some forces twice. Forces result from pairwise potentials. Hence, processing each pair of molecules once guarantees that all necessary calculations are done once only. We found it useful to clearly distinguish between things to be done for molecules and things to be done for pairs of molecules (see Section 4). In our code, this and the use of modular programming (See Section 2) has led to a better design of the data structures and the parallelisation using domain decomposition.

2 Software Engineering Aspects

For molecular dynamics software, good performance is a crucial property. Unfortunately, efficient programming often conflicts with good software engineering. On the basis of our data structure "ParticleContainer" used to store molecules we will explain how we try to resolve this conflict in our program. In our previous implementation, the particle container was not only responsible for storing the molecules but also for calculating the forces which act on the molecules. The reason for this is that only the particle container class knows how to efficiently access molecules and their neighbours. So the container class had to have information about how to calculate forces. There are two problems with this approach:

- The person implementing the data structure shouldn't have to know much about the calculation of forces.
- The program should be capable to handle different molecule models and a variety of intermolecular potentials. This requires a generic data structure.

We use the adapter design pattern[9] shown in Fig. 1 to get rid of both problems. An adapter class is basically just a wrapper class which connects the particle data structure with some class responsible for the pairwise interactions.

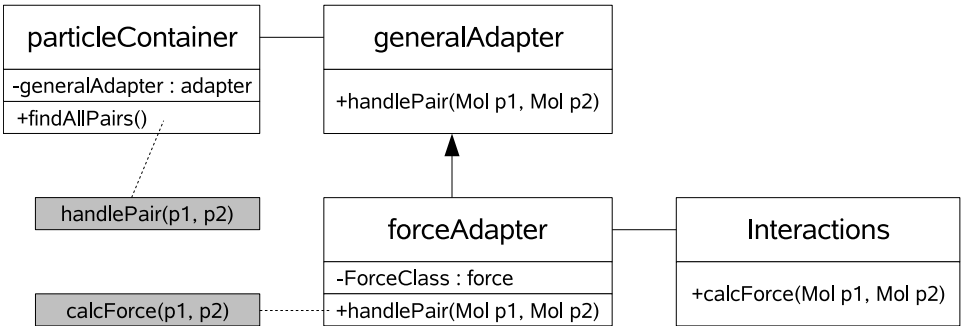


Figure 1. Adapter for pairs

The "ParticleContainer" does not contain any information about how the molecules are modelled and what type of interaction should be used. It just has to provide a method which iterates over all pairs and calls the adapter's "handlePair" method for each pair. Depending e.g. on the material that has to be simulated, a different implementation of the adapter will be used. Typical tasks for an adapter are:

- to call methods which evaluate the potential function for the pair,
- to increase the force on each of the two particles by the calculated value,
- to add contribution of the pair to macroscopic values (e.g. potential, virial,...).

Using this adapter we achieve a high modularity in our program. Different implementations of the ParticleContainer can easily be integrated.

We described how we achieved modularity only on the basis of the data structure "ParticleContainer", but also other parts of the program are designed as flexible modules, expecially the parallelisation and the integrator. This gives us the opportunity to easily develop and test new approaches, e.g. a new parallelisation, and plug them into the program with minimal effort.

3 Force Calculation Based on Parameter Streaming

In the present work molecular dynamics simulations of fluids are realized using rigid molecular models assembled from sites with non-bonded, short range potentials. First of all the Lennard-Jones 12-6 potential

$$U_{ij}^{LJ}\left(r_{ij}\right) = 4\epsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right) \tag{1}$$

covers repulsion and dispersive attraction.

In contrast to the distance $r_{ij} = \left|\vec{r_{ij}}\right| = \left|\vec{r_j} - \vec{r_i}\right|$ between two atoms i and j, the atom diameter σ as well as the energy parameter ϵ are constant. Mixtures of different molecule types, called components, are also supported. The modified Lorentz-Berthelot combining rules

$$\sigma_{AB} = \eta_{AB}\frac{\sigma_A + \sigma_B}{2} \tag{2a}$$

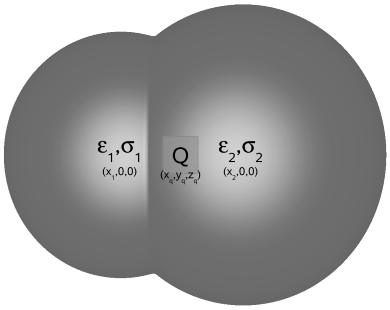
$$\epsilon_{AB} = \xi_{AB}\sqrt{\epsilon_A \epsilon_B} \tag{2b}$$

provide a good starting point to determine the unlike Lennard-Jones parameters for interactions between two different atoms. Whereas η is usually set to one, the binary interaction parameter ξ is a factor close to unity. Assemblies of LJ-centres are well suited for a wide range of anisotropic non-polar molecule types. Quadrupoles or dipoles have to be added to model polar molecules. The corresponding potentials are parameterized with a strength Q or μ for each quadrupole or dipole resp. The force F acting on an atom is directly related to the potential: $\vec{F}_{ij} = -\operatorname{grad} U\left(r_{ij}\right)$.

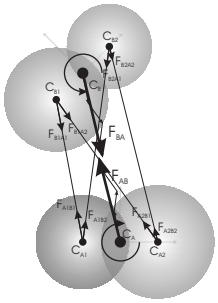
Regarding multi-centred molecules, the position for each site depends on the position and orientation of the molecule, as well as the local position of the site. The resulting force and torque acting on the molecule are calculated through a vector summation of all the corresponding site-site parts, where each LJ-centre of one molecule interacts with each LJ-centre of the other molecule as well as Dipoles and Quadrupoles interact with each other.

Even though the procedure and implementation works for mixtures of multi-centred molecules with an arbitrary number of sites, the method is demonstrated for the quadrupole two centred Lennard-Jones (2CLJQ) molecule type (cmp. fig. 2(a)). This subclass is suitable to model substances like Nitrogen (N_2), Oxygen (O_2) or Carbon Dioxide (CO_2) and was also successfully applied to binary and ternary mixtures[8]. The potential of and force between two 2CLJQ molecules i and j interacting with each other is given by

$$U_{ij}^{2CLJQ} = 4\sum_{\substack{k=1\\ comp(i)}}^{2}\sum_{\substack{l=1\\ comp(j)}}^{2}\epsilon_{kl}\left(\left(\frac{\sigma_{kl}^2}{r_{kl}^2}\right)^6 - \left(\frac{\sigma_{kl}^2}{r_{kl}^2}\right)^3\right) + \frac{3}{4}\frac{Q_i Q_j}{\left|r_{ij}\right|^5} f(\omega_i, \omega_j) \tag{3a}$$



(a) 2CLJQ molecule

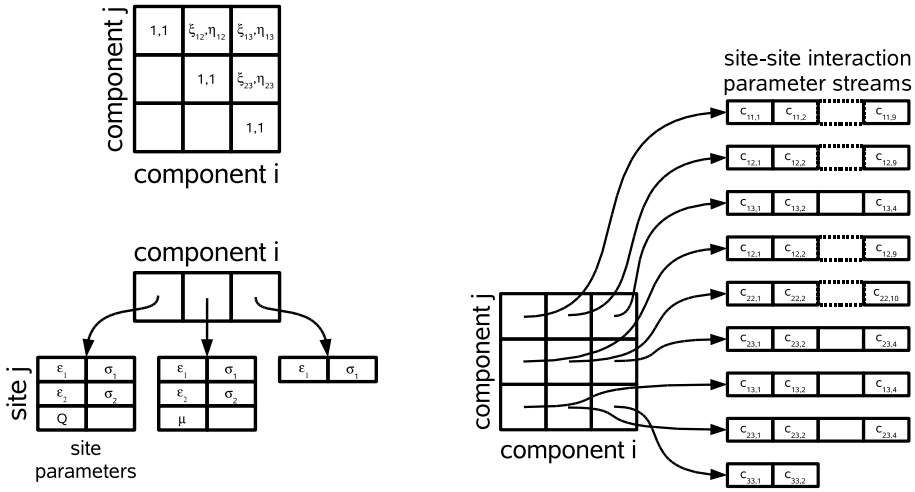


(b) Interactions between LJ centres

Figure 2. Exemplification component

$$\vec{F}_{ij}^{2CLJQ} = 24 \sum_{\substack{k=1 \\ comp(i)}}^{2} \sum_{\substack{l=1 \\ comp(j)}}^{2} \epsilon_{kl} \left(2 \left(\frac{\sigma_{kl}^2}{r_{kl}^2} \right)^6 - \left(\frac{\sigma_{kl}^2}{r_{kl}^2} \right)^3 \right) \frac{\vec{r}_{kl}}{r_{kl}^2} + \frac{15}{4} \frac{Q_i Q_j}{r_{ij}^6} f(\omega_i, \omega_j) \frac{\vec{r}_{ij}}{|r_{ij}|} \tag{3b}$$

f is a function related to the orientation ω of the quadrupoles.



(a) storing parameters in arrays

(b) site-site parameters streams

Figure 3. storing parameters

Each component $comp(i)$ has a certain number of sites and each site has a site type dependend number of site parameters. Although equation (2) indicates, that there's also a parameter ξ for each interaction of two LJ centres, a binary interaction parameter ξ will

Runtime per time step ([s])	Intel Core2 (Woodcrest) 2.66 GHz			Intel Itanium2 (Montecito) 1.4 GHz		
Components	Arrays	Streams	Gain	Arrays	Streams	Gain
Argon	0.7141	0.7007	1.88%	2.4744	2.4060	2.77%
Carbon dioxide (CO_2)	1.5962	1.5354	3.81%	5.3185	5.0916	4.27%
Cyclohexane (C_6H_{12})	9.4565	6.6986	29.16%	27.3789	25.7989	5.77%
Air (N_2+O_2+Ar+CO_2)	1.8305	1.5369	16.04%	5.3099	5.0465	4.96%
R227ea	25.6919	18.0649	29.69%	73.7758	68.7716	6.78%

Table 1. Runtime per step comparison of sequential MD simulations of 100000 molecules ($10 mol/l$). (using Intel compiler V10.0 with -fast option)

only be defined for each combination of two different components and used for all the corresponding LJ interactions.

The traditional way to store interaction parameters as well as site parameters is to use multidimensional arrays. A variant is shown in Fig. 3(a), where a two dimensional parameter array is stored for each component. An alternative approach will be presented in the following. We assume a small number of components, which is reasonable for practical experiments. Hence the memory consumption is still acceptable, if we store parameters for each component-component combination. Instead of just saving the site parameters, new site-site interaction parameters are defined, pre-calculated and stored in parameter streams like shown in Fig. 3(b). Note, that even extra parameter streams for the interaction between component i and j as well as j and i exist, where the only difference are the positions and therefore the order of the parameters. As a consequence, the calculation routine doesn't need to handle the molecules in a certain order and saves a conditional statement.

Regarding equations (3), we might define 9 compound parameters and push them in a queue such as $\boxed{c_{ij,1} \mid c_{ij,2} \mid c_{ij,3} \mid c_{ij,4} \mid c_{ij,5} \mid c_{ij,6} \mid c_{ij,7} \mid c_{ij,8} \mid c_{ij,9}}$ with

$$c_{ij,4k+2l-5} := 4\xi_{ij}\sqrt{\epsilon_k \epsilon_l} \text{ for } (k,l) = (1,1),(1,2),(2,1),(2,2) \tag{4a}$$

$$c_{ij,4k+2l-4} := (0.5 \cdot (\sigma_k + \sigma_l))^2 \text{ for } (k,l) = (1,1),(1,2),(2,1),(2,2) \tag{4b}$$

$$c_{ij,9} := 0.75 \cdot Q_i \cdot Q_j \tag{4c}$$

During the calculation of a molecule-molecule interaction these parameters are read and used in the same order again. Compared to the two times five 2CLJQ site and the interaction parameters, the number of parameters slightly decreased here. Since calculations of equation (4) are only performed once, the overall number of floating point operations will be reduced. Especially the savings of numerous square root calculations is advantageous here. In general however, there might be also a trade of between the number of parameters to be stored resulting in memory usage and the number of floating point operations to be saved, especially if site parameters are used multiple times in various combinations. The choice of optimal streaming parameters might be machine dependent then. An advantage of the stream concept is its flexibility to add new site types. Instead of changing the data structures, the stream will be just extended to include also the new parameter sets.

The implementation uses two functions sketched in listing, which are build up in a symmetric way: the initialization of the streams as a producer and the calculation as a consumer.

Listing 6.1. Initialization of a stream

```
for {k=0;k<2;++k}
 for {l=0;l<2;++l} {
  eps4 = 4.*xi*sqrt(eps[k]*eps[l]);
  strmij << eps4;
  sig2 = 0.5*(sigma[k]+sigma[l]);
  sig2* = sig2;
  strmij << sig2;
#ifndef NDEBUG
  marker = -1
  strmij << marker;

#endif

 }
 strmij << 0.75*Qi*Qj;
//
//
```

Listing 6.2. Calculation using a stream

```
for{k=0;k<2;++k}
 for{l=0;l<2;++l}{

  strmij >> eps4;


  strmij >> sig2;
#ifndef NDEBUG

  strmij >> marker;
  assert(marker!=-1);
#endif
// calculation with eps4 and sig2
 }
 strmij >> q075;
// calculation with q075
 assert(strmij.endofstream());
```

It's important that these functions match. To assure the consistency, markers will be added and the end of the stream will be checked if compiled in debug mode, where *NDEBUG* is not set.

Unlike common MPI programs, where only one source is written for a program, that will act as a sender or matching receiver at one point there are two separate functions and source code here. Due to performance reasons any unnecessary conditional statements have to be avoided and there's a need for an optimized calculation routine not carrying the initialization routine inside.

Runtime measurements of sequential MD simulations using a version of the program based on the array and the streaming data structure described above show the superiority of the latter (see Table 1). Modelled with a single LJ-centre, Argon as a small molecule doesn't benefit much opposed to Cyclohexane carrying six LJ centres and a quadrupole and the larger R227ea with its ten LJ-centres, a quadrupole and a dipole.

4 Parallelisation

The major parallelisation strategies for molecular dynamics are force decomposition and spatial decomposition[2]. We are focussing on distributed memory machines with a large number of processors. For such platforms, spatial decomposition has a better performance than domain decomposition[6]. Our implementation uses MPI for the parallelisation.

Pairs for which the two involved particles are owned by different processors require special treatment. Either one process does the calculations and sends the results to the neighbouring process, or both processes do the calculations. The communication costs do not depend on the number of sites in each molecule, as only force and torque have to be transferred. For the calculation of the force between two molecules with n sites, the costs are $\mathcal{O}(n^2)$. Thus, for big molecules, additional communication is preferred to additional computations. But in our applications, only very small molecules with few sites are simulated. That is why we have decided to calculate boundary pairs twice.

The contribution of each pair has to be added to the macroscopic values. This contribution must only be added once per pair. We have introduced ownership for pairs to guarantee this. That means that not only molecules but also pairs are assigned to processes. Each

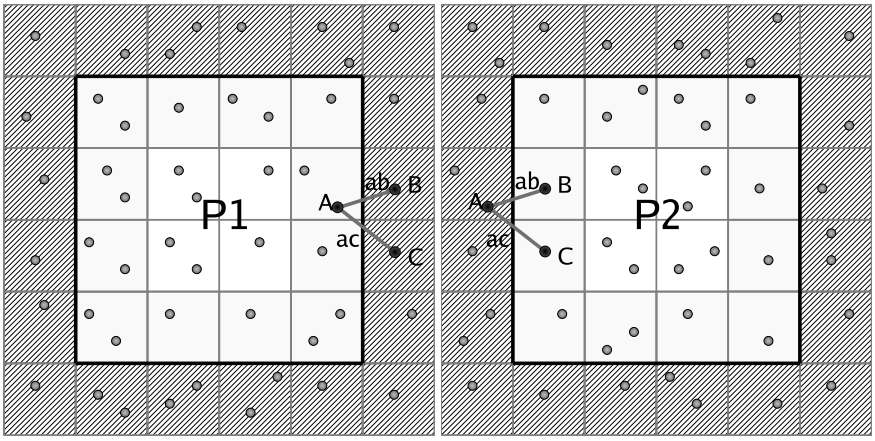


Figure 4. illustration of pairs which cross a process boundary

molecule and each pair belongs to exactly one process. Other molecules and pairs in the boundary region are considered to be copies. While the ownership for molecules is obvious as the processes have distinct spatial domains, the ownership for pairs is more complicated. To decide which process owns a pair, we use a global order of the molecules. A pair belongs to the process who owns the "first" of the two molecules. In our linked cell data structure, we use the index of the cell in which a molecule is stored to define the global order. As molecules which belong to different processes are certainly in different cells, this is sufficient for defining a global ordering. But also any other order criteria (e.g. the id of the particles) would do as well.

Figure 4 shows two neighbouring processes. The grey boundary stripes are the halo regions which contain copies of molecules from neighbouring processes. Particle A (belonging to process $P1$) has the neighbouring particles B and C on process $P2$. The cell of particle C has the lowest index, particle A's cell has a higher index and particle B's cell the highest of the three cells. Hence, the pair ab belongs to process $P1$ and pair ac belongs to process $P2$.

5 Performance and Conclusions

We tested the code on up to 16 nodes of a Linux Cluster. Each node has 8 GB RAM and four Opteron 850 processors with 2.4 GHz. The nodes are connected via an InfiniBand 4x network. First results are shown in Fig. 5. When the problem size is increased from 2.56 million particles to 5.12 million particles, the processing time on a single process increases significantly more than by a factor of two. This is due to insufficient memory on a single machine. Using more processors solves this problem, which explains the superlinear speedup that can be seen in the last row. The runtime comparison with the old version of the code shows only minor differences, hence we have the benefits from a modular and generic program without having to pay for it with reduced efficiency. We plan to conduct more runtime experiments with different configurations to confirm this. Further

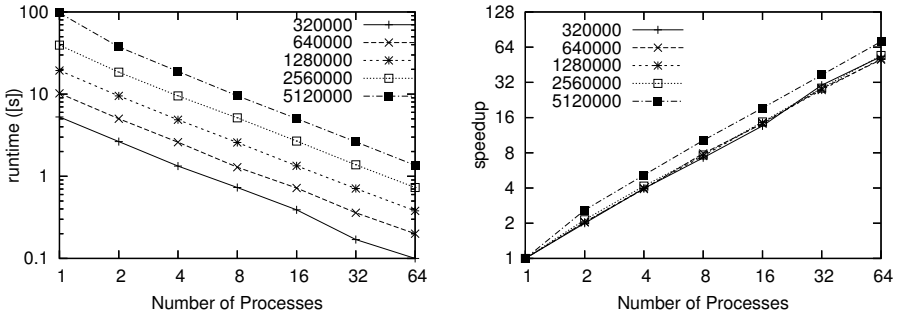


Figure 5. Runtime and speedup for different numbers of particles on up to 64 processors

experiments especially with heterogeneous particle distributions will be done to figure out the importance of load balancing and adaptive data structures for our application.

References

1. D. Fincham, *Parallel computers and molecular simulation*, Molecular Simulation, **1**, 1–45, (1987).
2. S. Plimpton, B. Hendrickson, *Fast Parallel Algorithms for Short-Range Molecular Dynamics*, J. Comp. Phys., **117**, 1–19, (1995).
3. M. P. Allen and D. J. Tildesley, *Computer Simulation of Liquids*, (Clarendon Press, Oxford, 1987)
4. M. Griebel, S. Knapek, G. Zumbusch and A. Caglar, *Numerische Simulation in der Molekül̈dynamik*, (Springer, 2004).
5. J. M. Haile, *Molecular Dynamics Simulation - Elementary Methods*, (Wiley, 1997).
6. M. Bernreuther and H. J. Bungartz, *Molecular Simulation of Fluid Flow on a Cluster of Workstations*, in: 18th Symposium Simulationstechnique ASIM 2005 Proceedings, pp. 117–123, (2005).
7. M. Bernreuther and J. Vrabec, *Molecular simulation of fluids with short range potentials*, in: High Performance Computing on Vector Systems: Proceedings of the Second Teraflop Workshop; Stuttgart, March 17–18, 2005, pp. 187–195, (2005)
8. J. Vrabec, J. Stoll and H. Hasse, *Molecular models of unlike interactions in fluid mixtures*, Molecular Simulation, **31**, 215–221, (2005)
9. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, (Addison-Wesley, 1999).

Massively Parallel Quantum Computer Simulations: Towards Realistic Systems

Marcus Richter, Guido Arnold, Binh Trieu, and Thomas Lippert

Jülich Supercomputing Centre (JSC)
Research Centre Jülich, D-52425 Jülich, Germany
E-mail: {m.richter, g.arnold, b.trieu, th.lippert}@fz-juelich.de

We discuss an extension of the *Massively Parallel Quantum Computer Simulator* by a gate level error model which covers operational errors and decoherence. Applying this error model to the Quantum Fourier Transformation (the kernel of Shor's algorithm) and Grover's quantum search algorithm, one finds that the QFT circuit is more robust to operational inaccuracies than Grover's algorithm on comparable scales. Critical parameters can be derived which give a first estimate of tolerable error thresholds. At present ion traps are regarded as the most promising technology for the realization of quantum computers due to the long coherence time of trapped ions. We discuss Hamiltonian based dynamical ion-trap simulations which have been developed in collaboration with the experimental working group of Prof. Rainer Blatt. In contrast to standard approaches no approximations like the rotating wave approximation or an expansion in the Lamb-Dicke parameter are required which allow for very accurate simulations. This permits to identify critical system parameters which limit the stability of the experiment.

1 Massively Parallel Quantum Computer Simulator

The *Massively parallel quantum computer simulator* is a software package available in Fortran 90 and C which has been developed to simulate universal quantum computers[3]. The software runs on various computer architectures, ranging from PCs to high-end (vector) parallel machines. The simulator can perform all the quantum operations that are necessary for universal quantum computation. The maximum number of qubits is set by the memory of the machine on which the code runs. On the supercomputer systems of the Research Centre Jülich `JUMP` and `JUBL` simulation results for quantum computers containing up to 37 qubits have been obtained. In view of the fact that the simulation of quantum systems, such as quantum computers, requires computational resources that grow exponentially with the system size, this represents a significant advance beyond the state of the art, which is currently around 32 qubits[4].

1.1 Quantum Computation

In a first step towards realistic quantum computer simulations we implement so called *ideal simulations*, where each gate is modeled by a quantum operation that acts instantaneously on the internal state of the quantum computer, neglecting both implementation imperfections and interactions with the environment.
In contrast to a classical bit the state of an elementary storage unit of a quantum computer, the quantum bit or qubit, is described by a two-dimensional vector of Euclidean length one. Denoting two orthogonal basis vectors of the two-dimensional vector space by $|0\rangle$ and $|1\rangle$, the state $|\psi\rangle$ of a **single qubit** can be written as a linear superposition of the basis states

$|0\rangle$ and $|1\rangle$:

$$|\psi\rangle_1 = a_0|0\rangle + a_1|1\rangle \ , \tag{1}$$

where the amplitudes a_0 and a_1 are complex numbers such that $|a_0|^2+|a_1|^2 = 1$. The state of a quantum computer with N qubits can be represented in the 2^N- dimensional Hilbert space as

$$|\psi\rangle_N = a_{0\ldots00}|0\ldots00\rangle + a_{0\ldots01}|0\ldots01\rangle + \ldots + a_{1\ldots10}|1\ldots10\rangle + a_{1\ldots11}|1\ldots11\rangle \tag{2}$$

Each operation on a quantum computer can be described by a 2^Nx 2^N dimensional unitary transformation $U = e^{-iHt}$ acting on the state vector $|\psi'\rangle = U|\psi\rangle$, with the hermitian matrix H being the Hamiltonian of the quantum computer model. An ideal quantum computer can be modeled by simple spin models such as the Ising model associating the two single-spin states $|\uparrow\rangle$ and $|\downarrow\rangle$ with the single-qubit basis states $|0\rangle$ and $|1\rangle$.

A quantum algorithm consists of a sequence of many elementary gates. These elementary gates are represented by very sparse unitary matrices. As the unitary transformation U may change all amplitudes simultaneously, a quantum computer is a massively parallel machine. A small set of elementary **one-qubit gates** (such as the Hadamard gate and the Phase shift gate) and a nontrivial **two-qubit gate** (such as the controlled NOT gate) are sufficient (but not unique) to construct a *universal* quantum computer. In the framework of ideal quantum operations any one- (two-) qubit operation can be decomposed into a sequence of 2x2 (4x4) matrix operations each acting on an orthogonal subspace of the 2^N dimensional Hilbert space.

1.2 Computational Resources and Performance

Due to the exponential growing Hilbert space with cumulative number of qubits, the simulation of quantum computers is clearly memory bounded. To represent a state of a quantum system of N qubits in a conventional, digital computer, we need at least 2^{N+4} bytes if each amplitude is represented by a complex double. Simple storage of the state vector in case of a 37 qubit system requires a memory of 2 TB. An efficient implementation of quantum operations on this state vector even requires 3 TB of memory (a detailed description of the implementation is given in[3]). In the following table we specify the typical simulation requirements depending on the system size N. The last row indicates the overall memory requirements to efficiently simulate quantum operations including the amount of memory to store the state vector.

#qubits N	**32**	**33**	**34**	**35**	**36**	***37***
#cpus (IBM p690+)	32	64	128	256	512	*1024*
#nodes (JUMP)[a]	1	2	4	8	16	*32*
#cpus (Blue Gene Light)	256	512	1024	2048	4096	*8192*
#nodes (JUBL)[b]	256	512	1024	2048	4096	*8192*
memory (state vector)	64 GB	128 GB	256 GB	512 GB	1 TB	*2 TB*
memory (operation)	96 GB	192 GB	384 GB	768 GB	1.5 TB	*3 TB*

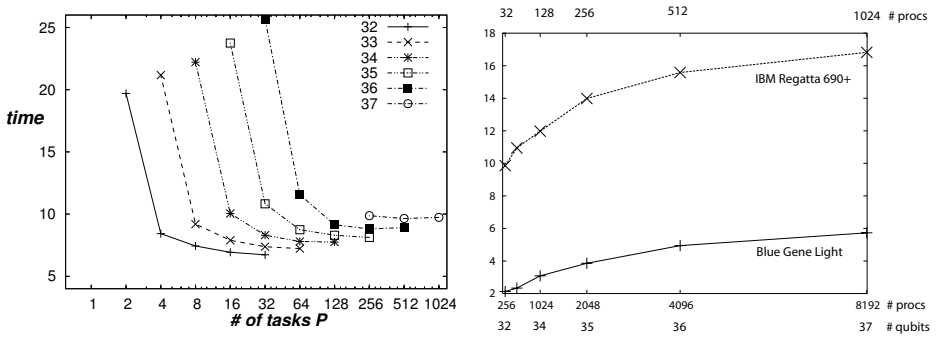


Figure 1. Left: average timings $t_{av}(N)$ on JUMP for a Hadamard operation on different system sizes N depending on the number of MPI tasks $P = 2^p$ using $T = 2^t$ OpenMP threads with $t + p = N - 27$. Right: scaling of the minimal average timings for the system sizes $N = 32, 33, 34, 35, 36, 37$.

From the user's perspective, the memory on a computer can be shared or distributed. On a shared memory computer, the state $|\psi\rangle$ of the quantum computer can be completely stored in the memory and all processors can access the entire memory. On a distributed memory machine, the elements of $|\psi\rangle$ are physically distributed over different nodes and each processor has direct access to its own local memory only. In the latter case, some extra programming is required to perform the communication between the processors. We use the standard Message Passing Interface (MPI) to perform the data communication.

Let us assume that the local memory (or more precise: one SMP-node) can store the 2^M amplitudes of the basis states of M qubits. Hence, to simulate a N-qubit quantum computer we need at least[c] $P = 2^N/2^M$ MPI processes. The amplitudes[d] $a_{x_{N-1}\ldots x_0}$ can be stored at the local memory address $A = \sum_{i=0}^{M-1} 2^i x_i$ of the MPI process with rank $R = \sum_{i=M}^{N-1} 2^{i-M} x_i$. In binary notation the local memory address and the rank of the processor reads $A = (x_{M-1} \ldots x_0)$ and $R = (x_{N-1} \ldots x_M)$, respectively. Recall that the qubits are numbered from 0 to $N-1$, that is qubit 0 corresponds to the least significant bit of the integer index, running from zero to 2^{N-1}, of the amplitude.

On a SMP-cluster like JUMP additionally OpenMP can be employed to parallelize the inter-node access. In this case the number of MPI-tasks obeys the obvious condition #nodes $\leq$ #tasks $\leq$ #CPUs. Since one JUMP-node can store the state vector of $N = 32$, a single CPU corresponds to $N = 27$ (maximal amount of memory per CPU) yielding

$$\underbrace{2^{N-32}}_{\#\text{nodes}} \leq \underbrace{2^{p}}_{\#\text{tasks}} \leq \underbrace{2^{N-27}}_{\#\text{CPUs}} \qquad \text{or} \qquad N - 32 \leq p \leq N - 27\,. \tag{3}$$

[a] The Juelich IBM p690+ (JUMP) is a cluster of 32 compute nodes each containing 32 Power 4+ processors (64bit) and 112 GB memory per node leading to 3.5 TB overall memory available to user access. Two processors share a L2 cache of 1.5 MB and each node shares a 512 MB L3 cache. Users normally only have access to max. 16 nodes equivalent to 512 processors.

[b] The Juelich Blue Gene Light System (JUBL) consists of 16384 PowerPC 440d CPUs driven by a clock rate of 700 MHz. Each node contains 2 processors which share 512 MB of memory. Depending on the application the nodes can be operated in two different modes: a coprocessor mode in which one CPU solely handles all the communication and a virtual node mode which allows to compute different tasks on each CPU.

[c] More processes are involved in case that the communication within one SMP-node is also realized via MPI.

[d] $x_i = 0, 1$ for $i = 0, \ldots, N-1$

The left half of Fig. 1 depicts the average timings for a single qubit quantum operation for different system sizes N against the number of MPI tasks $P = 2^p$. Almost in all cases the best performance is gained by choosing the maximal number of tasks. Only for $N = 36$ and $N = 37$ the performance increases marginally by using two OpenMP threads. The Juelich Blue Gene Light JUBL was operated in the coprocessor mode[e] in which one of the two CPUs per node handles all the communication. Therefore, OpenMP was not needed on this architecture.

The right side of Fig. 1 shows the scaling of the operation with increasing system size. Due to the little amount of memory per node, 8 times the number of CPUs is involved on JUBL compared to JUMP. Both architectures show nearly ideal weak scaling for a large number of qubits.

2 Operational Errors and Decoherence

Although the simulations described so far are gate-level based, it is nevertheless possible to include operational errors and decoherence effects. For this reason we have extended the *Massively parallel quantum computer simulator* by an error model which does not affect the intrinsic performance of the code.

To implement a basic model of operational errors every single qubit gate can be generated from *plane rotations*

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{4}$$

and *phase shifts*

$$P(\phi) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{pmatrix}. \tag{5}$$

This decomposition allows to introduce Gaussian distributed angle- and phase errors ϵ with standard deviation σ, such that $R_\epsilon(\theta) = R(\theta + \epsilon)$ and $P_\epsilon(\phi) = P(\phi + \epsilon)$ respectively. Computation of controlled two- and more qubit gates can be reduced to effective single qubit gate computation acting only on that part of the state vector whose control-bit(s) are set to $|1\rangle$. For each quantum gate operation we draw ϵ_1 and ϵ_2 randomly from independent central Gauss distributions $\rho(\epsilon) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{\epsilon^2}{2\sigma^2})$.

A simple decoherence error model (depolarizing channel) allows for a *bit-flip* $\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, a *phase-flip* $\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ or *both* $-i\sigma_y = \sigma_x\sigma_z = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$ with probability $p/3$ each. The state vector remains unchanged with probability $1 - p$. It is replaced with a completely mixed state with probability p. In other words: The error randomizes the state with probability p. We assume an approximately constant operation time δt for every single gate independent of the type and the qubit it operates on. This duration fixes the time scale for our basic decoherence model. After each serial operation ($t = k\,\delta t, \quad k$=1,2,...) within the quantum circuit *each* of the n qubits (stochastically independent) can be subject to one of the depolarizing operators σ_α. For this we use n independent random sequences

[e] The performance was slightly better than in virtual node mode.

each containing m uniformly distributed numbers, where m is the size of the ensemble (=number of experiment repetitions).

In different experiments we study the effects of gate imperfections and decoherence depending on the standard deviation σ and the probability p. Given a certain confidence level we find out numerically thresholds for these parameters in real applications such as Quantum Fourier transformation or Grover's search algorithm. We compute the error-norm of the final state-vector as the average of m individual measurements $e^2(\sigma, p) = |\psi - \psi_{corr}|^2$.

These results are to be compared with future calculations from dynamic simulations of quantum computer devices, taking into account the full time evolution according to a time dependent Hamiltonian describing both, the system and the environment.

2.1 Quantum Fourier Transformation

The Quantum Fourier Transformation (QFT) is of particular interest since it represents the kernel of Shor's factorization algorithm[f]. To analyze the error robustness of the QFT circuit we plot the error-norm in dependence of (σ, p) for system sizes $n = 8, 16$ with 100000 repetitions and $n = 24$ with 10000 repetitions per experiment each starting from the initial state-vector $|000\ldots0\rangle$.

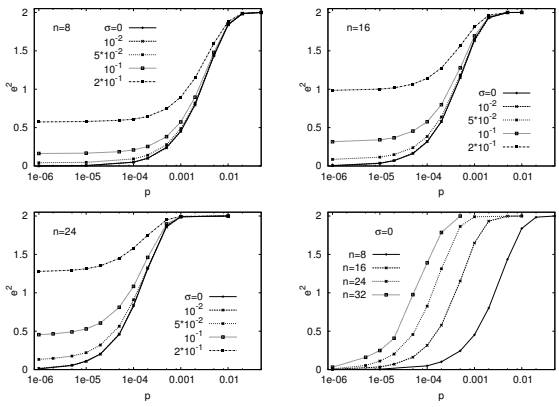


Figure 2. QFT: error-norm in dependence of σ and p for $n = 8, 16, 24$

Fig. 2 suggest a critical behaviour of the system in σ. The system is very robust against operational errors, since we find identical curves of $e^2(\sigma, p)$ for all $\sigma \leq 10^{-2}$ even for the largest system investigated. Larger operational errors increase the error-norm the more the larger the system is. To quantify the dependency of decoherence errors on the system size we have added the simulation results on a 32-qubit system gained from stochastic simulations of length $m = 1000$ each.

[f] Shor's algorithm allows to factorize natural numbers in polynomial time into a product of primes. This means, it could be used to break well-established encryption methods like the RSA scheme

2.2 Grover's Quantum Search Algorithm

Another well known quantum algorithm is Grover's algorithm for searching an unstructured database. In contrast to Shor's algorithm, the quantum search does not shift the problem into a different complexity class. Nevertheless, a quadratic speedup is gained which has been proven to be optimal.

Fig.(3) summarizes our simulation results for the system sizes 8+1, 16+1 and 23+1 qubits and demonstrates the effect of operational inaccuracies and decoherence errors on the amplitude $\psi(k)$ of the database element we are searching for. The (first) undisturbed maximum is expected after $l_{max} = 12, 201, 2274$ Grover iterations respectively. Due to the high cost for simulating larger systems we content ourselves with iterating the Grover circuit up to $l \approx l_{max}$ of the first period in case of n=16,23. The latter requires to simulate more than 35*10^6 quantum operations to collect statistics of m=100 runs. For the smaller systems we collect statistics of m=100000 (8+1) and m=10000 (16+1).

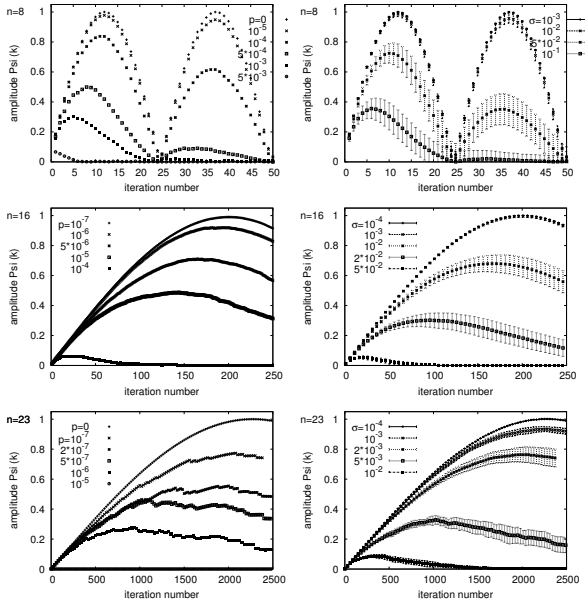


Figure 3. Decoherence (left column) or operational errors (right column) result in a damped amplitude $\psi(k)$ for the searched database entry k. The resultant maximum is shifted towards lower optimal number of Grover iteration steps.

In case of non-vanishing decoherence we can see damping of the value of the amplitude $|\psi_{max}(p, \sigma = 0)| < |\psi_{max,corr}| \approx \sqrt{1 - 1/N}$. The superposed decoherence process leads to a maximum shifted towards $l < l_{max}$ with a shift $\Delta = l_{max} - l$ growing for increasing system sizes n. We see amplitude damping in both cases, for operational and decoherence errors, but with very different sensitivities. In case of operational errors we state a more robust behaviour but switching to an appropriate deviation level we also see a clear shift of the maximal amplitude. To compare Grover's error sensitivity to the

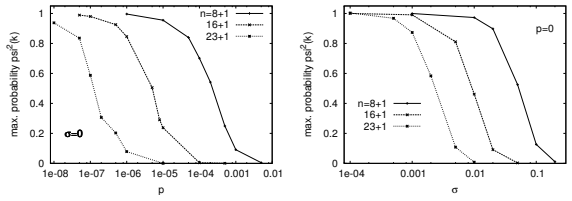


Figure 4. Comparison of the probability to find the correct database entry k for different system sizes in case of decoherence (left) or operational errors (right).

one of QFT we additionally plot the error-norm $e^2(\sigma, p) = |\psi - \psi_{corr}|^2$ in case of the 8+1 and (16+1) qubit system in fig.(5). In the presence of both error sources this investigation reveals a *system size dependent* threshold at $\sigma \approx 10^{-2}$ (and 10^{-3}) respectively. In contrast to the QFT algorithm this threshold is not constant but decreases significantly with the system size. For σ below these thresholds operational errors have nearly no impact. Beyond the threshold(s) we find a dramatically increased error-norm rising quicker with increasing system sizes than in case of QFT.

Asking for the maximal decoherence rate p at a given error tolerance e^2 in the under-critical σ regime the QFT algorithm allows a one (two) order(s) of magnitude higher decoherence probability respectively.

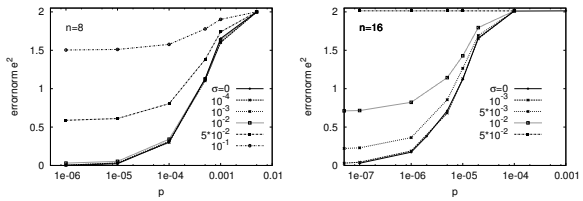


Figure 5. Comparing Grovers error-norm in the presence of both error sources for n=8,16 reveals a *system size dependent* threshold at $\sigma \approx 10^{-2}$ and 10^{-3} respectively.

3 Dynamic Simulation of Ion-Trap Devices

In order to simulate quantum devices from first principles, we have developed an additional package which efficiently solves the time-dependant Schrödinger equation. At present, ion traps are regarded as one of the most promising technologies for constructing a quantum computer, due to the long coherence time of the trapped ions. In a linear trap the ions (e.g. $^{40}\mathrm{Ca}^{+}$) are subject to a quasi harmonic potential $W(t)$ (see Eq.(7)) which keeps the ions fixed in space. The radio-frequency $\omega_{\mathrm{rf}}/2\pi$ induces only a so called micro-motion and is necessary, because in three dimensions it is impossible to trap electrically charged particles within a static potential. As long as the vacuum within the trap is sufficiently good, the ions are perfectly decoupled from the environment. Each ion represents one qubit in which the excitation levels $\mathrm{S}_{1/2}$ and $\mathrm{D}_{5/2}$ correspond to $|0\rangle$ and $|1\rangle$, respectively. The ions can be addressed via laser pulses (Eq. (9)) in order to induce internal transitions (cf.

Eq. (8)), excite bus phonons (which are needed to couple qubits for two-qubit operations), or measure a qubit.

The total time-dependant Hamiltonian consists of the following parts[7]

$$H_{\text{total}}(t) = H_{\text{motion}} + H_{\text{excitation}} + H_{\text{interaction}} \tag{6}$$

where

$$H_{\text{motion}} = \frac{p^2}{2m} + \frac{m}{2} W(t) x^2 \; ; \quad W(t) = \frac{\omega_{\text{rf}}^2}{4} \left[a_x + 2 q_x \cos(\omega_{\text{rf}} t) \right] \tag{7}$$

$$H_{\text{excitation}} = \frac{\hbar}{2} \tilde{\omega} \sigma_z \; ; \quad \tilde{\omega} = \omega_{\text{excited}} - \omega_{\text{ground}} \tag{8}$$

$$H_{\text{interaction}} = \hbar \Omega \sigma_x \cos(kx - \omega t + \phi) \; . \tag{9}$$

H_{motion} describes the collective motion of the ions in the trap potential $W(t)$, $H_{\text{excitation}}$ the internal excitation of the ions due to laser pulses and $H_{\text{interaction}}$ the laser ion interaction; σ_x and σ_z correspond to the Pauli matrices. Using a position space representation, the time evolution is calculated by a Lie-Trotter-Suzuki product formula (the momentum part $p^2/2m$ is solved via FFT).

In contrast to other approaches, no approximations – like the rotating-wave approx. or an expansion in the Lamb-Dicke parameter – are necessary, so that extremely accurate simulations can be carried out. First results demonstrate that effects which lead to a shift of the resonance frequencies – like the AC-Stark effect or off-resonant transitions – are correctly reproduced. This permits to identify critical system parameters which limit the stability of the experiment.

Acknowledgements

This research project was carried out in collaboration with the Computational Physics Group of Hans De Raedt (Univ. of Groningen) and Hartmut Häffner (Group of Rainer Blatt, Univ. of Innsbruck / IQOQI). We thank the DEISA consortium (co-funded by the EU, FP6 project 508830) for support within the DEISA Extreme Computing Initiative (`www.deisa.org`).

References

1. M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge (2000).
2. D. P. Di Vincenzo, `http://arxiv.org/abs/quant-ph/0002077`.
3. K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, Th. Lippert, H. Watanabe and N. Ito, *Massively Parallel Quantum Computer Simulator*, Comp. Phys. Comm. **176**, 121–136 (2007).
4. `http://www.qc.fraunhofer.de`
5. J. Niwa, K. Matsumoto, H. Imai, `http://arxiv.org/abs/quant-ph/0201042`.
6. G. Arnold, M. Richter, B. Trieu and Th. Lippert, *Improving Quantum Computer Simulations*, proceedings of the AQIS06 conference.
7. D. Leibfried, R. Blatt, C. Monroe and D. Wineland, Rev. Mod. Phys. **75**, 281 (2003).
8. J. I. Cirac and P. Zoller, Phys. Rev. Lett. **74**, 4091 (1995).

Image Processing and Visualization

Lessons Learned Using a Camera Cluster to Detect and Locate Objects

Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, and Otto J. Anshus

Dept. of Computer Science
Faculty of Science
N-9037 University of Tromsø, Norway
E-mail: {*daniels, phuong, jmb, otto*}*@cs.uit.no*

A typical commodity camera rarely supports selecting a region of interest to reduce bandwidth, and depending on the extent of image processing, a single CPU may not be sufficient to process data from the camera. Further, such cameras often lack support for synchronized inter-camera image capture, making it difficult to relate images from different cameras. This paper presents a scalable, dedicated parallel camera system for detecting objects in front of a wall-sized, high-resolution, tiled display. The system determines the positions of detected objects, and uses them to interact with applications. Since a single camera can saturate either the bus or CPU, depending on its characteristics and the image processing complexity, the system supports configuring the number of cameras per computer according to bandwidth and processing needs. To minimize image processing latency, the system focuses only on detecting where objects are, rather than what they are, thus reducing the problem's complexity. To overcome the lack of synchronized cameras, short periods of waiting are used. An experimental study using 16 cameras has shown that the system achieves acceptable latency for applications such as 3D games.

1 Introduction

This paper reports on lessons learned using a cluster of cameras to detect the position of objects in front of a wall-sized, high-resolution, tiled display. The system is used to support multi-user touch-free[a] interaction with applications running on a 220-inch 7x4 tiles, 7168x3072 pixels resolution display wall (Fig. 1). This requires that the system can accurately and with low latency determine the positions of fingers, hands, arms and other objects in front of the wall. To achieve this, a consistent and synchronized set of position data from each camera is needed.

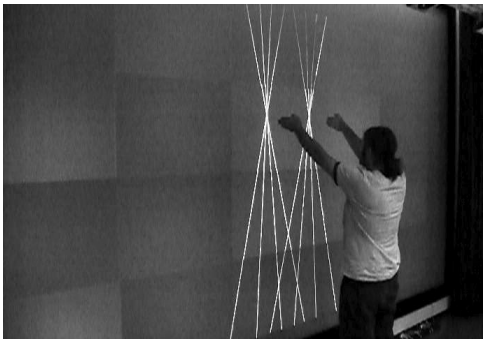


Figure 1. Using the system.

A grayscale camera producing images at a rate of 30 frames per second with a resolution of 640x480 pixels requires a bandwidth of about 8.78 megabytes/second. A FireWire 400 bus can accommodate at most three cameras producing data at this rate; higher-resolution or higher-framerate cameras further decrease this bound. To support more cameras, additional FireWire buses can be used on a single computer. Scalability may now

[a] As the display wall's canvas is not rigid, users must be able to interact with the display wall without actually touching it - thus the term "touch-free."

be limited by the CPU, either due to image processing complexity or deadlines on when results are needed. Finally, most commodity cameras have no support for hardware- or software-based inter-camera synchronization. This limits the accuracy of object positioning, as it reduces the system's ability to relate images captured from different cameras to each other.

This paper presents a parallel system for processing streaming video from several cameras. The system architecture comprises four layers: (i) Camera and image processing, (ii) object-position processing, (iii) event distribution, and (iv) end-application use of position data. The first layer uses 16 cameras connected pairwise to 8 computers. Each computer processes images from two cameras, locating objects and determining their one-dimensional position. When three or more cameras in the first layer see the same object, the second layer can determine the object's 2D position using triangulation. The third layer distributes position data between the other three layers. The fourth layer is comprised of applications using the position data for interaction. An experimental study has shown that the system achieves acceptable latency for common applications like the 3D games Quake 3 Arena and Homeworld (see Section 5 and Ref. 1).

The main contributions of this paper are the lessons learned from building and using the system, including: (i) The flexibility of the system architecture allows configuring available camera and processing resources to accommodate end-applications' needs, (ii) by reducing the complexity of image processing from identifying *what* objects are to identify *where* they are, processing is reduced, and (iii) despite the lack of synchronized cameras, useful results may still be obtained by introducing short periods of waiting.

2 Related Work

There exists much work on multi-camera systems. In Ref. 2, the authors demonstrate how a 100-camera array is used to capture very high-resolution video at 3800x2000 pixels at 30 FPS, or high-speed video with 640x480 pixels at 1560 FPS. Their implementation uses custom circuit boards to communicate with the FireWire cameras and relies on hardware synchronization of cameras, while the system presented in this paper is exclusively based on use of commodity components; cameras without support for synchronization and no use of custom hardware. In Ref. 3, the authors show how displays may be synchronized using an external synchronization source combined with software adjustment of display timings (software genlocking). Their use of a hardware synchronization signal precludes applying their technique to synchronize commodity camera capture.

Other work has used many low-resolution cameras to generate a 3D reconstruction of objects, either for collaborative applications[4] or for creating 3D models. Our system does not attempt to generate high-resolution video or imagery, or reconstruct 3D objects. Instead, the goal is to use a cluster of cameras to determine the 2D position of objects in a plane parallel to the display wall. Previous work reports on different ways of achieving this. In Ref. 5, the author combines internal reflection of infrared light with a camera mounted behind a (rigid) canvas to support multi-touch interaction. Our system differs in that it doesn't require users to actually touch the canvas in order to interact, and in the use of a parallel architecture for capturing and processing images. In Ref. 6, a set of cameras with on-board image processing is mounted in the corners of a large display, and used to detect multiple points of contact. Rather than build custom cameras, our system

uses commodity cameras mounted on the floor in front of the approximately 6 meter wide display wall, and performs all processing on a compute cluster.

3 Design

The system architecture is comprised of four layers, as detailed in the introduction and shown in Fig. 2. The camera and image processing layer captures and processes images from cameras used by the system. To allow for many cameras to be used simultaneously as well as flexibility in image processing complexity, this layer is designed to run in parallel. The layer produces 1D positions and radii for detected objects in each image for each camera. An object's 1D position is defined as the centre of a detected object along the horizontal axis of a captured image (the centner of the finger in Fig. 4), and its radius defined as half the width (in pixels) of the detected object. The object position processing layer combines the position data from each computer in the image processing layer using triangulation, to determine the each object's 2D position.

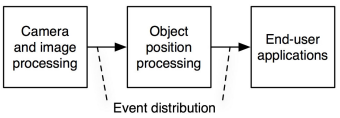


Figure 2. The system architecture.

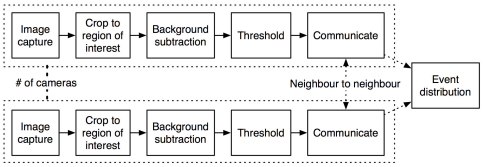


Figure 3. The camera and image processing layer design. The layer can operate in parallel with any number of cameras. Communication happens between each participant and its immediate neighbours.

The camera and image processing layer consists of several steps: (i) Image capture, (ii) cropping, (iii) background subtraction, (iv) thresholding and (v) communication, shown in Fig. 3. When the layer first starts up, it stores the current image from the camera it works with to a buffer. This image will be referred to as the background. As new images are captured, a horizontal region of interest (ROI) is isolated, before the pixel values in the ROI are subtracted from corresponding pixels in the background image. If the absolute difference between a pixel in the current and in the background image is beyond a given threshold, an object is detected at the position of the given pixel in the image. The ROI is determined dynamically when each camera starts capturing images, by identifying the two brightest, horizontal regions in the image[b].

Figure 4 shows an example of how a single image from a single camera is processed. The two horizontal lines (1) indicate two regions of interest in the image. A finger extends from the hand visible in the image, intersecting both ROIs. The background (2) is subtracted from the current image (3), resulting in (4), before the thresholding step is applied, yielding (5). Continuous regions of white indicate where objects have been found in the image.

To account for changes in lighting, the background is updated when too many objects are detected in a single frame from a given camera. An earlier implementation updated the background continuously by merging it with the current image. This did not work well, as users often point at the same location for longer periods of time (on the order of several

[b]The system makes use of a set of "Christmas lights" running along the ceiling, directly above the cameras, in order to create high contrast with intersecting objects.

seconds). The result was “ghost” objects appearing when the user eventually moved his hand.

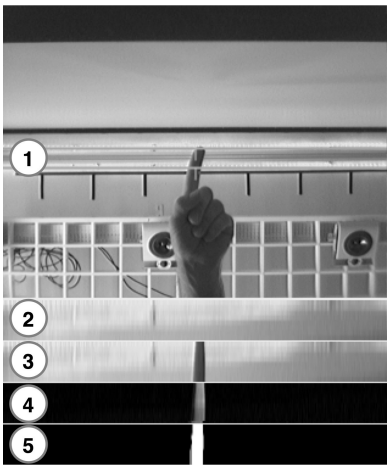


Figure 4. A sample image being processed by the image processing layer. The camera looks directly at the ceiling.

When all objects in the image have been found, the communication step begins. First, each participant sends the number of objects it has detected on the left- and right-hand side of the image to the neighbours on its left and right. The participant receives data from its neighbours, but to avoid introducing additional latency, the participant will use values that are up to 66 ms old[c]. The received values are used to determine if the participant’s results coincide with those of its neighbours. If the number of objects it has detected for the left or right side of the image is identical to the number of objects a neighbour has detected for the same side, no further processing is done. However, if the participant has detected fewer objects than its neighbour, it will re-perform the image processing sequence with a lowered threshold, in an attempt at discovering objects lost due to noise in the captured image. Similarly, if it detects more objects than a neighbour, the image processing sequence is re-performed with a raised threshold. Once this is done, the final 1D positions and radii are sent to the object position processing layer using the event distribution layer.

The object position layer receives 1D positions for located objects from the image processing layer, and uses the positions to triangulate their positions. In order to do this successfully, at least three 1D positions from three different cameras are required, as shown in Fig. 5; any less, and false positives occur when multiple objects are visible. The triangulation is performed by computing intersections between lines projecting from the cameras and up, at an angle determined by the 1D positions. An object’s position in 2D is successfully identified when two or more points of intersection from different cameras lie sufficiently close to each other. The final 2D position is computed as the average of the X and Y components of the 2D intersection points.

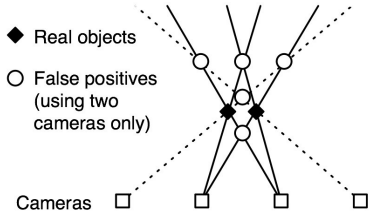


Figure 5. Line segments from each camera and passing through each object are generated. Each line segment is intersected with every other segment. At least three cameras are required to position an object, as using only two cameras results in many false positives.

4 Implementation

The system is comprised of 16 Unibrain Fire-i cameras, connected in pairs to a cluster of 8 Mac minis. In addition, a display cluster of 28 computers, each driving one projector at

[c]This is a tradeoff between system latency, and object detection accuracy. Data that is 66 ms = two frames old may still contain the correct number of (current) objects.

1024x768, is used to provide the graphics capabilities of the display wall, and a MacBook Pro is used to perform object position processing using data from the 8 Mac minis.

The cameras use IEEE-1394 (FireWire) to communicate with the Mac minis, and capture 640x480 grayscale (8-bit) images at 30 frames per second (FPS). They are mounted along the floor and spaced 32 cm apart, as shown in Fig. 6. The cameras do not support external or software-based triggers to synchronize the image capture of multiple cameras, not even when they are on the same FireWire bus. This means that two cameras may capture images spaced in time as far as 33 ms apart (the time between two frames at 30 FPS.).

The Mac minis are interconnected using Gigabit Ethernet. Each Mac mini captures and processes images from the two cameras it is connected to independently of the others, and runs a custom application for performing image capture and processing. This application is written in Objective-C, and uses libdc1394[7] to communicate with the cameras. Each camera is handled by a separate thread within the application, where each thread corresponds to one participant in the camera and image processing layer. Once a frame has been analyzed, the 1D positions and radii of any detected objects are sent to the the object position processing layer.

Figure 6. The image shows 12 of the 16 cameras mounted along the floor and looking at the ceiling.

The object position layer runs a loop operating at the same rate as the cameras, and uses the 1D object positions it receives to triangulate the positions of potential objects. To handle the lack of synchronized cameras, the object positioning software waits for up to 33 ms to receive (possibly empty) sets of object positions from all participating cameras. For the case when a camera has not detected an object, it will notify the object positioning layer of this for the first “no-detect” event only.

To triangulate the positions of objects, the cameras are placed in a coordinate system where cameras are spaced 1 unit apart (1 unit corresponds to 32 cm). For each camera, line segments starting at the camera’s position and passing through the centre of each detected object are generated (Fig. 5). The lines are then intersected with all lines from the two cameras to the current camera’s left and right. The resulting intersection points are compared, and points that are sufficiently close result in an object being identified. The identified objects’ 2D positions and radii are then sent to end-user applications. It is each end-user application’s responsibility to interpret the events to allow user interaction.

5 Evaluation

We have evaluated the system by measuring the latency incurred by the system’s different layers. In particular, we measure the latency for the following components: (i) Camera capture, (ii) image processing, (iii) event distribution, and (iv) object position processing.

To measure camera capture latency, one camera was connected to a computer and pointed at the computer’s display. A custom application fills the computer’s display with black, and then starts capturing images from the camera. At one-second intervals, the display is filled with white, and a timer is started. When the difference between average pixel

values from a 20x20 pixel square in the centre of the image in the previous and the current frame exceeds 150 (because the image goes from being black to being white), the timer is stopped, yielding the camera latency. The experiment was conducted on a Mac mini (1.66 GHz Intel Core Duo, 512 MB RAM) running Mac OS X 10.4.9 and a workstation (Intel Pentium 4 3.0 GHz, 2 GB RAM, HyperThreading enabled) running Ubuntu Linux 6.10 to investigate potential differences in latency caused by the operating system or hardware.

The image processing and object position processing latencies were measured by instrumenting the code that performs the two tasks and measure the execution time of 1000 iterations. The event layer's latency was measured using a ping-pong style benchmark, determining the round-trip time for one event sent back and forth. The resulting round-trip time was divided by 2 to find the one-way latency.

	Cam. capture	**Image proc.**	**Event distr.**	**Object pos.**	**Sum**
Samples	923 (852)	1000	1000	1000	-
Average	81 ms (93 ms)	1.16 ms	1.9 ms	31 ms	115 ms
Std. dev.	10 ms (9 ms)	0.11 ms	0.02 ms	10 ms	-
Minimum	58 ms (72 ms)	0.97 ms	1.6 ms	0.008 ms	62.7 ms
Maximum	104 ms (114 ms)	3.3 ms	3.8 ms	139 ms	250.1 ms

Table 1. Results from the latency experiments. Results for camera capture latency in parentheses are from running the experiment on the Linux workstation.

Table 1 shows the results from the experiments. The majority of total system latency of 115 ms is due to the cameras, with about 10 ms separating the measured latency on Mac OS X and Linux. The next biggest contributor to latency is object position processing (object pos.), which incurs an average latency of 31 ms, which is close to the rate at which the cameras deliver data (every 33 ms). Image processing in the system does not incur much latency. Event distribution (only counted once in the table, but generally incurred twice; once for sending events from the image processing layer to the object position layer, and then once more for sending events from the object position layer to end-user applications) incurs a negligible latency.

6 Discussion

The lack of synchronized cameras is the main limiting factor for accuracy in the system. As two cameras can capture images taken as much as 33 ms apart, the accuracy of the triangulation is significantly affected when the object is moving. The effect is further compounded because three cameras are required to accurately position an object. Filtering can reduce the impact of the uncertainty in position, but at the cost of higher latency. The object position processing layer already introduces up to 33 ms of latency to receive updated position data from all cameras. Although latency could be reduced by not waiting for all cameras, this has the effect of reducing the triangulation accuracy and the rate at which object positions are correctly triangulated drops.

Without synchronized cameras, the question of the system's accuracy can be raised. How fast can an object move while still being accurately positioned? Let p and r be the centre of an object O and its radius, respectively. Since the system uses only the horizontal

axis to position objects, position and movement of an object are implicitly assumed to be horizontal[d].

We observe that a position x of the object detected by a camera can be considered accurate as long as x lies within $[p-r, p+r]$. Therefore, the position of a moving object can be detected accurately if there exists a common position $x*$ that satisfies the accuracy requirement for three images taken by three adjacent cameras during the interval $t = 33ms$[e]. Let $p' > p$ be the new horizontal position of the object's centre due to the object movement during the interval t. The common position $x*$ must satisfy $x* \leq p + r$ and $x* \geq p' - r$. Such a common position exists if $p' - r \leq p + r$ or $p' - p \leq 2r$. That means the system can detect an object's position accurately if the object does not move longer than $2r$ - its diameter - during the interval t.

For instance, assume that the object diameter is 1 cm (e.g. the size of the index-finger). In this case, the object's position can be accurately determined if the object moves at a speed less than $\frac{1cm}{33ms} = 0.3m/s$. Higher framerates can increase this bound, since the maximum delay between two cameras capturing an image will decrease. Doubling the framerate makes the maximum delay go down from 33 ms to 16 ms, and also reduce the object position processing latency. Other limiting factors are the number of cameras detecting the same object, the resolution of the cameras, the speed of the objects, the camera shutter speed, and the accuracy of the image processing layer.

The total system latency of 115 ms is sufficiently low to support playing two games (Quake 3 Arena and Homeworld), as we show in Ref. 1. In that paper, the camera latency was measured to be 102 ms, 21 ms more than reported in this paper. We speculate that the difference is due to a newer OS release in between the first set of results and the results presented in this paper[f]. The results from the Linux workstation show that the operating system or hardware architecture has an impact on the latency from the time at which a camera captures an image, until that image can be processed.

7 Conclusion and lessons learned

We have presented a scalable, dedicated parallel system using a camera cluster to detect and locate objects in front of a display wall. The bottlenecks in such a system can range from the bandwidth required by multiple cameras attached to a single bus and CPU requirements to process images, to deadlines on when results from image processing must be available. Due to our system's parallel architecture, the system can scale both in terms of processing and number of cameras. We currently use two cameras per computer, but with either more cameras, higher-resolution cameras or cameras with higher framerates, the system can be scaled by adding more computers.

Processing images can be CPU-intensive. To avoid image processing incurring too much latency, we reduce the complexity of it by focusing on only detecting that an object is present in an image, rather than determining exactly what the object is. This means that the image processing done by our system can be done quickly, resulting in very low image processing latencies (about 1 ms).

[d] Vertical movement translates to slower shifts in the detected, horizontal position of objects.
[e] The maximum delay between two images taken by two different cameras.
[f] The initial results were gathered on Mac OS X 10.4.8, while the new results are from 10.4.9.

Another challenge in systems using cameras to detect and position objects is relating images from different cameras to each other. High-end cameras can resolve this issue by providing support for either software- or hardware-based synchronization. The commodity cameras used by our system supports neither. Our system resolves this by waiting for data from all cameras currently detecting objects, resulting in up to 33 ms of added latency. This still does not solve the problem of cameras capturing images at different points in time - however, it is better than not detecting objects at all because data from related cameras is processed in alternating rounds.

We have used the system for interacting with different applications on the display wall. This includes controlling the two games Quake 3 Arena and Homeworld[1], and control a custom whiteboard-style application with functionality for creating, resizing and moving simple geometric objects as well as drawing free-hand paths. The system works well for tasks that do not require higher levels of accuracy than our system can deliver, despite the intrinsic lack of synchronization between the cameras.

Acknowledgements

The authors thank Ken-Arne Jensen and Tor-Magne Stien Hagen. This work is supported by the Norwegian Research Council, projects 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and 155550/420 - Display Wall with Compute Cluster.

References

1. D. Stødle, T.-M. S. Hagen, J. M. Bjørndalen and O. J. Anshus, *Gesture-based, touch-free multi-user gaming on wall-sized, high-resolution tiled displays*, in: *Proc. 4th Intl. Symposium on Pervasive Gaming Applications, PerGames 2007*, pp. 75–83, (2007).
2. B. Wilburn, N. Joshi, V. Vaish, E.-V. Talvala, E. Antunez, A. Barth, A. Adams, M. Horowitz and M. Levoy, *High performance imaging using large camera arrays*, in: *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pp. 765–776, (ACM Press, NY, 2005).
3. D. Cotting, M. Waschbüsch, M. Duller and M. Gross, *WinSGL: synchronizing displays in parallel graphics using cost-effective software genlocking*, Parallel Comput., **33**, 420–437, (2007).
4. J. Mulligan, V. Isler and K. Daniilidis, *Trinocular stereo: A real-time algorithm and its evaluation*, Int. J. Comput. Vision, **47**, 51–61, (2002).
5. J. Y. Han, *Low-cost multi-touch sensing through frustrated total internal reflection*, in: *UIST '05: Proc. 18th Annual ACM symposium on User Interface Software and Technology*, pp. 115–118, (ACM Press, NY, 2005).
6. G. D. Morrison, *A camera-based input device for large interactive displays*, IEEE Computer Graphics and Applications, **25**, 52–57, (2005).
7. D. Douxchamps et. al., *libdc1394, an open source library for handling firewire DC cameras*. http://damien.douxchamps.net/ieee1394/libdc1394/.

Hybrid Parallelization for Interactive Exploration in Virtual Environments

Marc Wolter, Marc Schirski, and Torsten Kuhlen

Virtual Reality Group, RWTH Aachen University
E-mail: {wolter, schirski, kuhlen}@rz.rwth-aachen.de

Virtual Reality has shown to be a useful tool in the visualization of complex flow simulation data sets. Maintaining interactivity for the user exploring unstructured, time-varying data demands parallel computing power. We propose a decoupling of interactive and non-interactive tasks to avoid latencies. We introduce an optimized resampling algorithm for unstructured grids which remains scalable using hybrid parallelization. With a resampled region-of-interest, interactive execution of arbitrary visualization techniques is made possible. To validate findings, the user can access different error metrics made during resampling.

1 Introduction

With the growing size of scientific simulation output, efficient algorithms for the presentation of such results to a human user have become an important topic. The large amount of generated data must be explored by the user in order to extract the desired information. To compute visualizations on large, unsteady flow simulations with acceptable response times, parallelization of the corresponding algorithms has shown to be a worthwhile approach. Virtual Reality (VR) is a useful instrument for displaying and interacting with these visualizations. Time-varying 3D structures are perceived in a natural way, which provides a deeper insight into complex flow phenomena. However, the application of VR introduces an interactivity criterion to the system's response time. For direct interaction, Bryson[1] demands a response time of 100 ms to be considered interactive. Even with modern high performance computers, this threshold is not easily met for data sets of reasonable size. For image generation, dedicated visualization systems are used, which are normally locally and logically decoupled from high performance computing (HPC) systems. This introduces another source for latency, as all data generated on the HPC system has to be transmitted to the visualization system in order to be used for image synthesis.

To deal with this problem, we proposed a task distribution optimized for the user's interaction behaviour[2,3]. Tasks with frequent parameter changes are computed locally, while tasks for which parameters change less frequently or predictably are computed using hybrid parallelization on a remote HPC machine.

To render local visualization possible, we restrict it to a region-of-interest (ROI) reduced in size and complexity (as provided by a resampled Cartesian grid). Inside the ROI, the user can interactively explore the data using different visualization techniques. The resampling of a ROI for a time-varying dataset should be fast, but the interactivity criterion is relaxed somewhat, because this event is of significantly less frequent occurrence. Therefore, instead of parallelizing a specific visualization technique, we provide the user with arbitrary visualization tools for his data exploration by a general task distribution.

By distributing computational tasks onto processors and/or cores according to the data basis they operate on, our approach is especially targeted at the rising availability of multi-

core clusters. Distributed memory parallelization is used for tasks which do not share common data (i.e., data of different time steps), while shared memory parallelization is employed for tasks working on the same data (i.e., data of a single time step).

2 Related Work

Two main strategies for remote visualization are found in the literature: image-based and geometry-based approaches. Image-based remote visualization uses parallel computing to render the final images. This method is especially useful when handling very large data sets, as image space depends on viewport resolution only. Ma and Parker[4] describe software-based parallel rendering techniques for two visualization techniques, volume rendering and isosurface ray-tracing, for large unstructured meshes.

Strengert et al.[5] use hierarchical wavelet compression to keep the memory footprint low. They also integrate the user's viewing direction on a 2D viewport into the quality of the images rendered in parallel, providing some sort of implicit region of interest. Chen et al.[6] apply hybrid parallelization (MPI+OpenMP) on the Earthsimulator for volume rendering large data sets with additional vectorization of subtasks. They use dynamic load balancing between MPI nodes, as they distribute subvolumes among nodes.

However, the latency and restriction in viewport size generated by remote parallel rendering are not acceptable in an immersive virtual environment. A constantly high framerate (i.e., higher than 30 frames per second) in combination with head-tracked, user-centred stereoscopic projection is needed to maintain immersion. Due to stereoscopy and room-mounted displays, the amount of pixels to be covered exceeds normal desktop viewports by far.

In contrast to remote rendering, geometry-based remote visualization produces in parallel the requested results in form of geometry. Only the geometry is transmitted and is rendered locally at the visualization system. One of the first available systems for VR-based flow visualization was the Virtual Wind Tunnel and its follow-up Distributed Virtual Wind Tunnel[1]. The latter introduced a connection to a vectorized post-processing backend, which was then responsible for post-processing computations.

The Visualization ToolKit (VTK) provides different levels of parallelism[7]. Task, pipeline and data parallel execution of visualization pipelines are possible. They achieve scalable results while maintaining low complexity for the user. In addition, the VTK pipeline supports piecewise computation of results, which keeps the memory footprint low and enables the visualization even of large data sets.

3 Data Structures and Algorithmic Approach

As is typical for results of CFD computations, the problem domain is discretized in time and space, i.e. data is given on a number of unstructured grids. To allow fast interaction for the user, in our approach, data is to sent to the visualization system in the form of Cartesian grids of user-definable size and resolution. Thus, the unstructured source grids (or subportions thereof) have to be resampled into Cartesian grids. In previous work, we applied the algorithm provided by VTK for resampling[3]. This was the major bottleneck (i.e., $>$ 97 $\%$ was spent in the corresponding operations) even for rectilinear grids. As runtimes

increased significantly for unstructured grids, we now propose an optimized resampling algorithm for unstructured grids, which is easily parallelizable using OpenMP.

In a pre-processing step, the unstructured grids are converted into tetrahedral grids, which allow for a more efficient handling and cell search. Then, for every time step and for every target grid point, resampling is performed by locating the source grid cell which contains the target point, interpolating the data values of the points forming this cell, and storing this information at the corresponding point within the target grid. The most time-consuming part of this process is the cell search. For this, we employ a two-phase cell search strategy using a *k*d-tree and tetrahedral walks, providing significantly improved performance over the standard approach as implemented in VTK.

The basic cell search approach has originally been used for fast location of cells which contain seed points for interactively computed particle traces[8]. It comprises a broad phase using a *k*d-tree for the fast location of a grid point p_q located closely to the sought after target point q, followed by a tetrahedral walk from an incident cell to p_q to the final cell (see Fig. 1, left). However, as the tetrahedral walk can potentially fail if a boundary cell is encountered, we use an extended approach here, which considerably improves its reliability. The extension consists of restarting the tetrahedral walk from additional points p_{qi} and respective incident cells if it is not successful (see Fig. 1, right). These additional starting points are found by traversing the *k*d-tree upwards and using the encountered tree nodes as candidates. The query point q is considered as lying outside the source grid if and only if none of the tetrahedral walks is successful.

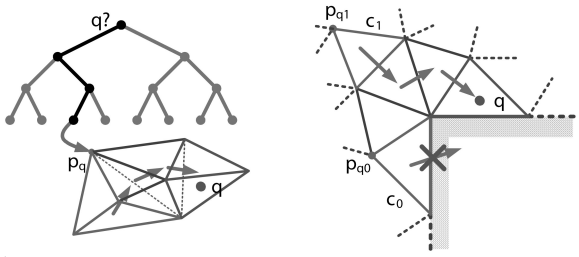


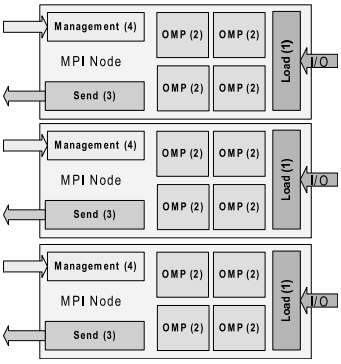
Figure 1. Point location is performed in a two-phase process consisting of a *k*d-tree search followed by a tetrahedral walk (left). Potential failures of the tetrahedral walk due to boundary cells are ameliorated by additional walks starting at supplementary candidate cells (right).

For an evaluation of the quality of the resampling process, a number of error metrics is employed. The corresponding information is then relayed to the user in order to let him assess the accuracy of the resampled grid, which directly influences the reliability of all further exploration efforts. For a global error estimation, the root mean square error is determined for all source grid points which lie within the boundaries of the target grid, followed by a normalization of these figures by a division through the data range of the involved points, thus leading to a relative root mean square error. In addition, the maximum relative error within this point set provides information about the upper bound of the interpolation error.

In order to determine the spatial distribution of local error, a relative root mean square

error is calculated for every target grid point as well, albeit with a restriction to localized vertex sets. It relies on those source grid points, for which the respective target grid point is the nearest neighbour in the whole target grid, and which lie within the boundaries of the target grid. This results in a spatial error distribution, which can be displayed to the user (e.g., via direct volume rendering – see Fig. 3, right) to allow for assessing the quality of the resampled grid. To confirm his findings, the user may request the ROI in its original structure, i.e. a subportion of the original data set, but hence potentially loses interactive exploration capabilities.

4 Parallel Data Extraction Phase




```
#pragma omp parallel shared(source, target) \
        shared(localmin[], localmax[])
{
  for (dimension k)
    for (dimension j)
      #pragma omp for nowait
      for (dimension i)
        p  = point(i,j,k)
        c  = locate_cell(source, p)
        if (c not found)
          c  = try_second_walk(source, p)
        if (c found)
          interpolate_point(c, p)
          update(localmin[], localmax[])
      endfor
    endfor
  endfor
}
update_globalvalues(localmin[], localmax[])
```

Figure 2. Left: Hybrid parallelization scheme. On each MPI node, at least four tasks are concurrently active: loading from filesystem (1), parallel computation using OpenMP (2), sending results to the visualization system (3) and managing control information (4). Right: Pseudocode for the resampling algorithm parallelized with OpenMP.

The parallel computation of the ROI is done using the Viracocha framework[9]. Single time steps of the data set are distributed using the scheduling scheme described by Wolter et al.[3]. Using this approach, the computational order of successive time steps is adapted to the user's exploration behaviour. Computed subtasks arrive just in time to be available for interactive computation. This is made possible by the constant animation duration of each time step and the predictability of successive time steps. However, a predictable and stable computation time per time step is crucial for a gap-free provision of a time-varying ROI.

Due to cell search and attribute interpolation being completely independent from each other for all sample points, a near-optimal speed-up can be achieved by distributing the corresponding computational load onto several processors or cores (see Section 6). However, since all computations for a single time step are performed on the same data basis, a shared memory system is the preferred computational environment, which makes OpenMP an ideal choice for the implementation (see Fig. 2, right). As an added benefit, our cell search approach results in a comparable amount of operations to be executed for every sample

point. This allows for very precise estimates of the time required for computation, which in turn helps prioritizing extraction operations for optimal user feedback.

A data manager component manages the amount of cached data on each node as well as prefetching required time steps.

For a fast ROI extraction, we apply a hybrid parallelization approach (see Fig. 2, left). Independent time steps are computed using MPI, which can be distributed across several nodes, as nearly no communication between nodes is necessary. A central scheduler process handles user requests and dynamic time step distribution.

To speed up the extraction for a single time step, pipelining and shared memory parallelization using OpenMP are applied on each MPI node (see Fig. 2, left). The three pipelining stages are (1) prefetching of the next time step data, (2) parallel extraction of the region of interest using OpenMP and (3) transmission (including serialization) of the simplified region. As time step data is predictable due to the continuous animation, simple OBL (one-block look-ahead) prefetching shows good hit rates for the prefetching thread (1). Preloading is overlapped with the current computation (2). The resulting Cartesian grid including error metrics must be serialized and sent over the network to the local visualization system (3). An additional thread (4) manages control messages (e.g., progress reports, update or cancellation of a task etc.). As all these threads need to communicate with other MPI nodes, a thread-safe MPI-2 implementation is required. Special care has to be taken not to waste CPU resources. Threads are suspended when their specific task is not required at the moment, but resume when a new task is available, which is realized with thread events. In addition, the MPI environment must be configured to prevent spinning of idle threads, which occupies CPU time otherwise available for the OpenMP computation. We currently use a spin time of one second, i.e. idle MPI calls check every second for new messages, which is enough for control messages.

The extracted region is transmitted to the local visualization system for further processing in the next phase.

5 Interactive Computation

Once the ROI is transmitted to the visualization host, the user can directly interact with the flow data and explore it using locally computed visualization algorithms. As an example, the user can specify particle seed points, seed isosurfaces, or position cut planes via an input device with six degrees-of-freedom inside the virtual environment. This information is directly used for a parametrization of the currently active visualization method, followed by an immediate computation and display of the result. The reduced grid data size (compared to the full-blown data set as stored on the HPC system) allows for visual feedback at interactive rates (i.e., < 100 ms)[3]. Any time the user wants to change the focus of the exploration, the ROI can be moved and/or resized, which results in a parallelized resampling of the data on the HPC system, again. If resampling times are low enough, the ROIs are computed just in time before they are displayed and move implicitly[3]. For larger resampling runtimes the user moves the ROI explicitly, which introduces a brief waiting time of a few seconds before he can continue with the exploration[2].

	16^3	32^3	64^3	80^3
# points	4096	32768	262144	512000
Vector error avg	6.69 %	3.47 %	1.48 %	1.08 %
Scalar error avg	3.62 %	1.69 %	0.68 %	0.51 %
Transmission time avg	0.097 s	0.107 s	0.733 s	1.432 s

Table 1. Global error values and ROI transmission times for three different resolutions of Cartesian ROIs.

	1 thread	2 threads	4 threads
1 node	10.335 s	9.174 s	8.75 s
2 nodes	8.437 s	7.887 s	7.69 s
4 nodes	7.663 s	7.326 s	7.287 s

Table 2. Total runtimes for resampling four time steps on the X2200 cluster, measured on the visualization system. The poor speedup is caused by transmission being the new bottleneck.

6 Results

We applied the system to explore a time-varying data set which consists of 100 time steps with approximately 3.3 million points per time step in an unstructured grid. The data show the turbulent mixing of a cooling jet in a hot boundary layer[10]. Measurements have been taken using a SunFire E2900 (12 UltraSparc IV dual core processors at 1.2 GHz, 48 GB of main memory), one SunFire V40z (four Opteron 875 dual-core processors at 2.2 GHz, 16 GB of main memory) and a cluster consisting of 16 SunFire X2200 nodes (two AMD Opteron 2218 dual core processors at 2.6 GHz, 16 GB of main memory, Gigabit Ethernet interconnect). The visualization system was connected to the remote HPC systems via a non-dedicated 100 Mbit/s network.

Table 1 shows the relative mean square error of a ROI with several resampling resolutions. A very turbulent region was chosen as region of interest in order to give an idea about worst case errors. The average resampling error is quite low and drops significantly with growing resampling resolution, as expected. Figure 3 (right) shows an example of a direct volume visualization of the local error. The lower part has a much higher local resampling error, as this part has a higher cell resolution in the unstructured grid. While the global error gives a general idea about the resampling error, the researcher should check for the local error to validate results. On the downside, the transmission times for the Cartesian grid grows linearly with the number of resampled points. The computed Cartesian grids contained one vector and one scalar field, both with double precision. The high transmission times are explained with the non-dedicated, slow network and the usage of reliable TCP without any compression.

Figure 3 (left) shows the speedup of the resampling algorithm using OpenMP. The parallelization scales well, as single point searches are independent from another and all data resides in shared memory. As an additional benefit, the root mean square error of the resampling duration decreases, which makes it easier to estimate runtimes. For the speedup measurement, the SunFire V40z with eight cores was used, which explains the performance drop at eight OpenMP threads, as one CPU core was reserved for the scheduling

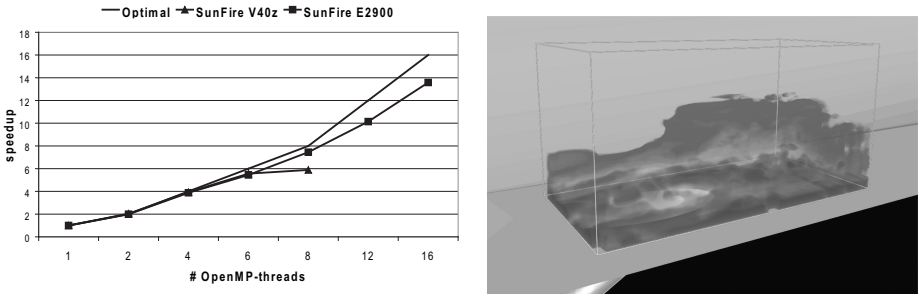


Figure 3. Left: Speedup for OpenMP parallelization of the resampling algorithm. Right: Displaying the spatial error distribution via direct volume rendering allows the user to assess the quality of the resampled grid ($64\times32\times32$). Local error values between 0% (transparent) - 10% (dark) are depicted.

component (cf. section 4).

Table 2 depicts the total runtimes for hybrid parallelization of four time steps. A region was resampled on a 80^3 resolution using up to four MPI nodes and up to four OpenMP threads. The runtimes were measured on the visualization system, and therefore depict total user waiting times. While a performance gain is achieved due to the speedup of the resampling algorithm itself, the total runtime savings do not scale just as well. Further analysis has shown that with the highly reduced resampling time two new bottlenecks arise: file I/O and data transmission to the visualization system. In the shown measurements we used already prefetched data in the data manager's cache, as the difference in loading and computing a time step was too long for efficient prefetching. For more efficient file I/O, external memory algorithms will be implemented as future work. MPI's parallel I/O is not useful in our case, as single files are processed by single MPI nodes. For the transmission times, which exceed computation time for all resolutions, compression in combination with a faster and dedicated network could improve the results. We have already measured high throughput (660 Mbit/s on a dedicated gigabit Ethernet) using such a network in previous work[11].

7 Conclusion

Hybrid parallelization minimizes user waiting times for newly extracted ROIs within large data sets considerably, thus speeding up the exploration process. By constraining the response requirements to the user's actions inside the ROI only, our approach allows for an interactive exploration even of large data sets in virtual environments. A drawback of this method in general is the user's restriction to a spatially bound region of interest. Although this corresponds to the common exploration pattern, the user does not get any information outside the specified region. The shift from directly parallelizing the visualization method to considering the region extraction as the target of parallelization leads to an efficient and scalable parallel algorithm. However, for large data sets and highly resampled regions, file system and network transmission emerge as new bottlenecks.

Acknowledgements

The authors would like to thank the Department for Mathematics CCES for providing access to the Opteron cluster.

References

1. S. Bryson and M. J. Gerald-Yamasaki, *The Distributed Virtual Windtunnel*, in: Proc. IEEE Supercomputing '92, pp. 275–284, (1992).
2. M. Schirski, C. Bischof and T. Kuhlen, *Interactive exploration of large data in hybrid visualization environments*, in: Proc. 13th Eurographics Symposium on Virtual Environments and 10th Immersive Projection Technology Workshop, pp. 69–76, (2007).
3. M. Wolter, C. Bischof, and T. Kuhlen, *Dynamic regions of interest for interactive flow exploration*, in: Proc. Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '07), pp. 53–60, (2007).
4. K.-L. Ma and S. Parker, *Massively parallel software rendering for visualizing large-scale data sets*, IEEE Computer Graphics and Applications, **21**, 72–83, (2001).
5. M. Strengert, M. Magallon, D. Weiskopf, S. Guthe and T. Ertl, *Hierarchical visualization and compression of large volume datasets using GPU clusters*, in: Proc. Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '04), pp. 41–48, (2004).
6. L. Chen, I. Fujishiro and K. Nakajima, *Optimizing parallel performance of unstructured volume rendering for the earth simulator*, Parallel Computing, **29**, 355–371, (2003).
7. J. Ahrens, C. Law, W. Schroeder, K. Martin and M. Papka, *A parallel approach for efficiently visualizing extremely large, time-varying datasets*, Tech. Rep. LAUR-001630, Los Alamos National Laboratory, (2000).
8. M. Schirski, C. Bischof and T. Kuhlen, *Interactive particle tracing on tetrahedral grids using the GPU*, in: Proc. Vision, Modeling, and Visualization (VMV) 2006, pp. 153–160, (2006).
9. A. Gerndt, B. Hentschel, M. Wolter, T. Kuhlen and C. Bischof, *Viracocha: An efficient parallelization framework for large-scale CFD post-processing in virtual environments*, in: Proc. IEEE Supercomputing '04, (2004).
10. P. Renze, W. Schroeder and M. Meinke, *LES of film cooling efficiency for different hole shapes*, in: 5th International Symposium on Turbulence and Shear Flow Phenomena, (2007).
11. T. Duessel, H. Zilken, W. Frings, T. Eickermann, A. Gerndt, M. Wolter, and T. Kuhlen, *Distributed Collaborative Data Analysis with Heterogeneous Visualisation Systems*, in: Proc. Eurographics Symposium on Parallel Graphics and Visualization (EGPGV '07), pp. 21–28, (2007).

Performance Modeling and Tools

Analysis of the Weather Research and Forecasting (WRF) Model on Large-Scale Systems

Darren J. Kerbyson, Kevin J. Barker, and Kei Davis

Performance and Architecture Lab
Los Alamos National Laboratory
Los Alamos, NM USA
E-mail: {djk, kjbarker, kei.davis}@lanl.gov

In this work we analyze the performance of the Weather Research and Forecasting (WRF) model using both empirical data and an accurate analytic performance model. WRF is a large-scale mesoscale numerical weather prediction system designed for both operational forecasting and atmospheric research. It is in active development at the National Center for Atmospheric Research (NCAR), and can use 1,000 s of processors in parallel. In this work we compare the performance of WRF on a cluster-based system (AMD Opteron processors interconnected with 4x SDR Infiniband) to that on a mesh-based system (IBM Blue Gene/L interconnected with a proprietary 3-D torus). In addition, we develop a performance model of WRF that is validated against these two systems and that exhibits high prediction accuracy. The model is then used to examine the performance of a near-term future generation supercomputer.

1 Introduction

The Weather Research and Forecasting (WRF) model is a community mesoscale numerical weather prediction system with nearly 5,000 users, developed by a consortium of government agencies together with the research community. It is used for both operational forecasting research and atmospheric research, particularly at the 1-10 km scale, and is capable of modelling events such as storm systems and hurricanes[10]. It is also being used for regional climate modelling, chemistry and air-quality research and prediction, large eddy simulations, cloud and storm modelling, and data assimilation. Features of WRF include dynamical cores based on finite difference methods and many options for physical parameterizations (microphysics, cumulus parameterization, planetary boundary layers, turbulence, radiation, and surface models) that are being developed by various groups. It includes two-way moving nests and can be coupled with other models including hydrology, land-surface, and ocean models.

WRF has been ported to various platforms and can utilize thousands of processors in parallel. Future computational requirements are expected to increase as a consequence of both increased resolution and the use of increasingly sophisticated physics models. Our performance model of WRF allows accurate prediction of the performance of WRF on near-future large-scale systems that may contain many hundreds of thousands of processors.

In this work we analyze the performance of WRF (version 2.2) on two very different large-scale systems: a cluster of 256 Opteron nodes (1,024 processing cores) interconnected using Infiniband, and a small Blue Gene/L system containing 1,024 nodes (2,048 processing cores) interconnected by a proprietary 3-D torus[1]. This comparison allows us to draw conclusions concerning the system sizes required to achieve an equivalent level of performance on WRF.

An important aspect of this work is the capture of key performance characteristics into an analytical performance model. This model is parameterized in terms of the main application inputs (iteration count, number of grid points in each dimension, the computation load per grid point, etc.) as well as system parameters (processor count, communication topology, latencies and bandwidths, etc.). The model also takes as input the time per cell when using all processing cores in a single node—this can be measured on an available system, or determined for a future system using a processor simulator. The utility of the model is its capability of predicting for larger-scale systems that are not available for measurement, and for predicting for future or hypothetical systems.

In Section 2 we provide an overview of WRF, two commonly used input decks, and the measured performance on the two systems. In Section 3 we detail the performance model and show that it has high prediction accuracy when compared to the measured data. In Section 4 we compare the performance of the two systems and quantify the size of the systems that are required to achieve equivalent performance. We then extend this work to a future large-scale system architecture using the analytic performance model we develop.

2 Overview of WRF

WRF uses a three-dimensional grid to represent the atmosphere at scales ranging from meters to thousands of kilometers, topographical land information, and observational data to define initial conditions for a forecasting simulation. It features multiple dynamical cores of which one is chosen for a particular simulation. In this work we have analyzed two simulations that are defined by separate input decks, *standard.input* and *large.input*, as used in the Department of Defense technology insertion benchmark suite (TI-06). Both input decks are commonly used in the performance assessment of WRF and are representative of real-world forecasting scenarios. The main characteristics of these input decks are listed in Table 1.

Both inputs define a weather forecast for the continental United States but at different resolutions, resulting in differences in the global problem size (number of cells) processed. As a consequence a smaller time step is necessary for *large.input*. Different physics are also used in the two forecasts, the details of which are not of concern to this work but nevertheless impact the processing requirements (and so processing time per cell).

The global grid is partitioned in the two horizontal dimensions across a logical 2-D processor array. Each processor is assigned a subgrid of approximately equal size. Strong scaling is used to achieve faster time to solution, thus the subgrid on each processor becomes increasingly smaller with increased processor count, and the proportion of time spent in parallel activities increases. The main parallel activities in WRF consist of boundary exchanges that occur in all four logical directions—39 such exchanges occur in each direction in each iteration when using *large.input*, and 35 when using *standard.input*. All message sizes are a function of the two horizontal dimensions of the subgrid as well as the subgrid depth and can range from 10's to 100's of KB at a 512 processor scale.

Each iteration on *large.input* advances the simulation time by 30 seconds. A total of 5720 iterations performs a 48-hour weather forecast. In addition, every 10 minutes in the simulation (every 20th iteration) a radiation physics step occurs, and every hour in the simulation (every 120th iteration) a history file is generated. These latter iterations involve large I/O operations and are excluded from our analysis. An example of the measured

		standard.input	*large.input*
Simulation	Resolution	12 km	5 km
	Duration	2 days	2 days
	Iteration time-step	72 s	30 s
	Total no. iterations	2,400	5,760
Grid dimensions	East-west	425	1010
	North-south	300	720
	Vertical	35	38
	Total grid points	4.5 M	27.6 M
Physics	Microphysics	WSM 3-class simple	WSM 5-class
	Land surface	Thermal diffusion	Noah model
	Radiation physics	10 min	10 min
	Lateral boundary updates	6 hr	3 hr
History file gen.		1 hr	1 hr

Table 1. Characteristics of the two commonly used WRF input decks

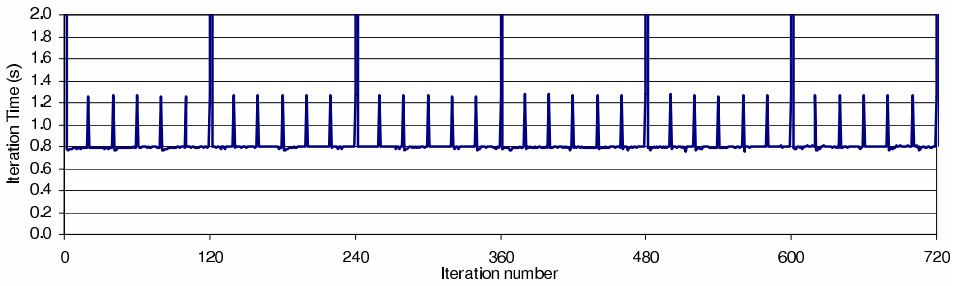


Figure 1. Variation in the iteration time on a 512 processor run for *large.input*

iteration time, for the first 720 iterations, taken from a 512 processor run using *large.input* on the Opteron cluster is shown in Fig. 1. Large peaks on the iteration time can clearly be seen every 120 iterations because of the history file generation. Note that the range on the vertical axis in Fig. 1 is 0 to 2 s while the history file generation (off scale) takes 160 s. The smaller peaks—every 20th iteration—result from the extra processing by the radiation physics. It should also be noted that the time for a typical iteration is nearly constant.

3 Performance Model Overview

An overview of the performance model we have developed is described below, followed by a validation of the model's predictive capability.

3.1 Model Description

The runtime of WRF is modeled as

$$T_{RunTime} = N_{Iter} \cdot T_{CompIter} + N_{IterRP} \cdot T_{CompIterRP} + (N_{Iter} + N_{IterRP}) \cdot T_{CommIterRP} \quad (1)$$

where N_{Iter} is the number of normal iterations, N_{IterRP} is the number of iterations with radiation physics, $T_{CompIter}$ and $T_{CompIterRP}$ are the modeled computation time per iteration for the two types of iterations, respectively, and $T_{CommIter}$ is the communication time per iteration.

The computation times are a function of the number of grid points assigned to each processor, and the measured time per cell on a processing node of the system:

$$\begin{aligned} T_{CompIter} &= \left\lceil \frac{N_x}{P_x} \right\rceil \cdot \left\lceil \frac{N_y}{P_y} \right\rceil \cdot N_z \cdot T_{CompPerCell} \\ T_{CompIterRP} &= \left\lceil \frac{N_x}{P_x} \right\rceil \cdot \left\lceil \frac{N_y}{P_y} \right\rceil \cdot N_z \cdot T_{CompRPPerCell} \end{aligned} \quad (2)$$

Note that the computation time per cell for both types of iteration may also be a function of the number of cells in a subgrid.

The communication time consists of two components: one for the east-west (horizontal x dimension), and one for the north-south (horizontal y dimension) boundary exchanges. Each exchange is done by two calls to *MPI_Isend* and two calls to *MPI_Irecv* followed by an *MPI_Waitall*.

From an analysis of an execution of WRF the number of boundary exchanges, $NumBX$, was found to be 35 and 39 for *standard.input* and *large.input*, respectively.

The time to perform a single boundary exchange is modeled as

$$T_{CommIter} = \sum_{i=1}^{NumBX} \left(T_{comm}(Size_{xi}, C_x) + T_{comm}(Size_{yi}, C_y) \right) \quad (3)$$

where the size of the messages, $Size_{xi}$ and $Size_{yi}$, vary over the $NumBX$ boundary exchanges.

A piece-wise linear model for the communication time is assumed which uses the latency, L_c, and bandwidth, B_c, of the communication network in the system. The effective communication latency and bandwidth vary depending on the size of a message and also the number of processors used (for example, in the cases of intra-node and inter-node communications for an SMP-based machine).

$$T_{comm}(S, C) = L_c(S) + C \cdot S \cdot \frac{1}{B_c(S)} \quad (4)$$

The communication model uses the bandwidths and latencies of the communication network observed in a single direction when performing bi-directional communications, as is the case in WRF for the boundary exchanges. They are obtained from a ping-pong type communication micro-benchmark that is independent of the application and in which the round-trip time required for bi-directional communications is measured as a function of the message size. This should not be confused with the peak uni-directional performance of the network or peak measured bandwidths from a performance evaluation exercise.

The contention that occurs during inter-node communication is dependent on the communication direction (x or y dimension), the arrangement of subgrids across processing

		Opteron cluster	*Blue Gene/L*
System	Peak	4.1 Tflops	5.7 Tflops
	Node Count	256	1,024
	Core Count	1,024	2,048
	Core Speed	2.0 GHz	0.7 GHz
Nodes	Peak	16 Gflops	5.6 Gflops
	Cores	4	2
	Memory	8 GB	512 MB
Network	Topology	12-ary fat tree	3D torus
	MPI (zero-byte) latency	4.0 μs	2.8 μs
	MPI (1 MB) bandwidth	950 MB/s	154 MB/s
	$n_{1/2}$	$\sim$15,000 B	$\sim$1,400 B

Table 2. Characteristics of the two systems used in the validation of the WRF model.

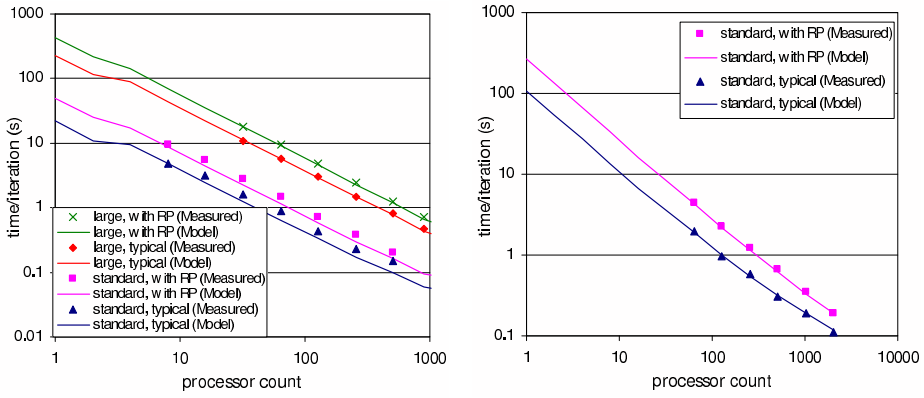
nodes, the network topology and routing mechanism, and the number of processing cores per node. The contention within the network is denoted by the parameters C_x and C_y in equation 4. For example, a logical 2-D array of sub-grids can be 'folded' into the 3-D topology of the Blue Gene interconnection network but will at certain scales result in more than one message requiring the use of the same communication channel, resulting in contention[3,8]. Contention can also occur on a fat tree network due to static routing (as used in Infiniband for example) but can be eliminated through routing table optimization[6]. The node size also determines the number of processing cores that will share the connections to the network and hence also impact the contention.

3.2 Model Validation

Two systems were used to validate the performance model of WRF. The first contains 256 nodes, each containing two dual-core AMD Opteron processors running at 2.0 GHz. Each node contains a single Mellanox Infiniband HCA having a single 4x SDR connection to a 288-port Voltaire ISR9288 switch. In this switch 264 ports are populated for the 256 compute nodes and also for a single head node. The Voltaire ISR9288 implements a two-level 12-ary fat tree. All communication channels have a peak of 10 Gb/s per direction. This system is physically located at Los Alamos National Laboratory.

The second system is a relatively small-scale Blue Gene/L system located at Lawrence Livermore National Laboratory (a sister to the 64K node system used for classified computing). It consisted of two mid-planes, each containing 512 dual-core embedded PowerPC440 processors running at 700 MHz. The main communication network arranges these nodes in a 3-D torus, and a further network is available for some collective operations and for global interrupts.

The characteristics of both of these systems are listed in Table 2. Note that the stated MPI performance is for near-neighbour uni-directional communication[4]. The quantity $n_{1/2}$ is the message size that achieves half of the peak bandwidth. It effectively indicates when a message is latency bound (when its size is less than $n_{1/2}$) or bandwidth bound.



(a) Opteron cluster (b) Blue Gene/L

Figure 2. Measured and modeled performance of WRF for both typical and radiation physics iterations

4 Performance Comparison of Current Systems

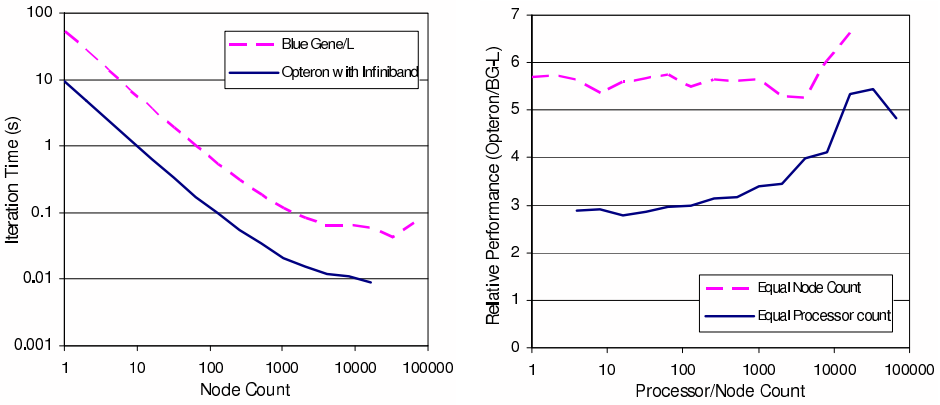
An interesting aspect of this work is the direct performance comparison of Blue Gene/L with the Opteron cluster on the WRF workload. We consider this in two steps, the first based on measured data alone and the second comparing larger scale systems using the performance model described in Section 3. In this way we show the utility of the performance model by exploring the performance of systems that could not be measured.

The time for the typical iteration of WRF on *standard.input* is shown in Fig. 3(a) for both systems using the measured performance up to 256 nodes of the Opteron system, and up to 1024 nodes of the Blue Gene/L system. The model is used to predict the performance up to 16K nodes of an Opteron system and up to 64K nodes of the Blue Gene/L system. It is clear from this data that the Blue Gene/L system has a much lower performance (longer run-time) when using the same number of nodes. It should also be noted that the performance of WRF is expected to improve only up to 32K nodes of Blue Gene/L—this limitation is due to the small sub-grid sizes that occur at this scale for *standard.input*, and the resulting high communication-to-computation ratio.

The relative performance between the two systems is shown in Fig. 3(b). When comparing performance based on an equal number of processors, the Opteron cluster is between 3 and 6 times faster than Blue Gene/L. When comparing performance based on an equal node count, the Opteron cluster is between 5 and 6 times faster than Blue Gene/L. Note that for larger problem sizes, we would expect that runtime on Blue Gene would continue to decrease at larger scales, and thus the additional parallelism available in the system would improve performance.

5 Performance of Possible Future Blue Gene Systems

The utility of the performance model lies in its ability to explore the performance of systems that cannot be directly measured. To illustrate this we consider a potential next-



(a) Time for typical iteration

(b) Relative performance (Opteron to BG/L)

Figure 3. Predicted performance of WRF on large-scale Blue Gene/L and Opteron/Infiniband systems.

		Blue Gene/P
System	Peak	891 Tflops
	Node Count	65,536
	Core Count	262,144
	Core Speed	850 MHz
Nodes	Peak	13.6 Gflops
	Cores	4
	Memory	4 GB
Network	Topology	3D torus
	MPI (zero-byte) latency	1.5 μs
	MPI (1 MB) bandwidth	500 MB/s
	$n_{1/2}$	$\sim$750 B

Table 3. Characteristics of the potential BG/P system.

generation configuration of Blue Gene (Blue Gene/P). The characteristics of Blue Gene/P that are used in the following analysis are listed in Table 3. It should be noted that this analysis was undertaken prior to any actual Blue Gene/P hardware being available for measurement. We also assume the same logical arrangement as the largest Blue Gene/L system presently installed (a 32x32x64 node 3D torus), but containing quad-core processors with an 850 MHz clock speed and increased communication performance. Note that the peak of this system is 891 Tflops.

The predicted performance of WRF using *standard.input* is shown in Fig. 4 and compared with that of Blue Gene/L (as presented earlier in Fig. 3). It was assumed that the processing rate per cell on a Blue Gene/P processing core would be the same as that on a Blue Gene/L processing core. It can be seen that we expect Blue Gene/P to result in improved performance (reduced processing time) when using up to approximately 10,000

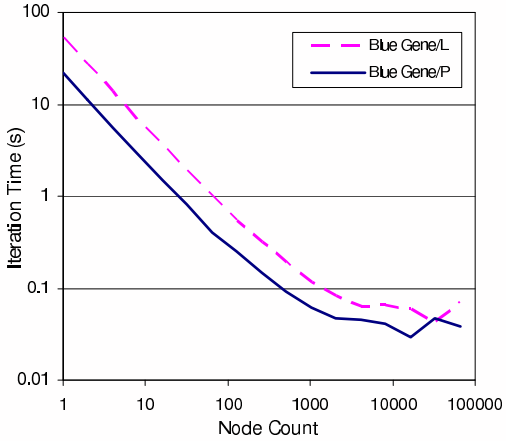


Figure 4. Comparison of Blue Gene/P and Blue Gene/L performance

	Processing time/cell (μs)	MPI latency (μs)	MPI bandwidth (MB/s)
-20%	24	1.80	400
-10%	22	1.65	450
Baseline	20	1.50	500
+10%	18	1.35	550
+20%	16	1.20	600

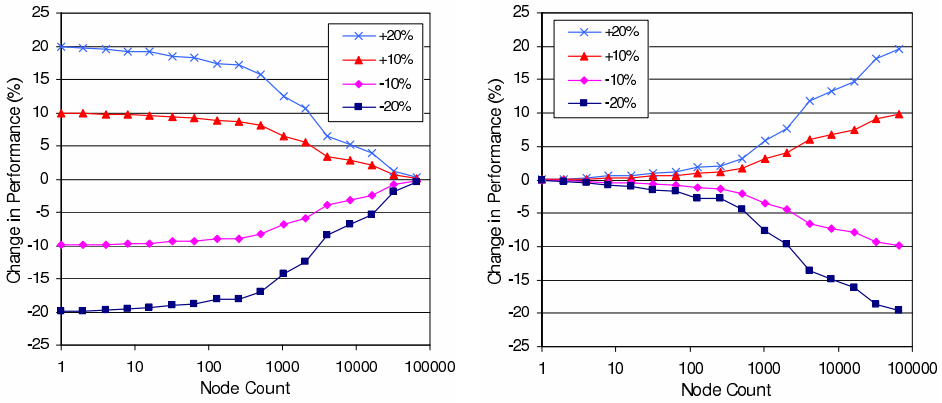
Table 4. Performance characteristics used in the sensitivity analysis of Blue Gene/P.

nodes. For larger input decks the performance should improve to an even larger scale.

The expected performance of WRF on Blue Gene/P is also analyzed in terms of its sensitivity to the compute speed on a single node, the MPI latency, and the MPI bandwidth. The expected performance has a degree of uncertainty due to the inputs to the performance model being assumed rather than measured for the possible configuration of Blue Gene/P.

A range of values for each of the compute performance, MPI latency and MPI bandwidth is used as listed in Table 4. Each of these values are varied from -20% to $+20\%$ of the baseline configuration. Each of the three values is varied independently, that is, one quantity is varied while the other two quantities are fixed at the baseline value.

Two graphs are presented: the range in computational processing rates in Fig. 5(a), and the range in MPI bandwidths in Fig. 5(b). The sensitivity due to MPI latency was not included since the performance varied by at most 0.1%, i.e. WRF is not sensitive to latency. It can be seen that WRF is mostly sensitive to the processing rate up to $\sim$4K nodes (i.e. compute bound in this range), and mostly sensitive to the communication bandwidth at higher scales (i.e. bandwidth bound). Improvements to the processing performance would be most beneficial to improve performance of coarsely partitioned jobs (large cell counts per processor), whereas increased network bandwidth would be most beneficial for jobs executing on larger processor counts.



(a) Sensitivity to compute performance

(b) Sensitivity to MPI bandwidth

Figure 5. WRF sensitivity analysis on Blue Gene/P

6 Conclusions

We have developed and validated an analytic performance model for the Weather Research and Forecasting (WRF) application and used it, in conjunction with empirical data, to quantitatively study application performance on two current generation and one near-term future generation supercomputer. Our analytic performance model was developed through careful study of the dynamic execution behaviour of the WRF application and subsequently validated using performance measurements on two current systems: a 256-node (1,024 core) AMD Opteron cluster using a 4x SDR Infiniband interconnection network, and a 1,024-node (2,048 core) IBM Blue Gene/L system utilizing a custom 3D torus network. In each case the average performance prediction error was less than 5%.

With a validated performance model in place, we are able to extend our analysis to larger-scale current systems and near-term future machines. At small node count, we can see that overall application performance is tied most closely to single processor performance. At this scale, roughly four times as many Blue Gene/L nodes are required to match the performance of the Opteron/Infiniband cluster. At larger scale, communication performance becomes critical; in fact WRF performance on Blue Gene/L improves very slowly beyond roughly 10K nodes due to the communication contention caused by folding the logical 2D processor array onto the physical 3D network.

This work is part of an ongoing project at Los Alamos to develop modelling techniques which facilitate analysis of workloads of interest to the scientific computing community on large-scale parallel systems[5].

Acknowledgements

This work was funded in part by the Department of Energy Accelerated Strategic Computing (ASC) program and by the Office of Science. Los Alamos National Laboratory is

operated by Los Alamos National Security LLC for the US Department of Energy under contract DE-AC52-06NA25396.

References

1. N. R. Adiga, et. al., *An Overview of the Blue Gene/L Supercomputer*, in: Proc. IEEE/ACM Supercomputing (SC'02), Baltimore, MD, (2002).
2. K. J. Barker and D. J. Kerbyson, *A Performance Model and Scalability Analysis of the HYCOM Ocean Simulation Application*, in: Proc. IASTED Int. Conf. on Parallel and Distributed Computing (PDCS), Las Vegas, NV, (2005).
3. G. Banot, A. Gara, P. Heidelberger, E. Lawless, J.C. Sexton, and R. Walkup, *Optimizing Task Layout on the Blue Gene/L Supercomputer*, IBM J. Research and Development, **49**, 489–500, (2005).
4. K. Davis, A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, S. Pakin, and F. Petrini, *A Performance and Scalability Analysis of the Blue Gene/L Architecture*, in: Proc. IEEE/ACM Supercomputing (SC'04), Pittsburgh, PA, (2004).
5. A. Hoisie, G. Johnson, D.J. Kerbyson, M. Lang, and S. Pakin, *A Performance Compariosn through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/L, Red Storm, and Purple*, in: Proc. IEEE/ACM Supercomputing (SC'06), Tampa, FL, (2006).
6. G. Johnson, D.J. Kerbyson, and M. Lang, *Application Specific Optimization of Infiniband Networks*, Los Alamos Unclassified Report, LA-UR-06-7234, (2006).
7. D.J. Kerbyson, and A. Hoisie, *Performance Modeling of the Blue Gene Architecture*, in: Proc. IEEE John Atanasoff Conf. on Modern Computing, Sofia, Bulgaria, (2006).
8. D.J. Kerbyson and P.W. Jones, *A Performance Model of the Parallel Ocean Program*, Int. J. of High Performance Computing Applications, **19**, 1–16, (2005).
9. V. Salapura, R. Walkup and A. Gara, *Exploiting Workload Parallelism for Performance and Power Optimization in Blue Gene*, IEEE Micro **26**, 67–81, (2006).
10. Weather Research and Forecasting (WRF) model, http://www.wrf-model.org.

Analytical Performance Models of Parallel Programs in Clusters

Diego R. Martínez[1], **Vicente Blanco**[2], **Marcos Boullón**[1], **José Carlos Cabaleiro**[1], **and Tomás F. Pena**[1]

[1] Dept. of Electronics and Computer Science
University of Santiago de Compostela, Spain
E-mail: {diegorm, marcos, caba, tomas}@dec.usc.es

[2] Dept. of Statistics and Computer Science
La Laguna University, Spain
E-mail: vicente.blanco@ull.es

This paper presents a framework based on an user driven methodology to obtain analytical models on parallel systems and, in particular, clusters. This framework consists of two interconnected stages. In the first one, the analyst instruments the source code and some performance parameters are monitored. In the second one, the monitored data are used to obtain an analytical model using statistical processes. The main functionalities added to the analysis stage include an automatic fit process that provides accurate performance models and the automatic data collection from monitoring. Some examples are used to show the automatic fit process. The accuracy of the models is compared with a complexity study of the selected examples.

1 Introduction

Performance prediction is important in achieving efficient execution of parallel programs. Understanding performance is important not only for improving efficiency of applications, but also for guiding enhancements to parallel architectures and parallel programming environments. As systems become more complex, as in the case of multiprocessor environments or distributed systems, accurate performance estimation becomes a more complex process due to the increased number of factors that affect the execution of an application. In addition to relatively static information, there are many dynamic parameters that must be taken into account in the performance estimation. These dynamic parameters may not be known until run time[10].

Performance models can be grouped into three categories: analytical modelling, simulation modelling, and measurement[18]. Analytical model evaluation takes a little time since it is based on solutions to mathematical equations. However, it has been less successful in practice for predicting detailed quantitative information about program performance due to the assumptions and simplifications built in the model. Simulation modelling constructs a reproduction, not only of the behaviour of the modeled system, but also its structure and organization. Simulation model should be more accurate than analytical models but it is more expensive and time consuming, and can be unaffordable for large systems. Measurement methods permits to identify bottlenecks on a real parallel system. This approach is often expensive because it requires special purpose hardware and software, and in real system they are not always available. Performance measurement can be highly accurate when a correct instrumentation design is carried out for a target machine.

Although analytical approaches are generally less accurate than simulation approaches, they are faster and more efficient since the behaviour is described through mathematical equations. Moreover, analytical modelling provides an abstract view of the hardware and software. Analytical modelling is a common issue in performance evaluation of parallel systems: parallel performance analytical models based on scalar parameters, like BSP[16], LogP[6] or LogGP[1], are widely used to evaluating parallel systems; the information of analytical models can be useful to optimize load balancing strategies[11]; Bosque et. al. propose a heterogeneous parallel computational model based on the LogGP model[4]. Some approaches combine empirical experimentation, analytical modelling and perhaps some light-weight forms of simulation[5,7,15]. The experimentation and simulation are used to characterize the application while the analytical model is used to predict the performance behaviour in terms of the characterization of the algorithm and the knowledge of the platform. This approach is getting relevance in Grid environments[2,9]. Its use is important not only for achieving efficient execution of parallel programs, but also for taking advantage of its qualities. For example, a quick evaluation of the performance behaviour of an application is desirable in a scheduler in order to obtain an efficient use of the computational resources of the grid[8,19].

The proposed framework uses both measurement and analytical modelling and provides an easy to use tool that allows the analyst to obtain analytical models to characterize the performance of parallel applications in clusters. The framework helps the analyst in the process of tuning parallel applications and automatically performs some tedious processes related to performance analysis. Its modular design makes it possible to introduce new tools in the framework, or even to substitute current functionalities, with a minimal impact on the rest of the framework. Analytical models are automatically obtained by a statistical analysis of measurements from real parallel executions. Therefore, the behaviour of an application can be characterized in terms of parameters such as the problem size, the number of processes, the effective network capacity, etc.

In this paper the methodology used to obtain analytical models from instrumentation data is described in detail, and some examples of its use are illustrated. Section 2 introduces the proposed framework based on two stages. The second stage where analytical models are obtained is described in detail in Section 3, and selected examples are used in Section 4 to show its use. Finally, Section 5 presents the main conclusions of this work and future developments.

2 The Modelling Framework

A methodology to obtain analytical performance models of MPI applications is introduced in this section. Fig. 1 shows a scheme of the whole framework that consists of two stages. The first stage is devoted to instrumentation, where information about the performance of the execution of the parallel application is monitored. The second stage uses this information to obtain an analytical model by means of statistical analysis to characterize the performance of the parallel code. This stage will be explained detailed in Section 3.

The instrumentation stage is based on CALL[3], which is a profiling tool for interacting with the code in an easy, simple and direct way. It controls external monitoring tools through the so called CALL drivers, that implement an application performance interface to those external tools. Therefore, CALL is a modular tool and new drivers can be developed

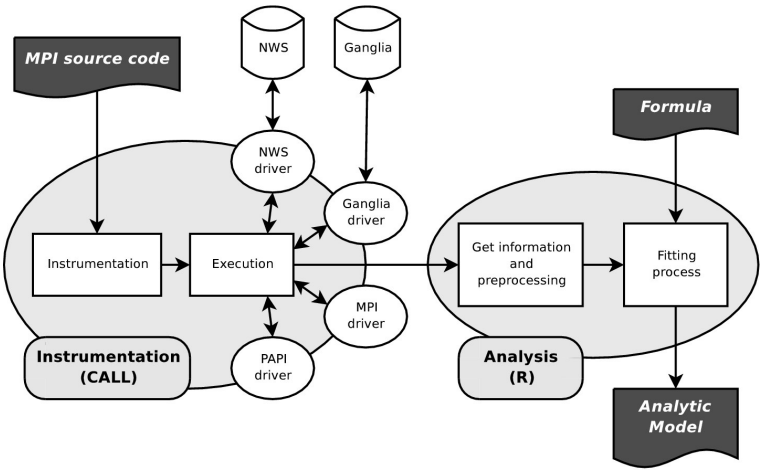


Figure 1. The two stages of the framework.

as new monitoring tools are available. CALL is based on pragmas that allow the user to define areas of interest in the source code. In these parts of the code, some selected parameters are monitored, so specific kernels or functions may be modeled instead of the whole application.

The measurements of the monitored parameters during the execution of the instrumented code are gathered and stored in XML files, one file for each CALL driver used to monitor the performance parameters. Some CALL drivers were specifically designed for cluster environments, like the NWS CALL driver and the Ganglia CALL driver[12]. The NWS CALL driver uses NWS[17] to monitor the effective latency and bandwidth of a parallel environment. Through a configuration file, the analyst can establish which nodes must be monitored during the execution of a NWS CALL experiment. The Ganglia CALL driver provides several performance metrics, both static and dynamic, of the nodes in a parallel system[13]. These two drivers play an important role in this stage, and their data are used in the analysis stage.

3 Analysis Stage

The second stage is the analysis one, that is based on R[14], a language and environment for statistical analysis. In early versions of CALL, the statistical analysis was integrated with the instrumented stage so that information about the behaviour of the CALL experiments had to be incorporated before the instrumented code was actually executed. The analysis stage was redesigned to get both stages fully uncoupled. Therefore, specific R functions were developed to process the data from multiple executions of a CALL experiment and to automatically perform analytical models of CALL experiments by means of an iterative fitting process. These functions are grouped into modules with well defined interfaces, so any change in a module produces a minimal impact on the others, and the capabilities of the analysis environment can be easily extended. Fig. 2 shows a scheme of the analysis

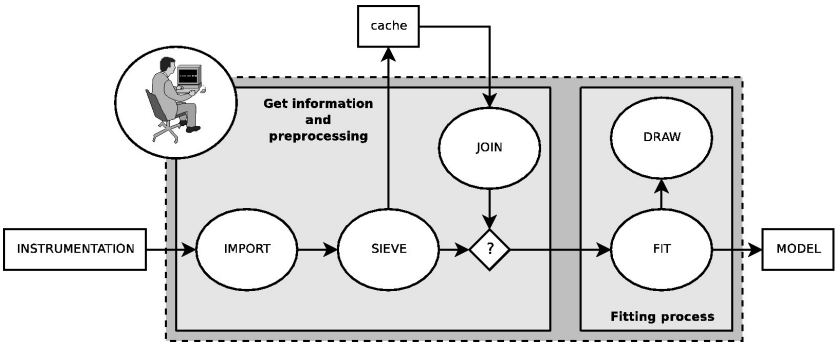


Figure 2. The modules of the analysis stage based on R.

stage and the relationship among its five modules.

The information stored in XML files from a particular CALL experiment is loaded into the environment, and it is stored in suitable structures for the statistical processing (IMPORT module in Fig. 2). At this point, some preprocessing may be necessary to obtain derived parameters. For example, NWS CALL driver provides information about the network state of some point-to-point communications. In general, this information consists of an array of values, where the impact of network fluctuations is present, and it could be necessary to obtain the mean of these values in order to associate only one value of latency and bandwidth to one execution of a CALL experiment.

The resulting object of the IMPORT module is an array of registers. Each register contains the monitored information related to a specific execution of the CALL experiment. In parallel applications, each parallel task has its own register. The number of fields of the register depends on the number of monitored parameters during the execution of the instrumented code. This object contains a huge amount of useful information, so the relevant data for a particular performance analysis must be extracted (SIEVE module in Fig. 2). For example, in a parallel program, the elapsed time of the slowest task is the elapsed time of the whole parallel program, so the elapsed time of the other tasks may be unnecessary; although this information can be useful if the user is interested in load balancing studies, for example. Therefore, a selection of useful information for the later analysis is performed guided by the user. In this step, the statistical outliers are checked, so executions with a elapsed time too far from the mean for executions under the same experimental conditions are not considered.

At this point, the framework has been designed to save this information in a file with a specific statistical format (cache in Fig. 2). Therefore, gathering information from CALL experiments becomes an independent process from the generation of a performance model. All the files concerning a specific CALL experiment can be stored, so that the volume of data increases as new executions of a instrumented code are performed. These files can be joined in the same data structure for the next fit process (JOIN module in Fig. 2). Note that only the common parameters in all instances of a CALL experiment are taken into account.

Once the information from all considered sources is ready, these data are fitted to an analytical expression which accurately describes the behaviour of the elapsed time of the CALL experiment as a function of the monitored parameters (FIT module in Fig. 2). This

is the main concern of the analysis stage. It starts from an initial attempt function as a first approximation of the elapsed time and continues with a set of iterations. In each iteration, the accuracy of the fit is established based on the standard error of the coefficients of the function. If the relative error of a coefficient is greater than a certain threshold (in our experiments, we consider 10%), it is assumed that such term can be neglected or that the model does not fit properly. Then, this term is removed and a new iteration is performed. This iterative process ends when all coefficients have an error smaller than the threshold. This process can start with a set of functions so that the process is applied to each individual function and the best fit is selected as the final result. The time consumption of the fitting process depends both on the number of terms of the initial function and on the size of data.

The user can provide the initial function or the initial set of functions of this process. Otherwise, a set of predefined and generic functions is used. These predefined functions implement a simple model based on the combination of a computational term (related to the number of processors) and a communication term (related to the latency and bandwith of the network). Each term is defined as a sum of different terms, which can be logarithms and powers of the monitored parameters, or a product of logarithmic and power terms, etc. Anyway, only expressions which have a theoretical base are considered.

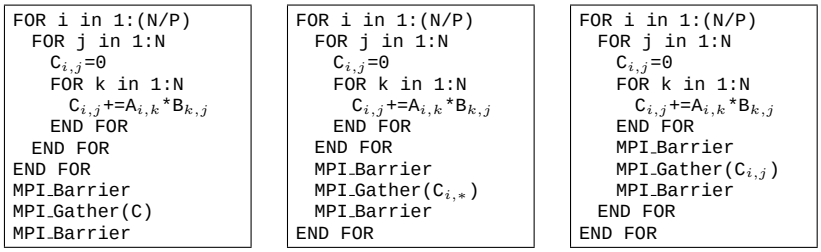
All the information about the resulting model of this process is shown to the user including coefficient of determination, standard error of coefficients, and some other useful statistical information. Besides, a comparison study between experimental measurements and the obtained model is performed through a specific visual environment (DRAW module in Fig. 2). In parallel applications, this environment shows both the experimental and predicted elapsed time versus the number of processes during the execution of the instrumented code. These graphics show the estimated times according to the model and the experimental values for all considered experimental conditions. The residuals of the model can be shown through a box-and-whisker plot.

4 Case of Study

Three different MPI versions of the parallel matrix product of two dense $N \times N$ matrix of single-precision floating-point numbers have been used as a case of study. Fig. 3 shows the pseudocode of these versions, that cover a broad spectrum from communication intensive to computation intensive applications. It is supposed that the distribution of the matrices has been performed in a previous step. The goal of the experiment is to compare both a theoretical model and the automatic model obtained from the proposed analysis stage using a set of predefined functions in a simple and well known parallel code.

The instrumented programs were executed in a homogeneous cluster of six 3 GHz Pentium 4 connected by a Gigabit Ethernet switch. The driver of the network interface card of the nodes allows the user to force the speed of the network. Therefore, different network states (10, 100, and 1000 Mbps) were obtained by using different input parameters of this driver. The NWS CALL driver was used to monitor the network state just before the execution of the programs. Each program was executed four times using three matrix sizes (N=300, 400 and 500) for each number of processors (from 2 to 6) and each network state.

A model of the three cases was automatically obtained using the developed analysis stage. As the structure of the code is known in the three cases, the parameters that may



a) CASE1 b) CASE2 c) CASE3

Figure 3. Pseudocode of the three versions of parallel matrix product (C=A×B), where the size of the matrices is $N \times N$, and P is the number of processes.

have influence in the application performance were identified. In this case, the following parameters were selected: N, P, L, and BW, where N is the size of the matrices, P the number of processors, L the mean latency of the network and BW is the mean bandwith of the network. The proposed initial set of functions for the iterative fitting process (described in Section 3) were identical in all cases. These functions cover all possible combination of the following expression:

$$t = A + BN^iP^j + CN^iP^j\lceil \log_2 P \rceil L + DN^iP^jBW^{-1}$$

where $i = 0, 1, 2, 3$, $j = 0, -1$ and the coefficients B, C and D can be set to 0.

In order to evaluate the accuracy of the proposed automatic fit process, the data from the instrumentation stage were also fitted to a theoretical expression based on the complexity analysis of the three codes. To build this theoretical model, a tree-base behaviour was supposed in global MPI communication. Besides, the effect of the communication-computation overlapping was supposed to be negligible.

Table 1 shows the final expressions obtained using the automatic approach of our analysis stage and the theoretical functions for each case. In these expressions, t is the elapsed time. The third column shows the coefficient of determination (R^2) for both the automatic and the theoretical fits from the complexity analysis. Note that the theoretical expressions presents accurate results, even for the third case in which the number of communications is huge.

5 Conclusions and Ongoing Work

A framework based on a methodology to obtain analytical models of MPI applications on multiprocessor environments is introduced in this paper. The framework consists of an instrumentation stage followed by an analysis stage, which are based on the CALL instrumentation tool and in the R language, respectively. A number of functionalities were developed in the analysis stage to help the performace analyst to study analytical models in parallel environments. One of the most relevant features is an automatic fit process to obtain an analytical model of parallel programs. This process provides accurate models as good as those based on a theoretical complexity analysis of source codes.

We are currently developing an efficient and flexible process to automatically generate and fit all analytical attempt functions. The process will permit the user to introduce in-

	Models	$\mathbf{R}^2$
CASE1	$t_{\text{Automatic}} = b\frac{N^3}{P} + d\frac{N^2}{BW}$	0.9945
	$t_{\text{Theoretical}} = A + B\frac{N^3}{P} + C\lceil\log_2 P\rceil L + D\frac{N^2}{BW}$	0.9946
CASE2	$t_{\text{Automatic}} = b\frac{N^3}{P} + cN\lceil\log_2 P\rceil L + d\frac{N^2}{BW}$	0.9966
	$t_{\text{Theoretical}} = A + B\frac{N^3}{P} + C\frac{N}{P}\lceil\log_2 P\rceil L + D\frac{N^2}{P}\frac{1}{BW}$	0.9904
CASE3	$t_{\text{Automatic}} = bN^2 + d\frac{N^2}{BW}$	0.980
	$t_{\text{Theoretical}} = A + B\frac{N^3}{P} + C\frac{N^2}{P}\lceil\log_2 P\rceil L + D\frac{N^2}{P}\frac{1}{BW}$	0.956

Table 1. The automatically obtained functions and the theoretical functions in each case. The third column is the corresponding coefficient of determination.

formation about the behaviour of the code, so that the precision of the obtained model will depend on the amount of the provided information. If no information is introduced, the monitored parameters will be used to build the initial functions. This process will provide a full search on all possible attempt analytical functions with physical meaning.

Acknowledgements

This work was supported by the Spanish Ministry of Education and Science through TIN2004-07797-C02 and TIN2005-09037-C02-01 projects and the FPI programme.

References

1. A. Alexandrov, M. F. Ionescu, K. E. Schauser and C. Scheiman, *LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation*, J. Parallel and Distributed Computing, **44**, 71–79 (1997).
2. R. M. Badía, J. Labarta, J. Giménez and F. Escale, *DIMEMAS: Predicting MPI applications behavior in Grid environments*, in: Workshop on Grid Applications and Programming Tools (GGF8), (2003).
3. V. Blanco, J.A. González, C. León, C. Rodríguez, G. Rodríguez and M. Printista, *Predicting the performance of parallel programs*, Parallel Computing, **30**, 337–356, (2004).
4. J. L. Bosque and L. Pastor, *A Parallel Computational Model for Heterogeneous Clusters*, IEEE Trans. Parallel Distrib. Syst., **17**, 1390–1400, (2006).
5. M. E. Crovella and Th. J. LeBlanc, *Parallel Performance Prediction Using Lost Cycles Analysis*, in: Proc. 1994 ACM/IEEE Conference on Supercomputing, (1994).
6. D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: Towards a realistic model of parallel computation*, in: 4th

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (1993).
7. Th. Fahringer, M. Gerndt, G. D. Riley and J. Larsson Träff, *Specification of Performance Problems in MPI Programs with ASL*, in: International Conference on Parallel Processing (ICPP'00), (2000).
8. I. Foster and C. Kesselman, *The Grid2: Blueprint for a New Computing Infrastructure*, Elsevier, Inc., (2004).
9. S. Jarvis, D. P. Spooner, H. N. Lim Choi Keung, J. Cao, S. Saini, and G. R. Nudd. *Performance prediction and its use in parallel and distributed computing systems*, Future Generation Computer Systems: Special Issue on System Performance Analysis and Evaluation, **2**, 745–754, (2006).
10. N. H. Kapadia, J. A. B. Fortes and C. E. Brodley. *Predictive Application-Performance Modeling in a Computational Grid Environment*, in: 8th IEEE International Symposium on High Performance Distributed Computing (HPDC '99), (1999).
11. D. R. Martínez, J. L. Albín, J. C. Cabaleiro, T. F. Pena and F. F. Rivera, *A load balance methodology for highly compute-intensive applications on grids based on computational modeling*, in: Proc. Grid Computing and its Application to Data Analysis (GADA'05) - OTM Federated Conferences and Workshops, (2005).
12. D. R. Martínez, V. Blanco, M. Boullón, J. C. Cabaleiro, C .Rodríguez and F. F. Rivera, *Software tools for performance modeling of parallel programs*, in: Proceedings of the IEEE International Parallel & Distributed Processing Symposium, (2007).
13. M. L. Massie, B. N. Chun and D. E. Culler, *The ganglia distributed monitoring system: design, implementation, and experience*, Parallel Computing, **30**, 817–840 (2004).
14. R Development Core Team (2006), *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.
15. A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. *A framework for performance modeling and prediction*, in: Proc. ACM/IEEE conference on Supercomputing, (2002).
16. L. G. Valiant, *A Bridging Model for Parallel Computation*, Commun. ACM, **33**, 103–111, (1990).
17. R. Wolski, N. Spring and J. Hayes, *The Network Weather Service: A distributed resource performance forecasting service for metacomputing*, J. Future Generation Computing Systems, **15**, 757–768, (1999).
18. X. Wu, *Performance Evaluation, Prediction and Visualization of Parallel Systems*, (Kluwer Academic Publishers, 1999).
19. Y. Zhang, C. Koelbel and K. Kennedy, *Relative Performance of Scheduling Algorithms in Grid Environments*, in: Seventh IEEE International Symposium on Cluster Computing and the Grid – CCGrid (2007).

Computational Force: A Unifying Concept for Scalability Analysis

Robert W. Numrich

Minnesota Supercomputing Institute
University of Minnesota
Minneapolis, MN 55455 USA
E-mail: rwn@msi.umn.edu

Computational force, also called computational intensity, is a unifying concept for understanding the performance of parallel numerical algorithms. Dimensional analysis reduces a formula for execution time, from a paper by Stewart, to an exercise in differential geometry for a single efficiency surface. Different machines move on the surface along different paths defined by curvilinear coordinates that depend on ratios of hardware forces to software forces.

1 Introduction

In an early paper on the scalability of parallel machines[1], Stewart reported formulas for execution time for eight different algorithms. The one algorithm he described in detail yielded the formula,

$$t = \alpha n^3/p + 2n\sqrt{p}(\sigma + \tau) + (\sqrt{p} + mn)(\sigma + n\tau/(m\sqrt{p})) \; . \tag{1.1}$$

Table 3 of Stewart's paper displayed the quantities involved in his formula and the units he used to measure them. The quantity t was the execution time; α was the reciprocal of the computational power of a single processor; σ was a startup time; τ was the reciprocal of bandwidth; n was the matrix size; m was a block size; and p was the number of processors. He measured length in units of eight-byte words ($l_0 = 1$ word), work in the units of eight-byte floating-point operations ($e_0 = 1$ flop), and time in units of seconds ($t_0 = 1$ s). The details of the algorithm are not important for the purpose of this paper. The interested reader should consult Stewart's original paper.

2 Dimensional Analysis

Dimensional analysis is based on the algebraic properties of the matrix of dimensions[2,3,4,5,6,7]. In the system of units used by Stewart, the quantities involved in formula (1.1) have the matrix of dimensions,

	l_0	e_0	t_0	t	α	σ	τ	n	m	p
L(word)	1	0	0	0	0	0	−1	0	0	0
E(flop)	0	1	0	0	−1	0	0	0	0	0
T(s)	0	0	1	1	1	1	1	0	0	0

(2.1)

The entries for each quantity are the powers for each unit that correspond to its dimension. For example, reciprocal bandwidth τ is measured in units (s/word) with entries (-1,0,1), and reciprocal power α is measured in units (s/flop) with entries (0,-1,1).

From matrix (2.1), it is clear that formula (1.1) is not dimensionally consistent. Stewart's analysis, along with many others like it, lacks a clear distinction between the unit of length l_0 and the unit of work e_0. The problem size, which is dimensionless, enters into the calculation of both the number of operations performed and the number of words moved. Because the work grows like the cube of the problem size, the first term in formula (1.1) should be written as $(\alpha n^3/p)e_0$ to give it the correct dimension of time. Furthermore, adding together the startup time σ and the reciprocal bandwidth τ makes no sense unless we multiply τ by the length of the transfer. In the first appearance of the quantity τ in formula (1.1), the length is presumably a single word of length l_0, and in the second appearance, it is nl_0.

To make the formula dimensionally consistent and to simplify it for the next step in the analysis, we define the square root of the number of processors,

$$q = \sqrt{p} \; , \tag{2.2}$$

and the work done per processor,

$$w = n(n/q)^2 e_0 \; . \tag{2.3}$$

We also define the dimensionless quantity,

$$s = n + 2nq + q \; , \tag{2.4}$$

and the length,

$$l = (2nq + (n/q)(n+q))l_0 \; , \tag{2.5}$$

noting that Stewart sets the parameter $m = 1$. Execution time then obeys the formula,

$$t = \alpha w + s\sigma + l\tau \; , \tag{2.6}$$

Rewriting the formula in correct dimensional form is not a pedantic quibble. It is an essential step in dimensional analysis[2,3,4,5,6,7].

We need, therefore, to consider execution time as a function of six dependent variables,

$$t = t(\alpha, w, s, \sigma, l, \tau) \; . \tag{2.7}$$

For the next step in the analysis, we switch to a system of measurement based on length, force, and time as primary units. Computational force, measured in the unit,

$$f_0 = e_0/l_0 = 1 \text{ flop/word} \; , \tag{2.8}$$

is also called computational intensity[8,9,10,11,12]. It is recognized as an important quantity in performance analysis, and we show, in what follows, that it provides a unifying concept for scalability analysis. In this new system of units, the quantities involved have the matrix of dimensions,

	l_0	f_0	t_0	t	α	w	σ	τ	l	s
L(word)	1	0	0	0	−1	1	0	−1	1	0
F(flop/word)	0	1	0	0	−1	1	0	0	0	0
T(s)	0	0	1	1	1	0	1	1	0	0

(2.9)

The fundamental assumption of dimensional analysis is that relationship (2.7) is independent of the system of units we use[2,3,4,5,6,7]. Under this assumption, we may pick

three parameters, $\alpha_L, \alpha_F, \alpha_T$, one for each primary unit, and scale each quantity in the relationship, using the entries in the matrix of dimensions (2.9), such that

$$\alpha_T t = t(\alpha_L^{-1}\alpha_F^{-1}\alpha_T\alpha, \alpha_L\alpha_F w, s, \alpha_T\sigma, \alpha_L l, \alpha_L^{-1}\alpha_T\tau) \; . \tag{2.10}$$

We can pick these scale factors in such a way to establish a new system of units where the number of parameters reduces from six to three and all quantities are scaled to dimensionless ratios. Indeed, if we pick them such that

$$\alpha_L^{-1}\alpha_F^{-1}\alpha_T\alpha = 1 \; , \qquad \alpha_L\alpha_F w = 1 \; , \qquad \alpha_L l = 1 \; , \tag{2.11}$$

we can solve for the three parameters to find

$$\alpha_L = 1/l \; , \qquad \alpha_F = l/w \; , \qquad \alpha_T = 1/(\alpha w) \; . \tag{2.12}$$

In this new system of units, formula (2.10) becomes

$$t/(\alpha w) = t(1, 1, s, \sigma/(\alpha w), 1, l\tau/(\alpha w)) \; . \tag{2.13}$$

We recognize the time,

$$t_* = \alpha w \; , \tag{2.14}$$

as the minimum time to complete the computation. The left side of formula (2.13) is, therefore, the reciprocal of the efficiency,

$$t/t_* = 1/e \; . \tag{2.15}$$

Scaling (2.6) with the parameters from (2.12), we find the efficiency function,

$$e = [1 + s\sigma/(\alpha w) + l\tau/(\alpha w)]^{-1} \; . \tag{2.16}$$

3 Computational Forces and the Efficiency Surface

The efficiency formula (2.16) depends on ratios of computational forces. To recognize this fact, we first observe that the efficiency formula assumes the simple form,

$$e = \frac{1}{1 + u + v} \; , \tag{3.1}$$

in terms of the two variables,

$$u = s\sigma/(\alpha w) \qquad v = l\tau/(\alpha w) \; . \tag{3.2}$$

Each of these variables is the ratio of opposing computational forces. Using the quantities defined by equations (2.3)-(2.5), we find the first variable,

$$u(n, q) = \frac{\phi_1^H}{\phi_1^S(n, q)} \; , \tag{3.3}$$

is the ratio of a hardware force,

$$\phi_1^H = (\sigma/l_0)/\alpha \; , \tag{3.4}$$

to a software force,

$$\phi_1^S = \frac{n(n/q)^2}{n + 2nq + q} f_0 \; . \tag{3.5}$$

In the same way, the second variable is the ratio of two other opposing forces,

$$v(n,q) = \frac{\phi_2^H}{\phi_2^S(n,q)}, \tag{3.6}$$

with a second hardware force,

$$\phi_2^H = \tau/\alpha \ , \tag{3.7}$$

and a second software force,

$$\phi_2^S(n,q) = \frac{n(n/q)^2}{2nq + (n/q)(n+q)} f_0 \ . \tag{3.8}$$

The two hardware forces are independent of the problem size and the number of processors. The first one is determined by the ratio of the startup time to the reciprocal of the computational power, and the second one is determined by the ratio of the reciprocal of the bandwidth to the reciprocal of the computational power. The two software forces, on the other hand, are functions of problem size and the number of processors, independent of the hardware. They are determined by this particular algorithm.

For fixed problem size, $n = n^*$, the curvilinear coordinates,

$$u^*(q) = u(n^*, q) \qquad v^*(q) = v(n^*, q) \ , \tag{3.9}$$

define paths along the surface,

$$e^*(q) = \frac{1}{1 + u^*(q) + v^*(q)} \ , \tag{3.10}$$

as the number of processors changes. Similarly, for fixed number of processors, $q = q_*$, the curvilinear coordinates,

$$u_*(n) = u(n, q_*) \qquad v_*(n) = v(n, q_*) \ , \tag{3.11}$$

define paths along the surface,

$$e_*(n) = \frac{1}{1 + u_*(n) + v_*(n)} \ , \tag{3.12}$$

as the problem size changes. Different machines follow different paths determined by their particular values of the hardware forces ϕ_1^H and ϕ_2^H.

4 Discussion

If we use the same numerical algorithm, the software forces have not changed in the decade and a half since the publication of Stewart's paper. But the hardware forces have changed as shown in the following table,

	Machine 1 ca. 1990	Machine 2 ca. 2007
$1/\alpha$	10^6 flop/s	10^9 flop/s
σ	10^{-4} s	10^{-6} s
$1/\tau$	10^6 word/s	10^9 word/s
ϕ_1^H	10^2 flop/word	10^3 flop/word
ϕ_2^H	1 flop/word	1 flop/word

(4.1)

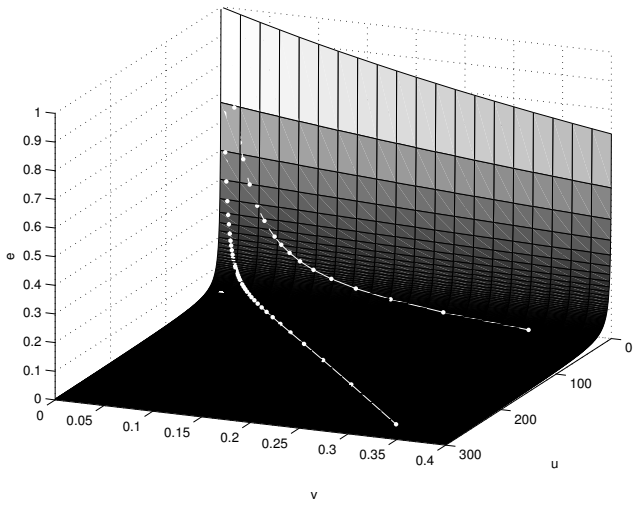


Figure 1. Two paths along the efficiency surface for fixed number of processors, $p = q^2 = 32^2 = 1024$. The higher path corresponds to the older Machine 1; the lower path to the modern Machine 2. The problem size increases from $n = 500$ at low efficiency in increments of 100. In the limit of very large problem size, both machines approach perfect efficiency. They approach the limit at different rates along different paths on the surface.

Because of these changes, modern machines follow paths lower on the efficiency surface than paths followed by older machines.

Fig. 1 shows two paths along the efficiency surface, one for each machine in Table 4.1 for a fixed number of processors, $p = q^2 = 32^2 = 1024$. Each point on a path $e_*(n)$, calculated from (3.12) using the curvilinear coordinates $u_*(n)$ and $v_*(n)$ from (3.11), corresponds to a different problem size. For large enough problem, both machines approach perfect efficiency. But they follow different paths at different rates along the surface to reach the summit. The modern machine follows a path lower on the efficiency surface at all values of the problem size and is less efficient than the older machine.

The reason for this loss of efficiency is clear. The modern machine has higher values for the coordinate $u_*(n)$ for each problem size. It travels a long way across the surface at low efficiency before mounting the ascent to the summit. The coordinate $u_*(n)$ depends on the hardware force ϕ_1^H, which depends on the startup time σ. As Table 4.1 shows, this force has increased in the last two decades. It is harder now to overcome long startup times than it was before. This difficulty translates directly into a harder job for the programmer who must provide a larger software force ϕ_1^S to overcome the larger hardware force. To increase the software force, the programmer must design a new algorithm for the problem.

5 Summary

Our approach to performance analysis is quite different from the more familiar approaches that have appeared in the literature. Our first impulse is to plot the efficiency surface, or

contours of the surface, as a function of the problem size and the number of processors. With that approach, each machine has its own efficiency surface depending on its particular values of the hardware forces ϕ_1^H and ϕ_2^H. It is difficult to compare machines because they lie on different surfaces.

Our approach produces one efficiency surface for all machines. Different machines follow different paths along the same surface depending on their hardware forces ϕ_1^H and ϕ_2^H. The distance between paths, along the geodesic on the surface, measures the distance between machines. Two machines with the same hardware forces are self-similar and scale the same way even though the particular values for their startup time σ, their bandwidth τ^{-1}, and their computational power α^{-1} may be quite different. Only the forces determined by the ratios of these quantities matter not their absolute values. These forces provide a unifying concept for scalability analysis.

Acknowledgement

The United States Department of Energy supported this research by Grant No. DE-FG02-04ER25629 through the Office of Science.

References

1. G. W. Stewart, *Communication and matrix computations on large message passing systems*, Parallel Computing, **16**, 27–40, (1990).
2. G. I. Barenblatt, *Scaling*, Cambridge University Press, (2003).
3. G. Birkhoff, *Hydrodynamics: A Study in Logic, Fact and Similitude*, Princeton University Press, 2nd edition, (1960).
4. L. Brand, *The Pi Theorem of Dimensional Analysis*, Arch. Rat. Mech. Anal., **1**, 35–45, (1957).
5. P. W. Bridgman, *Dimensional Analysis*, 2nd ed., (Yale Univ. Press, 1931).
6. R. W. Numrich, *A note on scaling the Linpack benchmark*, J. Parallel and Distributed Computing, **67**, 491–498, (2007).
7. R. W. Numrich, *Computer Performance Analysis and the Pi Theorem*, (submitted 2007).
8. D. Callahan, J. Cocke and K. Kennedy, *Estimating Interlock and Improving Balance for Pipelined Architectures*, J. Parallel Dist. Comp., **5** 334–358, (1988).
9. R. W. Hockney, *Performance parameters and benchmarking of Supercomputers*, Parallel Computing, **17**, 1111–1130, (1991).
10. R. W. Hockney, *The Science of Computer Benchmarking*, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, (1996).
11. J. D. McCalpin, *Memory Bandwidth and Machine Balance in Current High Performance Computers*, IEEE Computer Society; Technical committee on Computer Architecture Newsletter, (December 1995).
12. D. Miles, *Compute Intensity and the FFT*, in: Proc. Supercomputing 1993, pp. 676–684, (1993).

Distribution of Periscope Analysis Agents on ALTIX 4700

Michael Gerndt, Sebastian Strohhäcker

Technische Universität München, Fakultät für Informatik I10
Boltzmannstr.3, 85748 Garching, Germany
E-mail: gerndt@in.tum.de

Periscope is a distributed automatic online performance analysis system for large scale parallel systems. It consists of a set of analysis agents distributed on the parallel machine. This article presents the approach taken on the ALTIX 4700 supercomputer at LRZ to distribute the analysis agents and the application processes on to the set of processors assigned to a parallel job by the batch scheduling system. An optimized mapping reducing the distance between the analysis agents and the application processes is computed based on the topology information of the processors. This mapping is then implemented via the dplace command on the Altix.

1 Introduction

Performance analysis tools help users in writing efficient codes for current high performance machines. Since the architectures of today's supercomputers with thousands of processors expose multiple hierarchical levels to the programmer, program optimization cannot be performed without experimentation.

Performance analysis tools can provide the user with measurements of the program's performance and thus can help him in finding the right transformations for performance improvement. Since measuring performance data and storing those data for further analysis in most tools is not a very scalable approach, most tools are limited to experiments on a small number of processors.

Periscope[1,2,3] is the first distributed online performance analysis tool. It consists of a set of autonomous agents that search for performance properties. Each agent is responsible for analyzing a subset of the application's processes and threads. The agents request measurements of the monitoring system, retrieve the data, and use the data to identify performance properties. This approach eliminates the need to transport huge amounts of performance data through the parallel machine's network and to store those data in files for further analysis.

The focus of this paper is on the distribution of application processes and analysis agents in Periscope. Large scale experiments with Periscope are executed in form of batch jobs where in addition to the processors for the application additional processors are allocated for the analysis agents. The number of additional processors is currently decided by the programmer. The analysis agents provide feedback to the user, whether they were overloaded with the number of processes. In such a situation, the programmer might decide to use more processors for the analysis in a next experiment.

During startup of the experiment, Periscope determines the mapping of application processes and analysis agents to the processors. It is the goal, to place analysis agents near to the controlled processes to reduce the communication overhead. This paper describes the concepts and the implementation used on the ALTIX 4700 supercomputer at LRZ for placement.

The next section gives a short overview of related work. Section 3 presents Periscope's architecture. Section 4 introduces our target system. The mapping features of the batch system on the ALTIX are introduced in Section 5. Section 6 presents the details on the distribution of the application processes and the analysis agents. Section 7 presents results of our simulated annealing-based optimization approach.

2 Related Work

Several projects in the performance tools community are concerned with the automation of the performance analysis process. Paradyn's[4] Performance Consultant automatically searches for performance bottlenecks in a running application by using a dynamic instrumentation approach. Based on hypotheses about potential performance problems, measurement probes are inserted into the running program. Recently MRNet[5] has been developed for the efficient collection of distributed performance data. However, the search process for performance data is still centralized. To remedy the bottleneck this centralized approach presents for large-scale machines, work towards a Distributed Performance Consultant (DPC)[6] was recently conducted.

The Expert[7] tool developed at Forschungszentrum Jülich performs an automated post-mortem search for patterns of inefficient program execution in event traces. Potential problems with this approach are large data sets and long analysis times for long-running applications that hinder the application of this approach on larger parallel machines. More recently, Scalasca[8] was developed as a parallelized version of Expert.

Aksum[9], developed at the University of Vienna, is based on a source code instrumentation to capture profile-based performance data which is stored in a relational database. The data is then analyzed by a tool implemented in Java that performs an automatic search for performance problems based on JavaPSL, a Java version of ASL.

Hercule[10] is a prototype automatic performance diagnosis system that implements the model-based performance diagnosis approach. It operates as an expert system within the performance measurement and analysis toolkit TAU. Hercule performs an offline search based on model-specific measurements and performance knowledge.

Periscope goes beyond those tools by performing an automatic online search in a distributed fashion via a hierarchy of analysis agents.

3 Architecture

Periscope consists of a frontend and a hierarchy of communication and analysis agents – Fig. 1. Each of the analysis agents, i.e., the nodes of the agent hierarchy, searches autonomously for inefficiencies in a subset of the application processes.

The application processes are linked with a monitoring system that provides the Monitoring Request Interface (MRI). The agents attach to the monitor via sockets. The MRI allows the agent to configure the measurements; to start, halt, and resume the execution; and to retrieve the performance data. The monitor currently only supports summary information, trace data will be available soon.

The application and the agent network are started through the frontend process. It analyzes the set of processors available, determines the mapping of application and analysis

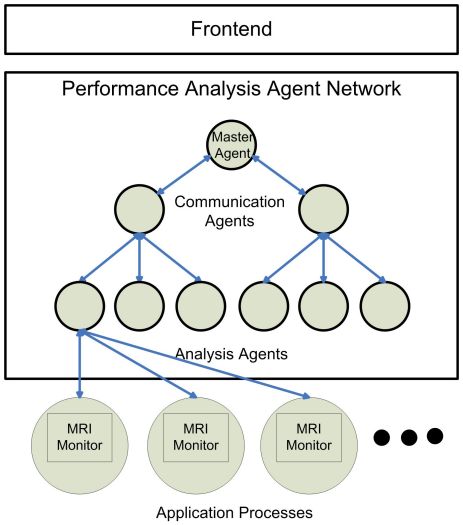


Figure 1. Periscope consists of a frontend and a hierarchy of communication and analysis agents. The analysis agents configure the MRI-based monitors of the application processes and retrieve performance data.

agent processes, and then starts the application. The next step is the startup of the hierarchy of communication agents and of the analysis agents. After startup, a command is propagated down to the analysis agents to start the search. At the end of the local search, the detected performance properties are reported back via the agent hierarchy to the frontend. The communication agents combine similar properties found in their child agents and forward only the combined properties.

To eliminate the need to transfer performance data through the whole machine, the analysis agents are placed near to their application processes. The next sections provide details on how the mapping is computed and implemented on the ALTIX 4700 supercomputer at LRZ, our main target system.

4 ALTIX 4700 Architecture

The ALTIX 4700 at the Leibniz Rechenzentrum (LRZ) in Garching consists of 19 NUMAlink4 interconnected shared memory partitions, each with 256 Intel Itanium 2 Montecito dual core processors and 2TB of memory. Within one partition the connection between processors is laid out as fat-tree.

Peak performance of the system with 9728 cores is 62,3 Teraflop/s. A large range of compilers, tools, libraries and other components is available for development and maintenance supporting OpenMP, various MPI flavours and other parallel programming paradigms.

5 PBS Jobs

The largest part of the ALTIX 4700 is reserved for batch processing. Batch jobs are annotated shell scripts where PBS-directives can be used to specify the required resources. This includes the number of processors, the amount of memory, and a runtime limit. The scripting part sets up the environment and starts the application.

MPI jobs can be run on processors from multiple partitions, while pure OpenMP jobs have to run within a single partitions and thus are limited to 512 threads. Hybrid application have the restriction that all threads of a single MPI process have to run in the same partition.

A job request is sent to the batch queue server using *qsub*. If enough resources are available they are assigned to the job and the script is executed. The environment contains information about the processors that are attributed to the current job in the form of cpusets. For each partition with processors selected for the current job, PBS specifies a cpuset which provides data about the physical CPU number of the allocated processors. Together with topological data of the ALTIX 4700 distance information can be generated.

MPI or hybrid applications are started from within a job script using *mpiexec*. It automatically starts application processes on all allocated processors (potentially spanning several nodes). In addition PBS allows starting remote application via *pbsdsh* which is used to place agents on their destined node.

Exact processor binding is achieved through the *dplace* utility. It can be used in combination with *pbsdsh* as well as with *mpirun*, which replaces the automatic placement via *mpiexec* in the batch script. *dplace* allows to map processes to processors of a single partition. If MPI or hybrid jobs span multiple partitions, the multi-program form of *mpirun* has to be used to start the MPI processes on each partition separately.

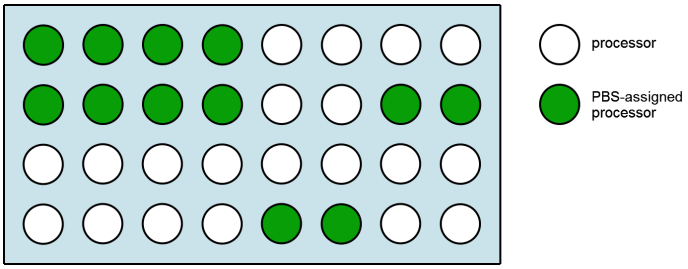
Hybrid programs have special properties both in their startup parameters as well as in the order processes and threads are created. When a process is started, it instantiates a number of helper threads before the actual threads are created. This information becomes relevant when *dplace* is used to force exact placement of both MPI processes and OpenMP threads as it relies on the order of process/thread creation.

6 Application and Agent Distribution

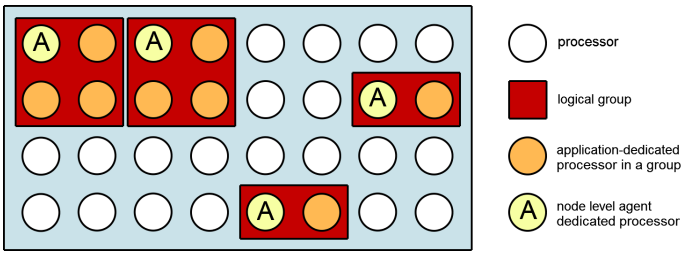
Performance measurement with Periscope is initiated by starting the frontend component. It requires parameters like the target application and the number of MPI processes and OpenMP threads (where their total number has to be smaller than the amount of requested processors so the agents can be started on unoccupied CPUs). Also some performance search configuration values can be overridden by parameters.

First the frontend evaluates data about the environment gathered from the PBS system, cpusets, and machine-dependent resources. It computes the mapping of application processes and analysis agents to the processors allocated to the job and starts the instrumented target application. The initialization code of the monitoring library inserted by source-level instrumentation halts the application processes so that the analysis agents can connect to and control the application processes.

Then a master agent is started with information about the target application components, i.e., processes and threads. It is the top level agent in the hierarchy, a recursive partitioning algorithm starts further communication agents until a single communication



(a) Processors available to a PBS job (12 CPUs requested)



(b) Assignment to analysis agents and program components

Figure 2. The process of aggregating logical clusters. Available processors as depicted in (a) are grouped together depending on their distance values (not shown here). This information is used for the actual program component and analysis agent placement which can be seen in (b).

agent resides on each partition. Then, instead of deepening the hierarchy of communication agents, a number of analysis agents are started and connected to disjoint subsets of the application processes. The analysis agents notify their successful startup by sending a message to the parent in the agent hierarchy. When all children of a communication agent are started this information is handed upwards until the frontend is aware of all agents being ready. Then a start message is sent to all analysis agents to begin with their performance property search.

The agent hierarchy is in the shape of a tree consisting of communication agents in its upper part and analysis agents which are connected to the target application components at the lowest level. Depending on the architecture of the machine the branching factor of the hierarchy is chosen, for the ALTIX 4700 rather few communication agents are required in the upper layers due to the low number of partitions. The amount of analysis agents is determined by the programmer by choosing more processors for the job than application processes or threads are started.

In order to keep the impact of the performance monitoring low, the analysis agents are to be placed close to the application processes or threads, they are responsible for. Thus the available processors are grouped into evenly-sized clusters where one processor slot is reserved for the analysis agent – see Fig. 2. This first step is guided by a specification of the maximum cluster size. The real cluster size is based on the number of CPUs in the partition, the number of desired clusters, and possibly by requirements from using a hybrid programming model.

Generation of the clusters works per partition with processors assigned to the job. To achieve tight grouping, the processor distance information of a partition is extracted from the topology file. Based on nearest processor assignment the clusters are created, after that several optimizing techniques can be deployed to minimize the distance from an analysis agent CPU to the other CPU in the cluster (Section 7).

Based on the computed mapping information the target application is started using *dplace* to enforce exact placement of all components. After the application is fully started up, the same clustering data are used for the analysis agent placing as well: when creating the agent hierarchy, placement strings and identifiers of application processes are passed along to be used at the lowest level to start the analysis agent on its dedicated processor and to connect its to the application processes.

7 Agent Placement Optimization

In order to keep the communication overhead low, two different methods of optimization are performed. First, the logical group tightness is improved by rearranging the mapping of application processes/threads and the analysis agent to processors within a cluster (Section 6). Second, the possibly available information about the message traffic between agents is used to calculate a good distribution of analysis agents over the logical groups.

The task of optimizing the initial clusters is modeled as the minimization of a sum of distances. This sum consists of all distance values from an application CPU to its respective analysis agent CPU. First, the processors in a cluster are treated without distinguishing between their purpose. For each cluster the optimal CPU number for the node-level agent is determined by calculating the sum of distances from one CPU to the others. The configuration that minimizes this sum is used to define the new processor number that will host the node-level agent.

During the second step the node-level agent position is fixed (as determined previously). The algorithm iterates over all pairs of clusters and over all pairs of CPUs selecting one processor number from each cluster. Then the theoretical improvement by swapping the CPU pair is calculated as the sum of the distances from the processors to the node-level agent dedicated processor. If the sum of this measure for both clusters is greater than zero the swap is performed. By this the total measure never increases.

To avoid local minima an exchange can also be performed if the total sum is raised, but only with a certain probability which is high at the beginning of the algorithm and decreases over time. This is similar to the optimization of a function using simulated annealing.

Optimizing the distribution of the analysis agents requires knowledge about the communication traffic. This might be information gathered from previous test runs, or a reasonable estimate. The underlying algorithm works similar to the simulated annealing based approach above. Instead of calculating the total communication costs for all permutations of the analysis agent to processor mapping, starting at an arbitrary configuration, analysis agent CPUs are exchanged (which implies a change of the mapping) if this results in lower total costs. Swap operations that raise the costs are only accepted with a decreasing probability.

In Fig. 3 the effect of the optimization based on simulated annealing is depicted. A configuration of 108 analysis agents is used, but the algorithms have been verified to work correctly and efficiently for much larger amounts of agents. The distance information of

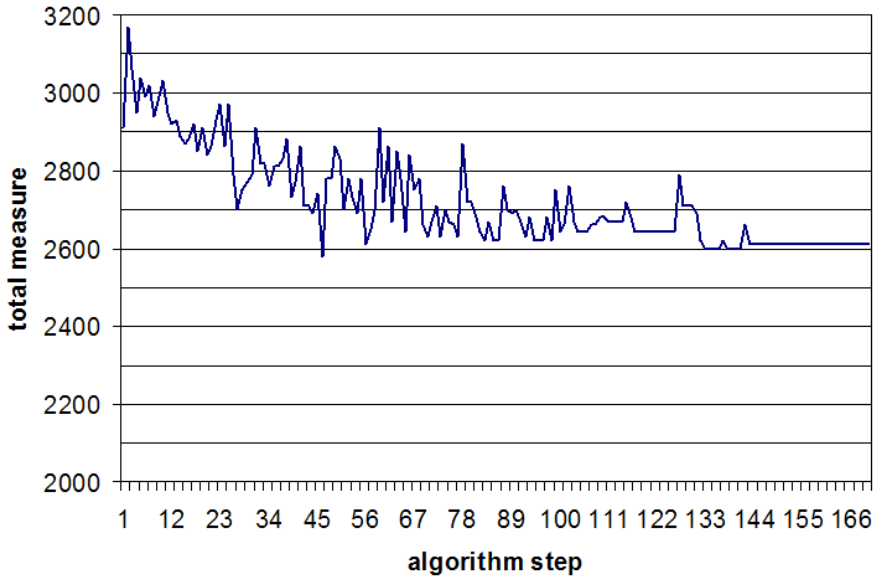


Figure 3. Result of optimization.

the Altix 4700 in form of the topology data is used, whereas random data for the communication intensity have been generated for the purpose of testing. Initially the probability of accepting exchanges that increase the total measure is set to 42% and decreased over time. The algorithm terminates if the probability drops below 0.6%, which results in a total runtime of under one second for up to 512 analysis agents (each shared memory partition of the Altix 4700 consists of 512 processor cores). The size of the decrements of the probability value as well as the final pre-defined probability can be used to adjust the runtime of the algorithm. Compared to testing all permutations of the analysis agent to processor mapping (which already becomes unfeasible for more than 120 agents) the exchange based optimization is fast and versatile.

Both optimization techniques are flexible to be used in various configuration scenarios. On large shared memory systems, like the Altix 4700, the logical grouping is the main target of optimization to reduce the access time to the shared data (ring buffers). The optimization of the communication traffic between the analysis agents is especially relevant on large distributed systems like IBM's Blue Gene.

8 Summary

Periscope is an automatic performance analysis tool for high-end systems. It applies a distributed online search for performance bottlenecks. The search is executed in an incremental fashion by either exploiting the repetitive behaviour of program phases or by restarting the application several times.

This article presented the startup of the parallel application and the agent hierarchy. We use a special algorithm to cluster application processes and analysis agents with respect to the machine's topology. The implementation of the startup on the ALTIX supercomputer at LRZ is based on the multi-program version of the *mpirun* command and the *dplace* utility for process to processor mapping.

In the current implementation, it is the responsibility of the programmer to choose the amount of resources for the analysis agent hierarchy. In future, we will work on an automatic selection based on historical information.

References

1. M. Gerndt, K. Fürlinger, and E. Kereku, *Advanced Techniques for Performance Analysis*, Parallel Computing: Current&Future Issues of High-End Computing (Proceedings of the International Conference ParCo 2005), Eds: G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, E. Zapata, NIC Series vol. **33**, ISBN 3-00-017352-8, pp. 15–26, (2006).
2. K. Fürlinger, *Scalable Automated Online Performance Analysis of Applications using Performance Properties*, PhD thesis, Technische Universität München, (2006).
3. E. Kereku, *Automatic Performance Analysis for Memory Hierarchies and Threaded Applications on SMP Systems*, PhD thesis, Technische Universität München, (2006).
4. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, *The Paradyn Parallel Performance Measurement Tool*, IEEE Computer, **28**, 37–46, (1995).
5. Philip C. Roth, Dorian C. Arnold, and Barton P. Miller, *MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools*, in: Proc. 2003 Conference on Supercomputing (SC 2003), Phoenix, Arizona, USA, (2003).
6. P. C. Roth and B. P. Miller, *The Distributed Performance Consultant and the Sub-Graph Folding Algorithm: On-line Automated Performance Diagnosis on Thousands of Processes*, in: Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06), (2006).
7. F. Wolf and B. Mohr, *Automatic Performance Analysis of Hybrid MPI/OpenMP Applications*, in: 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 13–22, (2003).
8. M. Geimer, F. Wolf, B. J. N. Wylie and B. Mohr, *Scalable Parallel Trace-Based Performance Analysis*, in: Proc. 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006), pp. 303–312, Bonn, Germany, (2006).
9. T. Fahringer and C. Seragiotto, *Aksum: A Performance Analysis Tool for Parallel and Distributed Applications*, in: Performance Analysis and Grid Computing, Eds. V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller, Kluwer Academic Publisher, ISBN 1-4020-7693-2, pp. 189–210, (2003).
10. L. Li, A. D. Malony and K. Huck, *Model-Based Relative Performance Diagnosis of Wavefront Parallel Computations*, in: Proc. 2nd International Conference on High Performance Computing and Communications 2006 (HPCC 06), M. Gerndt and D. Kranzlmüller, (Eds.), vol. **4208** of *LNCS*, pp. 200–209, (Springer, 2006).

Visualizing Parallel Functional Program Runs: Case Studies with the Eden Trace Viewer

Jost Berthold and Rita Loogen

Philipps-Universität Marburg, Fachbereich Mathematik und Informatik
Hans Meerwein Straße, D-35032 Marburg, Germany
E-mail: {*berthold, loogen*}*@informatik.uni-marburg.de*

This paper describes case studies with the Eden Trace Viewer (*EdenTV*), a post-mortem trace analysis and visualisation tool for the parallel functional language Eden. It shows program executions in terms of Eden's abstract units of computation instead of providing a machine-oriented low level view like common tools for parallelism analysis do. We show how typical inefficiencies in parallel functional programs due to delayed evaluation, or unbalanced workload can be detected by analysing the trace visualisations.

1 Introduction

Parallel functional languages like e.g. the parallel Haskell extension Eden[1] offer a highly abstract view of parallelism. While less error-prone, this sometimes hampers program optimisations, because it is very difficult to know or guess what really happens *inside* the parallel machine during program execution. A considerable number of profiling toolkits[2,3,4] monitor parallel program executions, but are less appropriate for high-level languages. These are implemented on top of a parallel runtime system (PRTS) which implements a parallel virtual machine. Standard profiling tools like `xpvm`[3] monitor the activity of the virtual processing elements (PEs or *machines*, usually mapped one-to-one to the physical ones), the message traffic between these PEs, and the behaviour of the underlying middleware like PVM[a] or MPI[b], instead of the program execution on top of the PRTS.

The Eden trace viewer tool (EdenTV) presented in this paper visualises the execution of parallel functional Eden programs at such a higher level of abstraction. EdenTV shows the activity of the *Eden* threads and processes, their mapping to the machines, stream communication between Eden processes, garbage collection phases, and the process generation tree, i.e. information specific to the Eden programming model. This supports the programmer's understanding of the parallel behaviour of an Eden program.

The Eden PRTS is instrumented with special trace generation commands activated by a runtime option. Eden programs themselves remain unchanged. Parts of the well-established Pablo Toolkit[5] and its *Self-Defining Data Format* (SDDF) are used for trace generation. EdenTV loads trace files after program execution and produces timeline diagrams for the computational units of Eden, i.e. threads, processes, and machines.

The aim of this paper is to show benefits of our high-level EdenTV profiling tool. Using a few simple case studies, we explain how the trace visualisations may help detect typical inefficiencies in parallel functional programs which may be due to delayed evaluation or bad load balancing. Section 2 sketches Eden's language concepts and its PRTS. Section 3

[a] `http://www.csm.ornl.gov/pvm/pvm_home.html`
[b] `http://www.mpi-forum.org`

shortly describes EdenTV. The central Section 4 discusses typical weaknesses of program executions that can be detected with EdenTV and how to eliminate them. Section 5 points at related work, and, at last, Section 6 concludes and indicates future work.

2 Eden

Eden[1] extends the functional language Haskell[c] with syntactic constructs for *explicitly* defining and creating parallel processes. The programmer has direct control over process granularity, data distribution and communication topology, but does not have to manage synchronisation and data exchange between processes. The latter are performed by the parallel runtime system through implicit communication channels, transparent to the programmer.

Coordination Constructs. The essential two coordination constructs of Eden are process abstraction and instantiation:

```
process :: (Trans a, Trans b) => (a -> b)    -> Process a b
( # )   :: (Trans a, Trans b) => Process a b -> a -> b
```

The function `process` embeds functions of type `a->b` into *process abstractions* of type `Process a b` where the context `(Trans a, Trans b)` states that both `a` and `b` must be types belonging to the `Trans` class of transmissible values.[d]

Evaluation of an expression `(process funct) # arg` leads to the creation of a new process for evaluating the application of the function `funct` to the argument `arg`. The argument is evaluated by new concurrent threads in the parent process and sent to the new child process, which, in turn, fully evaluates and sends back the result of the function application. Both are using implicit 1:1 *communication channels* established between child and parent process on process instantiation.

The type class `Trans` provides overloaded communication functions for *lists*, which are transmitted as streams, element by element, and for *tuples*, which are evaluated componentwise by concurrent threads in the same process. An Eden process can thus contain a variable number of threads during its lifetime.

As communication channels are normally connections between parent and child process, the communication topologies are hierarchical. In order to create other topologies, Eden provides additional language constructs to create channels dynamically, which is another source of concurrent threads in the sender process. Besides, Eden supports many-to-one communication by a nondeterministic merge function.

Parallel Runtime System. Eden is implemented on the basis of the Glasgow Haskell Compiler GHC[e], a mature and efficient Haskell implementation. While the compiler frontend is almost unchanged, the backend is extended with a *parallel* runtime system (PRTS)[6]. The PRTS uses suitable middleware (currently PVM or MPI) to manage parallel execution.
Concurrent threads are the basic unit for the implementation, so the central task for profiling is to keep track of their execution. Threads are scheduled *round-robin* and run through the straightforward state transitions shown in Fig. 1.

[c] `http://www.haskell.org`
[d] Type classes in Haskell provide a structured way to define overloaded functions. `Trans` provides implicitly used communication functions.
[e] `http://www.haskell.org/ghc`

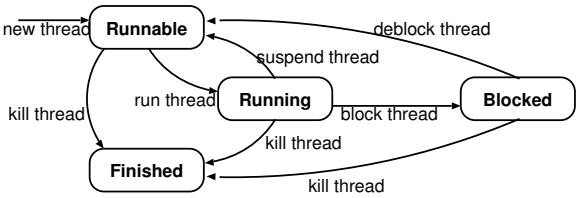


Figure 1. Thread State Transitions

An Eden process, as a purely conceptual unit, consists of a number of concurrent threads which share a common graph heap (as opposed to processes, which communicate via channels). The Eden PRTS does not support the migration of threads or processes to other machines during execution, so every thread is located on exactly one machine during its lifetime.

3 EdenTV — The Eden Trace Viewer

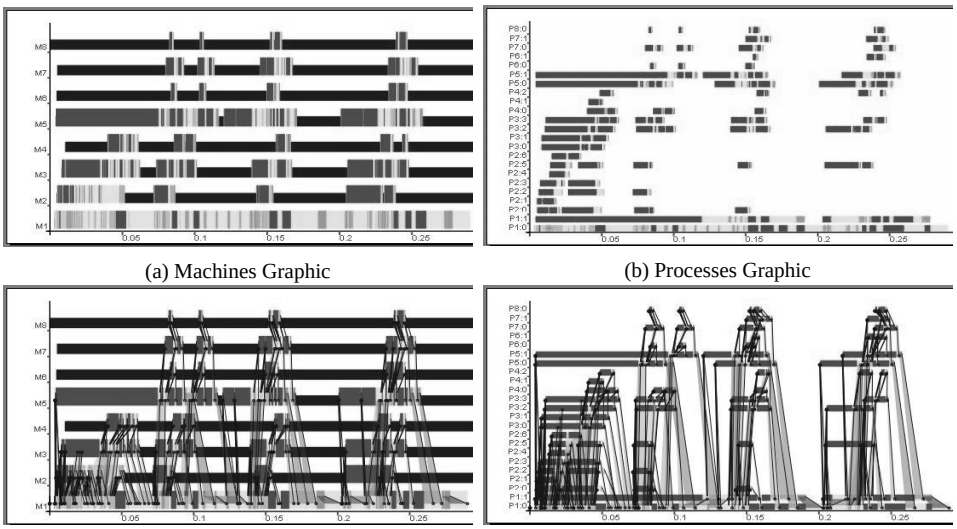
EdenTV enables a *post-mortem analysis* of program executions at the level of the PRTS. The steps of profiling are *trace generation* and *trace representation*, separated as much as possible in EdenTV, so that single parts can easily be maintained and modified for other purposes. Two versions of the trace representation tool are currently available. A Java implementation has been developed by Pablo Roldán Gómez[7]. Björn Struckmeier[8] did a re-implementation in Haskell which provides additional features.

Trace Generation. To profile the execution of an Eden program, we collect information about the behaviour of machines, processes, threads and messages by writing selected *events* into a trace file. These trace events, shown in Fig. 2, indicate the creation or a state transition of a computational unit. Trace events are emitted from the PRTS, using the Pablo *Trace Capture Library*[5] and its portable and machine-readable "Self-Defining Data Format". Eden programs need not be changed to obtain the traces.

Start Machine	End Machine
New Process	Kill Process
New Thread	Kill Thread
Run Thread	Suspend Thread
Block Thread	Deblock Thread
Send Message	Receive Message

Figure 2. Trace Events

Trace Representation. In the timeline diagrams generated by EdenTV, machines, processes, and threads are represented by horizontal bars, with time on the x axis. EdenTV offers separate diagrams for *machines*, *processes*, and *threads*. The machines diagrams correspond to the view of profiling tools observing the parallel machine execution. Fig. 3 shows examples of the machines and processes diagrams for a parallel divide-and-conquer program with limited recursion-depth. The trace has been generated on 8 Linux workstations connected via fast Ethernet. The diagram lines have segments in different colours, which indicate the activities of the respective logical unit in a period during the execution. As explained in Fig. 3(d), thread states can be directly concluded from the emitted events. Machine and process states are assigned following a fundamental equation for the thread count inside one process or machine. The example diagrams in Fig. 3 show that the program has been executed on 8 machines (virtual PEs). While there is some continuous activity on machine 1 (the bottom line) where the main program is started, machines 6 to 8 (the upper three lines) are idle most of the time. The corresponding processes graphic (see Fig. 3(a)) reveals that several Eden processes have been allocated on each machine. The diagrams show that the workload on the parallel machines was low — there were only



(a) Machines Graphic

(b) Processes Graphic

(c) . . . with Messages Overlay

Thread Count:	$0 \leq$ Runnable Threads + Blocked Threads $\leq$ Total Threads				
Condition	**Machine**	**Process**	**Thread**	Black/White	Colour
Total Threads $= 0$	Idle	n/a	n/a	**smaller bar**	**Blue**
Blocked + Runnable $<$ Total	Running	Running	Running	**dark grey**	**Green**
Runnable Threads > 0	System Time	Runnable	Runnable	**light grey**	**Yellow**
Runnable Threads $= 0$	Blocked	Blocked	Blocked	black	Red

(d) Explanation of Colour Codes

Figure 3. Examples of EdenTV Diagrams and Colour Codes Table

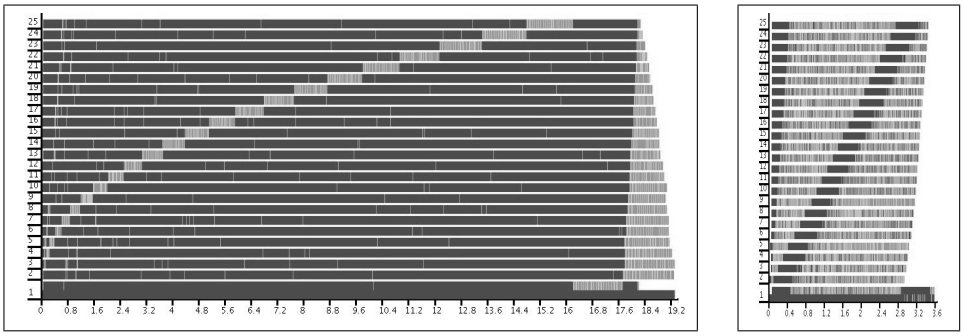
small periods where threads were running. Messages between processes or machines can optionally be shown by arrows which start from the sending unit line and point at the receiving unit line (see Fig. 3(c)). The diagrams can be *zoomed* in order to get a closer view on the activities at critical points during the execution.

Additional Features. EdenTV provides additional information about the program run, e.g. a summary of the messages sent and received by processes and machines (on the right for the trace in Fig. 3), stream communication is indicated by shading the area between the first and the last message of a stream (see Fig. 3(c)), garbage collection phases and memory consumption can be shown in the activity diagrams, and the process generation tree can be drawn.

Machine	Runtime (sec)	Processes	Messages sent	Messages recvd
1	0.287197	4	6132	6166
2	0.361365	18	1224	1206
		⋮		
8	0.362850	6	408	402
Total	0.371875	66	14784	14784

4 Case Studies: Tuning Eden Programs with EdenTV

Lazy Evaluation vs. Parallelism. When using a lazy computation language, a crucial issue is to start the evaluation of needed subexpressions early enough and to fully evaluate them



Runtime: 19.37 sec. without additional demand control

Runtime: 3.62 sec. with additional demand control

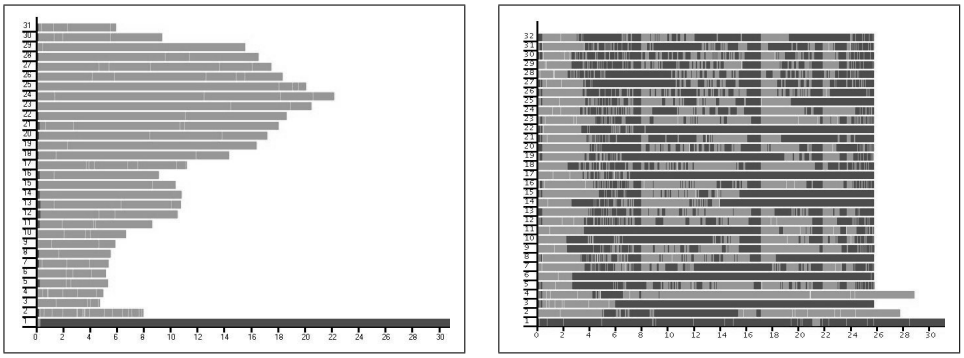
Figure 4. Warshall-Algorithm (500 node graph)

for later use. The basic choice to either evaluate a final result to weak head normal form (WHNF) or completely (to normal form (NF)) sometimes does not offer enough control to optimise an algorithm. Strategies[9] forcing additional evaluations must then be applied to certain sub-results. On the sparse basis of runtime measurements, such an optimisation would be rather cumbersome. The EdenTV, accompanied by code inspection, makes such inefficiencies obvious, as in the following example.

Example: **(Warshall's algorithm)** This algorithm computes shortest paths for all nodes of a graph from the adjacency matrix. We optimise a parallel implementation of this algorithm with a ring of processes. Each process computes the minimum distances from one node to every other node. Initialised with a row of the adjacency matrix, the direct distances are updated whenever a path of shorter distance via another node exists. All distance rows are continuously updated, and traverse the whole ring for one round.

The trace visualisations in Fig. 4 show EdenTV's *Processes* view for two versions of the program on a Beowulf cluster, with an input graph of 500 nodes (aggregated on 25 processors). The programs differ by a single line which introduces additional demand for an early update on the local row. The first program version (without demand control) shows bad performance. The trace visualization clearly shows that the first phase of the algorithm is virtually sequential. A period of increasing activity traverses the ring, between long blocked (black) periods. Only the second phase, subsequent updates after sending the local row, runs in parallel on all machines. The cause is that when receiving a row, the local row is updated, but demand for this update only occurs at the time when the local row is sent through the ring. The second version enforces the evaluation of the temporary local row each time another row is received, which dramatically improves runtime. We still see the impact of the data dependence, leading to a short wait phase passing through the ring, but the optimised version shows good speedup and load balance. ◁

Irregularity and Cost of Load Balancing. Parallel skeletons (higher-order functions implementing common patterns of parallel computation[10]) provide an easy parallelization of a program. The higher-order function `map` applies a function to all elements of a list. These computations do not depend on each other, thus a number of worker processes can compute results in parallel. Parallel `map` variants either distribute the list elements statically (which is called a *farm* skeleton), or dynamically, by using a feedback from worker results to



(a) Farm skeleton, irregular static distribution

(b) Non-hierarchical workpool skeleton

Figure 5. *Processes* diagrams for different parallelizations of the Mandelbrot program

worker inputs (called a *workpool*). However, when the tasks expose an irregular complexity, the farm may suffer from an uneven load distribution, and the machine with the most complex tasks will dominate the runtime of the parallel computation. The workpool can adapt to the current load and speed of the different machines, but the dynamic distribution necessarily has more overhead than a static one.

Example: **(Mandelbrot)** We compare different work distribution schemes using a program which generates Mandelbrot Set visualizations by applying a uniform, but potentially irregular computation to a set of coordinates. In the traces shown in Fig. 5, the pixel rows are computed in parallel, using either a farm or a workpool skeleton. Both programs have run on a Beowulf cluster with 32 processors, computing 5000 rows of 5000 pixels. The trace of the farm skeleton in Fig. 5(a) shows that the workers have uneven workload. The blockwise row distribution even reflects the silhouette of the Mandelbrot graphic (bottom-to-top) in the runtimes of the worker processes.

The trace of the workpool skeleton in Fig. 5(b) shows how the master process on machine 1 has to serve new tasks to 31 machines, and collect all results. The workers (2 to 32) spent a lot of time in blocked state, waiting for new work assigned by the heavily loaded master node. All workers stop at the same time, i.e. the system is well synchronized, but the single master is a severe bottleneck. ◁

Nesting and Process Placement. In Eden's PRTS, processes are placed either round-robin on available machines or randomly, but the implementation also supports explicit placement on particular machines. This can e.g. be used to optimally place processes in nested skeleton calls[11].

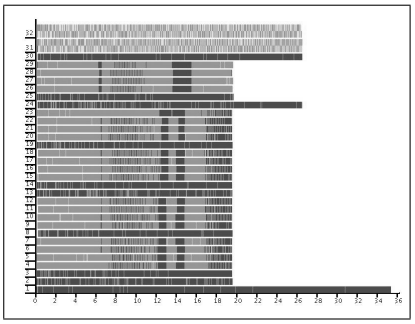


Figure 6. Hierarchical 3-level workpool

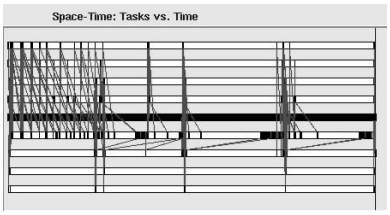
Example: **(Hierarchical Workpool)** The workpool master process can be relieved using a "hierarchy of workpools" with several stages of task distribution. Fig. 6 shows the trace of a hierarchical workpool in three levels: a toplevel master, three submasters and two subsubmasters per submaster, each con-

trolling four workers. Thus, 10 of the 32 PEs are reserved as (sub/subsub)master processes, only 24 worker processes are allocated, and the four worker processes of the final group share two PEs. Nevertheless, the trace shows much better workload distribution and worker activity than in the non-hierarchical system. Only the two machines with two worker processes need more time, which is due to the higher initial workload caused by the identical prefetch for all worker processes.

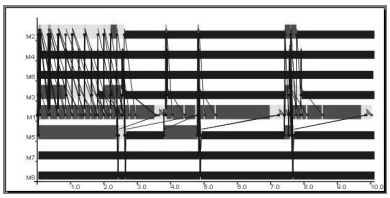
Collecting Results. All Mandelbrot traces suffer from long sequential end phases (omitted in Fig. 5) which dominate the runtimes of the various schemes. An inspection of the message traffic reveals that all processes return their results to the root process (bottom line) which merges them at the end of the computation. It can be observed that the end phase is shorter within the hierarchical scheme, because the merging is done level-wise. Nevertheless, it is still too long in comparison with the parallel execution times of the worker processes. ◁

5 Related Work

A rather simple (but insufficient) way to obtain information about a program's parallelism would be to trace the behaviour of the communication subsystem. Tracing PVM-specific and user-defined actions is possible and visualization can be carried out by `xpvm`[3]. Yet, PVM-tracing yields only information about virtual PEs and concrete messages between them. Internal buffering, processes, and threads in the PRTS remain invisible, unless user-defined events are used.



(a) `xpvm` Space-Time Graphic



(b) EdenTV Machines Graphic

Figure 7. `xpvm` vs EdenTV

As a comparison, we show `xpvm` execution traces of our first trace examples. The space-time graphic of `xpvm` shown in Fig. 7(a) corresponds to the EdenTV machine view of the same run in Fig. 7(b), if the machines are ordered according to the `xpvm` order. The monitoring by `xpvm` extremely slows down the whole execution. The runtime increases from 0.37 seconds to 10.11 seconds. A reason for this unacceptable tracing overhead might be that PVM uses its own communication system for gathering trace information.

Comparable to the tracing included in `xpvm`, many efforts have been made in the past to create standard tools for trace analysis and representation (e.g. Pablo Analysis GUI[5], the ParaGraph Suite[12], or the Vampir system[4], to mention only a few essential projects). These tools have interesting features EdenTV does not yet include, like stream-based online trace analysis, execution replay, and a wide range of standard diagrams. The aim of EdenTV, however, is a specific visualization of logical Eden units, which needed a more customized solution. The EdenTV diagrams have been inspired by the per-processor view of the Granularity Simulator GranSim[13], a profiler for Glasgow parallel Haskell (GpH)[9], which, however, does not trace any kind of communication due to the different language concept.

6 Conclusions

The Eden Trace Viewer is a combination of an instrumented runtime system to generate trace files, and an interactive GUI to represent them in interactive diagrams. We have shown case studies exposing typical traps of parallelism like missing demand or imbalance in computation or communication, which can be identified using EdenTV. Load balancing issues and communication structures can be analysed and controlled. Essential runtime improvements could be achieved for our example programs.

Acknowledgements. The authors thank Pablo Roldán Gómez and Björn Struckmeier for their contributions to the Eden Trace Viewer project.

References

1. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí, *Parallel Functional Programming in Eden*, Journal of Functional Programming, **15**, 431–475, (2005).
2. I. Foster, *Designing and Building Parallel Programs*, Chapter 9, (Addison-Wesley, 1995). http://www.mcs.anl.gov/dbpp/
3. J. A. Kohl and G. A. Geist, *The PVM 3.4 tracing facility and XPVM 1.1*, in: Proc. HICSS-29, pp. 290–299, (IEEE Computer Society Press, 1996).
4. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and analysis of MPI resources*, Supercomputer, **12**, January (1996).
5. D. A. Reed, R. D. Olson, R. A. Aydt *et al.*, *Scalable performance environments for parallel systems*, in: Sixth Distributed Memory Computing Conference Proceedings (6th DMCC'91), pp. 562–569, Portland, OR, (IEEE, 1991).
6. J. Berthold and R. Loogen, *Parallel Coordination Made Explicit in a Functional Setting*, in: IFL'06, Selected Papers, LNCS 4449, pp. 73–90, (Springer, 2007).
7. P. Roldán Gómez, *Eden Trace Viewer: Ein Werkzeug zur Visualisierung paralleler funktionaler Programme*, Masters thesis, Philipps-Universität Marburg, Germany, (2004, in German).
8. B. Struckmeier, *Implementierung eines Werkzeugs zur Visualisierung und Analyse paralleler Programmläufe in Haskell*, Masters thesis, Philipps-Universität Marburg, Germany, (2006, in German).
9. P. W. Trinder, K. Hammond, H-W. Loidl, and S. L. Peyton Jones, *Algorithm + Strategy = Parallelism*, J. Functional Programming, **8**, January, (1998).
10. F. A. Rabhi and S. Gorlatch, eds., *Patterns and Skeletons for Parallel and Distributed Computing*, (Springer, 2002).
11. J. Berthold, M. Dieterle, R. Loogen, and S. Priebe, *Hierarchical Master-Worker Skeletons*, in: Practical Aspects of Declarative Languages (PADL 08), (Springer LNCS, 2008).
12. M. T. Heath and J. A. Etheridge, *Visualizing the performance of parallel programs*, IEEE Software, **8**, (1991).
13. H. W. Loidl, *GranSim User's Guide*, Technical report, University of Glasgow, Department of Computer Science, (1996).

Automatic Phase Detection of MPI Applications

Marc Casas, Rosa M. Badia, and Jesús Labarta

Barcelona Supercomputing Center (BSC)
Technical University of Catalonia (UPC)
Campus Nord, Modul C6, Jordi Girona, 1-3, 08034 Barcelona
E-mail: {mcasas, rosab}@ac.upc.edu, jesus@cepba.upc.es

In the recent years, strong efforts have been dedicated on the study of applications' execution phases. In this paper, we show an approach focused on the automatic detection of the phases on MPI applications' execution. This detection is based on Wavelet Analysis, which provides a meaningful and fast methodology. Information derived from phase detection is very useful to study the performance of the application. To illustrate this importance, we show a study of the scalability of several applications based on phase detection.

1 Motivation and Goal

The development and consolidation of supercomputing as a fundamental tool to do research in topics such as genomics or meteorology has prompted the release of a huge set of scientific applications (CPMD[13] , WRF-NMM[11]....). The study of the execution of those applications is fundamental in order to optimize the code and, consequently, accelerate the research in the scientific topic which the application comes from.

However, the current scientific applications are executed using thousands of processors. For that reason, huge tracefiles [a] (more than 10 GB) are generated. In that context, the traditional visualization tools (Paraver[15], Vampir[4,5], ...) become useless or, at the most, they can be used after filtering or cutting the tracefiles. Another solution is tracing again the application limiting the number of events of the tracefile. In any case, the research on new methodologies that allow the analyst to make an automatic analysis or, at least, to automatically obtain partial information based on the execution of a given application is justified.

Furthermore, the first goal of this paper is to use signal processing techniques (wavelet transform) in order to provide a very fast automatic detection of the phases of MPI applications' execution. The criterion of such signal processing techniques in order to perform the phase detection is to separate regions according to their frequency behaviour, i. e., a region with a small iteration which is repeated many times will we separated from another region with no periodic behaviour. The second goal of this work is to use the information derived from signal processing techniques to acquire remarkable conclusions about the scalability if the applications and how could be improved.

An important point of our work is that it enables the analyst to acquire some information without requiring any knowledge of the source code of the application. Several authors argue that this is a minor point because the analyst can acquire these information directly looking to the source code. Others say that if a parallel application has to be optimized, the analyst will need anyhow the source code, and knowledge about it, in order to implement

[a] time stamped sequence of events

the optimizations. These two arguments do not take into account the emerging reality of the academic and industrial supercomputing centres. These centres have to deal with many application codes and the analyst who elaborates the performance reports is not, in many cases, the same person who will optimize the code. In that context, our tool makes possible obtaining a very fast report of the general characteristics of the application's execution and, for that reason, makes easier the work of the performance analyst. Obviously, in the final stages of the performance optimization process, strong knowledge about the source code will be needed.

There are other approaches to automatically detect program phases. In[6,7] the authors perform an analysis based on Basic Bloc Vectors (BBV), which are sections of code. The frequency with which that sections of code are executed is the metric to compare different sections of the application's execution. In[8] the authors use subroutines to identify program phases. If the time spent in a subroutine is greater than a threshold, it is considered an important phase. There is another approach in[9] where the authors define a program phase as the set of instructions touched in a given interval of time. There are many differences between these approaches and our. First, in this paper we analyze the execution of the application using signal processing techniques. Second, the approaches we talked above are focused on solve problems such us reduction of simulation time or study of energy consumption. Our approach is focused on the study of the performance of the application. This paper is organized as follows: First, in Section 2 there is a description of the signal that we work with and an explanation of Wavelet Analysis. After that, in Section 3 there is a characterization of execution phases of high performance computing applications and an example. In Section 4 we show results obtained from the study of several applications. Finally, Section 5 is dedicated to conclusions.

2 Signal Processing

2.1 Signal Definition

Our approach is based on tracefile techniques, since such techniques allow a very detailed study of the variations on space (set of processes) and time that could affect notably the performance of the application. On our work, we will use OMPItrace tracing package[10] during the process of obtaining tracefiles. In order to apply signal processing techniques, we derive a signal from the data contained in the tracefile. More exactly, the signal we will work with will be the sum of durations of running bursts. This signal is generated taking into account the running states contained in the tracefile. For all of the threads, we generate a signal which contains, for each instant of time t, the duration of the running burst that is being executed in this moment t. If there is no running burst in that moment, the value assigned to t is 0. Finally, the signals generated for each of the threads are added. Therefore, we obtain the sum of durations of running bursts. In Fig. 1 there is an example in which a sum of durations of running bursts signal is generated.

2.2 Overview of Wavelet Transform

Wavelet transform is a well known signal processing tool used in a wide and heterogeneous range of topics. Specifically, this transform and the techniques based on it have

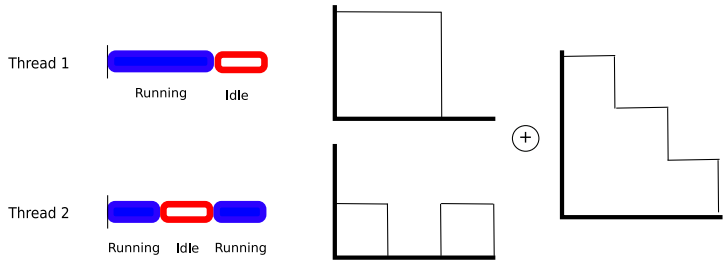


Figure 1. Generation of sum of durations of running bursts signal from a simple running burst distribution

become useful for relevant issues such as elimination of signal noises or image compression[3]. However, they have not been widely used in the study of execution of applications. One of the few examples of its utilization in this topic is in10. When we talk about signal processing, Fourier transform is the most famous technique to study the signal from the frequency domain point of view. Wavelet and Fourier transforms are related in the fact that they analyze the function in its frequency domain. However, there are two key differences between Fourier and Wavelet transforms: First, Fourier transform uses sinusoid functions to study the input signal while Wavelet transform uses a pair of analysis functions: the scaling function and the wavelet function. The first interprets low-frequency information and the second captures high-frequency information. The second difference is prompted by the fact that Fourier analysis of a signal only gives information about the frequencies while wavelet analysis, which gives information about the frequencies that appear in the signal, determines also where these frequencies are located.

The characteristics of the Wavelet transform we have explained make it specially appropriate for detect execution phases on MPI applications for several reasons: First, because the time-frequency localization is fundamental in order to overcome the non-stationary behaviour of the programs. If a given application changes its execution behaviour suddenly and starts to perform high-frequency iterations, Wavelet transform will detect this change.

Second, wavelet analysis allows one to choose the analysis function that best captures the variations shown by signals generated from tracefiles of MPI applications. We use the Haar Wavelet Function[2] because it is attuned to the discontinuities that often appear in the signals we work with.

To provide a more precise understanding of wavelet transform and a concrete explanation of the role of scaling and wavelet functions, we show the mathematical formulation of wavelet transform. First, consider a division of the temporal domain of the signal $x(t)$ in 2^N partitions:

$$a_k = \int_{-\infty}^{\infty} 2^{\frac{j_0}{2}} \phi(2^{j_0}t - k)x(t)dt \text{ where } 0 \leq k < 2^{j_0} \tag{2.1}$$

$$b_{j,k} = \int_{-\infty}^{\infty} 2^{j} \psi(2^{j}t - k)x(t)dt \text{ where } 0 \leq k < 2^{j} \text{ and } j_0 \leq j < N \tag{2.2}$$

The scaling function, $\phi(t)$ is used to obtain the coefficients a_k. We have one of these coefficients for everyone of the time regions we have divided the domain of the original

signal, $x(t)$. Their value is an average of $x(t)$ over a window of size 2^{j_0}. The value of j_0 allows us to define a maximum granularity of wavelet analysis. On the other hand, coefficients $b_{j,k}$ capture higher frequency behaviours. They are indexed by two variables: j, which corresponds to frequency, and k, which corresponds to time.

From the previous presentation of wavelet analysis, we can derive an interesting feature of it. As we increase the value of j, we obtain more coefficients, 2^j, for the same time domain, that is, the accuracy of the time resolution increases too. However, the accuracy of the frequency value decreases since the range of frequencies we are trying to detect increases as we increase j. In summary, this multi-resolution analysis provides a good time resolution at high frequencies, and good frequency resolution at low frequencies. For this reason, our analysis will be focused on the detection of phases of high frequency behaviour during the application's execution. Wavelet analysis warrants a good accuracy in this kind of detection.

Finally, from the computational point of view, the Discrete Wavelet Transform (DWT) has an implementation called FastWavelet Transform(FWT)[1]. This implementation has an algorithmic complexity of O(n). This property allows allow us to find different execution phases in a tracefile with only O(n) operations.

3 Execution Phases

Typically, high performance computing applications have severalwell defined phases. First of all, the execution has an initialization phase characterized by initial communications in order to assign values to the variables, domain decomposition, etc... This phase has not a periodic structure, that is, the impact of high frequencies is negligible on it.

Second, HPC applications have an intensive computation phase. It has typically a periodic behaviour prompted by the usual application's structure, which performs iterations on space (physical domain of the problem) and time interleavedwith communication (point to point or collectives). It is in this second phase where the most parallelizable code is being executed, since there are no input/output operations. Finally, there is a final phase phase. It consists, mainly, in output data operations. As well as the initialization, this phase has not a periodic structure.

These three phases constitute a common execution of HPC application. Wavelet transform is specially indicated to detect them because the differences between the three phases have an impact in the frequency domain. The good accuracy of Wavelet analysis detecting the places where high frequencies occur enables it to detect accurately the localization of the computation phase.

However, there are non-typical applications which do not follow the basic HPC application structure. In these cases, the Wavelet analysis will identify the ad-hoc structure if the differences between phases could be expressed from the frequency domain point of view, that is, if the phases could be characterized by occurrence of high-frequencies or low-frequencies.

3.1 Example

In this section, we discuss in detail results obtained applying wavelet analysis to a real application in order to give an overview of the information provided by this analysis. The

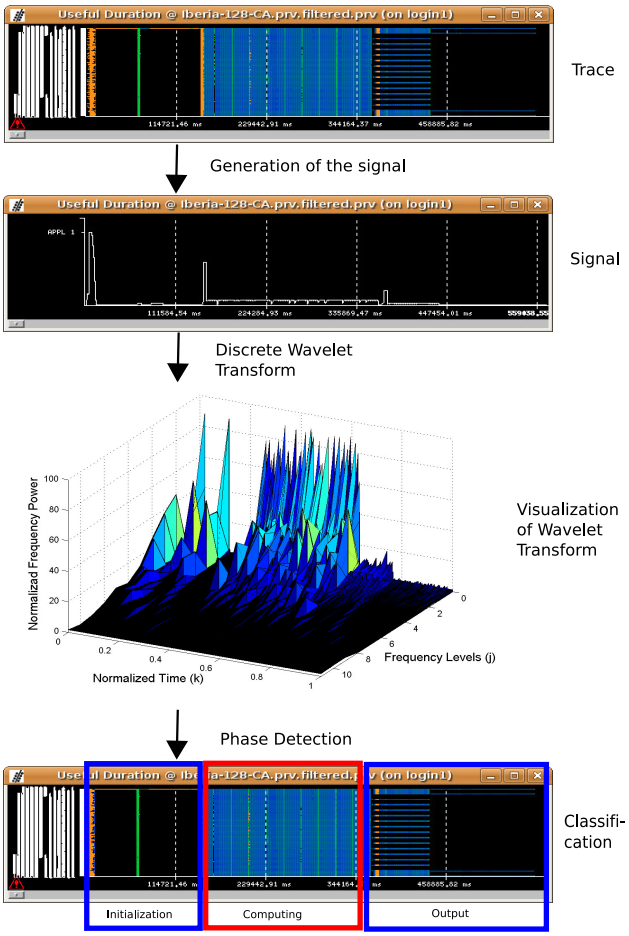


Figure 2. Example of Wavelet Analysis applied on WRF-NMM tracefile. First, running bursts distribution generated from the tracefile. After, sum of durations of running bursts signal generated from the distribution. In the third cell, numerical results of the wavelet analysis and, finally, phase detection derived from wavelet analysis.

application selected is WRF-NMM[11] over a 4 km grid of the Iberian Peninsula with 128 processors.

First of all, application has been traced. In Fig. 2 we see, on the top, a visualization of the running burst distribution contained in the tracefile. It shows clearly the typical structure derived from the execution of a HPC application: First, an initialization phase with non-periodic structure, second, a computation phase with strong periodic behaviour and, finally, an output phase. In Fig. 2, we also show the signal that has been generated from the tracefile using the methodology explained in Section 2.1. In this signal it is possible again to see the typical structure of HPC application. Wavelet analysis has been applied to the signal in order to detect execution phases. Numerical output of wavelet analysis is shown at the third picture of Fig. 2: First, in the x-axis, we draw the normalized

time of the execution of the application. In the y-axis we show the Frequency Levels. Each of these levels corresponds to a set of $b_{j,k}$ coefficients defined in 2.1 with the same j. In the Level 0 axe we show the coefficients which are sensitive to high frequencies. As the Level number increases, the sensitivity to high frequencies decreases and the sensitivity to low frequencies grows. Finally, in the Level 10 axe, we show the value of the coefficient a_k, defined in 2.2. In order to give a more intuitive visualization of the sense of the results, the values of the coefficients have been normalized by the window's size used in every level.

In the fourth cell of Fig. 2 we show the phase classification, based on Wavelet analysis results, that automatically our system has made. This classification corresponds to the division that we can do intuitively. It has been done without any knowledge of the source code and without the need of a visualization of the tracefile.

The most important thing to take into account from Fig. 2 is that wavelet analysis has been able to detect the regions of the tracefile where high frequencies occur, that is, regions where strong periodic behaviour characterizes the execution of the application.

4 Results

We have applied Wavelet Analysis to the Weather Research and Forecasting system. We have analyzed the Nonhydrostatic Mesoscale Model (NMM)[11] core and the Applied Research Weather (ARW)[12] one. Each core has been executed with 128, 256 and 512 processors. These executions have been made using, first, a 4 kilometers grid of the Iberian Peninsula and, second, a 12 kilometers grid of Europe. In Table 1 the results generated from WRF-NMM application for Iberia are shown. It is clear that the scalability of the application is severely affected by the fact that the initialization phase not only does not scale well, it also increases the time span required to be carried out. On the other hand, computation phase and output one show a better behaviour. They are far, however, to reach lineal scalability.

In Table 1 the results obtained from WRF-NMM application for Europe are shown, that is, the same code we have used to generate results from WRF-NMM-Iberia but different input data. The behaviour of the execution is similar to the Iberia case, that is, time span required to carry out the initialization does not decrease as the number of processor increases. On the other hand, computation phase and output phase show a decreasing span of time but far to lineal.

In Table 2 the results generated from WRF-ARW application for Iberia are shown, that is, the same input data used to generate Table 1 but different code. We see as WRF-ARW is faster than WRF-NMM but shows a worst scalability. This fact is explained by the poor scalability shown by the computation phase of the WRF-ARW core. This is not a trivial issue: Indicates that the resolution of the same physical problem could be solved faster by WRF-ARW if we have several hundreds of processors but WRF-NMM could be better if we can perform the execution on several thousands of processors. We highlight the fact that all this information is generated automatically without the need of knowledge of the source code.

Finally, in Table 2 the results obtained from WRF-ARW application for Europe are shown. The same behaviour as we have seen in Iberian case is detected.

Table 1. WRF-NMM. Time spans are expressed in microseconds.

Number of Processors	Initialization	Computation	Output	Elapsed Time
WRF-NMM Iberian Peninsula				
128	145338	226795	199179	571312
256	167984	143703	140392	452079
512	212785	88102	91359	392246
WRF-NMM Europe				
128	209953	203590	206924	620467
256	155502	134578	130064	420144
512	203207	90168	93760	387135

Table 2. WRF-ARW. Time spans are expressed in microseconds.

Number of Processors	Initialization	Computation	Output	Elapsed Time
WRF-ARW Iberian Peninsula				
128	140363	102442	93412	336217
256	153783	73624	63674	291081
512	189981	62973	57071	310025
WRF-ARW Europe				
128	126081	95575	86108	307764
256	141139	69705	61591	272435
512	181811	60603	54515	296929

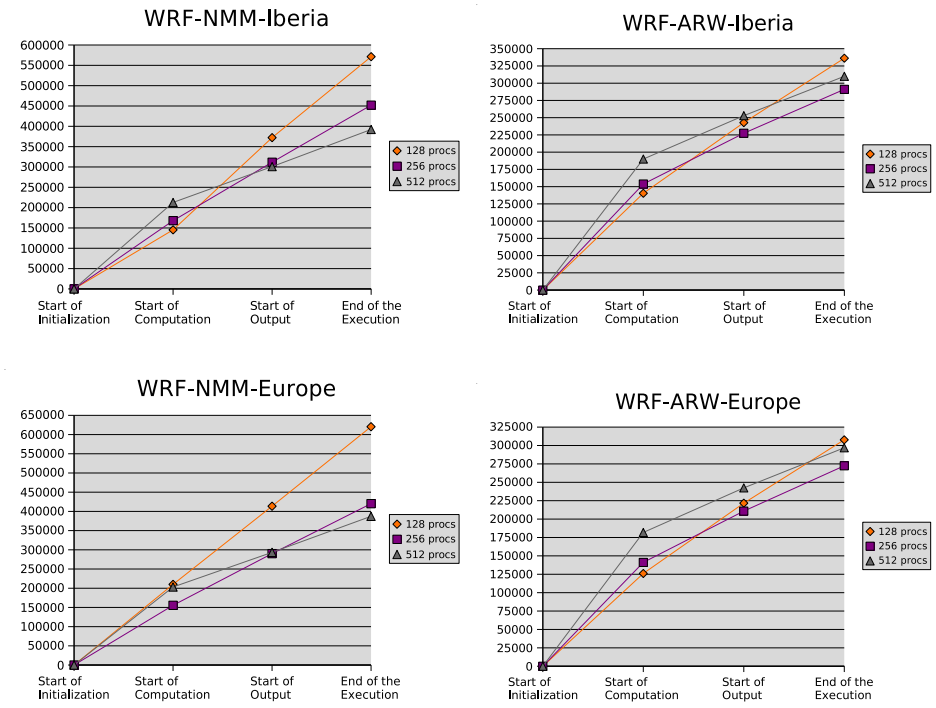


Figure 3. Representation, in microseconds, of the accumulated sum of the results shown in tables 1 and 2. In the Start of Computation axe, time span of initialization is shown. In the Start of Output axe, the sum of Initialization and Computation time spans is shown. Finally, in the End of Execution axe, the total elapsed time is shown.

5 Conclusions and Future Work

In this paper, we have shown clearly the relevance of automatic phase detection of MPI applications in order to obtain automatically non-trivial information. Specifically, to study the scalability of applications, we have shown how automatic phase detection allows the analyst to detect several execution regions and determine which of these regions are undermining the global execution scalability.

This automatic detection is performed using Wavelet analysis, which is a well known signal processing tool used in a wide and heterogeneous range of topics but very few used to study the execution of applications. A very important feature of Wavelet analysis is that it can be carried out in O(n) operations. In the future, we will use the methodologies explained in[16] to study the periodicities of the computing phase and to extract the length of the iterations. Using these methodologies we will also extract a representative set of iterations of the computing phase. After that, an analytical model of the scalability will be applied to this set of iterations. The final goal is generate automatically a complete report that guides the programmer to improve application's performance.

References

1. S. G. Mallat, *A Wavelet Tour on Signal Processing*, (Elsevier, 1999).
2. I. Daubechies, *The Wavelet Transform, time-frequency localization and signal analysis*, IEEE Transactions on Information Theory, (1990).
3. C. S. Burrus, R. A. Gopinath and H. Guo,*Introduction to Wavelets and Wavelet Transform*, (Prentice-Hall, 1998).
4. A. Knuepfer, H. Brunst and W. E. Nagel, *High Performance Event Trace Visualization*, in: Proc. PDP 2005, pp. 258–263.
5. H. Brunst, D. Kranzlmuller and W. E. Nagel, *Tools for Scalable Parallel Program Analysis - Vampir VNG and DeWiz*, DAPSYS, pp. 93–102, (2004)
6. T. Sherwood, E. Perelman, G. Hamerly and B. Calder, *Automatically Characterizing Large Scale Program Behavior*, in: Proc. 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 45–57, (2002).
7. T. Sherwood, E. Perelman, G. Hamerly, S. Sair and B. Calder, *Discovering and Exploiting Program Phases*, in: IEEE Micro: Micro's Top Picks from Computer Architecture Conferences pp. 84–93, (2003).
8. M. Huang, J. Renau and J. Torrellas, *Positional adaptation of processors: application to energy reduction*, in: Proc. the 30th Annual Intl. Sym. on Computer Architecture, pp. 157–168, (2003).
9. J. E. Smith and A. S. Dhodapkar *Dynamic microarchitecture adaptation via co-designed virtual machines*, in: Intl. Solid State Circuits Conference 2002, pp. 198–199, (2002).
10. OMPItrace manual `www.cepba.upc.es/paraver/docs/OMPItrace.pdf`
11. Nonhydrostatic Mesoscale Model (NMM) core of the Weather Research and Forecasting (WRF) system. `http://www.dtcenter.org/wrf-nmm/users/`
12. Advanced Research WRF (ARW) core of the Weather Research and Forecasting system. `http://www.mmm.ucar.edu/wrf/users/`
13. Plane wave/pseudopotential implementation of Density Functional Theory, particularly designed for ab-initio molecular dynamics. `http://www.cpmd.org/`
14. Dimemas: performance prediction for message passing applications, `http://www.cepba.upc.es/Dimemas/`
15. Paraver: performance visualization and analysis, `http://www.cepba.upc.es/Paraver/`
16. M. Casas, R. M. Badia and J. Labarta,*Automatic Extraction of Structure of MPI Applications Tracefiles*, in: Proc Euro-Par 2007, pp. 3–12 (2007).

Biomedical Applications

Experimenting Grid Protocols to Improve Privacy Preservation in Efficient Distributed Image Processing

Antonella Galizia[1], **Federica Viti**[2], **Daniele D'Agostino**[1], **Ivan Merelli**[2], **Luciano Milanesi**[2], **and Andrea Clematis**[1]

[1] Institute for Applied Mathematics and Information Technologies
National Research Council, Genova, Italy
E-mail: {*clematis, dago, antonella*}*@ge.imati.cnr.it*

[2] Institute for Biomedical Technologies
National Research Council, Segrate (MI), Italy
E-mail: {*federica.viti, ivan.merelli, luciano.milanesi*}*@itb.cnr.it*

In order to share interesting data located in various institutions, it may be necessary to ensure secure accesses and privacy preservation. For example, it is the case of biomedical images, since they are related to confidential information corresponding to real patients. This paper describes an approach to enable the use of $\mathrm{PIMA(GE)^2}$ Lib, Parallel IMAGE processing GEnoa Library, in the the analysis of distributed medical images complying with privacy preservation policy imposed by the owner institutions. To achieve this goal, a set of specialized I/O functions was developed to enable secure remote data access and elaboration, exploiting the Grid infrastructure and without copy data from their original position. The work considers the use of the EGEE infrastructure and the GFAL API. The approach was experimented in the analysis of images obtained through the Tissue MicroArray technique; the tests gave positive results also in the execution time.

1 Introduction

In the image processing community, a topic of interest is represented by privacy preservation and secure handling of sensitive data, such as biomedical images. In recent years microarray technology has increased its potential and now it produces a large amount of genomic, transcriptomic and proteomic data. Sharing such images between different institutions is an essential point to increase the number of case studies to analyze and compare. Actually in pathology, the data are related to real patients, and confidential information, such as patient identity, its clinical situation, sample treatment, are usually stored in sensitive metadata. Thus the preservation of such metadata is required.

The pathology departments usually shield the patient identity using anonymous identifiers to store data and metadata. Through such identifiers, only authorized people may know the important information related to patients. This is a good method for a local storage, but it is not safe in the sharing of data among scientific communities; it is necessary to add a further security level. In fact, in distributed systems aimed to support data sharing and multi-institutional collaborations, the transfer of confidential patient information and genomic data around the world must be completely protected. In most cases, to copy patient data in places different from the original database is forbidden by the owner institutions.

Therefore it is possible to outline a double necessity in biomedical imaging: computational resources to share and analyze large data sets, and secure sharing policies to

shield sensitive data belonging to different institutions. One of the possible architectures to achieve these goals is the Grid[1], that provides storage facilities and high computational power under a solid security infrastructure. In fact to access the resources in a Grid, users must be members of a Virtual Organization (VO). Depending on the agreements between the VO partners, an authorization policy is defined to ratify who can access and exploit the resources shared in the Grid. Furthermore the Grid architecture provides an efficient security system with the Grid Security Infrastructure (GSI), it is mainly based on the use of public key cryptography. GSI ensures secure communications, supports secure sharing across organizational boundaries, and the single sign-on for users of the Grid.

This paper describes an approach to add privacy preservation to $\mathrm{PIMA(GE)}^2$ Lib, Parallel IMAGE processing GEnoa Library[2], in order to enable its use to process distributed medical images. The key point is to allow the interaction between $\mathrm{PIMA(GE)}^2$ Lib and computational grid; in this way the library is able to exploit grid security policies, and obtain computational and storage resources. This strategy does not compromise the user-friendliness of the library. In particular this work considers the EGEE (Enabling Grids for E-sciencE) middleware[3], that provides the Grid File Access Library (GFAL) API. Using GFAL, a set of specialized I/O functions has been developed and added to the library to enable secure data access. These functions allow to read/write remote data verifying the user authorization; in this way it is possible to avoid sensitive data transfer and to enable their elaborations complying with the VO policy. The performance aspects have been evaluated comparing GFAL with other protocols for data acquisition. The strategy has been tested with the analysis of images obtained considering the Tissue MicroArray technique[4]. The experimentation of Grid protocols to analyse remote TMA images in the EGEE environment gives positive results, also in the performance evaluation.

The paper is organized as follows: in the next Section related works are discussed. In Section 3 the EGEE infrastructure and GFAL are described. In Section 4, $\mathrm{PIMA(GE)}^2$ Lib is presented, with the major emphasis on the I/O operations. In Section 5 a Tissue MicroArray elaboration is provided. Performance analyses are detailed in Section 6 and in the last Section the conclusions are given.

2 Related Works

In order to handle and mining TMA data, a number of databases can be used. Open source software tools that could be considered the most utilized in this context[5,6,7,8] were tested. They provide a web interface to local databases, and enable the storage of information related with TMA images, such as patient clinical data, specimens, donor blocks, core, and recipient block. These tools may also supply different algorithms to enable the analysis of the TMA images. Nevertheless, none of the available systems promote data sharing in a secure environment, nor the possibility of remotely storing data without HTTP time-consuming protocol.

Instead considering well-known projects related with accessing distributed sensitive data[9,10], the security level of the systems is mainly guaranteed by the Globus implementation of GSI. However it is possible to find different architectures introducing new secure data managing systems mainly based on cryptography techniques. Other groups are developing distributed platforms increasing the user-friendliness of the Globus security service.

In Angeletti et al.[11], an agent based architecture for DNA-microarray data integration

is proposed. They provide a tool to enable the communications and coordinations of DNA-microarray data disseminated in different databases. To deal with sensible data and ensure privacy preservation, they exploit techniques of cryptography and certified agents.

The cancer Biomedical Informatics Grid ($\mathrm{caBIG}^{\mathrm{TM}}$) project[12] connects scientists and organizations to combine strengths and expertise in an open environment. Issues related to privacy and security are addressed in the Data Sharing and Intellectual Capital Workspace (DSIC WS). They faced security aspects following the GSI method, and developed an architecture very similar to the EGEE system.

The Grid Enabled Microarray Experiment Profile Search (GEMEPS) project[13] aims to develop a Grid infrastructure for discovery, access, integration and analysis of microarray data sets. They simplify the Globus authorization and authentication, users can access resources by using their own home security domain to perform the authentication. It is made through a portal; once logged-in a user can exploit the portlets depending on its level of privilege and without repeating the authentication process.

A similar approach has been proposed in the Biomedical Research Informatics Delivered by Grid Enables Services (BRIDGES)[14], which deals with federation and elaboration of distributed biomedical data among a VO. They consider the use of a portal to check the resource access and to simplify the authentication and authorization phases. They exploit PERMIS to provide a fine grained security (authorization). The data management is implemented with different security levels. Actually data integrated in BRIDGES can be public domain or restricted for usage only by specific VO partners.

The $\mathrm{PIMA(GE)}^2$ Lib approach also exploits GSI to guarantee the boundaries imposed by the VO partners; this work utilizes the EGEE security infrastructure to perform the authorization and the authentication processes. Moreover using the GFAL API, a further level of security is added in the elaboration of sensitive data, since it enables the data acquisition without actually transfer or copy such files from their original positions. In this way, data are still under a restrict access policy, and the metadata security is not compromised; any possible data intrusion is avoided. Furthermore the level of easiness in the use of the $\mathrm{PIMA(GE)}^2$ Lib functions is not compromised. The user has to develop a library application, and execute it in the EGEE infrastructure.

3 The EGEE Infrastructure and GFAL

This work has been performed using the EGEE infrastructure, a wide area Grid platform for scientific applications. EGEE relies on the Globus Toolkit[15] (GT4), an ideal communication layer between the different Grid components that is considered the de facto standard in the Grid community.

The EGEE infrastructure is a network of several Computing Elements (CE), that are the gateways for the Worker Nodes (WN) on which jobs are performed, and a number of Storage Elements (SE), on which data are stored. To exploit the resources the users must be members of a VO. Through the User Interface (UI), an authorized user can access data, submit jobs, monitor their status and retrieve the outputs if jobs are terminated successfully or resubmit them in case of failure. The distribution of the computational load is performed by the Resource Broker (RB) delegated for routing each job to an available CE. Jobs are submitted from a UI using the Job Description Language (JDL) scripts, which specify the necessary directives to perform the task. Important features specified by the JDL are the

files to access on the SE, the data to exchange between the UI and the WN and the software requirements for job executions.

EGEE provides a set of tools in order to manage remote data in a distributed way. The LCG File Catalogue (LFC) allows to locate files, or their replicas, on the Grid. The users must create an alias to refer to data, i.e. the Logical File Name (LFN), and a non-human-readable unique identifier is created, i.e. the Globally Unique Identifier (GUID). The LFC maintains the mappings between the LFNs and the GUIDs of the data file. Both, the GUID and the correspondent LFN can be used to access data. The LCG system provides the functionality to the data management. They enable a transparent interaction with file catalogs. Files are visible to the users belonging to the same VO, but the owner can manage permissions, and avoid the access by non-authorized users.

However it is likely to find data which is not allowed to transfer or copy on different SEs. In this case, an effective tool provided by EGEE is the GFAL (Grid File Access Library) API, a library aimed at the file management that is available for different programming languages. It enables the access of files located in SEs, and their elaboration without copying data on the WNs. GFAL verifies the user certificate, and forbids the acquisition of data without the proper authorization. Moreover avoiding multiple physical copies of data, also any possible data intrusion is unlikely. In fact data are not present on a machine different form the SE where they have been allocated under a strict access policy.

4 $\mathrm{PIMA(GE)^2}$ Lib and I/O Operations

The $\mathrm{PIMA(GE)^2}$ Lib provides robust and high performance implementation of the most common low level image processing operations, according to the classification presented in Image Algebra[18]. It was implemented using C and MPICH 2, and allows the development of compute intensive applications, achieving good speed up values and scalability.

The library elaborates 2D and 3D data sets, and allows to perform operations both in sequential and in parallel, depending on the user requirements. In the second case the parallelism is hidden from the users and completely managed by the library. The transparence of the parallelism is mainly obtained through the definition of an effective and flexible interface that appears completely sequential[19]. The main advantage of this approach is that $\mathrm{PIMA(GE)^2}$ Lib shields users from the intrinsic complexity of a parallel application.

An optimization policy is applied in the library, which tackles the optimization issue at different levels: a strategy is oriented to a suitable management of communication and memory operations; the data distribution ensures load balancing, and is performed during the I/O phases through a parallel I/O. Other efforts are dedicated to the exploitation of memory hierarchies. Also the optimization aspects are transparent to the users.

The operations provided by the $\mathrm{PIMA(GE)^2}$ Lib have been grouped in eight *conceptual objects*, that represent the core of $\mathrm{PIMA(GE)^2}$ Lib code, i.e. all the operations that are visible to the users. The objects underline the operation similarity, and each of them follows the same parallelization policy, called *parallelizable pattern*. This paper considers the **I/O Operations**, which are responsible for the I/O and memory management operations. A set of I/O functions has been developed to exploit different protocols in the access data. They present a single interface with specific parameters to select the appropriate implementation. The possibilities are:

- **Local Parallel I/O**: if I/O data are local, a parallel I/O developed with MPI2 I/O functionalities[17] is considered. Data are stored using a parallel file system namely PVFS[16], that combined with MPI 2, allows the exploitation of an efficient data access and distribution. In this way the I/O time speeds up;
- **Distributed Parallel I/O**: if I/O data are remote and accessible, data transfer services using the FTP protocols is considered. Images are provided through their URIs, Uniform Resource Identifier, copied in the PVFS parallel file system, and acquired through an MPI 2 parallel I/O;

Considering biomedical images located in various pathology departments, it is likely that data acquisition through an anonymous FTP service is forbidden, and only a restricted data access is allowed. In such situations a user identification and authorization may be required, thus the $\mathrm{PIMA(GE)}^2$ Lib functionalities could not be utilized. In order to adapt the library code for the processing of biomedical image, it is possible to exploit the Grid infrastructure. In fact combining the use of the Grid Secure Infrastructure, an appropriate VO sharing policy, and the GFAL API restricted data access can be ensured[20]. In this way, it is possible to elaborate data without compromising the security level imposed by the VO. The VO maintains the control and the responsibility of the data, and $\mathrm{PIMA(GE)}^2$ Lib refers to them in order to guarantee the effectiveness of that policy. The GFAL API has been used to develop a further set of specialized I/O functions, called

- **Distributed master-slave I/O**: if I/O data are remotely located and restricted by authorized access in the EGEE infrastructure. Even in this case images are provided through their URIs, however the raw data are transfered directly in main memory and not on a physical disk. It has been obtained using the GFAL API. The data distribution has been implemented with a master/slave approach.

The distributed master-slave I/O functionalities have been included in the I/O Operations of $\mathrm{PIMA(GE)}^2$ Lib. Thus, another possibility has been added in the selection of the protocols to use in the I/O management, and the use of the GFAL API becomes transparent. In this way,the interaction between $\mathrm{PIMA(GE)}^2$ Lib and the Grid is obtained without an actual modification to the library interface. Indeed in order to exploit the Grid, the user has to develop its application, and execute it on the Grid in accordance with a standard EGEE submission.

5 A Tissue MicroArray Analysis

The presented strategy was tested by executing an edge detection of images stored on Grid SE. The images to process have been obtained using the Tissue MicroArray technique, that enables the elaboration of hundreds of tissues as input, to study a single gene, transcript or protein in a parallel way on the same paraffin block. The test considers 10 images stored in a set of SEs, acquiring raw data and the metadata not related with patient information. They are acquired one at a time using the distributed master-slave I/O functions, processed on a CE, in the specific case a remote workstation, and then the output image is produced with respect to the defined privacy policy.

The edge detection was developed using the library functions and in particular the distributed master-slave I/O functionalities to access data. The application has been executed

through a CE on which the user has already been identified as member of the VO. Through the UI, the application has been submitted using the JDL script; in this way the user describes how the job has to be computed on the WN. An example of the analysis is shown in Figure 1. Figure 1(a) is the input image and it represents an haematoxylin-eosin reaction: this coloration differentiates basic and acid entities in the tissue slice. Basic elements (like cytoplasm) are highlighted in a static grey while acid elements (like nuclei) are in light grey. Figure 1(b) represents the result of a first attempt of the edge detection algorithm.

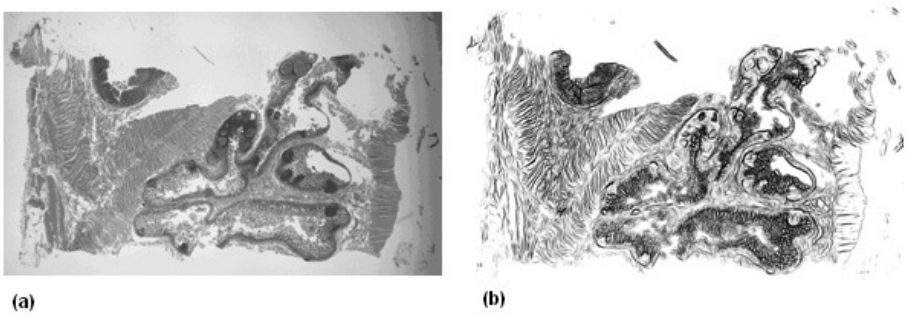


Figure 1. TMA image input 1(a) and output 1(b) of remote image analysis, where is applied an edge detection algorithm.

6 Performance Evaluations

In order to demonstrate the effectiveness of the distributed master-slave I/O functions, the same application has been executed using different protocols in the data acquisition. In particular, the attention was focused on the time that is spent on the data acquisition, since the $\mathrm{PIMA(GE)}^2$ Lib performance are not interesting in this context.

The aim of the tests is to compare different situantions for distributed image processing. In all the cases there are: a client that drives the execution, a remote data storage and a remote computing node. The point is to evaluate performance and security in data transfer between the storage and the computing element. In a first test the three elements interact unsing FTP protocol outside of the Grid infrastructure. In the other two cases the elements interact within EGEE Grid infrastructure, in one test using LCG, in the other GFAL to transfer data.

The most efficient operation is the FTP data transfer, requiring 12,229 seconds in the management of a 10 MB image. It is an almost evident result, because this protocols is not involved in a complex Grid infrastructure and does not suffer from the overhead related to it. exploiting the grid. Furthermore this service does not apply any strategy to obtain a secure data transfer, in fact data are sent in clear text, and can be intercepted by everyone. This step also improves the service performance, however its use in the bioimaging community has to be discouraged.

The similar operation provided by the EGEE infrastructure, i.e. LCG-copy, requires about one minute (65,081 seconds) to manage the same image. In fact in this case the data

transfer suffers from the overhead related to the Grid protocol, but data are transferred in a secure way. This service should be used instead of the previous service.

The data acquisition through the use of GFAL requires 27,654 seconds. This service also suffers from the Grid overhead, however its performance is really better than the LCG-copy one, and it still provides a secure data acquisition. Actually, since the data are not physically transferred, it represents the most secure way to access data. The use of GFAL is the suitable solution for the processing of data bounded by a strict authorization access policy, as in the bioimaging context described in the paper.

7 Conclusions

Security is a topic of interest in the applied medicine, in order to warrant patient privacy and confidentiality. This issue is really stressed in distributed systems aimed to support data sharing and multi-institutional collaborations, in fact the transfer of patient clinical information and genomic data around the world must be completely protected. Also in the biomedical imaging community it represents a topic of interest, since the necessity of data sharing has increased.

This paper presents how to add privacy preservation and secure data access to $\mathrm{PIMA(GE)^2}$ Lib exploiting the grid infrastructure. In particular this work considers the EGEE middleware and the GFAL API; using GFAL, a set of specialized I/O functions for secure data access was developed. These functions allow to read/write remote data verifying the user authorization; in this way it is possible to avoid sensitive data transfer and to enable their secure elaborations complying with the VO sharing authorization policy. Furthermore the user-friendliness of the $\mathrm{PIMA(GE)^2}$ Lib is not compromised in exploiting the Grid. From the user point of view, the development of an application is not actually modified, and the execution phase corresponds to a standard EGEE submission. Also the evaluation performance gives positive results.

Allowing different data acquisition possibilities, $\mathrm{PIMA(GE)^2}$ Lib represents a flexible and efficient tool. It can be exploited to process remote and local images. In particular it can be employed in biomedical imaging communities since it is compliant with the possibility of respect the local policy of data access and preservation of sensitive data. Even in these situations $\mathrm{PIMA(GE)^2}$ Lib still ensures efficient executions and a user friendly interface.

Acknowledgements

This work has been supported by the regional program of innovative actions PRAI-FESR Liguria.

References

1. I. Forster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, 2nd Edition. (Morgan Kaufmann Publishers Inc, 2004).
2. A. Clematis, D. D'Agostino and A. Galizia, *The Parallel IMAGE processing GEnoa Library:* $\mathrm{PIMA(GE)^2}$ *Lib.* Technical Report IMATI-CNR-Ge, n. 21 (2006).

3. EGEE Home Page, http://www.eu-egee.org/
4. J. Kononen, L. Bubendorf, A. Kallioniemi, M. Barlund, P. Schraml, S. Leighton, J. Torhorst, M. J. Mihatsch, G. Sauter and O. P Kallioniemi, *Tissue microarrays for high-throughput molecular pro_ling of tumor Specimens*, Nature Medicine, **4**, 844–847, (1998).
5. J. D. Morgan, C. Iacobuzio-Donahue, B. Razzaque, D. Faith and A. M. De Marzo, *TMAJ: Open source software to manage a tissue microarray database*, Proc. APIII Meeting, (2003).
6. F. Demichelis, A. Sboner, M. Barbareschi and R. Dell'Anna, *TMABoost: an integrated system for comprehensive management of Tissue Microarray data*, IEEE Trans Inf Technol Biomed 2006, **10**, 19–27, (2006).
7. V. Della Mea, I. Bin, M. Pandolfi and C. Di Loreto, *A web-based system for tissue microarray data management*, Diagnostic Pathology, **1**, 36, (2006).
8. A. Sharma-Oates, P. Quirke and D. R. Westhead, *TmaDB: a repository for tissue microarray data*, BMC Bioinformatics, **6**, 218, (2005).
9. caGrid: https://cabig.nci.nih.gov/workspaces/Architecture/caGrid
10. MyGrid: http://www.mygrid.org.uk/
11. M. Angeletti, R. Culmone and E. Merelli, *An intelligent agents architecture for DNA-microarray data integration*, Proceedings of NEETAB (2001).
12. caBIG: https://cabig.nci.nih.gov/
13. R. Sinnott, C. Bayliss and J. Jiang, *Security-oriented Data Grids for Microarray Expression Profiles*, Proc. HealthGrid2007, (2007).
14. R. Sinnott, M. Bayer, D. Houghton, D. Berry and M. Ferrier, *Bridges: Security Focused Integration of Distributed Biomedical Data*, Proceedings of UK e-Science All Hands Meeting (2003).
15. The Globus Alliance, http://www.globus.org/
16. The parallel virtual file system (PVFS), http://www.pvfs.org/pvfs/
17. ROMIO: A high-performance, portable MPI-IO implementation, http://www.mcs.anl.gov/romio
18. G. Ritter and J.Wilson, *Handbook of Computer Vision Algorithms in Image Algebra*, 2nd edition. (CRC Press, 2001).
19. A. Clematis, D. D'Agostino and A. Galizia, *An Object Interface for Interoperability of Image Processing Parallel Library in a Distributed Environment*. In Proceedings of ICIAP 2005, LNCS 3617, 584-591, (Springer, 2005).
20. S. Bagnasco, F. Beltrame, B. Canesi, I. Castiglioni, P. Cerello, S.C. Cheran, M.C. Gilardi, E. Lopez Torres, E. Molinari, A. Schenone and L. Torterolo, *Early diagnosis of Alzheimer's disease using a grid implementation of statistical parametric mapping analysis*, Stud Health Technol Inform, **120**, 69–81, (2006).

A Parallel Workflow for the Reconstruction of Molecular Surfaces

Daniele D'Agostino[1], **Ivan Merelli**[2], **Andrea Clematis**[1], **Luciano Milanesi**[2], **and Alessandro Orro**[1]

[1] Institute for Applied Mathematics and Information Technologies, National Research Council
Via De Marini 6, 16149 Genova, Italy
E-mail: {*dago, clematis*}*@ge.imati.cnr.it*

[2] Institute for Biomedical Technologies, National Research Council
Via Fratelli Cervi 93, 20090 Segrate (MI), Italy
E-mail: {*ivan.merelli, luciano.milanesi, alessandro.orro*}*@itb.cnr.it*

In this paper a parallel workflow for the reconstruction of molecular surfaces based on the isosurface extraction operation is proposed. The input is represented by the atomic coordinates of a molecule, the output is both its volumetric description and the isosurface that, on the basis of the user selection, corresponds to the Van der Waals, Lee & Richards or Connolly surface. The main feature of the workflow is represented by the efficient production of high resolution surfaces. This is a key aspect in Bioinformatics applications, considering that the amount of data to process may be very huge. This goal is achieved through a parallel implementation of the stages of the workflow.

1 Introduction

The modelling of molecular surfaces and their visualization is assuming an increasing importance in many fields of Bioinformatics. This is particularly true in the study of molecule-molecule interactions, usually referred as molecular docking, that represents one of the subject of our researches.

Superficial complementarities play a significant role to determine the possible binds between pairs of molecules[1,2]. Mechanisms such as enzyme catalysis and recognition of signals by specific binding sites and docking in fact depend on morphological characteristics. It is clear that a correct superficial description plays a crucial role for the definition of a valid docking. Usually a molecule is represented through the set of its 3D atomic coordinates. This is for example the format adopted by the Protein Data Bank (PDB)[3], one of the most important repositories. Such format well suits structural analysis operations, but not those based on the molecular shape. This is the reason why other representations, that can be derived from this one, have to be taken into account.

In particular for molecular docking the Van der Waals[4], the Lee & Richards[5] and the Connolly[6] surfaces are the most important ones. All of them are computed by modelling the atoms with spheres, and the result is the production of polygonal (mostly triangular) meshes, but they differ for the considered atomic radii and the possibility to close some small superficial cavities.

Different algorithms were proposed in the literature to compute such surfaces, and many implementations are available. Most of them rely on a pure geometrical analysis of the atoms' positions and on the estimation of their volumes. One example is MSMS[7],

that is probably the most used tool. The main characteristic of this approach is its computational efficiency, but disregarding the volumetric description some information about binding sites buried under the surface is lost. This is a major drawback for the docking analysis. Other algorithms rely on the volumetric description of the molecule using the isosurface extraction operation. This approach permits an accurate modelling of its inside structures, but is more costly than the previous one. The most representative tool in this case is GRASP[8]. Both these tools have the drawback than they run out of memory for large surfaces, therefore they are able to provide only limited resolution surface representations. This may be an important issue considering the need to compute the buried surface upon molecular formation, therefore MSMS and GRASP are not perfectly suited for an in-depth analysis of internal functional sites.

In this paper a parallel workflow for the reconstruction of molecular surfaces based on the isosurface extraction operation is proposed. Two are the main advantages, the performace and the surface quality. With regards to the performance, the tests showed that also using the sequential implementation of the workflow it is possible to compute the molecular surfaces faster than MSMS and GRASP.

Moreover the use of the simplification operation permits to produce high quality surfaces with different levels of detail. The purpose of this operation is to preserve the morphological information in correspondence of irregular zones, that are the most interesting ones, with great accuracy, while reducing the number of triangles in the other parts. The resulting simplified surface is therefore made up by less triangles than the original one, but it preserves all the important features of the molecule. This aspect is of particular importance for molecular docking, that has a cost proportional to the size of the surfaces to process. In this case the parallel implementation of the workflow is fundamental, because it permits to obtain fast results also for surfaces made up by several million triangles, that otherwise cannot be processed.

The paper is organized as follows. In Section 2 the structure of the workflow and the parallel implementations of the various stages is described. Section 3 presents the experimental results, while conclusions and future work are outlined in Section 4.

2 The Workflow Design and Implementation

The architecture of the workflow is represented in Fig. 1. The main input of the system is a PDB file containing the atomic coordinates of the atoms that form a molecule, while the output is represented by both the volumetric description of the molecule and the isosurface corresponding to one of the three aforementioned surfaces. The workflow is made up by five operations: *Grid Generation*, *Connolly Correction*, *Median Filter*, *Isosurface Extraction* and *Simplification*.

Grid Generation
The first operation is the generation of the three-dimensional grid containing the volumetric data representing the molecule. The size of the grid is determined on the basis of the coordinates of the atoms and on the required sampling step. Typical step values are chosen between 0.7 and 0.1 Å, according to the desired level of resolution. Low step values correspond to dense grids and high resolution surfaces, and vice versa.

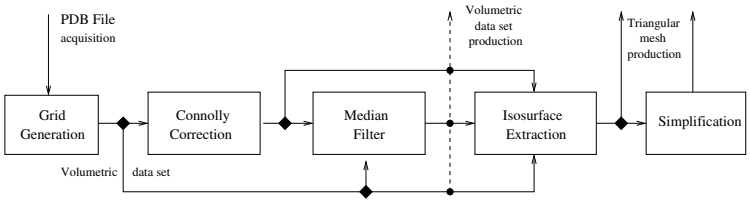


Figure 1. The stages of the parallel workflow for the reconstruction of molecular surfaces. The results are both the volumetric data set, produced before the execution of the isosurface extraction, and the triangular mesh.

Atoms are modeled in the grid with spheres having different radii. The points within a sphere assume negative values, that increase gradually from the centre to the hull, where they become positive.

Two are the possible different partitioning strategies for the parallelization of this operation. The first one is the even subdivision of the grid along the Z axis, the other one is a partitioning driven by the spatial subdivision of the atoms. Both the strategies present pros and cons, therefore they were experimentally evaluated.

The latter strategy permits to achieve the best performance considering only the execution of this operation, because the computational cost of each chunk of the grid is proportional to the number of atoms a process has to model. However this solution has the drawback to worsen considerably the performance of the whole pipeline. This is because the partitioning is not suitable for the following two operations, therefore there would be the need to perform at least one rebalancing step.

On the contrary, the former strategy permits to exploit the same partitioning for the three operations. In fact the resulting unbalancing of this stage of the workflow is less costly than the data exchange that otherwise is necessary, therefore it was decided to adopt it. Moreover this solution permits processes to keep the data in main memory until the isosurface extraction operation, if there is a sufficient amount of aggregate memory.

Connolly Correction

The second operation is performed only when the Connolly surface is required. This surface consists of the border of the molecule that could be accessed by a probe sphere representing the solvent. When this sphere rolls around a pair of atoms closer than its radius, it traces out a saddle-shaped toroidal patch of reentrant surface. If the probe sphere is tangent to three atoms, the result is a concave patch of reentrant surface. The main difference with respect to the Van der Waals and the Lee & Richards surfaces is that these patches close the superficial small cavities.

The Connolly Correction operation consists in changing the values of the points of the volume that become internal (and so with a negative value) considering these new patches. It is performed in two steps, the identification of the pairs of close atoms and the modification of the values of the points in the neighbourhood of these pairs.

The first step requires to compare the distances among all the pairs of atoms and the solvent radii. It is implemented by subdividing all the possible pairs among the processes, with the exchange of the results at the end.

The second step is implemented using the data parallel paradigm. Also in this case the

considerations made describing the parallelization strategy of the previous operation holds true, therefore its partitioning is exploited .

It is worthwhile to note that, while in the previous operation the unbalancing produced by the uneven spatial distribution of the atoms is negligible, due to their limited number, here it may be required to model the patches of more than one million pairs, therefore the unbalancing may be more relevant.

Median Filter
After the creation of the volumetric data set a median filter can be applied in order to produce a smoother isosurface. This operation is useful when medium-high step values are used, otherwise it may be disregarded.

The data parallel paradigm is the suitable choice also for implementing this operation. In this case in fact the amount of work is exactly proportional to the number of points to process, therefore the even partitioning of the volume guarantees high performance.

Note that the parallel processes need to exchange border regions in order to correctly compute the values of the points on the boundary of the parts. This represents a negligible overhead, except for small grids.

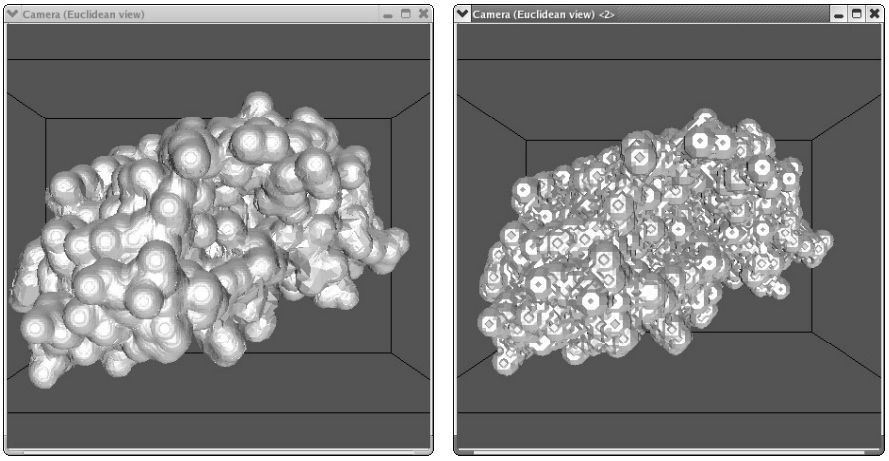
After the last of these three operations the volumetric data set is written on the hard disk, in order to make it available for further analysis operations.

Isosurface Extraction
The fourth operation is the extraction of the isosurface representing the molecular shape. The Connolly surface is extracted considering the isovalue 0 if the Connolly Correction is performed. Otherwise the Van der Waals and the Lee & Richards surfaces are extracted considering, respectively, the isovalues 0 and 1.4.

Two parallel versions of the Marching Cubes algorithm[9], the mostly used algorithm to extract isosurfaces from volumetric data sets, were implemented. The first one is based on the farm on demand paradigm, and provides high performance figures when the simplification operation is not performed. Otherwise the second one[10], based on a data parallel approach with an intermediate step for balancing the triangles among the parallel processes, is used. The two versions are respectively called *Farm on Demand* (FD) and *Load Balanced on Active Cells* (LBAC). A comparison of these two algorithms is discussed in[11].

Simplification
The Marching cubes algorithm has the characteristics of producing a large number of small planar triangles. With the simplification operation it is possible to obtain a smaller but equivalent surface by merging them. This operation is optional. It may be executed for example if the size of the isosurface overcomes a given threshold. The result is characterized by the non-uniform level of detail, because the most irregular zones are represented using more triangles than the regular ones.

If the objective is only the reduction of triangles a bigger step for Grid Generation may be used. However in this case the resulting mesh has an uniform coarse grain level of detail. In Fig. 2 two Connolly surfaces of the same molecule made up by about 310,000 triangles are presented. If the left one, obtained using a finer grid and the simplification operation with a simplification percentage of 40%, is compared with the right one, obtained using a coarser grid, it appears that the best result is obtained using the simplification.



(a) The simplified surface (b) The surface obtained using a coarser grid

Figure 2. A comparison between two different representations of the Connolly surface for the molecule of the bovine insulin "2INS". Both the images are made up by about 310,000 triangles, but the left one is obtained using a finer grid and the simplification operation with a percentage of 40%, the right one using a coarser grid.

Several simplification algorithms were proposed in the literature. Here the Garland-Heckbert algorithm[12] was chosen because it produces high quality results. A parallel version of it was developed using the data parallel approach with a coordinator process that globally selects the triangles to merge, in order to minimize the introduced representation error[13]. The boundaries of the various parts are preserved, therefore the mesh resulting after the merge of the partitions does not present any crack.

It is to note that the simplification algorithm works in core. This means that the sequential version is not able to process large meshes. The parallel version, instead, is able to process arbitrary meshes if a sufficient amount of aggregate memory is available. This aspect represents a further advantages of the parallel workflow, beside the efficiency production of the results.

3 Experimental Results

Experimental results were collected considering the execution of all the stages of the workflow, in order to analyze the contribution of each operation to the global execution time.

A Linux-based Beowulf Cluster of 16 PCs equipped with 2.66 GHz Pentium IV processor, 1 GB of Ram and two EIDE 80 GB disks in RAID 0 was used. The nodes are linked using a dedicated switched Gigabit network, and they share a PVFS file system.

Three molecules of the Protein Data Bank repository, chosen on the basis of their size, were considered. The smallest one is the structure of DES-PHE B1 bovine insulin, identified as 2INS and made up by 778 atoms, followed by the crystal structure of a wild type human estrogen receptor, identified as 1G50 and made up by 5,884 atoms, and by the crystal structure of the large ribosomal subunit from Deinococcus Radiodurans, identified as 1NKW and made up by 65,300 atoms.

Molecule	Atoms	Pairs	Grid	Grid file	Triangles	Mesh file
2INS	778	14870	160x114x150	5.2 MB	270568	4.7 MB
1G50	5884	121039	389x294x265	57.8 MB	1938128	33.3 MB
1NKW	65300	1280539	687x833x723	789.2 MB	21813684	374.4 MB

Table 1. This table summarizes the characteristics of the three considered molecules. The number of the pairs of atoms considered for the Connolly Correction is denoted with "Pairs". The size of the files containing the volumetric data and the isosurfaces is computed considering a binary format without any header. In particular the meshes are represented through the coordinates of the vertices and the triangle/vertex incidence relation.

A step of 0.3 Å was considered for the Grid Generation operation, because it permits to create also for 1NKW a grid that fits in main memory using only one node of the cluster.

The characteristics of the molecules, of the resulting volumetric data sets and the Connolly surfaces are shown in Table 1, while Table 2 shows the performance of the sequential and the parallel implementations of the workflow.

At first the sequential times were compared with those of MSMS. In particular MSMS corresponds to the execution of the Grid Generation, the Connolly Correction and the Isosurface Extraction operations. The output of the mesh was disregarded because MSMS stores it in two plain ASCII files, while here binary files are produced. It appears that the performance is the same for 2INS, while MSMS takes 2 second more for 1G50 and it is not able to produce any result for 1NKW because of the large number of atoms to model. Note furthermore that with the PCs used MSMS cannot produce meshes larger than 2 million triangles.

Considering the parallel workflow it can be seen that the efficiency is of about 0.5. This is an important result considering all the issues related to the parallelization of the workflow operations.

In particular it can be seen that the incidence of the possible unbalancing of atoms and pairs resulting from the even partitioning of the grid is less relevant when a large number of

	2INS			1G50			1NKW		
Nodes	1 (sec).	8	16	1 (sec).	8	16	1 (sec).	8	16
Grid G.	0.2	5.3	6.2	1.5	5.0	7.5	19.8	6.6	11.6
Connolly C.	0.8	3.0	4.8	7.4	3.5	5.3	215.8	5.2	8.1
M. Filter	0.4	6.3	8.1	4.2	7.9	15.3	60.0	8.0	15.8
Grid Out.	1.3	3.3	3.1	3.4	3.1	2.4	27.1	2.5	2.4
Isoextr.	2.2	2.5	3.8	7.2	2.6	4.7	70.5	3.4	5.6
Simpl.	6.7	5.7	11.8	48.6	4.8	8.5	(571.3)	N.A.	8.3
Iso. Out.	0.9	8.0	12.1	7.2	8.0	14.4	(112.5)	N.A.	13
Tot.	**12.5**	**4.6**	**7.2**	**79.5**	**4.5**	**7.6**	**(1077)**	**N.A.**	**8.5**

Table 2. This table presents the times, in seconds, for executing the sequential implementation of the workflow (denoted as "1"), and the speed up values of the parallel version using 8 and 16 nodes. The output of the volumetric data set is indicated with "Grid Out.", and the output of the Connolly surface with "Iso. Out.". As regards the Simplification operation the production of a mesh made up by half of the original triangles is required. With the cluster used is not possible to simplify meshes larger than 2 million triangles with only one node. For this reason an estimation of the sequential execution times for 1NKW is provided in brackets.

them have to be modeled. Furthermore this partitioning results in high speed up values for the Median Filter operation, except for 2INS. In this case in fact the overhead represented by the time to exchange the border regions among the workers is comparable to the time to process the data.

The production of the volumetric data has to be done on a shared file system, because they will be used for the isosurface extraction operation. Despite the use of PVFS, that is one of the most performant open parallel file systems, this step achieves low performance when compared with the use of a local disk. On the contrary the part of the isosurface extracted by each process is stored on the local disk of the node on which it is running, together with additional data that will be used for the efficient sewing of these chunks in a unique surface[10]. It is worthwhile to note that these additional data represent an overhead with respect to the sequential case, and furthermore their size is proportional to the number of triangles and parts to merge. For this reason the performance is slight lower for increasing values of the parallelism degree and of the isosurface size.

As said before, the execution of the parallel simplification operation implies the use of the LBAC algorithm. This algorithm presents low performance with respect to FD, but considering also the execution of the simplification it permits to achieve the highest performance[11].

Note that the parallel implementation of the workflow permits the exploitation of the aggregate memory of the cluster. This is of particular importance when considering larger grids, because in these cases the sequential workflow needs to use out-of-core techniques. For example if 1G50 with a sampling step of 0.1 Å is considered, a grid with a size of 1.5 GB (1165x884x797) has to be processed. In this case the parallel workflow achieves a speed up of 12 using 16 nodes.

Disregarding the efficiency aspects, the use of the aggregate memory has the further advantage of making possible the simplification of large meshes in order to produce smaller results with different levels of detail. With a single PC of the cluster used in fact it is not possible to simplify meshes larger than 2 million triangles, while using the cluster meshes of more than 20 million triangles as 1NKW can efficiently be processed. It is estimated that, even if it is possible to process 1NKW with a single node, the execution time should be about 18 minutes, while the parallel implementation is able to produce the result in 2 minutes.

4 Conclusions and Future Work

Surfaces play a fundamental role in protein functions, because chemical-physical actions are driven by their mechanic and electrostatic properties, with scarce influence of the internal structures. In this paper a method for describing molecular surfaces in detail is presented. The aim is to develop a high performance docking screening system, that represents an innovative high performance approach to the protein-protein interaction study.

The key aspects of the workflow are the efficiency and the quality of the results. These aspects are strictly coupled, in particular considering that high resolution representations of molecules is of fundamental importance for the effectiveness of the following analysis operations, but their modelling is a costly process.

As regards the performance, the parallel workflow is able to produce efficiently molecular surfaces also considering very fine grain resolutions. In particular the use of the sim-

plification operation permits an effective reduction of the surface size, preserving however a high level of detail for the irregular zones, that are the most interesting ones.

The next objective of the research is the exploitation of these surfaces in order to determine the possible docking between pairs of molecules. Also in this case the performance represents an important issue, because of the large amount of data to take into exams.

Acknowledgements

This work has been supported by the regional program of innovative actions PRAI-FESR Liguria.

References

1. K. Kinoshita, and H. Nakamura, *Identification of the ligand binding sites on the molecular surface of proteins*, Protein Science, **14**, 711–718, (2005).
2. T. A. Binkowski, L. Adamian, and J. Liang, *Inferring functional relationships of proteins from local sequence and spatial surface patterns*, J. Mol. Biol., **332**, 505–526, (2003).
3. H. M. Berman, T. N. Bhat, P. E. Bourne, Z. Feng, G. Gilliland, H. Weissig, and J. Westbrook, *The Protein Data Bank and the challenge of structural genomics*, Nature Structural Biology, **7**, 957–959, (2000).
4. M. L. Connolly, *Solvent-accessible surfaces of proteins and nucleicacids*, Science, **221**, 709–713, (1983).
5. B. Lee, and F. M. Richards, *The Interpretation of Protein Structures: Estimation of Static Accessibility*, J. Mol. Biol., **55**, 379–400, (1971).
6. M. L. Connolly, *The molecular surface package*, J.Mol.Graphics, **11(2)**, 139–141, (1993).
7. M. F. Sanner, A. J. Olson, and J. Spehner, *Fast and Robust Computation of Molecular Surfaces*, in: Proc. 11th ACM Symp. Comp. Geometry, C6–C7, (1995).
8. A. Nicholls, K. A. Sharp, and B. Honig, *Protein Folding and Association: Insights From the Interfacial and Thermodynamic Properties of Hydrocarbons*, Proteins: Stuc., Func. and Genet., **11**, 281–296, (1991).
9. W. E. Lorensen, and H. E. Cline, *Marching cubes: A high resolution 3-D surface construction algorithm*, Computer Graphics, **21(3)**, 163–169, (1987).
10. A. Clematis, D. D'Agostino, and V. Gianuzzi, *An Online Parallel Algorithm for Remote Visualization of Isosurfaces*, Proc. 10th EuroPVM/MPI Conference, LNCS No. 2840, 160–169 (2003).
11. A. Clematis, D. D'Agostino, and V. Gianuzzi, *Load Balancing and Computing Strategies in Pipeline Optimization for Parallel Visualization of 3D Irregular Meshes*, Proc. 12th Euro PVM/MPI Conference, LNCS No. 3666, 457–466 (2005).
12. M. Garland, and P. S. Heckbert, *Surface Simplification Using Quadric Error Metrics*, Computer Graphics, **31**, 209–216, (1997).
13. A. Clematis, D. D'Agostino, V. Gianuzzi, and M. Mancini, *Parallel Decimation of 3D Meshes for Efficent Web based Isosurface Extraction*, Parallel Computing: Software Technology, Algorithms, Architectures & Applications, Advances in Parallel Computing series, **13**, 159–166, (2004).

HPC Simulation of Magnetic Resonance Imaging

Tony Stöcker, Kaveh Vahedipour, and N. Jon Shah

Institute of Medicine
Research Centre Jülich, 52425 Jülich, Germany
E-mail: {*t.stoecker, k.vahedipour, n.j.shah*}*@fz-juelich.de*

High performance computer (HPC) simulations provide helpful insights to the process of magnetic resonance image (MRI) generation, e.g. for general pulse sequence design and optimisation, artefact detection, validation of experimental results, hardware optimisation, MRI sample development and for education purposes. This manuscript presents the recently developed simulator JEMRIS (Jülich Environment for Magnetic Resonance Imaging Simulation). JEMRIS is developed in C++, the message passing is realised with the MPI library. The approach provides generally valid numerical solutions for the case of classical MR spin physics governed by the Bloch equations. The framework already serves as a tool in research projects, as will be shown for the important example of multidimensional, spatially-selective excitation.

1 Introduction

Bloch equation-based numerical simulation of MRI experiments is an essential tool for a variety of different research directions. In the field of pulse sequence optimisation, e.g. for artefact detection and elimination, simulations allow one to differentiate between effects related to physics and hardware imperfection. Further, if the simulation environment is able to simulate the hardware malfunction, then results may be used for the optimisation of the hardware itself. Another prominent application is the design of specialised RF pulses which is often based on numerical simulations of the excitation process. In general, the interpretation and validation of experimental results benefit from comparisons to simulated data, which is especially important in the context of MRI sample development, e.g. for the development of implants. Another important direction of application is image generation for the purpose of image processing – here, complete control of the properties of the input data allows a tailored design of image processing algorithms for certain applications.

The numerical simulation of an MRI experiment is, in its most general form, a demanding task. This is due to the fact that a huge spin ensemble needs to be simulated in order to obtain realistic results. As such, several published approaches have reduced the size of the problem in different ways. The most prominent method is to consider only cases in which analytical solutions to the problem exist[1]. However, for the case of radiofrequency (RF) excitation in the presence of time varying gradient fields no analytical solution exists and, thus, the important field of selective excitation cannot be studied with such an approach. Apart from the computational demand, the complexity of the MRI imaging sequence is an additional obstacle. The difficulty of MRI sequence implementation using the software environments of commercial MRI scanner vendors – painfully experienced by many researchers and pulse programmers – can be significantly reduced with appropriate software design patterns.

The JEMRIS project was initiated taking all the aforementioned considerations into

account. It takes advantage of massive parallel programming on HPC (high performance computing) architectures. The aim of the project was to develop an MRI simulator which is general, flexible, and usable. In detail, it provides a general numerical solution of the Bloch equations on a huge ensemble of classical spins in order to realistically simulate the MRI measurement process. Further, it takes various important off-resonance effects into account.

2 Theory

2.1 MR Signal Evolution

The JEMRIS simulator is based on a classical description of MRI physics by means of the Bloch equations, describing the sample by its physical properties of equilibrium magnetisation, M_0, and the longitudinal and transverse relaxation times, T_1 and T_2, respectively. It provides an exact description for the magnetisation vector, $\mathbf{M}(\mathbf{r},t)$, of non-interacting spin isochromates under the influence of an external magnetic field. For MRI, the field decomposes to the strong static component, B_0, a temporally and spatially varying field along the same direction (the imaging gradients $\mathbf{G}$) and the orthogonal components of the RF excitation field, B_1. The total field is thus given by

$$\mathbf{B}(\mathbf{r},t) \;=\; [B_0 + \mathbf{G}(t)\cdot\mathbf{r}]\mathbf{e}_z + B_{1x}(\mathbf{r},t)\mathbf{e}_x + B_{1y}(\mathbf{r},t)\mathbf{e}_y \;. \tag{2.1}$$

A mathematical and numerical treatment is greatly simplified in the rotating frame of reference, in which the effect of the main field is not seen since the coordinate system rotates with the speed of the spin precession. Here, the formulation of the Bloch equation in cylindrical coordinates is very well suited for a numerical implementation

$$\begin{pmatrix}\dot{M}_r\\ \dot{\varphi}\\ \dot{M}_z\end{pmatrix} \;=\; \begin{pmatrix}\cos\varphi & \sin\varphi & 0\\ -\frac{\sin\varphi}{M_r} & \frac{\cos\varphi}{M_r} & 0\\ 0 & 0 & 1\end{pmatrix}\cdot\left[\begin{pmatrix}-\frac{1}{T_2} & \gamma B_z & -\gamma B_y\\ -\gamma B_z & -\frac{1}{T_2} & \gamma B_x\\ \gamma B_y & -\gamma B_x & -\frac{1}{T_1}\end{pmatrix}\cdot\begin{pmatrix}M_r\cos\varphi\\ M_r\sin\varphi\\ M_z\end{pmatrix}+\begin{pmatrix}0\\0\\ \frac{M_0}{T_1}\end{pmatrix}\right] \tag{2.2}$$

where γ is the gyromagnetic ratio of the nuclei (usually protons) under consideration. The complex MR signal is then described by the signal equation

$$S(t) \;\propto\; \int_V M_r(\mathbf{r},t)\exp[\mathrm{i}\varphi(\mathbf{r},t)]\;d^3r \tag{2.3}$$

which integrates all components within the RF coil volume, V. For the description of the MR measurement process, the time evolution of each spin isochromate is completely different and the problem is ideally suited for numerical treatment with parallel processing. In MRI, the measurement process is expressed by the MRI sequence[2] which describes the timing of pulsed currents in various coils to produce the RF field for excitation and the gradient field for encoding spatial information to the phase/frequency of the MR signal: $\varphi(\mathbf{r},t) \;=\; \gamma\int\mathbf{G}(t)\,dt\cdot\mathbf{r} \;\equiv\; \mathbf{k}(t)\cdot\mathbf{r}$. Inserting this expression in Eq. (2.3) shows that the MR signal, $S(t)$, can be reordered as a function of the k-vector, $S(\mathbf{k})$. Then, the MR image, $M_r(\mathbf{r})$, is given by the Fourier transformation of the acquired signal. Thus, the timing of the gradient pulses defines the so called k-space trajectory, $\mathbf{k}(t)$, along which the necessary information for image generation is acquired during the measurement. In

general the MR image, $M_r(\mathbf{r}) = M_r(M_0(\mathbf{r}), T_1(\mathbf{r}), T_2(\mathbf{r}))$, depends on the timing of the MRI sequence through the time evolution of the Bloch equation. In this way, images with various desirable properties such as soft-tissue contrast can be obtained. The image contrast is based on differences in the proton density, M_0, and/or the relaxation times, T_1, T_2, in biological tissue components; this unique feature provides a variety of medical imaging applications and it is the basis for the success of MRI in medicine.

2.2 Selective Excitation

Multidimensional, spatially-selective excitation[4] is an important concept of growing interest in MRI, e.g. in the field of *in vivo* spectroscopy or for the challenging task of correcting subject-induced B_1 field inhomogeneities at ultra high fields[5]. However, thus far the computation of these pulses is based on a simplified physical model neglecting relaxation and off-resonance effects during the pulse. Under such conditions, the calculation of the unknown RF pulse shape, B_1, from a given target excitation pattern, M_p, reduces to a linear system:

$$M_p(\mathbf{r}) \;=\; \mathrm{i}\gamma M_0(\mathbf{r}) \int_0^T B_1(t) \exp[\mathrm{i}\mathbf{r}\cdot\mathbf{k}(t)]\; dt \tag{2.4}$$

where M_0 is the equilibrium magnetisation and $\mathbf{k}(t)$ is a given k-space trajectory. A spatially and temporally discrete version of Eq. (2.4) can be solved for B_1 by suitable generalised matrix inversion methods. In contrast, the present approach provides a numerical method to design selective pulses under realistic experimental conditions. The simulator computes the effective transverse magnetisation, $M_e(\mathbf{r}, t)$, which is used to correct the RF pulse in order to account for effects not governed by Eq. (2.4). Thus, a minimisation problem is formulated and individually solved for all time steps $n\Delta t \;\; (n = 1, ..., N)$, where $N\Delta t = T$ equals the pulse length:

$$||M_p(\mathbf{r}) \;-\; M_e[M_0(\mathbf{r}), T_1(\mathbf{r}), T_2(\mathbf{r}), \Delta\omega(\mathbf{r}), B_1(t), \mathbf{k}(t)]\,|| \;=\; \min_{B_1} \tag{2.5}$$

Here, the difference between the desired magnetisation pattern and the effective magnetisation pattern is minimised with respect to the real and the imaginary parts of $B_1 = B_{1x} + \mathrm{i}B_{1y}$. The starting point (B_{1x}, B_{1y}) for each of the N consecutive 2D minimisation problems is taken from the solution of Eq. (2.4). Note that the temporal sampling of B_1 is taken from the discrete version of Eq. (2.4), whereas the time evolution of the effective magnetisation is computed with much higher accuracy by the simulator, i.e. within each interval Δt the Bloch equation is individually solved for each spin to compute the norm in Eq. (2.5) for the minimisation routine. Once a minimum is found for the n-th step of the RF pulse, the final magnetisation states are taken as the starting condition for the next step.

3 Software Engineering

3.1 General Design of the Framework

The software design of JEMRIS had to meet two competing premises: Obviously, the object design had to reflect the physical world. At the same time the objects and members

were to remain highly maintainable, reusable and yet easy to handle.
A simple reflection of the class hierarchy is presented in Fig. 1. The object model in JEMRIS is based on the definition of the four main classes: *Sample*, *Signal*, *Model*, and *Sequence*, respectively. The *Sample* class describes the physical properties of the object, currently defined by the set $P = (M_0, T_1, T_2)$ at every spatial position $\mathbf{r} = (x, y, z)$ of the sample. Since MPI has no functionality for object serialisation, send/receive of sub-samples is realised with appropriate MPI data-types. Similarly, the *Signal* holds information about the 'MR signal' consisting of the net magnetisation vector $\mathbf{M}(t) = [M_x(t), M_y(t), M_t(t)]$ at every sampled time point. MPI functionality is implemented in the same way as for the *Sample* class. The *Model* class contains the functionality for solving the physical problem.
The design of the *Sequence* class and its underlying sequence framework proved to be a very demanding task. It supplies the most complex part of the simulator. For this a novel object-oriented design pattern for MRI is introduced which derives the basic parts, *Sequence*s (loopable branch nodes) and *Pulse*s (leaf nodes) from an abstract *Module* class. To further reduce the complexity in, and promote the encapsulation of, the objects in this part of the framework, an abstract factory approach using prototypes is implemented as suggested by Gamma et al[6].
A sequence represents a left-right ordered tree. Fig. 2 depicts by example how the different modules of the well-known EPI sequence[2], loops and pulses, can be arranged in an ordered tree. Trees can be effectively accessed by recursion and are very well suited for building as well as accessing values of the sequence. The atomic sequences are containers for the pulses and, thus, display the functionality to emit pulses of various types. The pulses themselves are defined in a seceded class hierarchy.
The sequence tree is internally handled via XML. Thus, sequences themselves in turn can

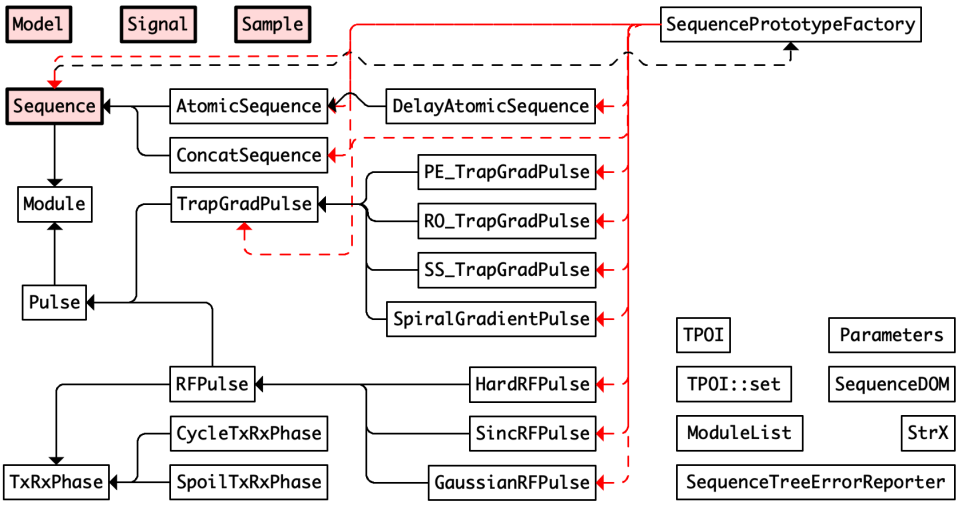


Figure 1. *Class hierarchy of the basic JEMRIS components, Sample, Signal, Model, and Sequence class, respectively.*

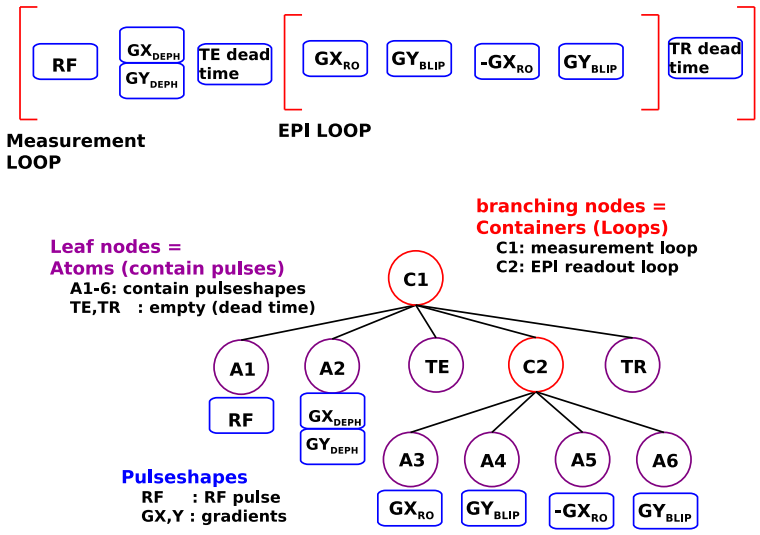


Figure 2. **Top:** *Sketch of a native EPI pulse sequence diagram, consisting of an outer loop (e.g. slices) in which the RF excitation module, the dephasing gradients, dead times, and the inner loop for the EPI readout are nested.* **Bottom:** *Representation of the same diagram with a left-right ordered tree. Branching nodes represent loops in which other modules are concatenated. Leaf nodes represent the modules in which the actual pulse shapes are played out.*

natively be read from, or written to, XML files. The Xerces C++ parser provided by the Apache Software Foundation[a] is used for serialising and de-serialising the sequence object. Further, a GUI was implemented in MATLAB[b] which allows one to interactively build as well as manipulate the sequence tree, view the pulse diagram, and perform the simulation. For the latter, MATLAB calls JEMRIS via ssh on the remote HPC site.

3.2 Parallel Implementation

Pseudo code of the parallel workflow of JEMRIS is shown in the parallel algorithm 1. (Comments are written with a reduced font size.) The (mostly sleeping) master process subdivides the problem in the beginning and harvests the result at the end. The basic functionality of the slave processes is hidden in the *Solve()* method of the *Model* class, where the solution for each spin is obtained during a recursive travel through the sequence tree. All functionality for introducing off-resonance effects is hidden in lines 9 and 19 of the slave process. Settings about this functionality, as well as the setting of the sample, is provided in the simulation XML file, which is parsed in the beginning.

The computation of selective excitation pulses utilising the simulator is shown in the parallel algorithm 2. The input parameters for the discrete problem consisting of N_t steps and the appendant solution of the linear problem in Eq. (2.4) is read from an XML file. After splitting the sample and the target pattern, the slave processes instantiate a sequence object which consists of a root node and N_t child nodes, each representing one time segment with

[a] http://xml.apache.org/xerces-c/
[b] http://www.mathworks.com

Algorithm 1. *Simulation Routine*

initialise N+1 MPI processes (n=0,...,N)

MASTER process (p=0)

1: parse simulation XML file
2: instantiate *Sample* object
3: split *Sample* into N *SubSample* objects
4: **for** n=1 to N **do**
5: **send** n-th *SubSample* to n-th slave
6: **end for**

7: instantiate *Signal*
8: **for** n=1 to N **do**
9: **receive** *SubSignal* from n-th slave
10: *Signal* += *SubSignal(n)*
11: **end for**
12: save *Signal*

SLAVE processes (p=1,...,N)

1: parse simulation XML file
2: instantiate *Model* object
3: parse sequence XML file
4: instantiate *Sequence* object
⟶ 5: **receive** *SubSample*

- functionality of *Model::Solve()* -

6: instantiate *SubSignal*
7: N_s = *SubSample::getNumOfSpins()*
8: **for** $s = 1$ to N_s **do**
9: instantiate *Spin(s)*
10: *Sequence::Prepare()*
11: N_r = *Sequence::getNumOfRepetitions()*
12: N_c = *Sequence::getNumOfChildren()*
13: **for** $r = 1$,to,N_r **do**
14: **for** $c = 1$,to,N_c **do**
15: **if** $N_c > 0$ (case *ConcatSequence*) **then**
16: *Sequence=Sequence.getChild(k)*
17: go to line 10: - recursion -
18: **else** {compute solution}
19: instantiate CVODE solver
20: obtain *Solution*
21: **end if**
22: **end for**
23: **end for**
24: *SubSignal* += *Solution*
25: **end for**
⟵ 26: **send** *SubSignal*

a constant RF pulse (B_{1x}, B_{1y}). Then, within a loop over the time steps, the master solves a conjugate gradient search to minimise the difference between the target pattern and the excited magnetisation by varying (B_{1x}, B_{1y}). Thus, each slave repeatedly calculates its contribution to this difference each time with a new RF pulse at the current segment. Once a minimum is found, the master stores the result and continues with the next time step at which the slaves proceed with the final magnetisation state of the previous step.

4 Results

4.1 Benchmarks

The simulations were performed on a 16 dual-core CPU Opteron cluster. This can be seen as small-scale HPC; it is sufficient to perform 2D MRI simulations within minutes. Thus, the simulations presented here reduce to 2D examples, though the simulator is also able to treat 3D simulations. The performance of the simulator is depicted in Fig. 3. The results from sequential program profiling show, that most of the time is spent in the highly optimised CVODE libraries. However, a significant amount of time, 28 %, is spent in the *Sequence.getValue()* function, leaving room for future optimisation of data retrieval from

Algorithm 2. *Selective Excitation Routine*

initialise N+1 MPI processes (n=0,...,N)

MASTER process (p=0)		SLAVE processes (p=1,...,N)
1: parse selective excitation XML file		1: parse selective excitation XML file
		2: instantiate *Sequence* and *Model*
2: split and **send** *Sample* and *Target*	$\longrightarrow$	3: **receive** *SubSample* and *SubTarget*
- loop over timesteps -		4: **for** t=1 to N_t **do**
3: **for** $t = 1$ to N_t **do**		5: bool *bCont* = true
4: bool *bCont* = true		
		- repeatedly call simulation routine -
- 2D conjugate gradient search for $B_1(t)$ -		6: **while** *bCont* **do**
5: **while** *bCont* **do**		7: *Model::Solve()* in time interval $[t-1, t]$
6: **for** $n = 1$ to N **do**		- send difference between target and excited magnetisation -
7: **receive** ε_n from n-th slave	$\longleftarrow$	8: **send** $\varepsilon_n = \sum \lvert M_p(x_i) - M_e(x_i, t) \rvert$
8: **end for**		
9: *bCont* = $[\sum \varepsilon_n \neq \min]$		
10: select next B_1 of the gradient search		
11: **broadcast** (B_{1x}, B_{1y}) and *bCont*	$\longrightarrow$	9: **receive** (B_{1x}, B_{1y}) and *bCont*
12: **end while**		10: *Sequence.getChild(t)::setB1*(B_{1x}, B_{1y})
		11: **end while**
13: store final $B_1(t)$		
		12: **end for**
14: **end for**		

the sequence tree. As expected, parallel performance scales nearly perfectly, i.e. the speed increases linearly with the number of processors.

4.2 MRI Simulation Examples

Fig. 4 shows an example of the GUI for sequence development, which allows interactive building of the sequence tree and various representations of the corresponding pulse diagram. The right part of Fig. 4 depicts the simulation GUI, showing an example of a simple

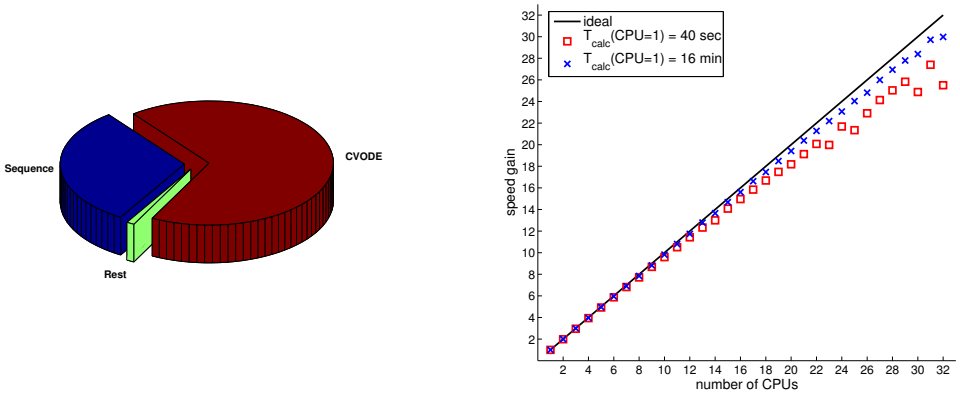


Figure 3. *JEMRIS performance testing.* **Left:** *Sequential profiling shows that ≈ 70 % of the computing time is spent in the highly optimised CVODE library.* **Right:** *The speed gain due to parallelisation (right) is close to optimal for large scale problems, as expected.*

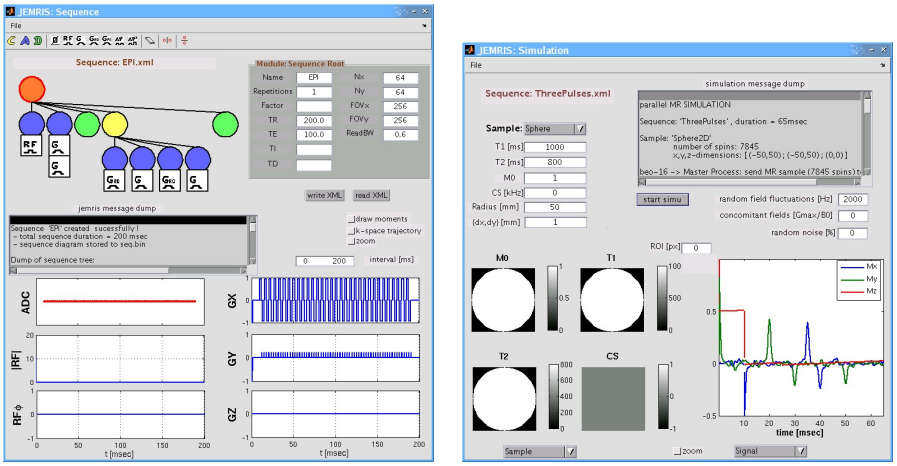


Figure 4. **Left:** *Example of the JEMRIS GUI for sequence development showing the EPI sequence already shown in Fig. 2. In the top left, the sequence is interactively built and on the right the parameters of the individual nodes are set. In the bottom, the corresponding sequence diagram is shown. Possible error output is given in the slider-window.* **Right:** *Example of the JEMRIS GUI for simulations: a simple three pulse sequence (60°-90°-90° at 0-10-25 ms) applied to a homogeneous sphere under strong field inhomogeneities results in the well-established five echoes: three spin echoes, one reflected spin echo, and the stimulated echo.*

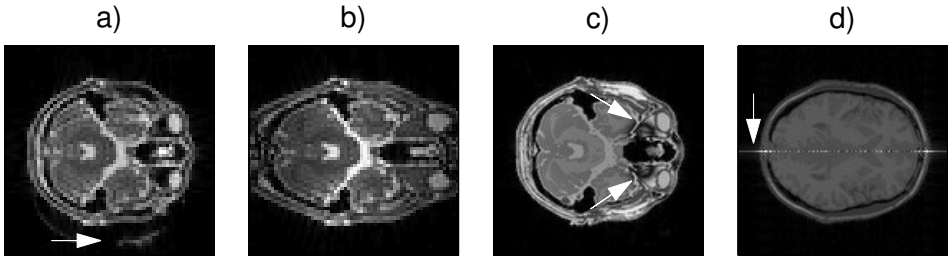


Figure 5. *Example of artefact simulations on a human head model:* **a** *EPI with chemical shift.* **b** *EPI distortions due to a nonlinear read gradient.* **c** *TrueFISP banding artefacts resulting from susceptibility-induced field inhomogeneities.* **d** *Artefact in a spin echo sequence with a long refocusing pulse in the order of T_2 of the sample.*

3-pulse experiment without any gradients involved. However, strong random field variations are applied yielding a very strong T_2^* effect, i.e. signal decays rapidly to zero. The well-established five MR echoes generated by such a three pulse experiment are visible. The tracking of magnetisation can be accurately performed at any desired time scale also during the application of RF pulses, since a generally valid numerical solution is computed. Examples of MRI artefact generation are given in Fig. 5: a) EPI simulation considering chemical shift effects yield the prominent fat-water shift in MRI; b) malfunctioning MR scanner hardware simulation with a nonlinear gradient field results in a distorted EPI image; c) TrueFISP simulation including susceptibility-induced field inhomogeneity yields the well-known banding artefacts in the human brain; d) artefact in spin echo imaging due

Figure	Number of Spins	Sequence	calculation time [min]
4 (right)	10.000	ThreePulses	0.1
5 a),b)	60.000	EPI	1
5 c)	60.000	TrueFISP	10
5 d)	60.000	Spin Echo	10

Table 1. *Calculation times of the simulation examples. Note, that the calculation time strongly depends on the type of imaging sequence.*

to a very long inversion pulse exciting transverse magnetisation. The corresponding calculation times are given in Table 1. Note that the last example, Fig. 5 d), cannot be realised with any simulator relying on analytical solutions of the Bloch equation due to neglect of relaxation effects during the RF pulse. The exact simulation of the simultaneous occurrence of RF pulses and imaging gradients with JEMRIS is the foundation for the derivation of new selective excitation pulses presented in the next section.

4.3 RF Pulse Design for Selective Excitation

For the demonstration of selective excitation, a homogeneous spherical object and a desired target magnetisation pattern were defined and the RF pulses of the common model and the new model were computed according to Eq. (2.4) and (2.5), respectively. Simulations were performed for $\approx$ 30,000 spins. The results are depicted in Fig. 6. Note that the excitation patterns are not the result of a (simulated) imaging sequence but they are the effective

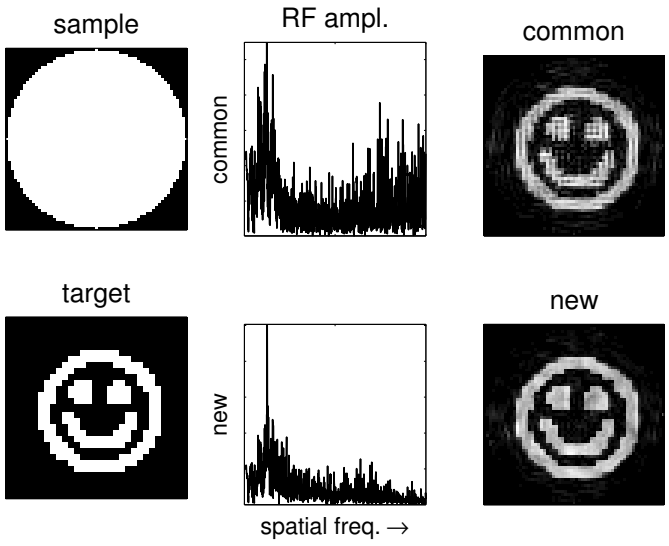


Figure 6. *Example of multidimensional excitation.* **Left Column:** *2D MR sample and the target pattern for selective excitation.* **Middle Column:** *RF pulses computed with the common approach and with the proposed method based on the JEMRIS framework.* **Right Column:** *Simulation results showing the excited spin system after applying the corresponding HF pulses.*

patterns of excited spins directly after the pulse. In comparison, the optimised RF pulse computed with the new approach shows fewer higher spatial frequency components than the common approach. This is a well-known problem of the common linear approach, leading to high frequency artefacts on selectively excited images, as can be seen from the excited target pattern in Fig. 6. In comparison, the new approach excites a pattern that better approximates the target pattern.

5 Conclusion

JEMRIS performs (classically) general MRI simulations. It allows time tracking of the net magnetisation, separation of different effects, and therefore accurate investigation of MR samples (e.g. implants) and MRI sequences under a variety of experimental conditions. The parallel HPC implementation allows for the treatment of huge spin ensembles resulting in realistic MR signals. On small HPC clusters, 2D simulations can be obtained on the order of minutes, whereas for 3D simulations it is recommended to use higher scaling HPC systems. As shown, the simulator can be applied to MRI research for the development of new pulse sequences. The impact of these newly designed pulses in real experiments has to be tested in the near future. Further, the rapidly evolving field of medical image processing might benefit of a "gold standard", serving as a testbed for the application of new methods and algorithms. Here, JEMRIS could provide a general framework for generating standardised and realistic MR images for many different situations and purposes. This aspect is especially interesting, since the image processing community usually has adequate access to HPC systems.

References

1. H. Benoit-Cattin, G. Collewet, B. Belaroussi, H. Saint-Jalmes and C. Odet, *The SIMRI project: a versatile and interactive MRI simulator*, Journal of Magnetic Resonance, **173**, 97–115, (2005).
2. E.M. Haacke, R.W. Brown, M.R. Thompson and R. Venkatesan, *Magnetic Resonance Imaging: Physical Principles and Sequence Design*, (Wiley & Sons, 1999).
3. S. Cohen and A. Hindmarsh, *CVODE, a stiff/nonstiff ODE solver in C*, Computers in Physics, **10**, 138–143, (1996).
4. J. Pauly, D. Nishimura and A. Macovski, *A k-space analysis of small tip-angle excitation pulses*, Journal of Magnetic Resonance, **81**, 43–56, (1989).
5. T. Ibrahim, R. Lee, A. Abduljalil, B. Baertlein and P. Robitaille, *Dielectric resonances and B_1 field inhomogeneity in UHFMRI: computational analysis and experimental findings*, Magnetic Resonance Imaging, **19**, 219–226, (2001).
6. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*, (Addison Wesley, 1995).

A Load Balancing Framework in Multithreaded Tomographic Reconstruction

José Antonio Álvarez, Javier Roca Piera, and José Jesús Fernández

Departamento de Arquitectura de Computadores y Electrónica
Universidad de Almería, 04120 Almería, Spain
E-mail: {jaberme, jroca, jose}@ace.ual.es

Clusters are, by now, one of most popular architectures. Hiding latencies as well as getting an optimal assignment of processors, are two issues for many scientific applications, specially if non dedicated clusters are used. Traditionally, high performance scientific applications are parallelized using MPI libraries. Typical implementations of MPI, minimize dynamic features required to face latencies or shared resource usage. Switching from the MPI process model to a threaded programming model in the parallel environment, can help to achieve efficient overlapping and provide abilities for load balancing. Gains achieved by $BICAV$ (our research group's tomographic reconstruction software) in a multithreaded framework, are exposed.

1 Introduction

Multithreaded programming[1] provides a way to divide a program into different separated pieces that can run concurrently. Using user level threads[2,3] in places where concurrence can be exploited, would allow us to achieve latency hiding, better scalability, skills to avoid overhead on processors due to faster context switching and, also, key abilities to migrate threads for load balancing purposes. In the design of a parallel and multithreaded strategy for applications related to image processing, such as 3D tomographic image reconstruction algorithms[4], data distributions have to be carefully devised so locality is preserved as much as possible. Specially in systems like non-dedicated clusters, where the workload needs to be dynamically reassigned. The strategy presented, RakeLB (inspired on Rake[5] algorithm), has been implemented following a centralized scheme and seizes the facilities provided by AMPI [6,7], which is the top level layer of the Charm [8] framework. AMPI embeds MPI processes into user level threads (*virtual processors* or *VPs* in Charm framework), allowing more than one active flow of control within a process, therefore aspects like context switching and migration of tasks are possible with a relative efficiency. Applications like $BICAV$ can run on non-dedicated clusters without significant performance loss, if latency hiding and adaptability are achieved and combined.

This work presents and analyzes results obtained by $BICAV$[9] when it was ported to a multithreaded environment. Section 2 describes $BICAV$, Section 3 exposes gains for the multithreaded version of $BICAV$ due to latency hiding. Section 4, describes the load balancing strategy, RakeLB, compared to more general load balancing strategies. Section 5 evaluates RakeLB and finally, conclusions are presented in Section 6.

2 Iterative Reconstruction Methods: BICAV

Series expansion reconstruction methods assume that the 3D object, or function f, can be approximated by a linear combination of a finite set of known and fixed basis functions,

with density x_j. The aim is to estimate the unknowns, x_j. These methods are based on an image formation model where the measurements depend linearly on the object in such a way that $y_i = \sum_{j=1}^{J} l_{i,j} \cdot x_j$, where y_i denotes the i^{th} measurement of f and $l_{i,j}$ the value of the i^{th} projection of the j^{th} basis function. Under those assumptions, the image reconstruction problem can be modeled as the inverse problem of estimating the x_j's from the y_i's by solving the system of linear equations aforementioned. Assuming that the whole set of equations in the linear system may be subdivided into *B* blocks, a generalized version of component averaging methods, *BICAV*, can be described. The processing of all the equations in one of the blocks produces a new estimate. All blocks are processed in one iteration of the algorithm. This technique produces iterates which converge to a weighted least squares solution of the system. A volume can be considered made up of 2D slices. The use of the spherically symmetric volume elements, blobs[10], makes slices interdependent because of blobs' overlapping nature. The main drawback of iterative methods are their high computational requirements. These demands can be faced by means of parallel computing and efficient reconstruction methods with fast convergence.

An adaptive parallel iterative reconstruction method is presented. The block iterative version of the component averaging methods, $BICAV$, has been parallelized following the Single Program Multiple Data (SPMD) approach[9]. The volume is decomposed into slabs of slices that will be distributed across the nodes (for MPI) or across the threads (for AMPI). Threads can, thereby, process their own data, however, the interdependence among neighbour slices due to the blob extension makes necessary the inclusion of redundant slices into the slabs. In addition, there must be a proper exchange of information between neighbour nodes. These communication points constitute synchronization points in every pass of the algorithm in which the nodes must wait for the neighbors. The amount of communications is proportional to the number of blocks and iterations. Reconstruction yields better results as the number of blocks is increased.

3 Latency Hiding with User Level Threads for BICAV

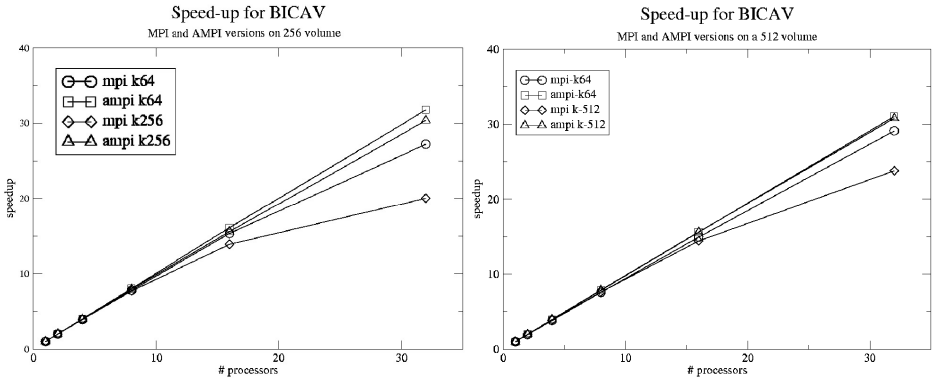
An analysis on the application's communication pattern has been carried out. Such analysis provides estimations on improvements to be achieved if the concurrence is really effective. Concurrence using AMPI user level threads offers the advantage of having more virtual processors than physical processors. Therefore more than one virtual processor can coexist in a physical processor efficiently. Tests carried out showed gains obtained by the multithreaded version of $BICAV$ compared to those obtained by the MPI version where latency hiding technique was included with non blocking Sends/Recvs. Scaling experiences were also performed for both versions, varying the number of threads/processor and the number of processors. It is important to note that for $BICAV$, as the number of blocks (*K*) increases, the convergence of the algorithm is faster, but the amount of communications also increases. This scenario harms the MPI version whereas AMPI is expected to keep good performance. Two volumes were used for testing $BICAV$, a 256 volume and a 512 volume.

All experiences were performed on *Vermeer*, our research cluster. This cluster runs a Gentoo Linux, Kernel version was 2.4.32 SMP. Each of its *32* computing nodes have two Pentium IV xeon 3.06 Ghz with 512 KB L2 Cache and 2 GB of ecc ddr sdram. Nodes were connected with 1 Gb Ethernet links.

Table 1. % Relative differences between CPU and WALL times for k 256 and k 512

	K 256 (volume 256)		K 512 (volume 512)	
	MPI	AMPI	MPI	AMPI
Procs	%	%	%	%
2	2.9	0.1	2.8	0.0
4	3.5	0	3.2	0.0
8	5.6	0	4.7	0.3
16	17.1	0.8	9.7	0.7
32	62.4	1.5	32.3	0.2

The relative differences between cpu and wall times are shown in Table 1, for both problem sizes, using the higher possible value for *K*, for each case. For AMPI, wall and cpu times are alike, which means that cpu was mostly in use, in contrast to the MPI version in which differences turn out to be significant. Taking into account that there are neither I/O operations nor function/method calls apart from those related to the algorithm itself, it can be said that for our multithreaded version, the concurrency is maximized. To further analyze the gains obtained by the multithreaded version of $BICAV$, the speedup was computed for the tests carried out for Table 1, see Fig. 1.



(a) 256 volume (b) 512 volume

Figure 1. Speedup in Vermeer cluster for 256 volume (left) and 512 volume (right)

In Fig. 1, wall times for MPI and AMPI (128 vps) versions are shown, for several numbers of blocks K and for 256 and 512 volume sizes. It can be observed that below a threshold of K=64 both versions seem to behave similarly, showing slight improvement in AMPI. But above that threshold, and as K increases, AMPI behaves better than MPI especially for more than *16* processors. $BICAV$'s AMPI version is getting benefits from the hidden concurrency. This means that the amount of communications needed due to a high value of *K* can be overlapped efficiently. These speedup curves show that with MPI, $BICAV$ loses the linear speedup when using more than eight processors. However, AMPI

keeps its speedup almost linear.

Once an optimal number of processors is found, an optimal number of threads per processor is also desirable, for both volumes. It was found that a number of threads ranging from four to eight were optimal. This means that having fewer threads than four, concurrency may not be exploited, on the other side, having a higher number than advised, performance can be degraded due to thread management overheads.

For non-dedicated clusters, concurrency, if exploited correctly, can play an important role for performance. Therefore, this criteria is implemented as a complement to the load balancing strategy. AMPI offers a threaded framework in which latencies due to communications can be handled. The following section shows how the load balancing strategy, RakeLB, takes advantage of latency hiding.

4 Adaptability

In general, load balancing strategies do not take into consideration data locality. Consequently, pieces of work sharing data can be placed on different processors. The strategy for preserving locality, RakeLB[5], was implemented as a centralized load balancer. For RakeLB, two processors are neighbours if each one has, at least, one thread with data in common. RakeLB has been implemented taking advantage of facilities provided by AMPI where the migration of workload among nodes is carried out in terms of threads. Employing user level threads make load balancing issues easier in SPMD applications. Each thread owns an universal ID, the rank. This rank establishes an order relationship between them. When computation is ignited, each thread picks up a chunk of data to work on, thereby threads with consecutive ranks will need, at some point in time to communicate one another. Therefore, thread migration decisions should conserve as much as possible the established thread relationship. To maintain the aforementioned order, a FIFO class structure has been devised. When threads migration is advised, those with the minimal data locality restrictions in the node will be migrated.

RakeLB behaviour has been contrasted with standard centralized, dynamic load balancing strategies, like Refine and Greedy[11]. Greedy strategy does not consider any previous thread-processor assignment, simply builds two queues, processors and threads, and reassigns load to reach average. Refine, in contrast, only migrates threads from overloaded processors until the processor load is pushed under average.

4.1 Implementation

The purpose of this load balancing algorithm is to redefine the initial threads distribution by applying a certain strategy, sensible to the criteria stated in Section 3. Inputs for strategy evaluation are provided by Charm (threads per processor, load per processor, cpu and wall time per thread,. . .). For preserving data locality, RakeLB has to determine the set of available processors in the cluster to create logical links between processors that host *correlative* sets of threads (according to the established order relationship). RakeLB determines, for each processor, its previous and next available processors, from a logical point of view. A procedure called $LinkProcs$ has been developed to provide such a facility. $LinkProcs$ is responsible of maintaining processors virtually linked, providing therefore, a consistent way to prevent migrations between sets of non correlative threads. This procedure also

Algorithm 3. : RakeLB

1. **for** $i = 1$ **to** $numAvail - Resd$ — *First phase: $numAvail - Resd$ iterations*
2. **for** $j = 1$ **to** NP — *Compute total load as*
3. $Burd_j = PRCS_j.bkgLoad + PRCS_j.cmpLoad$; — *bg load + threads' load*
4. **endfor**
5. **for** $j = 1$ **to** NP
6. **while** ($Burd_j > AvLoad$ & $numThreads_j > 0$) — *while proc is overloaded*
7. deAssign($THRD.id, PRCS_j$); — *move away threads, preserving locality*
8. Assign($THRD.id, PRCS_j.next$); — *to next processor*
9. $numThreads_j = numThreads_j - 1$; — *update #threads*
10. $Burd_j = Burd_j - THRD.cmpLoad$; — *update load*
11. **endwhile**
12. **endfor**
13. **endfor**
14. **for** $i = 1$ **to** $Resd$ — *Second phase: $Resd$ iterations*
15. **for** $j = 1$ **to** NP
16. $Burd_j = PRCS_j.bkgLoad + PRCS_j.cmpLoad$;
17. **endfor**
18. **for** $j = 1$ **to** NP
19. **while** ($Burd_j >$ AvLoad + ΔLoad &$numThreads_j > 0$) — *note $\Delta Load$*
20. deAssign($THRD.id, PRCS_j$);
21. Assign($THRD.id, PRCS_j.next$);
22. $numThreads_j = numThreads_j - 1$;
23. $Burd_j = Burd_j - THRD.cmpLoad$;
24. **endwhile** — *During migrations, Assign and Deassign methods*
25. **endfor** — *together with $LinkProcs$ and the FIFO structure*
26. **endfor** — *devised, preserved data locality for threads*

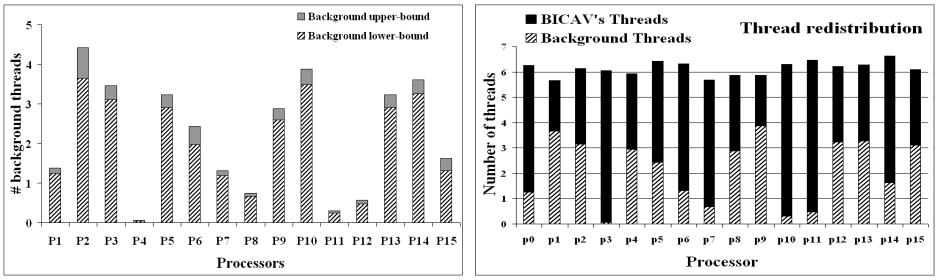
maintains useful information such as the number of available processors ($numAvail$), the average load ($AvLoad$) which is worked out accumulating all the load in each available processor. Hence $AvLoad$ is the main parameter concerning thread migration, together with $Resd$, which is the remainder obtained when calculating $AvLoad$. Another important parameter is the total number of $BICAV$ threads, $NThreads$.

RakeLB convergence is assured in $numAvail$ iterations. These $numAvail$ iterations are performed in two main steps. The first step consists of $numAvail - Resd$ iterations, the second step consists of *Resd* iterations. At the first step, RakeLB examines processors whose load is over the average ($AvLoad$) and determines which threads migrate to *next* processor. For every iteration, the total load in each processor is re-computed and stored in the variable $Burd$. There is an upper-bound limit, $numThreads$, to the number of threads that can be migrated from an overloaded processor. Note that the total number of threads in the application is given by $NThreads = \sum_{i=1}^{numAvail} numThreads_i$. At the second step only those processors whose load is over $AvLoad$+Δ $Load$ are involved. Δ $Load$ is set, during these experiences, to the minimum thread load.

5 Evaluation of Load Balancing Strategies

Preserving data locality and minimizing latencies are two issues that the RakeLB strategy exploits. In this section, a study of RakeLB, GreedyLB and RefineLB (Non-Locality Load Balancing Algorithms) behaviour is presented. GreedyLB and RefineLB are implementations for Greedy and Refiner strategies.

$BICAV$ is used to evaluate the dynamic load balancing algorithms. $BICAV$ exhibits strong data dependence and hence emphasizes the influence of data locality preservation on the performance. Load balancer will be invoked just once.



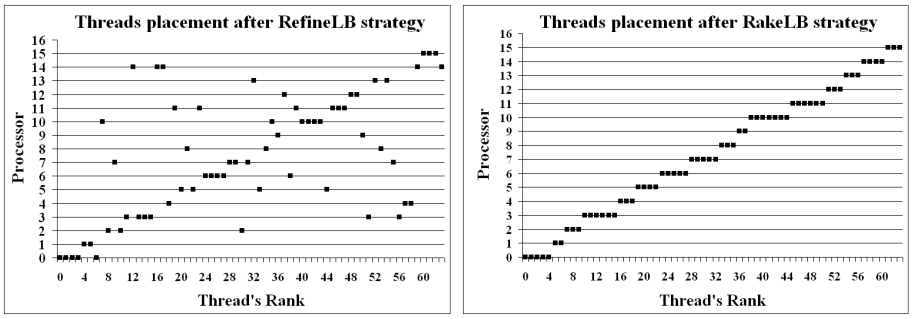
(a) Background (max and min values) per processor. (b) Threads after RefineLB and RakeLB strategies

Figure 2. Background load and resulting data redistribution after load balancing.

Background load scenarios were prepared using threads from a secondary application, in order to create a controlled imbalance. Fig. 2(a) exposes the number of background threads placed per processor. The background load evolution simulates other user's interactions with the cluster. For every experiment, the maximum value for the background activity is controlled by a parameter, $maxback$, adapted to the current number of $BICAV$'s threads per processor, that is, $NThreads \div numAvail$. Background load is assigned randomly to a processor, having a minimum value of zero and a maximum value of $maxback$, see Fig. 2(a). At the beginning, computing threads for the application are evenly distributed among nodes.

Fig. 2 shows how the strategies under study react to load imbalanced scenarios. As discussed in Section 3, from sixteen processors $BICAV$ show significant divergences between the AMPI version and the MPI version. A number ranging from *4* to *8* vps per processor was advised. Thus the sixteen processors and sixty four threads ($NThreads = 64$) case is selected for testing load balancing strategies.

After migration, see Fig. 2(b), RakeLB and RefineLB reacted alike. The decision that GreedyLB took follows the trend established by RakeLB and RefineLB, although its distribution diverges slightly from that taken by its counterpart strategies. What can be deduced from these figures is that all three strategies react in a very similar way to the effect of the background load. There's a key point, viz. *data dependency*. An analytical proof for what is asserted in Fig. 2 was performed employing the standard deviation of the whole system load, normalized to the average load, σ. The initial imbalance value reflected by σ was over 0.5. After applying the balancing strategies a similar σ value (0.051 for GreedyLB



(a) Disorder after RefineLB (b) Locality preservation after RakeLB

Figure 3. Placement of threads

Table 2. Walltime ratio and overlap ratio for $BICAV$

	Greedy	Refine	Rake
Cputime / Walltime ratio	0.34	0.52	0.70
Master's Walltime (% relative to Greedy)	100	66.59	50.29

and 0.045 for RakeLB and RefineLB) was achieved, but there's a key point *data dependency*.

In Fig. 2(b), it can be seen that after the load balancer is done, a complete homogeneous load distribution is achieved (σ almost zero). Although load balancers strategies achieve an almost equal load distribution (alike for RakeLB and Refine) it is important to note that these distributions are completely different in terms of data locality, see Fig. 3, aspect that turns out to be an issue for $BICAV$'s performance. Only RefineLB and RakeLB figures are shown to state that although both σ values are similar, data distributions are different. GreedyLB distribution was remarkably messy.

Maintaining the advisable number of neighboring threads in the same processor as RakeLB does, leads to a performance improvement in the application. This performance improvement begins to be much more significant for $BICAV$ when balanced with the RakeLB strategy in contrast with the cases where it is balanced with Refine or Greedy strategies, from the case where sixteen processors are used, see Table 2. This is the reason why in general terms, with RakeLB, $BICAV$, is faster on *vermeer* cluster. This is found in the combination of both issues under study. With RakeLB, latencies are lower than with any of the other two strategies, having enough computation to overlap communication, which is not the case with RefineLB and GreedyLB. This overlap is seized by threads sharing the same processor to advance computation.

6 Conclusions

Two concepts, adaptability and latency hiding abilities, were studied together, holding the hypothesis that for scientific applications where data is under a tight data locality rela-

tionship, both could be combined in such a way that executions on non dedicated clusters could obtain good results indeed in contrast to not considering these concepts. On the one hand, the use of user level threads allowed multiple flows of control in one processor, this characteristic together with their fast context switching operation times, makes it very easy to hide communication operations with other thread computations. On the other hand, load balancing with strategies that maintains a thread's neighbourhood -regarding data locality- avoids overloaded processors that harm the performance of the applications. If such strategies, as shown, maintain data locality, then latencies to be hidden are reduced, having as only handicap the possible overhead caused by the thread's management. The dynamic load balancing strategy which preserves data locality, was proved to be efficient for applications like $BICAV$.

Acknowledgements

This work has been supported by grants MEC-TIN2005-00447 and JA-P06-TIC01426.

References

1. C. M. Pancake, *Multithreaded Languages for Scientific and Technical Computing*, Proceedings of the IEEE, **81**, 2, (1993).
2. S. Oikawa, H. Tokuda, *Efficient Timing Management for User-Level Real-Time Threads*, in: Proc. IEEE Real-Time Technology and Applications Symposium, 27–32, (1995).
3. G. W. Price, D. K. Lowenthal, *A comparative analysis of fine-grain threads packages*, Parallel and Distributed Computing, **63**, 1050–1063, (2003) College Station, Texas.
4. A. C. Kak, M. Slaney, *Principles of Computerized Tomographic Imaging*, (SIAM Society for Industrial and Applied Mathematics, 2001).
5. C. Fonlupt, P. Marquet, J. L. Dekeyser, *Data-parallel load balancing strategies*, Parallel Computing, **24**, 1665-1684, (1998).
6. Ch. Huang, O. Lawlor, L. V. Kale, *Adaptive MPI*, in: Proc. 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, Texas, 306–322, (2003).
7. Ch. Huang, G. Zheng, S. Kumar, L. V. Kale, *Performance evaluation of Adaptive MPI* in: Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (2006).
8. L. V. Kale, S. Krishnan, *Charm++: a portable concurrent object oriented system based on C++*, in: Proc. Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 91–108, (1993).
9. J. Fernández, A. F. Lawrence, J. Roca, I. García, M. H. Ellisman, J. M. Carazo, *High performance electron tomography of complex biological specimens*, Journal of Structural Biology, **138**, 6–20, (2002).
10. S. Matej, R. Lewitt, G. Herman, *Practical considerations for 3-D image reconstruction using spherically symmetric volume elements*, IEEE Trans. Med. Imag., **15**, 68–78, (1996).
11. G. Aggarwal, R. Motwani, Zhu, *The load rebalancing problem*, Journal Algorithms, **60**, 42–59, (2006).

Parallel Algorithms

Parallelisation of Block-Recursive Matrix Multiplication in Prefix Computations

Michael Bader, Sebastian Hanigk, and Thomas Huckle

Institut für Informatik, Technische Universität München
Boltzmannstraße 3, 85748 Garching, Germany
E-mail: {*bader, hanigk, huckle*}*@in.tum.de*

We present a performance and scalability study on the parallelisation of a series of matrix multiplications within a quantum control problem. As a time-critical sub-task, all matrix products $A_1 \cdots A_k$ for matrices $A_1, \ldots, A_m$ have to be computed (prefix problem). The parallelisation can either be coarse-grain, where parallel prefix computation is used to concurrently execute the individual matrix multiplications; however, there is also the obvious fine-grain parallelisation approach to parallelise the subsequent matrix multiplications, and mixtures of these two principal approaches are possible. We compare two parallel multiplication algorithms that are based on block-structuring – SRUMMA and a modified, block-recursive approach based on Peano space-filling curves – and study their efficiency on a compute cluster with 128 processors. The Peano approach proves to be advantageous especially for moderately sized matrices. Hence, we compare the Peano algorithm's performance for coarse-grain, fine-grain, and hybrid parallelisation of the prefix problem. The Peano algorithm proves to be scalable enough to allow a purely fine-grain parallelisation of the prefix problem.

1 Introduction and Intended Application

The matrix multiplication task studied in this paper occurs within the parallelisation of an optimal-control-based quantum compiler[2,3]. There, the subsequent evolvement of the quantum states needs to be simulated and optimised. The respective iterative optimisation scheme is based on the so-called GRAPE-algorithm[3], which computes a sequence of evolutions $U^{(r)}(t_k)$ of the quantum system at given time steps t_k:

$$U^{(r)}(t_k) \;=\; e^{-i\Delta t H_k^{(r)}} e^{-i\Delta t H_{k-1}^{(r)}} \cdots e^{-i\Delta t H_1^{(r)}}, \tag{1.1}$$

which is actually the multiplication of a series of complex-valued matrices $e^{-i\Delta t H_k^{(r)}}$, which are computed as the exponential function of Hamiltonian matrices $H_k^{(r)}$. Hence, the computation of the matrices $U^{(r)}(t_k)$ is an instance of the so-called *prefix problem*, which is to compute all matrix products $A_1 A_2 \cdots A_k$ for $k = 1, \ldots, m$ for a given set of matrices $A_1, \ldots, A_m$.

For typical quantum control problems, the number of products m to be computed is in the range of 100–1000, and the size n of the individual matrices $A_1, \ldots, A_m$ grows exponentially with the number q of quantum bits, i.e. $n = 2^q$. Hence, on usual workstation computers, typically only systems with up to 7 or 8 spin qubits are treatable. On small and medium sized compute clusters, the treatable system size increases to about 10–12 qubits, however, we of course want to increase the treatable system size to as many qubits and timesteps as possible, which requires porting the system to parallel high-performance computers. Above all, it requires a scalability study on the parallelisation of the most time-consuming subtasks, one of which is the parallel prefix problem.

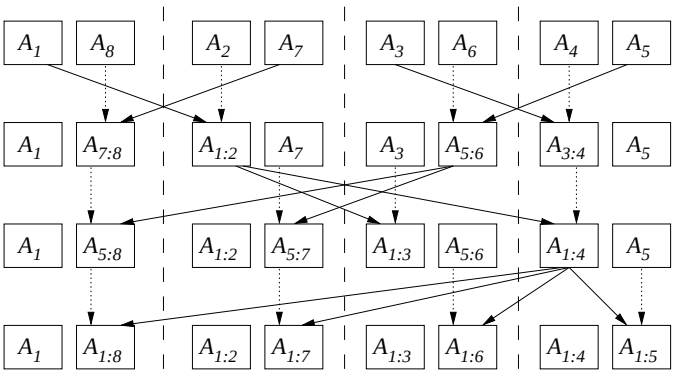


Figure 1. Multiplication tree in parallel prefix computation. $A_{k_1:k_2}$ denotes the product $A_{k_1} \cdots A_{k_2}$. The dashed lines define the processor scopes; solid lines denote communication between processors, and dotted lines indicate where intermediate results may be overwritten by the next tree level.

2 Parallel Prefix Computation vs. Parallel Matrix Computation

For the parallel prefix problem, there are two principal approaches for parallelisation. One obvious approach is to compute the matrix products

$$A_{1:k} := \prod_{\nu=1}^{k} A_\nu, \qquad k = 1, \ldots, m \tag{2.1}$$

step by step, as we would do on a single-processor computer, but use a parallel implementation of the individual matrix products. The other approach aims at a concurrent computation of the matrix products $A_{1:k}$, and is obtained by organising the multiplications via a tree-like scheme as given in Fig. 1. Then, up to $\frac{m}{2}$ multiplications can be performed in parallel. Hence, this *parallel prefix computation* scheme[5] can lead to a considerable speed-up on parallel hardware, even though the total computational work increases from $m-1$ to $\frac{m}{2}\log_2 m$ matrix multiplications. The parallel prefix scheme can actually be started on any level of the scheme. For example, in the scheme given in Fig. 1, we can first sequentially compute the products A_1 to $A_{1:4}$ and A_5 to $A_{5:8}$ on two (groups of) processors, and then compute the products $A_{1:5}$ to $A_{1:8}$ of the last level in parallel. In general, using sequential multiplication for the first l_1 levels, and the prefix scheme for the remaining l_2 levels, $l_1 + l_2 = \log_2 m$, requires $\left(m - 2^{l_1}\right) + l_2 \frac{m}{2}$ matrix multiplications.

In a straightforward implementation of the prefix scheme, at most $\frac{m}{2}$ processor can be used. Hence, on a massively parallel system, the individual matrix multiplications would need to be parallelised, as well. The performance of the combination of these two schemes will thus depend on several parameters, such as the number of available processors, the size of matrix blocks to be treated on one single processors, the speed of the communication network, and probably more. The parallel prefix scheme also contains a possible inherent communication bottleneck, because on the last level of the computation scheme, all processors need to access the intermediate result $A_{1:\frac{m}{2}}$ stored on a single processing unit (also, similarly, on the previous levels to only few intermediate results).

In a previous work[2], we showed that the parallel prefix scheme leeds to faster runtimes, once the parallel multiplication algorithm runs into heavy scalability problems, which can both result from too small problem sizes on each processor and from increased communication load due to the transfer of matrix blocks between processors. In this paper, we test two different, block-structured algorithms for parallel multiplication that promise better scalability.

3 Block-Recursive Matrix Multiplication

For the parallelisation of individual matrix multiplications, we consider two approaches, which both rely on a block-structured formulation of matrix multiplication. Consider, for example, a blockwise matrix multiplication such as

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix} . \tag{3.1}$$

Then the subblocks C_{ik} of the result matrix C are computed as $C_{ik} = \sum_j A_{ij} B_{jk}$. The block multiplications $A_{ij} B_{jk}$ may be performed in parallel. The SRUMMA algorithm[4], our first considered approach, uses an *owner-computes* scheme, which basically relies on the strategy to perform all computations for a given block C_{ik} on the same processor. The operand blocks A_{ij} and B_{jk} need to be transferred to the local memory of the respective processor. SRUMMA concentrates on hiding the respective communication costs by overlapping the current block multiplication with the prefetching of the next operand blocks.

Our second considered approach is based on a block-structured multiplication scheme based on the recursive substructuring of matrices and using a space-filling-curve layout of the matrices to profit from improved locality properties[1]. In each recursion step, we split the current block matrix into 3×3 submatrices:

$$\begin{pmatrix} A_0 & A_5 & A_6 \\ A_1 & A_4 & A_7 \\ A_2 & A_3 & A_8 \end{pmatrix} \begin{pmatrix} B_0 & B_5 & B_6 \\ B_1 & B_4 & B_7 \\ B_2 & B_3 & B_8 \end{pmatrix} = \begin{pmatrix} C_0 & C_5 & C_6 \\ C_1 & C_4 & C_7 \\ C_2 & C_3 & C_8 \end{pmatrix} . \tag{3.2}$$

Here the indices of the matrix blocks indicate the order in which the subblocks are stored in memory, which is obtained from the iterations of a Peano space-filling curve. The resulting block-recursive scheme (see Fig. 2) has excellent locality properties, which in our previous works[1] was mainly used to guarantee optimal cache efficiency of the algorithm. For example, it can be shown that on any set of j^2 elements, at least $\mathcal{O}(j^3)$ element multiplications are performed. In the present context, we utilise these locality properties to improve data locality during parallelisation of the subblock multiplications. The resulting algorithm will be called the *Peano algorithm* in the remainder of this paper.

4 Parallel Implementation

For both approaches, our parallelisation of the individual block matrix multiplications followed the SRUMMA idea[4]: The key concept is to use two sets of buffers, one of which is always used for the computation of a matrix block operation, while in the other buffers,

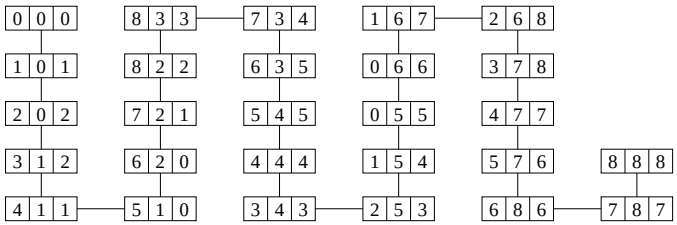


Figure 2. Optimal sequence of block multiplications for the matrix multiplication given in equation (3.2). Each triple (i, j, k) represents a (block) multiplication $C_k := C_k + A_i B_j$.

the matrix blocks for the next block multiplication are loaded. We use the ARMCI library[6] and its non-blocking, one-sided communication calls to ensure that communication and computation can be overlapped. Hence, during computation the required data for the next block operation may be fetched from a remote processor via a non-blocking RDMA transfer without loss of computing time for communication. We use a shared memory paradigm to store the matrices; however, via ARMCI we can enforce whether matrix blocks are stored on a common processing node, on the hardware-supported shared memory of an SMP node, or on distributed compute nodes.

In SRUMMA's *"owner computes"* strategy, each processor generates a task list of the block-matrix multiplications $C_{ik} = \sum_j A_{ij} B_{jk}$ to be executed on this processor. For the Peano algorithm, each processor generates a subsequence of the entire sequence of block multiplications without considering the localisation of the matrix blocks at all. However, the localisation properties of the underlying space-filling curve ensure that matrix blocks can be re-used frequently, and the number of accessed memory blocks is small in general. Of course, a write-back operation to store each updated C-block is now necessary, which we again try to overlap with the following block-multiplication.

Both, SRUMMA's block-structuring and the block-recursive Peano algorithm generate a sequence of "atomic" block multiplications, which are executed sequentially. For these atomic operations, the respective BLAS routine was called. As BLAS libraries typically reach their full performance only for sufficiently large matrix sizes, the choice of the size of the atomic matrix blocks has a strong impact on the parallel performance:

- Using a large block size generally increases BLAS performance; moreover, the ratio between computation and communication is more favourably for larger blocks, such that it is easier to hide communication costs.
- On the other hand, using large blocks leads to using a large number of processors (or groups of processors) for the tree operations in parallel prefix computation, which results in an increase of overall computational work (recall the $\log m$-factor) and also creates a possible communication bottleneck in the parallel prefix algorithm: in the last computation step, all processors will have to obtain the product $A_1 \cdots A_{m/2}$ as one of the operands. Moreover, the heavy-weight sequential operations may negatively affect load balancing.

Our flexible storage scheme allows to vary the *block size*, i.e. the size of the "atomic" matrix blocks for which the BLAS routine is called, over almost the entire possible range:

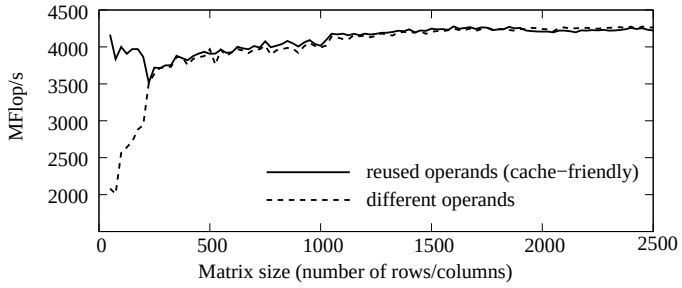


Figure 3. Performance of ACML's sequential dgemm routine on the Infiniband cluster (in MFlop/s, each element update `c[i,k] += a[i,j]*b[j,k]` is counted as 2 operations).

from distributing each matrix over all available nodes (and, hence, not using the parallel prefix scheme at all) via allocating matrices to a group of processors or SMP node up to assigning entire matrices to single CPUs and thus following the classical parallel prefix algorithm without parallelising individual matrix multiplications. Our goal is to find an optimal balance between the coarse-grain parallel prefix approach, and the more fine-grain parallelisation of the individual matrix multiplications.

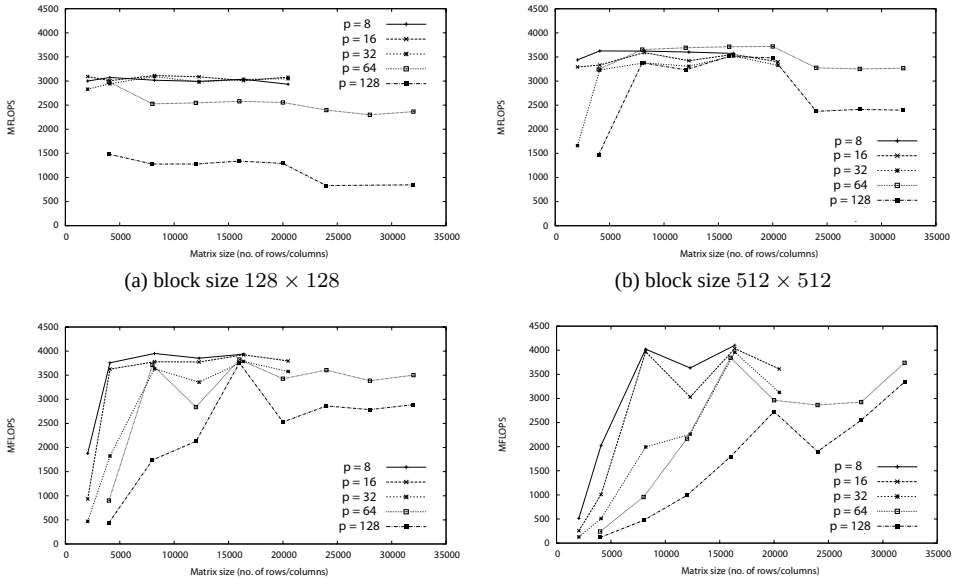
5 Performance Tests

The described algorithms were implemented and tested on an *Infiniband cluster* consisting of 36 Opteron nodes; each node contains four AMD Opteron 850 processors (2.4 GHz) that have access to 8 GB (16 GB for some of the nodes) of shared memory, and is equipped with one MT23108 InfiniBand Host Channel Adapter card for communication.

5.1 Performance of Parallel Matrix Multiplication

The performance of our two examined algorithms for parallel matrix multiplication, the Peano algorithm and SRUMMA, depends considerably on the efficiency of the underlying implementation of the individual block multiplications. Fig. 3 shows, for example, a performance benchmark we did for the vendor-supplied BLAS library ACML (AMD Core Math Library, version 3.6.0) on the Infiniband cluster. We compared the MFlop/s rate when performing a sequence of block multiplications on the same matrix blocks, and on different sets of matrix blocks. Especially for small matrix blocks, we can see that the performance does strongly depend on whether matrix blocks are re-used between two block multiplications, which is obviously a caching effect. This does, of course, affect the performance of any parallel implementation that is based on such block multiplications.

In the following, we compare the performance of our Peano algorithm with that of the normal SRUMMA algorithm on the Infiniband cluster. In order to really compare the actual algorithms, we tried to keep the implementations of the algorithms as similar as possible. Hence, both implementations were based on first creating a task list for each processor. Based on this task list, the actual block multiplications, but also the prefetching actions, were executed. Nevertheless, the two approaches are difficult to compare, because their



(a) block size 128×128 (b) block size 512×512

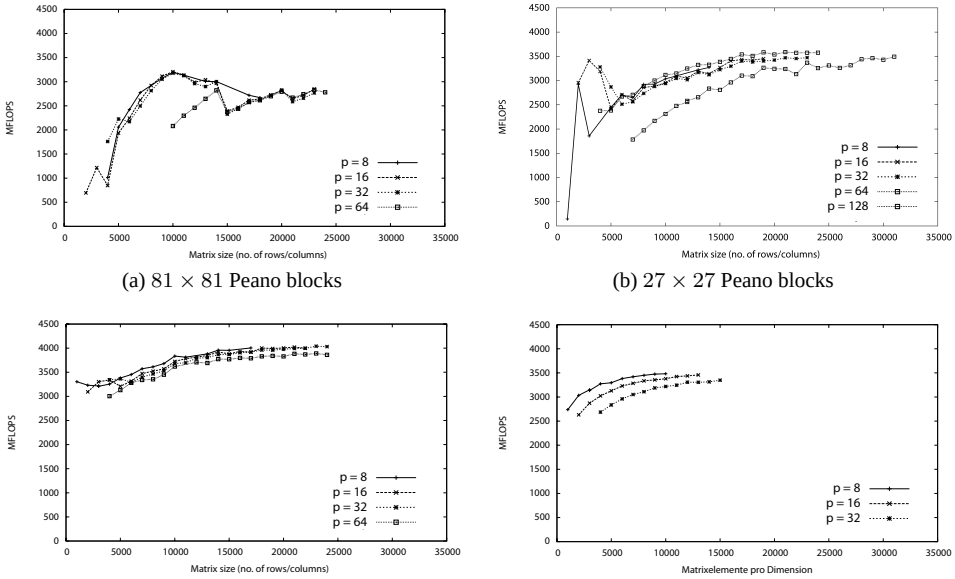
(c) block size 1024×1024 (d) block size 2048×2048

Figure 4. Performance of SRUMMA (given in MFlop/s per processor) for increasing size of matrices computed using different block sizes for the parallelised block multiplications.

parallelisation is based on two slightly different paradigms. SRUMMA uses an *owner-computes* approach, where the distribution of the total work to the processors is strictly determined by the partitioning of the matrices, which again is more or less determined by the number of available processors. On the other hand, the Peano algorithm does not consider the location of matrix blocks at all. Instead, it divides the entire sequence of block multiplications into equal-sized parts, and distributes these onto the processors.

Hence, Fig. 4 shows the performance of the SRUMMA algorithm for different size of the block partitions. The block sizes are powers of 2 in accordance with the matrix sizes of our intended target application. In Fig. 5, we present the performance of the Peano algorithm. Here, to simplify the Peano layout, we only considered block layouts where the number of blocks per dimension is a power of 3. Hence, we present the performance for different block layouts, which means that the size of the matrix blocks is always proportional to the size of the global matrices.

For sufficiently large matrix partitions, both algorithms are able to come very close to the theoretically optimal performance determined by the ACML library. SRUMMA seems to have a slight advantage with respect to the highest achievable MFlop/s rates. In our opinion, this is an advantage due to the *owner-computes* scheme: in contrast to the Peano algorithm, SRUMMA does not require communication operations to write back results of block multiplications to the result matrix. On the other hand, our SRUMMA implementation revealed severe deficits when 64 or even 128 processors were used on the Infiniband cluster. Here, SRUMMA shows a substantial decrease of the performance, especially for matrices of moderate size, whereas the performance loss for the Peano algorithm is com-



(a) 81×81 Peano blocks

(b) 27×27 Peano blocks

(c) 9×9 Peano blocks

(d) 3×3 Peano blocks

Figure 5. Performance of the Peano algorithm (given in MFlop/s per processor) for increasing size of matrices. Diagrams (a)–(d) show the performance for different Peano-blocking schemes; the respective size of parallelised matrix blocks grows proportionally to the matrix size.

parably small. We believe this general performance decrease to be an effect of the special communication hardware of the Infiniband cluster. As on each cluster node, 4 processors share only 1 Infiniband adapter for communication, the available communication bandwidth and latency deteriorates when using 64 or even all 128 processors of the cluster. The Peano algorithm is obviously less affected by the weaker performance of the communication hardware, which it apparently compensates by the stronger locality properties of the element access.

5.2 Performance of Parallel Prefix Computation

Our next step is to use the Peano algorithm for parallelising individual matrix multiplications in the parallel prefix scheme. As indicated in Section 2, the parallel prefix scheme makes it possible to parallelise the computation of the matrix products $A_{1:k}$, but the required tree-like computation scheme increases the overall work. Fig. 6 illustrates how the increase of the total work depends on the number of processors used for the prefix scheme. It also shows how many matrix multiplications each processor has to perform sequentially, which indicates the optimal speedup that can be achieved by the prefix scheme alone. In the illustrated example ($m = 128$), the prefix scheme is more expensive by nearly a factor 4. However, in a previous paper[2], it was shown that the prefix scheme can still outperform the iterative computation (with parallelised matrix multiplication) for large m and large number of available processors. The respective performance loss in parallel matrix multiplication resulted both from less efficient matrix multiplication on small matrix blocks and

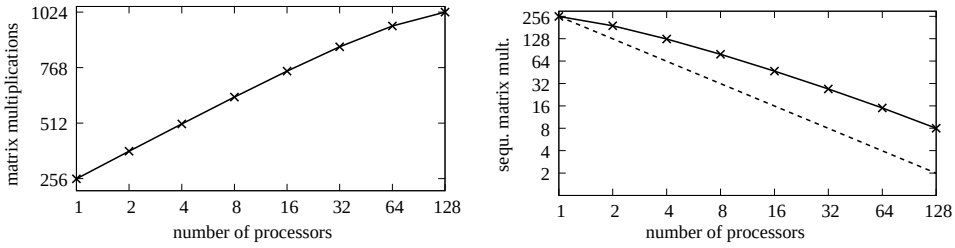


Figure 6. Number of matrix multiplications required to compute the parallel prefix scheme for $m = 256$ matrices. The left diagram shows the total number of multiplications depending on the number p of processors used for the prefix scheme ($p = 1$: sequential computation; $p = 128$: full prefix scheme). The right diagram shows the number of multiplication performed by each processor (solid line), which characterises the ideal speedup when using all p processors for the prefix scheme. The dashed line indicates the perfect speedup for the straightforward, iterative computation of the scheme (using p processors to parallelize the $m - 1$ matrix multiplications).

from communication overhead due to the multiplication scheme.

In Fig. 7, we present the runtimes for computing the parallel prefix scheme for different matrix sizes and different values of m (i.e. numbers of matrices in the scheme). All 128 processors of the Infiniband cluster were used. However, we varied the number of processors used for parallelising the prefix scheme and for parallelising the individual matrix multiplications. As the leftmost value, we also added the runtime for computing the iterative prefix scheme using all 128 processors to parallelise the individual matrix multiplications. Several effects can be observed:

- The runtime generally decreases considerably the less Peano blocks are used for parallelisation. This is a direct consequence of the larger block sizes of the parallelised block multiplications, which both leads to a better performance of the respective BLAS routine, and leads to heavy-weight block operations, which makes it easier to "hide" the communication operations by executing them in parallel to computations.
- For fixed blocking scheme and increasing number of processors used for the parallel prefix scheme, the computation time increases, which is simply a result of the increased overall work (compare Fig. 6).
- For the 3×3 Peano blocking, using a large number of processors for parallelising individual matrix multiplications leads to comparably bad speedups, because the resulting 27 block multiplications cannot be distributed homogeneously to more than 8 (or at most 16) processors.

Overall, it can be observed that the iterative computation of the prefix scheme, using all 128 available processors to parallelise matrix multiplication, leads to the lowest runtimes for all presented scenarios, except for the smallest problem – scenario (a), where the pure prefix scheme became the fastest option. For small matrices (512×512 or 1024×1024), the "atomic" matrix blocks obviously become too small to get a useful performance out of the sequential block multiplications, and also too small to hide communication costs behind the block computations. However, for larger problem sizes, the fine-grain approach to parallelise only the individual matrix multiplications is clearly the best solution – the margin getting even larger for growing matrix sizes.

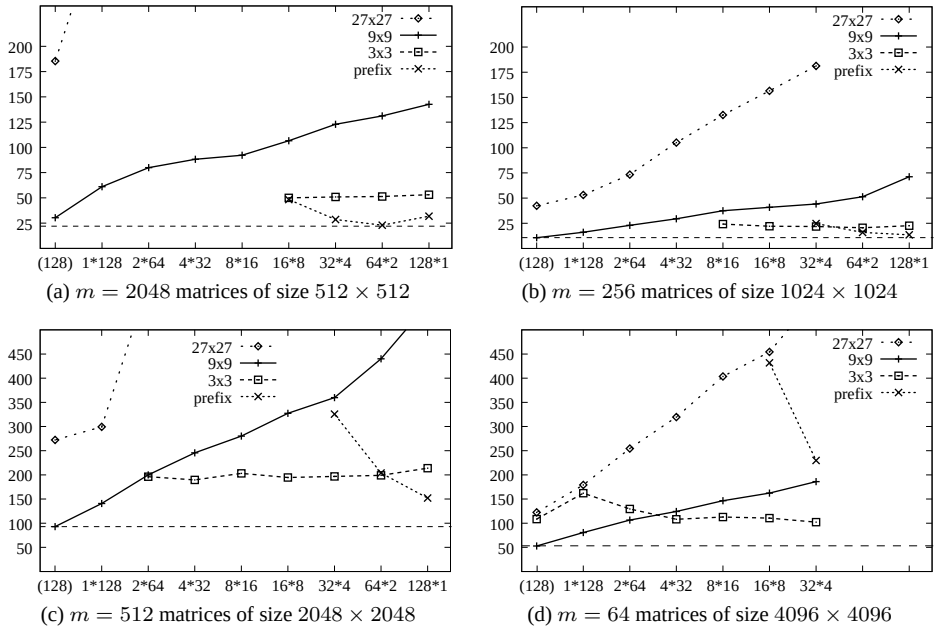


Figure 7. Runtimes to compute the parallel prefix scheme on 128 processors. Figures (a)–(d) present the runtime in dependence of the number of processor groups used for the parallelisation of the prefix scheme. The leftmost value shows the runtime for the sequential prefix scheme (using all 128 processors to parallelise the individual matrix multiplications). The dashed line indicates the lowest achieved runtime.

This observation is to some extent in contradiction to previous studies[2], where the parallel prefix scheme was shown to be preferable for large numbers of available processors. However, in that paper, a different algorithm was used for parallelising matrix multiplications (using matrix slices instead of square matrix blocks), which suffered more strongly from performance losses and communication overhead for parallelising multiplications of small submatrices. Hence, the improved scalability of the Peano algorithm – especially when using small atomic matrix blocks – allows to use the much simpler iterative scheme for the prefix problem.

6 Conclusion

We presented a performance study which compared the parallel performance of the Peano algorithm, a new variant of the SRUMMA algorithm, with the classical SRUMMA approach. The Peano algorithm is based on a block-recursive algorithm for matrix multiplication based on the Peano space-filling curve, and exploits the locality properties of the Peano curve to improve data locality and communication patterns in a parallel implementation. Following the SRUMMA idea, block multiplication is overlapped with prefetching of the next matrix blocks in order to hide communication costs. While the straightforward SRUMMA approach is slightly superior to the Peano algorithm, when large block matrices may be used, and fast shared-memory communication is available, the Peano algorithm

outperforms SRUMMA in situations, where speed of communication becomes more critical. For example, on the Infiniband cluster used for the presented work, four processors share a single Infiniband adapter, which leads to communication bottlenecks, when all available processors of the cluster are used. Here, the Peano algorithm was able to deliver a noticeably higher MFlop/s rate than the classical SRUMMA approach.

For our intended application, an instance of the so-called prefix problem, we integrated the Peano algorithm into the tree-structured parallel prefix scheme, such that two options for parallelisation became available: to parallelise the prefix scheme, which offers coarse grain parallelism, but leads to an increase of the total work, and to parallelise individual matrix multiplications, which leads to a fine-grain parallelisation, but can suffer from scalability problems, once the parallelised matrix blocks become too small. Our performance tests show that the Peano algorithm scales well enough to make the parallel prefix scheme dispensable – in all larger test scenarios, iterative (sequential) prefix computation combined with parallel individual matrix multiplications (using all available processors) was the fasted option. This is in contrast to previous results for less scalable algorithms for parallel multiplication, where the prefix scheme was the better option, especially for very many matrices of comparably small size.

From a programmers point of view, the Peano algorithm offers an additional advantage, which results from the SRUMMA approach: as block matrices are prefetched "on demand" from remote processors, the placement of the individual matrices of the prefix scheme onto different processors becomes less important. In addition, only the parallel implementation of the matrix multiplication needs to issue data transfer operations, which keeps the implementation comparably simple.

References

1. M. Bader and C. Zenger, *Cache oblivious matrix multiplication using an element ordering based on a Peano curve*, Linear Algebra and its Applications, **417**, 301–313, (2006).
2. T. Gradl, A. Spörl, T. Huckle, S. J. Glaser and T. Schulte-Herbrüggen, *Parallelising Matrix Operations on Clusters for an Optimal Control-Based Quantum Compiler*, in: Nagel et al. (eds.), Euro-Par 2006 Parallel Processing, Lecture Notes in Computer Science **4128**, 751–762, (2006).
3. N. Khaneja, T. Reiss, C. Kehlet, T. Schulte-Herbrüggen and S. J. Glaser, *Optimal Control of Coupled Spin Dynamics: Design of NMR Pulse Sequences by Gradient Ascent Algorithms*, Journal of Magnetic Resonance, **172**, 296–305, (2005).
4. M. Krishnan and J. Nieplocha, *SRUMMA: A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems*, In: Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), (2004).
5. R. E. Ladner and M. J. Fischer, *Parallel Prefix Computation*, Journal of the Association for Computing Machinery, **27**, 831–838, (1980).
6. J. Nieplocha and B. Carpenter, *ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems*, in: Proceedings of RTSPP IPPS/SDP, (1999).
   ARMCI web page: http://www.emsl.pnl.gov/docs/parsoft/armci/

Parallel Exact Inference

Yinglong Xia[1] **and Viktor K. Prasanna**[2]

[1] Computer Science Department
University of Southern California, Los Angeles, U.S.A.
E-mail: yinglonx@usc.edu

[2] Ming Hsieh Department of Electrical Engineering
University of Southern California, Los Angeles, U.S.A.
E-mail: prasanna@usc.edu

In this paper, we present complete message-passing implementation that shows scalable performance while performing exact inference on arbitrary Bayesian networks. Our work is based on a parallel version of the classical technique of converting a Bayesian network to a junction tree before computing inference. We propose a parallel algorithm for constructing potential tables for a junction tree and explore the parallelism of rerooting technique for multiple evidence propagation. Our implementation also uses pointer jumping for parallel inference over the junction tree. For an arbitrary Bayesian network with n vertices using p processors, we show an execution time of $O(nk_m^2 + (wn^2 + wN \log n + r^w wN + r^w N \log N)/p)$, where w is the clique width, r is the number of states of the random variables, k is the maximum node degree in the Bayesian network, k_m is the maximum node degree in the moralized graph and N is the number of cliques in the junction tree. Our implementation is scalable for $1 \leq p \leq n$ for moralization and clique identification, and $1 \leq p \leq N$ for junction tree construction, potential table construction, rerooting and evidence propagation. We have implemented the parallel algorithm using MPI on state-of-the-art clusters and our experiments show scalable performance.

1 Introduction

A full joint probability distribution for any real-world systems can be used for inference. However, such a distribution grows intractably large as the number of variables used to model the system grows. Bayesian networks[2] are used to compactly represent joint probability distributions by exploiting conditional independence relationships. They have found applications in a number of domains, including medical diagnosis, credit assessment, data mining, etc.[6,7]

Inference on a Bayesian network is the computation of the conditional probability of the *query variables*, given a set of *evidence variables* as knowledge. Such knowledge is also known as *belief*. Inference on Bayesian networks can be *exact* or *approximate*. Since exact inference is known to be NP hard[5], several heuristics exist for the same. The most popular exact inference algorithm for multiply connected networks was proposed by Lauritzen and Speigelhalter[2], which converts a given Bayesian network into a junction tree, and then performs exact inference on the junction tree.

Several parallel implementations of exact inference are known[4,5,6]. However, while some of them only deal with singly connected network; others only consider a part of the independent operations. Shared memory implementations of the various stages of exact inference in parallel exist[3,4], but to the best of our knowledge, this is the first parallel *message passing* implementation of the *entire* process: structure conversion from Bayesian network to junction tree, the construction of potential table, rerooting the tree according to

the evidence provided, and inference calculation with multiple evidence on junction tree. The scalability of our work is demonstrated by experimental results in this paper.

The paper is organized as follows: Section 2 gives a brief background on Bayesian networks and junction trees. We explore the parallel conversion from Bayesian network to junction tree in Section 3. Section 4 discusses parallel exact inference on junction tree. Experimental results are shown in Section 5 and Section 6 concludes the paper.

2 Background

A *Bayesian network* exploits conditional independence to represent a joint distribution more compactly. A Bayesian network is defined as $\mathrm{B} = (\mathbb{G}, \mathbb{P})$ where $\mathbb{G}$ is a *directed acyclic graph* (DAG) and $\mathbb{P}$ is the parameter of the network. The DAG $\mathbb{G}$ is denoted as $\mathbb{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{A_1, A_2, \ldots, A_n\}$ is the node set and $\mathcal{E}$ is the edge set. Each node A_i represents a random variable. If there is an edge from A_i to A_j i.e. $(A_i, A_j) \in \mathcal{E}$, A_i is called a *parent* of A_j. $pa(A_j)$ denotes the set of all parents of A_j. Given the value of $pa(A_j)$, A_j is conditionally independent of all other preceding variables. The parameter $\mathbb{P}$ represents a group of *conditional probability tables* (CPTs) which are defined as the conditional probability $P(A_j|pa(A_j))$ for each random variable A_j. Given the Bayesian network, a joint distribution $P(\mathcal{V})$ can be rewritten as $P(\mathcal{V}) = P(A_1, A_2, \cdots, A_n) = \prod_{j=1}^{n} Pr(A_j|pa(A_j))$. Fig. 1 (a) shows a sample Bayesian network.

The *evidence variables* in a Bayesian network are the variables that have been instantiated with values e.g. $\mathrm{E} = \{A_{e_1} = a_{e_1}, \cdots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \ldots, n\}$. Given the evidence, we can inquire the distribution of any other variables. The variables to be inquired are called *query variables*. The process of *exact inference* involves propagating the evidence throughout the network and then computing the updated probability of the query variables.

It is known that traditional exact inference using Bayes' rule fails for networks with undirected cycles[5]. Most inference methods for networks with undirected cycles convert a network to a cycle-free hypergraph called a *junction tree*. A junction tree is defined as $\mathrm{J} = (\mathbb{T}, \hat{\mathbb{P}})$ where $\mathbb{T}$ represents a tree and $\hat{\mathbb{P}}$ denotes the parameter of the tree. Each vertex $\mathcal{C}_i$ (known as a clique) of $\mathbb{T}$ is a set of random variables. Assuming $\mathcal{C}_i$ and $\mathcal{C}_j$ are adjacent, the *separator* between them is defined as $\mathcal{C}_i \cap \mathcal{C}_j$. All junction trees satisfy the *running intersection property* (RIP)[2]. $\hat{\mathbb{P}}$ is a group of *potential tables* (POTs). The POT of $\mathcal{C}_i$, denoted as $\psi(\mathcal{C}_i)$, can be viewed as the joint distribution of the random variables in $\mathcal{C}_i$. For a clique with w variables, each taking r different values, the number of entries in the POT is r^w. Fig. 1 (d) shows a junction tree.

An arbitrary Bayesian network can be converted to a junction tree by the following steps: *Moralization*, *Triangulation*, *Clique identification*, *Junction tree construction* and *Potential table construction*. Fig. 1 (b) and (c) illustrate a moralized graph and a triangulated graph respectively. Sequential algorithms of the above steps were proposed by Lauritzen et al.[2]. To the best of our knowledge, parallel algorithm for potential table construction has not been addressed in the literature.

In a junction tree, exact inference proceeds as follows: Assuming evidence is $\mathrm{E} = \{A_i = a\}$ and $A_i \in \mathcal{C}_j$, E is *absorbed* at $\mathcal{C}_j$ by instantiating the variable A_i, then renormalizing the remaining constituents of the clique. The effect of the updated $\psi(\mathcal{C}_j)$ is propagated to all other cliques by iteratively setting $\psi^*(\mathcal{C}_x) = \psi(\mathcal{C}_x)\psi^*(\mathcal{S})/\psi(\mathcal{S})$ where $\mathcal{C}_x$ is

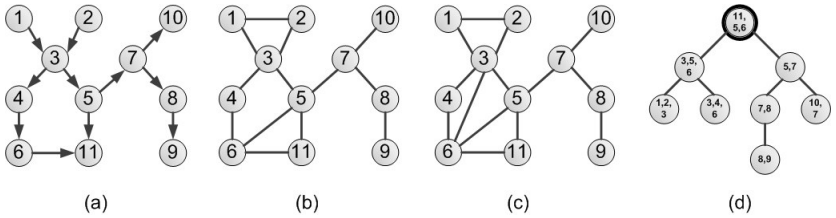


Figure 1. An example of (a) Bayesian network; (b) Moralized graph; (c) Triangulated graph; (d) Junction tree.

the clique to be updated; $\mathcal{S}$ is the separator between $\mathcal{C}_x$ and its neighbour that has been updated; ψ^* denotes the updated POT. After all cliques are updated, the distribution of a query variable $Q \in \mathcal{C}_y$ is obtained by $P(Q) = \sum_{\mathcal{R}} \psi(\mathcal{C}_y)/Z$ where $\mathcal{R} = \{z : z \in \mathcal{C}_y, z \neq Q\}$ and Z is a constant with respect to $\mathcal{C}_y$. This summation sums up all entries with respect to $Q = q$ for all possible q in $\psi(\mathcal{C}_y)$. The details of sequential inference are proposed by Lauritzen et al.[2]. Pennock[5] has proposed a parallel algorithm for exact inference on Bayesian network, which forms the basis of our work.

For the analysis of our work, we have assumed a Concurrent Read Exclusive Write Parallel Random Access Machine (CREW PRAM) model. In implementation, each processor contains a portion of the data, and they interact by passing messages.

3 Parallel Construction of a Junction Tree from a Bayesian Network

Given an arbitrary Bayesian network $\mathrm{B} = (\mathbb{G}, \mathbb{P})$, it can be converted into a junction tree $\mathrm{J} = (\mathbb{T}, \hat{\mathbb{P}})$ by five steps: moralization, triangulation, clique identification, junction tree construction and potential table construction. In this section, we discuss the parallel algorithms used by us in our implementation.

3.1 Structure Conversion

In parallel moralization, additional edges are inserted so that all parents of each node are pairwise connected. The input to our parallel moralization algorithm is a DAG $\mathbb{G} = (\mathcal{V}, \mathcal{E})$. The output is a moralized graph $\mathbb{G}_m = (\mathcal{V}, \mathcal{E}_m)$. The DAG $\mathbb{G}$ is represented as an adjacency list. Each processor is sent a segment of the adjacency array. Assuming node v_i is assigned to processor p_i, p_i generates a set $pa(v_i)$ as the set of v_i's parents and sends $pa(v_i)$ to the processors where v_i's parents are assigned. Each processor works in parallel. Then, every processor p_i receives a set pa_k from its k-th child. If v_i is not connected to a node $v_j \in pa_k$, they are connected so that v_i is moralized.

In triangulation, edges are inserted to $\mathcal{E}_m$ so that in the moralized graph all cycles of size larger than or equal to 4 have chords. A *chord* is defined as an edge connecting two nonadjacent nodes of a cycle. The optimal triangulation minimizes the maximal clique width in the resulting graph. This problem is known to be NP hard[5]. The input to our triangulation algorithm is a moralized graph $\mathbb{G}_m = (\mathcal{V}, \mathcal{E}_m)$. The output is a triangulated graph $\mathbb{G}_t = (\mathcal{V}, \mathcal{E}_t)$. $\mathcal{E}_t$ is the union of the newly added edges and $\mathcal{E}_m$. In order to improve the speed of exact inference in the junction tree, the triangulation method should minimize

the maximum clique width in the resulting graph, which is known to be NP hard[8]. In this step, each processor is in charge of a subset of $\mathbb{G}_m$. The cycles of size equal or larger than 4 are detected and chorded by inserting edges.

In parallel clique identification, the cliques of the junction tree are computed on each processor. The input is a triangulated graph $\mathbb{G}_t = (\mathcal{V}, \mathcal{E}_t)$ and the output is a group of cliques $\{\mathcal{C}_1, \cdots, \mathcal{C}_N\}$. $\mathbb{G}_t$ is also represented as an adjacency array. Each processor obtains a segment of $\mathbb{G}_t$ and forms a candidate clique for each node v_i in parallel. Each processor then uses the details of all the cliques to verify if there exist $\mathcal{C}_i, \mathcal{C}_j$ s.t. $\mathcal{C}_i \subseteq \mathcal{C}_j$. If so, the candidate clique $\mathcal{C}_i$ is removed. Each processor performs the candidate clique verification in parallel. The survived cliques form the output.

Parallel junction tree construction assigns a parent to each clique identified in the previous step. The input to our parallel junction tree construction is a group of cliques $\{\mathcal{C}_1, \cdots, \mathcal{C}_N\}$. The output is junction tree structure $\mathbb{T}$ which is an adjacency array representing the connections between cliques. To satisfy the RIP, we need to compute the union $\mathcal{U}_j = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \cdots \cup \mathcal{C}_{j-1}$ for $\mathcal{C}_j$, $j = 1, \cdots, N$. We use the well-known parallel technique of *pointer-jumping*[1] to perform the computation in $O(\log N)$ time for N cliques. Then, each processor computes in parallel the intersection $\mathcal{I}_j = \mathcal{C}_j \cap \mathcal{U}_j$ and finds $\mathcal{C}_i \supseteq \mathcal{I}_j$ from $\mathcal{C}_1, \cdots, \mathcal{C}_{j-1}$ so that $\mathcal{C}_i$ can be assigned as the parent of $\mathcal{C}_j$.

3.2 Potential Table Construction

Potential table construction initializes a POT for each clique of a junction tree using the CPTs of the Bayesian network from which the junction tree is converted, and then converts the initial POTs to chain set POTs. The input to the parallel potential table construction is the Bayesian network $\mathrm{B} = (\mathbb{G}, \mathbb{P})$ and the structure of the junction tree $\mathbb{T}$. The output is $\hat{\mathbb{P}}$, a group of POTs for the cliques of junction tree $\mathrm{J} = (\mathbb{T}, \hat{\mathbb{P}})$. In parallel potential table construction, each processor is in charge of one or several cliques. The processors work in parallel to identify those nodes A that satisfy $A \cup pa(A) \subseteq \mathcal{C}_i$ for each clique $\mathcal{C}_i$. Then, the CPTs of these identified nodes are multiplied in parallel to produce the initial POT for each clique. Each processor then converts the initial POTs to chain set POTs in parallel. The conversion process is as the same as the inference in junction tree except for the absence of evidence variable. We use a pointer jumping based technique to parallelize the conversion from initial POTs to chain set POTs. The details of the pointer jumping based technique is addressed in Section 4.

3.3 Complexity Analysis

The analysis of the execution time is based on the Concurrent Read Exclusive Write Parallel Random Access Machine (CREW PRAM) model. i.e. concurrent read access to the same data by different processors is allowed, but concurrent write is not.

The execution time of moralization is $O(nk^2/p)$ where n is the number of nodes, k is the maximal number of neighbors in $\mathbb{G}$, and p is the number of processors. Triangulation and clique identification take $O(k_m^2 n + wn^2/p)$ time, where w is the clique width and k_m is the maximal number of neighbors in $\mathbb{G}_m$. The execution time for junction tree construction is $O((wN \log n + wN^2)/p)$ where N is the number of cliques of the junction tree. The potential table construction takes $O(r^w N(w + \log N)/p)$ time where r is the

number of states of a variable. Both moralization and clique identification are scalable over $1 \leq p \leq n$; while junction tree construction and potential table initialization are scalable over $1 \leq p \leq N$.

4 Parallel Inference in Junction Trees

We discuss parallel inference in junction tree in two situations: when the evidence variable is present at the root (see the clique with thick border in Fig. 1 (d)) of the junction tree and when the evidence variable is not present at the root of the junction tree.

When the evidence variable is present at the root, the root absorbs the evidence by instantiating the evidence variable in its POT. Then, the pointer jumping technique is used to propagate the evidence throughout the complete tree: Each clique $\mathcal{C}_i$ sends its POT $\psi(\mathcal{C}_i)$ to its children $ch(\mathcal{C}_i)$ and receives a POT from its parent $pa(\mathcal{C}_i)$. Each clique's POT $\psi(\mathcal{C}_i)$ is updated using $\psi(pa(\mathcal{C}_i))$ independently. As $\mathcal{C}_i$ may have multiple children, the sending of the POT to each child is performed in parallel. Then, we set $ch(\mathcal{C}_i) = ch(ch(\mathcal{C}_i)),\ pa(\mathcal{C}_i) = pa(pa(\mathcal{C}_i))$ and perform the evidence propagation again. Assuming each clique is assigned to a processor, the whole tree is updated in $O(\log D)$ time where D is the depth of the junction tree. Each update requires rewriting a CPT, taking $O(r^w)$ time with one processor.

When the evidence variable is not present at the root, we extend the parallel tree rerooting technique[5] to make the clique that contains the evidence variable as the root of a new junction tree. We compute the depth-first search (DFS) order α of the cliques starting from the clique with evidence variable. Each processor in parallel gets a copy of α and modifies the edge direction if it is inconsistent with α. There is no change in POTs and separators. The execution time of parallel rerooting technique is $O(|\mathrm{E}|Nw/p)$ where $|\mathrm{E}|$ is the number of cliques with evidence. Once the clique containing evidence variable becomes the root, we perform the parallel evidence inference as the same as that in the first situation.

Since the number of cliques containing evidence is bounded by N and $D \leq N$, the execution time of parallel inference algorithm is bounded by $O(r^w N(\log N)/p)$ where $1 \leq p \leq N$.

5 Experiments

5.1 Computing Facilities

We ran our implementations on the DataStar Cluster at the San Diego Supercomputer Center (SDSC)[9] and on the clusters at the USC Center for High-Performance Computing and Communications (HPCC)[10]. The DataStar Cluster at SDSC employs IBM P655 nodes running at 1.5 GHz with 2 GB of memory per processor. This machine uses a Federation interconnect, and has a theoretical peak performance of 15 Tera-Flops. Furthermore, each channel is connected to a GPFS (parallel file system) through a fibre channel. The DataStar Cluster runs Unix with MPICH. IBM Loadleveler was used to submit jobs to batch queue.

The USC HPCC is a Sun Microsystems & Dell Linux cluster. A 2-Gigabit/second low-latency Myrinet network connects most of the nodes. The HPCC nodes used in our experiments were based on Dual Intel Xeon (64-bit) 3.2 GHz with 2 GB memory. The operating system is USCLinux, a customized distribution of the RedHat Enterprise Advanced Server

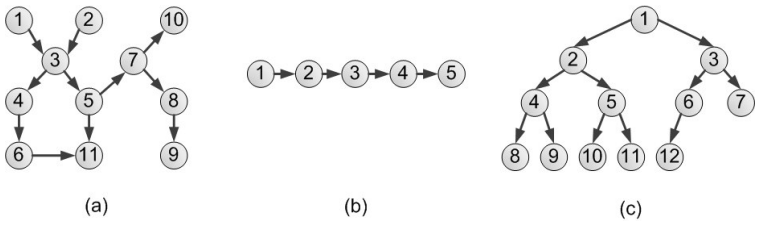


Figure 2. An example of (a) Random Bayesian network; (b) Linear Bayesian network; (c) balanced Bayesian network.

3.0 (RHE3) Linux distribution, with MPICH for communication. The Portable Batch System (PBS) was used for resource allocation and management.

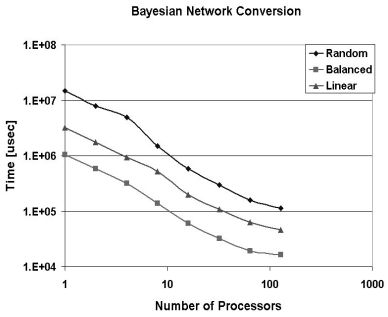
5.2 Input Bayesian Networks

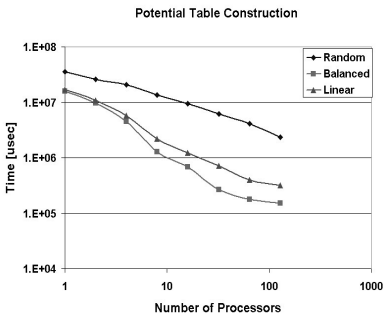
We experimented with three types of Bayesian network as the input: random, linear and balanced. Each Bayesian network has 1024 nodes and each node is binary (r=2). We ran the program with 1, 2, 4, 8, 16, 32, 64 and 128 processors for each of three networks.

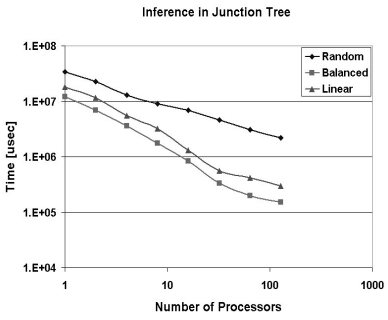
The random Bayesian network (Fig. 2 (a)) was initially generated as a random graph. Representing the initial graph as a adjacency matrix, we modified the matrix to ensure the following constraints: (1) It is an upper-triangular matrix; (2) The graph is connected; (3) The number of 1s in each row and column is limited by a given constant known as *node degree*. These constraints leads to a legal Bayesian network structure. Assuming all nodes are binary, the CPT for a node A was generated as a table of non-negative floating point numbers with 2 rows and $2^{|pa(A)|}$ columns where $|pa(A)|$ is the number of parents of A. In our experiment, the clique width $w = 13$. The linear Bayesian network (Fig. 2 (b)) is a chain: each node except the terminal ones has exactly one incoming edge and one outgoing edge. As $|pa(v)| \leq 1\ \forall v$ and $w = 2$, the CPT size is no larger than 4. The balanced Bayesian network (Fig. 2 (c)) was generated as tree, where the in-degree of each node except the root is 1 and the out-degree for each node except the leaves and last non-leaf node is a constant. So, we have $|pa(v)| \leq 1$ and $w = 2$.

5.3 Experimental Results

In our experiments, we recorded the execution time of parallel Bayesian network conversion, potential table construction and exact inference on junction trees. We added together the above times to obtain the total execution time. The results from DataStar Cluster are shown in Fig. 3 and the results from HPCC in Fig. 4. We are concerned with the scalability of the implementation instead of the speedups. From these figures, we can see that all the individual steps and the total execution time exhibit scalability. Scalability is observed for all three types of Bayesian networks. The execution time for random networks is larger than others because the maximal clique width of the junction tree created from random Bayesian network is larger. The experimental results reflect the scalability of our implementation, confirming that communication does not overshadow computation over a significant range of p.







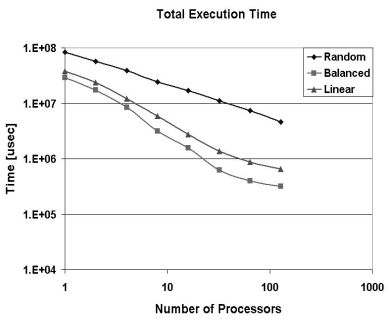
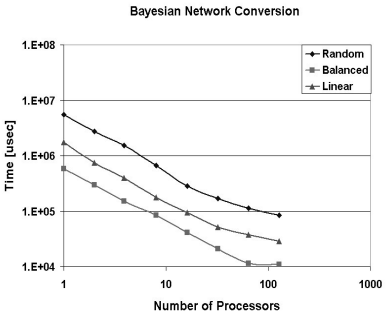
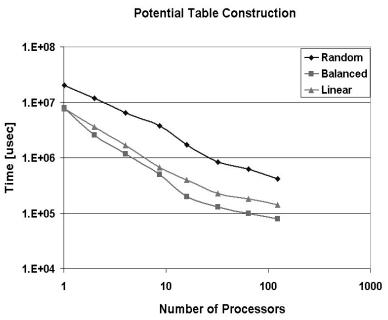
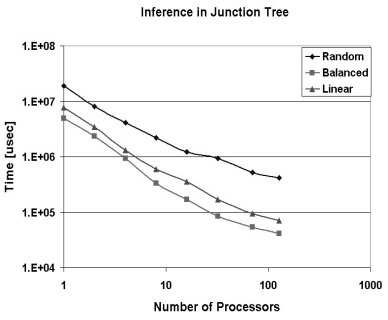


Figure 3. The execution time of exact inference on DataStar.







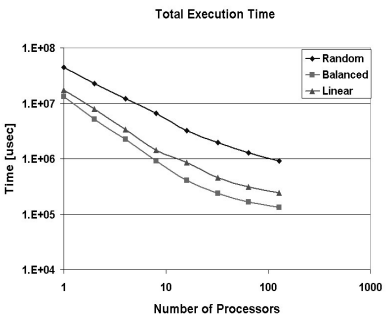


Figure 4. The execution time of exact inference on HPCC.

6 Conclusion

In this paper, we presented our scalable parallel message passing implementation of all the steps involved in exact inference starting from an arbitrary Bayesian network. We also estimate the execution time for each step in this paper. Our implementation was done using MPI and we presented the experimental results from state-of-the-art clusters to show the scalability of our work. For future work, we intend to investigate the parallelization of exact inference at different levels while maintaining scalability.

Acknowledgements

This research was partially supported by the National Science Foundation under grant number CNS-0613376 and utilized the DataStar at the San Diego Supercomputer Center. NSF equipment grant CNS-0454407 is gratefully acknowledged.

References

1. J. JáJá, *An Introduction to Parallel Algorithms*, (Addison-Wesley, Reading, MA, 1992).
2. S. L. Lauritzen and D. J. Spiegelhalter, *Local computation with probabilities and graphical structures and their application to expert systems*, J. Royal Statistical Society B., **50**, 157–224, (1988).
3. V. K. Namasivayam, A. Pathak and V. K. Prasanna, *Scalable parallel implementation of Bayesian network to junction tree conversion for exact inference*, in: Proc. 18th International Symposium on Computer Architecture and High Performance Computing, pp. 167–176, (2006).
4. V. K. Namasivayam and V. K. Prasanna, *Scalable parallel implementation of exact inference in Bayesian networks*, in: Proc. 12th International Conference on Parallel and Distributed Systems, pp. 143–150, (2006).
5. D. Pennock, *Logarithmic time parallel Bayesian inference*, in: Proc. 14th Annual Conference on Uncertainty in Artificial Intelligence, pp. 431–438, (1998).
6. L. Yin, C.-H. Huang, and S. Rajasekaran, *Parallel data mining of Bayesian networks from gene expression data*, in: Poster Book of the 8th International Conference on Research in Computational Molecular Biology, pp. 122–123, (2004).
7. E. Segal, B. Taskar, A. Gasch, N. Friedman and D. Koller, *Rich probabilistic models for gene expression*, in: 9th International Conference on Intelligent Systems for Molecular Biology, pp. 243–252, (2001).
8. T. Klocks, *Treewidth: Computations and Approximations*, Springer-Verlag, Berlin, (1994).
9. http://www.sdsc.edu/us/resources/datastar/
10. http://www.usc.edu/hpcc/

Efficient Parallel String Comparison

Peter Krusche and Alexander Tiskin

Department of Computer Science
The University of Warwick, Coventry CV4 7AL, United Kingdom
E-mail: {peter, tiskin}@dcs.warwick.ac.uk[†]

The longest common subsequence (LCS) problem is a classical method of string comparison. Several coarse-grained parallel algorithms for the LCS problem have been proposed in the past. However, none of these algorithms achieve scalable communication. In this paper, we propose the first coarse-grained parallel LCS algorithm with scalable communication. Moreover, the algorithm is work-optimal, synchronisation-efficient, and solves a more general problem of semi-local string comparison, improving in at least two of these aspects on each of the predecessors.

1 Introduction

Computing longest common subsequences of strings is a common method of comparing strings, which is also referred to as string or sequence alignment. It has applications in biology, signal processing and many other areas. Finding the length of the longest common subsequence (LCS) is of interest as a measure of string similarity and is equivalent to finding the Levenshtein string edit distance[9,16]. In the BSP model[15,6], the LCS of two strings of length n can be computed in $O(n^2/p)$ computational work using p processors, $O(n)$ communication and $O(p)$ supersteps using the standard dynamic programming algorithm[16] combined with the grid dag method[10] (see also[2,7]). In addition, various algorithmic applications[12] require computing the LCS lengths for a string against all substrings of the other string, and/or the LCS lengths for all prefixes of one string against all suffixes of the other string. These additional tasks can be performed in the BSP model at no extra asymptotic cost by combining the grid dag method with the sequential algorithm by Alves et al.[4] or the one by Tiskin[14] (both based on an algorithm by Schmidt[11]).

Alternative algorithms by Tiskin[13,14] compute LCS lengths using a fast method for $(\max, +)$ multiplication of highest-score matrices in an implicit (critical point) representation. Using this method, two highest-score matrices with n critical points each can be multiplied in time $O(n^{1.5})$. Overall, parallel computation of the implicit highest-score matrix for two given strings can be performed in local computation $\mathsf{W} = O(n^2/p)$, communication $\mathsf{H} = O(n \log p)$ and $\mathsf{S} = \log p$ supersteps.

None of the described algorithms achieve scalable communication, i.e. communication $O(n/p^\alpha)$ with $\alpha > 0$. In this paper, we propose the first BSP algorithm with scalable communication, running in local computation $\mathsf{W} = O(n^2/p)$, communication $\mathsf{H} = O(n \log p/\sqrt{p})$ and $\mathsf{S} = \log p$ supersteps. The key idea of the algorithm is to carry out the highest-score matrix multiplication procedure in parallel and in a constant number of supersteps. An overview of previous results and new improvements is given in Table 1.

[†]The authors acknowledge the support of DIMAP (the Centre for Discrete Mathematics and its Applications) during this work.

Table 1. Parallel algorithms for LCS/Levenshtein distance computation

global / str.-substr. / prefix-suffix	W	H	S	*References*
• / - / -	$O(\frac{n^2}{p})$	$O(n)$	$O(p)$	[10]+[16]
• / • / •	$O(\frac{n^2}{p})$	$O(n)$	$O(p)$	[10]+[4,14]
• / • / •	$O(\frac{n^2 \log n}{p})$	$O(\frac{n^2 \log p}{p})$	$O(\log p)$	[1]
• / • / -	$O(\frac{n^2}{p})$	$O(pn \log p)$	$O(\log p)$	[3]
• / • / -	$O(\frac{n^2}{p})$	$O(n \log p)$	$O(\log p)$	[13,4]
• / • / •	$O(\frac{n^2}{p})$	$O(\frac{n \log p}{\sqrt{p}})$	$O(\log p)$	NEW

2 The BSP Model

The BSP model introduced by Valiant[15] describes a parallel computer with three parameters (p, g, l). The performance of the communication network is characterised by a linear approximation, using parameters g and l. Parameter g, the *communication gap*, describes how fast data can be transmitted continuously by the network (in relation to the computation speed of the individual processors) after the transfer has started. The *communication latency* l represents the overhead that is necessary for starting up communication. A BSP computation is divided into *supersteps*, each consisting of local computations and a communication phase. At the end of each superstep, the processes are synchronised using a barrier-style synchronisation. Consider a computation consisting of S supersteps. For each specific superstep $1 \leq \mathsf{s} \leq \mathsf{S}$ and each processor $1 \leq q \leq p$, let $\mathsf{h}^{in}_{\mathsf{s},q}$ be the maximum number of data units received and $\mathsf{h}^{out}_{\mathsf{s},q}$ the maximum number of data units sent in the communication phase on processor q. Further, let w_s be the maximum number of operations in the local computation phase. The whole computation has separate *computation cost* $\mathsf{W} = \sum_{\mathsf{s}=1}^{\mathsf{S}} \mathsf{w}_\mathsf{s}$ and *communication cost* $\mathsf{H} = \sum_{\mathsf{s}=1}^{\mathsf{S}} \mathsf{h}_{\mathsf{s},q}$ with $\mathsf{h}_{\mathsf{s},q} = \max_{1 \leq q \leq p}(\mathsf{h}^{in}_{\mathsf{s},q} + \mathsf{h}^{out}_{\mathsf{s},q})$. The total running time is given by the sum $\mathsf{T} = \sum_{\mathsf{s}=1}^{\mathsf{S}} \mathsf{T}_\mathsf{s} = \mathsf{W} + g \cdot \mathsf{H} + l \cdot \mathsf{S}$.

3 Problem Analysis and Sequential Algorithm

Let $x = x_1 x_2 \ldots x_m$ and $y = y_1 y_2 \ldots y_n$ be two strings over an alphabet Σ. A *substring* of any string x can be obtained by removing zero or more characters from the beginning and/or the end of x. A *subsequence* of string x is any string that can be obtained by deleting zero or more characters, i.e. a string with characters x_{j_k}, where $j_k < j_{k+1}$ and $1 \leq j_k \leq m$ for all k. The *longest common subsequence* of two strings is the longest string that is a subsequence of both input strings. A common approach to finding the LCS of two strings is to define a grid directed acyclic graph, which has vertical and horizontal edges of weight 0, and diagonal edges of weight 1 for every character match between the input strings. Let this *alignment dag* be defined by a set of vertices $v_{i,j}$ with $i \in \{0, 1, 2. \ldots, m\}$ and $j \in \{0, 1, 2. \ldots, n\}$ and edges as follows. We have horizontal and vertical edges $v_{i,j-1} \rightarrow v_{i,j}$ and $v_{i-1,j} \rightarrow v_{i,j}$ of weight 0, and diagonal edges $v_{i-1,j-1} \rightarrow v_{i,j}$ of

weight 1 that are present only when $x_i = y_j$. Longest common subsequences of a substring $x_i x_{i+1} \dots x_j$ and y correspond to longest paths in this graph from $v_{i-1,0}$ to $v_{j,m}$. In addition to the standard LCS problem in which only strings x and y are compared, we consider its generalisation *semi-local LCS*, which includes computation of the lengths of the string-substring LCS, the prefix-suffix LCS and symmetrically the substring-string LCS and the suffix-prefix LCS. Solutions to the semi-local LCS problem can be represented by a matrix $A(i, j)$, where each entry is related to the length of the longest common subsequence of y and substring $x_i \dots x_j$ (or of substrings $x_1 \dots x_i$ and $y_j \dots y_n$, etc.). Without loss of generality, we will assume from now on that both input strings have the same length n. In this case, matrix A has size $N \times N$ with $N = 2n$.

The sequential algorithm by Tiskin[14] computes a set of N *critical points* that can be used to query the length of the LCS of string y and any substring $x_i \dots x_j$. These critical points are given as (odd) half-integer pairs $(\hat{\imath}, \hat{\jmath})$ (corresponding to positions between the vertices of the alignment dag). Throughout this paper we will denote half-integer variables using a $\hat{}$, and denote the set of half-integers $\{i+\frac{1}{2}, i+\frac{3}{2}, \dots, j-\frac{1}{2}\}$ as $\langle i : j \rangle$ (analogously, we denote the set of integers $\{i, i+1, \dots, j\}$ as $[i : j]$). A point (r_1, c_1) is *dominated* by another point (r_2, c_2) if $r_1 \geq r_2$ and $c_1 \leq c_2$. It has been shown[14] that there are N critical points, such that every entry $A(i, j)$ can be computed as $A(i, j) = j - i - a(i, j)$, where $a(i, j)$ is the number of critical points that are dominated by the pair of integers (i, j). This set of critical points defines a permutation matrix D_A, which is used as an implicit representation of the highest-score matrix A. Having computed all critical points, querying the values for $A(i, j)$ is possible in $O(\log^2 N)$ time per query by using a range tree[5], or by using an asymptotically more efficient data structure[8].

Furthermore, Tiskin[14] describes a procedure which, given two highest-score matrices that correspond to adjacent strips in the grid dag, computes the highest-score matrix for the union of these two strips. This procedure can be reduced to the multiplication of two integer matrices in the $(\min, +)$ semiring. As an input, we have the nonzeros of two $N \times N$ permutation matrices D_A and D_B. The procedure computes an $N \times N$ permutation matrix D_C, such that the permutation distribution matrix d_C is the $(\min, +)$ product of permutation distribution matrices d_A and d_B. The permutation distribution matrix d_C is defined as

$$d_C(i, k) = \sum_{(\hat{\imath}, \hat{k}) \in \langle i:N \rangle \times \langle 0:k \rangle} D_C(\hat{\imath}, \hat{k}), \tag{3.1}$$

and d_A and d_B are defined analogously. Matrices D_A, D_B and D_C have half-integer indices ranging over $\langle 0 : N \rangle$. Furthermore, we assume without loss of generality that N is a power of 2. By exploiting the monotonicity properties of permutation-distribution matrices, the procedure runs in time $O(N^{1.5})$ for matrices of size $N \times N$, given by their implicit (critical point) representation. The method uses a divide-and-conquer approach that recursively partitions the output matrix into smaller blocks in order to locate its nonzeros.

We will now describe the matrix multiplication method in more detail. As an input, we have the nonzeros of the permutation matrices D_A and D_B, which are both of size $N \times N$. The procedure computes the permutation matrix D_C. At every level of the recursion, we consider a square block in D_C corresponding to the set of indices $\langle i_0 - h : i_0 \rangle \times \langle k_0 : k_0 + h \rangle$. We will call such a block a *C-block*, and denote every such block by the triple (i_0, k_0, h). For every C-block, we compute the number of nonzeros contained within. We

can stop recursing in two cases: if the C-block does not contain any nonzeros, or if it is of size 1×1 and thus specifies the location of a nonzero. The number of nonzeros in a C-block is computed as follows. For each C-block, we define a subset of *relevant* nonzeros in D_A with indices $\mathcal{I}^{(i_0,k_0,h)} = \{(\hat{\imath},\hat{\jmath}) \in \langle i_0 - h : i_0\rangle \times \langle 0 : N\rangle$ and $D_A(\hat{\imath},\hat{\jmath}) = 1\}$ and a subset of relevant nonzeros in D_B with indices $\mathcal{K}^{(i_0,k_0,h)} = \{(\hat{\jmath},\hat{k}) \in \langle 0 : N\rangle \times \langle k_0 : k_0 + h\rangle$ and $D_B(\hat{\jmath},\hat{k}) = 1\}$. We can split a given set of relevant nonzeros in D_A and D_B into two sets at a position $j \in [0 : N]$, and determine the numbers of relevant nonzeros in D_A up to and including column $j - \frac{1}{2}$:

$$\delta_A^{(i_0,k_0,h)}(j) = \mid \{(\hat{\imath},\hat{\jmath}) \in \mathcal{I}^{(i_0,k_0,h)} \text{ and } \hat{\jmath} < j\} \mid = d_A(i_0 - h, j) - d_A(i_0, j), \quad (3.2)$$

as well as the number of relevant nonzeros in D_B starting at row $j + \frac{1}{2}$:

$$\delta_B^{(i_0,k_0,h)}(j) = \mid \{(\hat{\jmath},\hat{k}) \in \mathcal{K}^{(i_0,k_0,h)} \text{ and } \hat{\jmath} > j\} \mid = d_B(j, k_0 + h) - d_B(j, k_0). \quad (3.3)$$

These sequences can be obtained in time $O(N)$ by a scan of the relevant nonzeros in D_A and D_B, which are given as input[a][b]. At lower levels of the recursion, it is possible to compute the values from the sequences that were computed at the previous level. Since, for a fixed C-block, δ_A and δ_B only change at the values of j at which also the number of nonzeros in the relevant part of D_A or D_B changes, we can define contiguous sets of j, which we call the "*j-blocks*", corresponding to a value of $d \in [-h : h]$ which uniquely identifies this block. We define $\mathcal{J}^{(i_0,k_0,h)}(d) = \{j \mid \delta_A^{(i_0,k_0,h)}(j) - \delta_B^{(i_0,k_0,h)}(j) = d\}$. When $\mathcal{I}^{(i_0,k_0,h)}$ and $\mathcal{K}^{(i_0,k_0,h)}$ are given, we can determine the j-blocks by an $O(h)$ scan[c] of these sets. Notice that a j-block need not exist for every d. Particularly for small C-blocks, there will be few j-blocks, as the number of relevant nonzeros decreases with the block size. On the set of existing j-blocks, we define sequences

$$\Delta_A^{(i_0,k_0,h)}(d) = \operatorname*{any}_j \delta_A^{(i_0,k_0,h)}(j) \quad \text{and} \quad \Delta_B^{(i_0,k_0,h)}(d) = \operatorname*{any}_j \delta_B^{(i_0,k_0,h)}(j). \quad (3.4)$$

The predicate "any" is taken over all j in the j-block $\mathcal{J}^{(i_0,k_0,h)}(d)$ corresponding to d. This is sufficient since for a fixed value of d, the corresponding values of $\delta_A(j)$ and $\delta_B(j)$ are equal for all j in the j-block defined by d. When the value of either δ_A or δ_B changes, the corresponding value of d changes, too. For each C-block, we are also interested in the sequence of minima

$$M^{(i_0,k_0,h)}(d) = \min_j (d_A(i_0, j) + d_B(j, k_0)), \text{ with } j \in \mathcal{J}^{(i_0,k_0,h)}(d). \quad (3.5)$$

The predicate "$\min$" is taken over all j in the j-blocks corresponding to d since the values $d_A(i_0, j) + d_B(j, k_0)$ can be different inside a j-block.

At the top level, the set $\mathcal{J}^{(N,0,N)}(d)$ always contains either one or zero elements, and the corresponding sequence $M^{(N,0,N)}(d) = 0$ for all d. At lower levels of the recursion

[a] If the nonzeros are given as tuples, pre-sorting all tuples $(\hat{\imath},\hat{\jmath})$ from D_A and $(\hat{\jmath},\hat{k})$ from D_B by $\hat{\jmath}$ can be used to simplify the scanning procedure as this allows to access the tuples $(\hat{\imath},\hat{\jmath})$ in $\mathcal{I}^{(i_0,k_0,h)}$ and accordingly $(\hat{\jmath},\hat{k})$ in $\mathcal{K}^{(i_0,k_0,h)}$ in order of $\hat{\jmath}$. The order can be preserved through the different levels of the recursion, and the sorting can be carried out by bucket sorting without increasing the memory or computation cost.

[b] If the whole matrix D_C is used as a block for starting the recursion, these sequences are trivial, having $\delta_A^{(N,0,N)}(j) = j$ and $\delta_B^{(N,0,N)}(j) = N - j$, as $\mathcal{I}_{(N,0,N)}$ contains all nonzeros in the permutation matrix D_A and $\mathcal{K}_{(N,0,N)}$ contains all nonzeros in the permutation matrix D_B.

[c] We have $\mid \mathcal{I}^{(i_0,k_0,h)} \mid = \mid \mathcal{K}^{(i_0,k_0,h)} \mid = h$ because D_A and D_B are permutation matrices.

tree (i.e. for smaller block sizes h), the number of relevant nonzeros in D_A and D_B decreases, and the individual j-blocks contain more elements. At any level of the recursion tree, sequence $M^{(i_0,k_0,h)}$ can be computed by an $O(N)$ scan of the sets $\mathcal{I}^{(i_0,k_0,h)}$ and $\mathcal{K}^{(i_0,k_0,h)}$. However, at lower levels of the recursion we can also determine the sequence M corresponding to any C-subblock of any C-block (i_0, k_0, h) using an $O(h)$ scan of sequences $M^{(i_0,k_0,h)}$, $\Delta_A^{(i_0,k_0,h)}$ and $\Delta_B^{(i_0,k_0,h)}$. Using sequences $M^{(i_0,k_0,h)}$, $\Delta_A^{(i_0,k_0,h)}$ and $\Delta_B^{(i_0,k_0,h)}$, it is possible to obtain the values of d_C at the four corners of the current C-block in time $O(h)$ by taking the minimum over all values $d \in [-h : h]$ for which a j-block exists:

$$\begin{aligned} d_C(i_0, k_0) &= \min_d M^{(i_0,k_0,h)}(d), \\ d_C(i_0 - h, k_0) &= \min_d (\Delta_A^{(i_0,k_0,h)}(d) + M^{(i_0,k_0,h)}(d)), \\ d_C(i_0, k_0 + h) &= \min_d (\Delta_B^{(i_0,k_0,h)}(d) + M^{(i_0,k_0,h)}(d)), \\ d_C(i_0 - h, k_0 + h) &= \min_d (\Delta_A^{(i_0,k_0,h)}(d) + \Delta_B^{(i_0,k_0,h)}(d) + M^{(i_0,k_0,h)}(d)). \end{aligned} \tag{3.6}$$

From these values, the number of nonzeros in the current block can be obtained by computing $d_C(i_0 - h, k_0 + h) - d_C(i_0 - h, k_0) - d_C(i_0, k_0 + h) + d_C(i_0, k_0)$. If this number is zero, the recursion can terminate at the current C-block. Otherwise, the algorithm proceeds to recursively partition the block into four subblocks of size $\frac{h}{2}$ in order to locate the nonzeros. In order to partition the block, it is necessary to determine the sequences Δ_A, Δ_B and M for all four C-subblocks $(i', k', \frac{h}{2})$ with $(i', k') \in \{i_0, i_0 - \frac{h}{2}\} \times \{k_0, k_0 + \frac{h}{2}\}$. To determine the sequences Δ_A and Δ_B, it is sufficient to scan the $\frac{h}{2}$ relevant nonzeros in every block. To establish the sequence M for every C-subblock for all $d' \in [-\frac{h}{2} : \frac{h}{2}]$, we define sequences $\bar{\Delta}$ that contain the number of relevant nonzeros in subblock $(i', k', \frac{h}{2})$ for every j-block $\mathcal{J}(d)$ of the current C-block as

$$\bar{\Delta}_A^{(i',k',\frac{h}{2})}(d) = \operatorname*{any}_{j \in \mathcal{J}(d)} \delta_A^{(i',k',\frac{h}{2})}(j) \quad \text{and} \quad \bar{\Delta}_B^{(i',k',\frac{h}{2})}(d) = \operatorname*{any}_{j \in \mathcal{J}(d)} \delta_B^{(i',k',\frac{h}{2})}(j). \tag{3.7}$$

Using these sequences, it is possible to compute the sequences M for every C-subblock from only the minima corresponding to the current C-block and the relevant nonzeros in the C-subblock by taking

$$\begin{aligned} M^{(i_0,k_0,\frac{h}{2})}(d') &= \min_d M^{(i_0,k_0,h)}(d), \\ M^{(i_0,k_0+\frac{h}{2},\frac{h}{2})}(d') &= \min_d M^{(i_0,k_0,h)}(d) + \bar{\Delta}_B^{(i_0,k_0+\frac{h}{2},\frac{h}{2})}(d), \\ M^{(i_0-\frac{h}{2},k_0,\frac{h}{2})}(d') &= \min_d M^{(i_0,k_0,h)}(d) + \bar{\Delta}_A^{(i_0-\frac{h}{2},k_0,\frac{h}{2})}(d), \\ M^{(i_0-\frac{h}{2},k_0+\frac{h}{2},\frac{h}{2})}(d') &= \min_d M^{(i_0,k_0,h)}(d) + \bar{\Delta}_A^{(i_0-\frac{h}{2},k_0+\frac{h}{2},\frac{h}{2})}(d) + \bar{\Delta}_B^{(i_0-\frac{h}{2},k_0+\frac{h}{2},\frac{h}{2})}(d) \end{aligned} \tag{3.8}$$

over all d such that $\bar{\Delta}_A^{(i',k',\frac{h}{2})}(d) - \bar{\Delta}_B^{(i',k',\frac{h}{2})}(d) = d'$ with $d' \in [-\frac{h}{2} : \frac{h}{2}]$. This is equivalent to computing the j-blocks for the C-subblocks. Notice the difference in index between sequences Δ and $\bar{\Delta}$: $\bar{\Delta}$ counts the numbers of relevant nonzeros corresponding to the current C-block, whereas the sequences Δ count numbers of relevant nonzeros in the C-subblocks of the current C-block.

4 Parallel Algorithm

The sequential highest-score matrix multiplication procedure can be used to derive a parallel algorithm[13] that solves the semi-local LCS problem by partitioning the alignment dag into p strips. The problem is solved independently on one processor for each strip using dynamic programming[11,4] to compute the implicit highest-score matrices, and then "merging" the resulting highest-score matrices in a binary tree of height $\log p$. This procedure requires data of size $O(N)$ to be sent by every processor in every level of the tree. The sequential merging requires time $O(N^{1.5})$ for computing the resulting highest-score matrix.

By parallelising the highest-score matrix multiplication algorithm and partitioning the alignment dag into a grid of $\sqrt{p} \times \sqrt{p}$ square blocks, we reduce the number of critical points that need to be transferred in every level of the tree to $O(N/\sqrt{p})$ per processor. We assume w.l.o.g. that $\sqrt{p}$ is an integer, and that every processor has an unique identifier q with $0 \leq q < p$. Furthermore, we assume that every processor q corresponds to exactly one pair $(q_x, q_y) \in [0 : \sqrt{p} - 1] \times [0 : \sqrt{p} - 1]$.

We now describe the parallel version of the highest-score matrix multiplication algorithm. The initial distribution of the nonzeros of the input matrices is assumed to be even among all processors, so that every processor holds $\frac{N}{p}$ nonzeros of D_A and D_B. The recursive divide-and-conquer computation from the previous section has at most p independent problems at level $\frac{1}{2}\log_2 p$. In the parallel version of the algorithm, we start the recursion directly at this level, computing relevant nonzeros and sequence M from scratch as follows.

First we redistribute the nonzeros to strips of width $\frac{N}{p}$ by sending each nonzero $(\hat{\imath}, \hat{\jmath})$ in D_A and each nonzero $(\hat{\jmath}, \hat{k})$ in D_B to processor $\lfloor(\hat{\jmath} - \frac{1}{2}) \cdot p/N\rfloor$. This is possible in one superstep using communication $O(\frac{N}{p})$. To compute the values of sequence M for every processor (M-values), we compute the elementary $(\min, +)$ products $d_A(q_x \cdot \frac{N}{\sqrt{p}}, j) + d_B(j, q_y \cdot \frac{N}{\sqrt{p}})$ for all $j \in [0 : N]$ and every pair (q_x, q_y). Every processor holds all $D_A(\hat{\imath}, \hat{\jmath})$ and all $D_B(\hat{\jmath}, \hat{k})$ for $\hat{\jmath} \in \langle q \cdot \frac{N}{\sqrt{p}} : (q+1) \cdot \frac{N}{\sqrt{p}}\rangle$. Since d_A and d_B are defined from D_A and D_B by (3.1), we can compute the values $d_A(q_x \cdot \frac{N}{\sqrt{p}}, j)$ and $d_B(j, q_y \cdot \frac{N}{\sqrt{p}})$ in blocks of $\frac{N}{p}$ on every processor by using a parallel prefix (respectively parallel suffix) operation. We have $\sqrt{p}$ instances of parallel prefix (respectively parallel suffix), one for each value of q_x (respectively q_y). Therefore, the total cost of the parallel prefix and suffix computation is $\mathsf{W} = O(\frac{N}{p} \cdot \sqrt{p}) = O(\frac{N}{\sqrt{p}})$; the communication cost is negligible as long as $\frac{N}{p} \geq p \;\Rightarrow N \geq p^2$. The parallel prefix and suffix operations can be carried out in $\mathsf{S} = O(1)$ supersteps by computing intermediate (local) prefix results on every processor, performing an all-to-all exchange of these values, and then locally combining on every processor the local results with the corresponding intermediate values. After the prefix and suffix computations, every processor holds N/p values $d_A(q_x \cdot \frac{N}{\sqrt{p}}, j) + d_B(j, q_y \cdot \frac{N}{\sqrt{p}})$ for $j \in [q \cdot \frac{N}{\sqrt{p}} : (q+1) \cdot \frac{N}{\sqrt{p}}]$.

Now we redistribute the data, assigning a different C-block $(q_x \cdot \frac{N}{\sqrt{p}}, q_y \cdot \frac{N}{\sqrt{p}}, \frac{N}{\sqrt{p}}, \frac{N}{\sqrt{p}})$ to each processor q. To be able to continue the recursive procedure at this level, every processor must have the $O(\frac{N}{\sqrt{p}})$ values of sequence $M^{(q_x \cdot \frac{N}{\sqrt{p}}, q_y \cdot \frac{N}{\sqrt{p}}, \frac{N}{\sqrt{p}}, \frac{N}{\sqrt{p}})}$, and the sets of relevant

nonzeros in D_A and D_B. Each processor holds at most N/p nonzeros in D_A and the same number of nonzeros in D_B. Imagine that the nonzeros are added one by one to initially empty matrices D_A and D_B. Each nonzero in D_A (respectively in D_B) can increase the overall number of j-blocks by at most 1 for each of the $\sqrt{p}$ C-blocks where this nonzero is relevant; a nonzero does not affect the number of j-blocks for any other C-blocks. Each j-block is assigned one value in the corresponding sequence M. Therefore, the total number of M-values per processor before redistribution is at most $N/p \cdot \sqrt{p} + p = N/\sqrt{p} + p$. The total number of M-values per processor after redistribution is $N/\sqrt{p}$, therefore the communication is perfectly balanced, apart from the maximum of p values that can arise due to processor boundaries "splitting" a j-block. Since every processor holds N/p nonzeros before redistribution, and every nonzero is relevant for $\sqrt{p}$ C-blocks, redistributing the relevant nonzeros can also be done in $O(N/\sqrt{p})$ communication. After this, every processor has all the data that are necessary to perform the sequential procedure from the previous section on its C-block. The resulting parallel highest-score matrix multiplication procedure has BSP cost $\mathsf{W} = O((N/\sqrt{p})^{1.5}) = O(N^{1.5}/p^{0.75})$, $\mathsf{H} = O(N/\sqrt{p})$ and $\mathsf{S} = O(1)$.

When applied to semi-local LCS computation, this algorithm is used at every level of a quadtree merging scheme. At the bottom level, the alignment dag is partitioned into a regular grid of p sub-dags of size $n/\sqrt{p} \times n/\sqrt{p}$. The highest-score matrix for each sub-dag is computed sequentially by a separate processor in computation work $O(N^2/p)$. Then the matrices are merged sequentially with computation work $O((N/\sqrt{p})^{1.5}) = O(N^{1.5}/p^{0.75})$. At higher levels of the quadtree, blocks are merged in parallel. In particular at level $\log r$, $1 \leq r \leq p$, the block size is $N/\sqrt{r}$, and each merge is performed by a group of p/r processors in computation work $\mathsf{W} = O(\frac{(N/r^{0.5})^{1.5}}{(p/r)^{0.75}}) = O(\frac{N^{1.5}/r^{0.75}}{p^{0.75}/r^{0.75}}) = O(\frac{n^{1.5}}{p^{0.75}})$ and communication $\mathsf{H} = O(\frac{(n/r^{0.5})}{(p/r)^{0.5}}) = O(\frac{n}{p^{0.5}})$. This analysis includes the root of the quadtree, where $r = 1$. Overall, the new algorithm runs in local computation $\mathsf{W} = O(\frac{n^2}{p} + \frac{N^{1.5}\log p}{p^{0.75}}) = O(n^2/p)$ (assuming that $n \geq p^2$), communication $\mathsf{H} = O(\frac{n \log p}{\sqrt{p}})$ and $\mathsf{S} = O(\log p)$ supersteps.

5 Conclusions and Outlook

In this paper, we have presented an efficient coarse grained parallel algorithm for semi-local string comparison based on a parallel highest-score matrix multiplication procedure. Our algorithm reduces the BSP overall communication cost H to $O(n \log p/\sqrt{p})$, running in local computation $\mathsf{W} = O(n^2/p)$ and using $\mathsf{S} = O(\log p)$ supersteps. The local computation cost can be reduced slightly by using a subquadratic sequential algorithm[14]. Thus, our algorithm is the first coarse-grained parallel LCS algorithm with scalable communication. Moreover, the algorithm is work-optimal, synchronisation-efficient, and solves a more general problem of semi-local string comparison. It is worth mentioning here that the algorithm can be extended to allow querying the actual longest common subsequences (as opposed to just their lengths). Also, the data structures used by this algorithm allow compact storage of the results, which is of advantage when comparing very large strings. Since this algorithm is a useful building block for solving various algorithmic problems[12], we intend to implement this algorithm in order to investigate its practicality and to provide an "algorithmic plug-in" for these applications.

References

1. C. E. R. Alves, E. N. Cáceres, F. Dehne and S. W. Song, *Parallel dynamic programming for solving the string editing problem on a CGM/BSP*, in: Proc. of 14th ACM SPAA, pp. 275–281, (2002).
2. C. E. R. Alves, E. N. Cáceres, F. Dehne and S. W. Song, *A parallel wavefront algorithm for efficient biological sequence comparison*, in: Proc. ICCSA, vol. **2668** of LNCS, (2003).
3. C. E. R. Alves, E. N. Cáceres and S. W. Song, *A BSP/CGM algorithm for the all-substrings longest common subsequence problem*, in: Proc. 17th IEEE/ACM IPDPS, pp 1–8, (2003).
4. C. E. R. Alves, E. N. Cáceres and S. W. Song, *A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem*, Algorithmica, **45**, 301–335, (2006).
5. J. L. Bentley, *Multidimensional divide-and-conquer*, Comm. ACM, **23**, 214–229, (1980).
6. R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*, (Oxford University Press, 2004).
7. T. Garcia and D. Semé, *A load balancing technique for some coarse-grained multicomputer algorithms*, in: SCA 21st International Conference on Computers and Their Applications (CATA '06), pp. 301–306, (2006).
8. J. Jájá, C. Mortensen and Q. Shi, *Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Range Counting*, in: Proc. 15th ISAAC, vol. **3341** of LNCS, (2004).
9. V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, Sov. Phys. Dokl., **6**, 707–710, (1966).
10. W. F. McColl, *Scalable Computing*, in: Computer Science Today: Recent Trends and Developments, vol. **1000** of LNCS, pp. 46–61, (Springer-Verlag, 1995).
11. J. P. Schmidt, *All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings*, SIAM Journal on Computing, **27**, 972–992, (1998).
12. A. Tiskin, *Semi-local string comparison: Algorithmic techniques and applications*, Mathematics in Computer Science, `arXiv:0707.3619[cs.DS]`, (2007, in press).
13. A. Tiskin. *Efficient representation and parallel computation of string-substring longest common subsequences*, in: Proc. ParCo05, vol. **33** of NIC Series, pp. 827–834, (John von Neumann Institute for Computing, 2005).
14. A. Tiskin, *All semi-local longest common subsequences in subquadratic time*, *Journal of Discrete Algorithms*, (2008, in press).
15. L. G. Valiant, *A bridging model for parallel computation*, Comm. ACM, **33**, 103–111, (1990).
16. R. A. Wagner and M. J. Fischer, *The string-to-string correction problem*, J. ACM, **21**, 168–173, (1974).

Parallel Programming Models

Implementing Data-Parallel Patterns for Shared Memory with OpenMP

Michael Suess and Claudia Leopold

University of Kassel, Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
E-mail: {msuess, leopold}@uni-kassel.de

AthenaMP[1] is an open-source parallel pattern library in C++ and OpenMP that is currently under development. This paper reports on the implementations of several data-parallel patterns: `modify_each`, `transmute`, `combine`, `reduce`, and `filter`. We describe the patterns as well as our experiences implementing them in C++ and OpenMP. The results of two benchmarks showing no performance losses when compared to a pure OpenMP implementation are also included.

1 Introduction

With the advent of multi-core processors, parallel programming for shared memory architectures is entering the mainstream. However, parallel programming in general is considered to be difficult. One solution to this problem lays in the use of libraries that hide parallelism from the programmer. This has been practised in the field of numerics for a long time, where premium quality libraries that encapsulate parallelism are available. For the more general field of programming, design patterns are considered a good way to enable programmers to cope with the difficulties of parallel programming[2].

Although there are a variety of libraries available both commercially and in a research stage, what is missing from the picture is a pattern library in OpenMP. Since its introduction in 1997, OpenMP has steadily grown in popularity, especially among beginners to parallel programming. Its ease of use is unmatched by any other system based on C/C++ or Fortran available today, while at the same time offering an at least acceptable performance. Therefore, design patterns implemented in OpenMP should be a viable learning aid to beginners in the field of concurrent programming. At the same time, they are able to encapsulate parallelism and hide it from the programmer if required. For this reason, the AthenaMP project[1] was created that implements various parallel programming patterns in C++ and OpenMP.

The main goal of AthenaMP is to provide implementations for a set of concurrent patterns, both low-level patterns like advanced locks (not described here), and higher-level patterns like the data-parallel patterns described in this paper or task-parallel patterns like taskpools or pipelines. The patterns demonstrate solutions to parallel programming problems, as a reference for programmers, and additionally can be used directly as generic components. Because of the open source nature of the project, it is even possible to tailor the functions in AthenaMP to the need of the programmer. The code is also useful for compiler vendors testing their OpenMP implementations against more involved C++-code. A more extensive project description is provided by one of the authors in his weblog[a].

[a] http://www.thinkingparallel.com/2006/11/03/a-vision-for-an-openmp-pattern-library-in-c/

Another contribution of this paper is a description of implementation problems that we faced while developing our patterns: the insufficient state of the compilers with regard to OpenMP and C++, as well as the restricted ability to influence the scheduling of parallel loops at runtime.

Section 2 starts with a general introduction to the features of our patterns/generic components and goes on to highlight each one of them in detail. The performance of our solutions is described in Sect. 3 using two benchmarks. Some implementation details and problems with current compilers are described in Sect. 4. Sect. 5 shows related work, and Sect. 6 closes the paper with a short summary.

2 Data-Parallel Patterns in Detail

This section introduces the data-parallel patterns contained in AthenaMP to date. First, we explain some features of our implementation common to all of them.

The interfaces of our functions realizing the patterns have been designed using the Standard Template Library (STL) as an example where applicable. Just like in the STL, all functions presented here are polymorphic through the use of templates. It is possible to mix the library calls freely with normal OpenMP code. This is useful e.g. for employing nested parallelism, where the user specifies the first level of parallelism in his own code and the second level is opened up by the library.

All functions take two or more iterators specifying one or more ranges of elements as parameters, as is customary in the STL. Behind the scenes, depending on the iterators supplied, there are two versions of the patterns: one that takes random-access iterators (as supplied by e.g. std::vectors or std::deques), and one that takes forward or bidirectional iterators (as supplied by e.g. std::lists). Using template metaprogramming (iterator traits), the right version is selected transparently to the user by the compiler.

The first version makes use of the `schedule`-clause supplied with OpenMP. Although it is set to static scheduling, this is easily changeable in the source of the library. When the functor supplied takes a different amount of time for different input values, using a dynamic schedule is a good idea. This random-access version is fast and has easy to understand internals, because it uses OpenMP's `parallel for` construct.

The second version is for forward iterators. Use of the scheduling functionality supplied by OpenMP would be very costly in that case, as to get to a specific position in the container, $O(n)$ operations are necessary in the worst case. This operation would need to be carried out n times, resulting in a complexity of $O(n^2)$. For this reason, we have implemented our own static scheduling. The internal `schedule_iter` template function is responsible for it. It takes references to two iterators. One of them marks the start and the other the end of the range to be processed. The function calculates the number of elements in the range, divides it among the threads, and changes the iterators supplied to point to the start and the end of the range to be processed by the calling thread (by invoking `std::advance`). This results in a total complexity of $O(n * t)$, where t is the number of threads involved. Iff each thread in a team calls the function and works on the iterators returned, the whole range is processed with only a few explicit calls to `std::advance`. However, the forward iterator version is slower and less flexible (because changing the scheduling policy is not possible with this version) than the random access iterator version, as can be observed in Section 3.

The parallelism inherent in the patterns described is totally hidden from the user, except of course when looking into the AthenaMP source code. This does not necessarily lead to smaller sources (as a lot of scaffolding code for the new functors is required), but to less error prone ones. A lot of mistakes are common when working with OpenMP[3] (or any other parallel programming system), and these can be reduced by deploying the patterns.

The last parameter to each function is the number of threads to use in the computation. If the parameter is left out, it defaults to the number of threads as normally calculated by OpenMP. Unless otherwise specified in the description of the pattern, the original order of the elements in the range is preserved and no critical sections of any kind are required.

It is also possible to nest patterns inside each other, e.g. by calling `reduce` inside a functor supplied to `modify_each` for two-dimensional containers. Nested parallelism is employed in this case, which becomes especially useful if you have a large number of processors to keep busy.

The following patterns and their implementations are discussed in the next sections:

- `modify_each`: also commonly known as *map*, modifies each element supplied
- `transmute`: also known as *transform* applies a function to each element provided and returns a new range containing the results of the operation
- `combine`: combines elements from two sources using a binary function and returns a new range containing the results of the operation
- `reduce`: also known as *fold*, combines all elements into a single result using an associative operation
- `filter`: filters the elements supplied according to a predicate function and returns the results using a new container

2.1 Modify_each

The `modify_each` pattern provides a higher-order template function to apply a user-supplied functor to each element in a range in parallel. No provisions are made to protect any internal data in the functor or any side-effects of the functor from concurrent access. Fig. 1 shows an example, where the pattern is used to add ten to all elements in an std::vector. The user is relatively free with regards to the functor supplied, as long as it contains an `operator()` method that is a unary function and takes a non-const reference as argument. If `operator()` has a return value, it is ignored.

2.2 Transmute

The `transmute` pattern is similar to `modify_each` in the way that it applies a user-supplied unary functor to all elements in a range in parallel. While `modify_each` works on the original elements and modifies them, `transmute` stores its results in a different range and is therefore even able to change the type of each element. It is known in the STL as `transform` (but similar to many of our patterns, we could not use this name to avoid name-clashes with the STL-version).

```
1
2  /** a user-supplied functor that adds diff to its argument */
3  class add_func : public std::unary_function<int, void> {
4    public:
5      add_func (const int diff) : diff_ (diff) {}
6      void operator() (int& value) const { value += diff_; }
7    private:
8      const int diff_;
9  };
10
11 /* add 10 to all targets in place. */
12 athenamp::modify_each (targets_.begin(), targets_.end(),
13     add_func (10));
```

Figure 1. `modify_each` in action

```
/* combine all elements from sources1_ with all elements from
 * sources2_ and store the results in sources1_ */
athenamp::combine (sources1_.begin(), sources1_.end(),
    sources2_.begin(), sources1_.begin(), std::plus<int>());
```

Figure 2. `combine` in action

The list of parameters it takes is similar to `modify_each` again, except for the fact that it takes an additional iterator that points to the location where the results are to be stored. The user is responsible for making sure that there is enough room to store all results. The user-supplied functor is similar to the one supplied for `modify_each` again, except for the fact that it cannot work on its argument directly, but returns the result of its computation instead. The result is then put into the appropriate location by our library function `transmute`.

No provisions are made to protect any internal data in the functor or any side-effects of the functor from concurrent access. While it is possible and correct to use `transmute` to apply changes inplace by overlapping its supplied ranges, it is recommended to use `modify_each` in that case because it is faster, since it involves less copying of elements.

An example is omitted here to keep the paper short and because of the similarity of this method to the already explained `modify_each` pattern.

2.3 Combine

`combine` is a relatively simple pattern that is used to combine elements from two different ranges using a binary operation and put the result into a third range. It is similar to the `transmute` pattern explained above, except that it works on two ranges instead of one. Similar restrictions as for `transmute` also apply here: the user is responsible for making sure there is enough room to put the results in and the internals of the functor are also not protected from concurrent access. It is also possible to store the results inplace, as shown in Fig. 2. What is also shown in this figure is that it is possible to use the functors provided by the STL for this pattern (`std::plus` in this example). If this is the case, many lines of code can be saved when compared to the parallel version without patterns.

```
/** A functor that can be used to find the maximum and sum of all
 * elements in a range by repeatedly applying operation() to all
 * elements in the range. */
class max_sum_functor : public std::unary_function<int, void> {
  public:
    max_sum_functor (int max, int sum) : max_(max), sum_(sum) {};
    int max () const { return max_; }
    int sum () const { return sum_; }

    void operator() (const int arg1)
    {
        max_ = std::max (arg1, max_);
        sum_ += arg1;
    }

    void combine (const max_sum_functor& func)
    {
        max_ = std::max (func.max (), max_);
        sum_ += func.sum ();
    }

  private:
    int max_;
    int sum_;
};

max_sum_functor func (-1, 0);
/* calculate maximum and sum of all elements in vector targets_ */
athenamp::reduce (targets_.begin(), targets_.end(), func);
/* check result */
std::cout << func.max() << std::endl << func.sum() << std::endl;
```

Figure 3. `reduce` in action

2.4 Reduce

The `reduce` pattern combines all elements in a given range into a single result using a binary, associative operation. It is also commonly known as *fold* or *for_each*. Many parallel programming systems feature a reduce-operation, among them OpenMP. The reduce-operation in OpenMP has a disadvantage, though: it is limited to a few simple, predefined operators, such as $+$ or $*$. These operators can only be applied to a few data-types, such as `int`'s.

Our `reduce`-method solves these problems, as a user-defined functor is specified as an operator. This functor can work on any data-type. Multiple reductions in a single pass are possible as well, with a functor that does two or more operations at the same time. An example (see Fig. 3) will make things clearer, before we go into more details. The functor in the example stores the variables `max_` and `sum_` internally. They are initialized appropriately in the constructor and can be read after the operation has completed using the `max` and `sum` methods. `operator()` is applied to each element in the range to find the maximum element and the sum of all elements.

Internally, the `reduce`-method creates a copy of the functor for each thread involved in the calculation. For this reason, the functors supplied must also have a copy constructor. In our example, the compiler takes care of this correctly, therefore we have omitted it. At the end of the reduction, the `combine`-method (which has nothing in common with the `combine` pattern mentioned in the last section!) is used to correctly combine the

```
/* filter all odd numbers and store them into results_ */
results_ = athenamp::filter<std::list<int> >(targets_.begin(),
   targets_.end(), std::bind2nd (std::modulus<int>(), 2));
```

Figure 4. `filter` in action

different functors from each thread. As can be seen in the example, there is no need to protect anything from concurrent access, as the `reduce`-method does this automatically. As a downside, this also means that our implementation has a critical region that must be carried out once by all threads involved, which of course decreases the performance slightly, especially for quick operations on few elements.

2.5 Filter

The `filter` pattern filters a range of objects according to a predicate functor and stores all results for which the predicate returns true into a new container. The target container must be specified as template parameter and must offer both `push_back` and `insert`-methods in its interface (all STL sequence containers do). A small example to make things clearer is shown in Fig.4.

In this example, all odd numbers are filtered out of the targets_-vector and stored into the results_-list. The predicate functor can either be a standard one from the STL (as shown in the example) or a user-defined one. In all cases, no protection of the internals of the functor from concurrent access is guaranteed. This should not be a problem, as most functors used in this case do not have any internals to protect.

The implementation has no critical sections, but a small sequential part at the end where each thread copies its objects into a new container that is returned afterwards.

3 Performance

Measuring performance of generic components such as the ones provided here is hard, as it depends heavily on the user code that is to be parallelized. Most patterns do not need locks or perform at most one locking operation per thread. Therefore, they are able to scale to a large number of processors. If the amount of work to be done in the user-supplied functor is too small, however, bus contention will become an issue and performance may not be satisfactory – but this is the case for pure OpenMP as well.

We have performed two different tests on our components. For brevity, we can only report the results for the `modify_each` pattern here. For the first test, we incorporated the pattern into the game-like application OpenSteerDemo[4], which is a testbed for the C++ open-source library OpenSteer written by Reynolds. It simulates and graphically displays the steering behaviour of autonomous computer-controlled characters, called agents, in real-time. No difference at all was measurable between the pure OpenMP version and our pattern version. We expect this to be the case in most applications.

The second test refers to the case that the user-supplied functor is too small. Our benchmark uses `modify_each` to add one to all elements in a vector or list, respectively. The results are shown in Fig. 1. Of course, this benchmark is not representative in any way,

Platform (Threads)	`modify_each` (lists)	OpenMP variant (lists)
AMD (4)	0.11 (8.06)	0.09 (7.56)
SPARC (8)	2.39 (35.8)	2.38 (34.8)
IBM (8)	0.15 (3.35)	0.14 (3.40)

Table 1. Wall-clock times in seconds for the `modify_each` function. The values in braces are measurements for the version of `modify_each` using forward iterators. All tests carried out on containers with 100.000 elements, using 10.000 repetitions.

as it is clearly limited by the available memory bandwidth. Yet, since this is true for the pure OpenMP-version as well, it shows that the overhead introduced by using the pattern is negligible even for this case.

It can also clearly be observed that the version of our patterns using forward iterators (as found in lists) is several orders of magnitude slower than the one for random access iterators. The reasons for this behaviour have been explained in Section 2. For our other patterns, similar results can be observed.

4 General Remarks and Evaluation

This section lists some implementation details and problems that were common to all patterns described here. The biggest problems while implementing the patterns was the state of the compilers available today. OpenMP and C++ are rarely used together in practice, and therefore many compilers are still buggy when it comes to this combination. This manifests itself in valid code that cannot be compiled on some compilers, or in subtle bugs that creep in as soon as high optimization settings are turned on. These high optimization settings are needed, though, because the way of programming shown here heavily relies on inlining. For example, without inlining there will be one function called for every element in the range to be processed by our patterns. When the function itself does not do much (as the one shown in Sect. 2.1), all performance gains from parallelization are lost. Good optimization by the compilers is therefore crucial to our work.

Another problem was the lack of support for changing the scheduling policy for parallel loops at runtime. With this feature, it would be possible to pass a parameter to any data-parallel pattern and let the user decide which scheduling policy works best for his particular problem. As an example, when the functors supplied to the pattern take a different amount of time to execute depending on their input, dynamic or guided scheduling usually works best. Currently, the user has to change the parameter inside the library or copy the code into his own program, where it can be modified.

Changing the scheduling policy at runtime is only possible with an environment variable in OpenMP. This feature is mostly used as a debugging aid, because one can only change the schedule of all loops (with a schedule set to `runtime`) at once. Use of environment variables is also not practical for a library. Letting the `schedule`-clause take a user-supplied parameter (e.g. a string) would solve this problem and is presently discussed in the OpenMP language committee.

5 Related Work

Many of the data-parallel patterns described here are similar to functionality provided by the Standard Template Library (STL). There are a variety of parallel STL-implementations in a research stage, among them STAPL[5] or PSTL[6]. Similar functionality is also starting to appear in commercial projects, such as the Intel Threading Building Blocks or in QT Concurrent. What differentiates AthenaMP from these projects is its strong focus on OpenMP on one hand (which no other project we know of provides), and its ability for programmers to study the source and learn from that. AthenaMP does not restrict itself to patterns found in the STL, but there are also task-parallel patterns (e.g. taskpool, pipeline), synchronization patterns (e.g. scoped locks, guard objects) and many more available in the library, which are not described in this paper, though. By using the expressiveness of OpenMP, its code is far easier to understand and adapt than any of the other libraries we are aware of.

6 Summary

This paper shows how it is possible to implement data-parallel patterns modelled after the Standard Template Library (STL) with OpenMP. The implementations we have described are part of the AthenaMP project[1]: `modify_each`, `transmute`, `combine`, `reduce`, and `filter`. Common characteristics are their iterator-based interface, the dynamic selection of the best suited implementation depending on the iterator-type supplied, and their ability to nest inside each other. The patterns were described in detail and with examples, along with performance measurements for a simple benchmark and the game-like application OpenSteerDemo. Afterwards, some implementation problems we had with OpenMP were shown, the biggest one being the state of the OpenMP-compilers today and the lack of support for runtime scheduling in OpenMP. The paper closed with a summary of related work.

References

1. M. Suess, AthenaMP, `http://athenamp.sourceforge.net/`, (2006).
2. T. G. Mattson, B. A. Sanders and B. L. Massingill, *Patterns for Parallel Programming (Software Patterns Series)*, (Addison-Wesley Professional, 2004).
3. M. Suess and C. Leopold, *Common mistakes in OpenMP and how to avoid them*, in: Proc. International Workshop on OpenMP - IWOMP'06, (2006).
4. B. Knafla and C. Leopold, *Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP*, in: Proc. International Conference on Parallel Computing (PARCO 2007), Juelich, Germany, (2007).
5. P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato and L. Rauchwerger, *STAPL: An Adaptive, Generic Parallel C++ Library*, Workshop on Languages and Compilers for Parallel Computing, pp. 193–208, (2001).
6. E. Johnson and D. Gannon, *HPC++: experiments with the parallel standard template library*, Proc. 11th international conference on Supercomputing, pp. 124–131, (1997).

Generic Locking and Deadlock-Prevention with C++

Michael Suess and Claudia Leopold

University of Kassel, Research Group Programming Languages / Methodologies
Wilhelmshöher Allee 73, D-34121 Kassel, Germany
E-mail: {*msuess, leopold*}*@uni-kassel.de*

Concurrent programming with threads heavily relies on locks. The locks provided by most threading systems are rather basic and suffer from a variety of problems. This paper addresses some of them, namely deadlocks, lack of exception-safety, and their procedural style. We describe higher-level locks that can be assembled from the more basic ones. Throughout the paper, we refer to C++ and OpenMP for parallelization, but most of the functionality is generic and independent from OpenMP. This work is part of the AthenaMP project.

1 Introduction

Locks are one of the most important building blocks of concurrent programming today. As with the advent of multi-core processors, parallel programming starts to move into the mainstream, the problems associated with locks become more visible, especially with higher-level languages like C++. This paper addresses the following ones:

- lock initialization and destruction is not exception-safe and very C-ish. Lock destruction is forgotten frequently.
- setting and releasing locks is not exception-safe and C-ish, as well. Unsetting Locks may be forgotten in complicated code-paths with a lot of branches.
- deadlocks are possible when using multiple locks

To solve the first two problems, a common C++ idiom called *RAII* is used. RAII stands for *Resource Acquisition is Initialization*[1] and combines acquisition/initialization and release/destruction of resources with construction and destruction of variables. Our solution to the first problem is called lock adapter and has the effect that locks are initialized and destroyed in constructors and destructors, respectively. Our solution for the second problem is already well known as guard objects or *scoped locking* and means that locks are set and unset in constructors and destructors, respectively.

The third problem is solved in two ways: first by extending the guard objects to multiple locks and internally choosing the locking order in a deterministic way, and second by introducing so-called leveled locks that enable the creation and automatic control of a lock hierarchy that detects possible deadlocks at runtime.

The generic locking functionality presented here is part of the AthenaMP open source project[2]. Its main goal is to provide implementations for a set of concurrent patterns (both low-level patterns like advanced locks, and higher-level ones) using OpenMP and C++. These patterns demonstrate solutions to parallel programming problems, as a reference for programmers, and additionally can be used directly as generic components. The code is also useful for compiler vendors testing their OpenMP implementation against more

involved C++-code, an area where many compilers today still have difficulties. A more extensive project description is provided by one of the authors in his weblog[a].

The term *generic* warrants some more explanations at this point. From all the functionality introduced in this paper, only the adapters are tied to a particular lock type as provided by the parallel programming system (PPS). New adapters are trivial to implement for different threading systems, and we have done so as a proof-of-concept for POSIX Threads. As soon as these adapters exist, higher-level functionality built on top of the adapters can be used. A slight deviation from this rule are levelled locks — they have a configurable backend to actually store the level-information, and one of these backends uses thread-local storage as is available in OpenMP. To make up for this, an otherwise identical backend has been implemented that is as generic as the rest of the components.

Sect. 2 describes the generic lock adapters and guard objects. With this infrastructure in place, Sect. 3 concentrates on the important problem of deadlocks. Benchmarks to evaluate the performance penalties associated with the additional functionality provided are given in Sect. 4. Some implementation problems we had are shown in Sect. 5. Related work and our contributions are highlighted in Sect. 6, a summary closes the paper in Sect. 7.

2 Generic Locking

In this section, we introduce the basic locking constructs provided by the AthenaMP library in detail, along with short examples on how to use them. The first class of locking constructs (called lock adapters) is described in Sect. 2.1. The concept of *scoped locking* is applied to these adapters in Sect. 2.2, leading to guard objects.

2.1 Lock Adapters

A lock adapter is a simple wrapper-object for a traditional lock as found in most threading systems. In AthenaMP, lock adapters are provided for the OpenMP types `omp_lock_t` and `omp_nest_lock_t`. The interface for an `omp_lock_ad` is shown in Fig. 1, along with a simple example of how to use them in Fig. 2. The interface should be self-explanatory, possibly except for the `get_lock`-method. It is useful, if you need to call library routines that expect the native lock type, as explained by Meyers[3]. In the example, a simple bank transfer function is sketched (stripped to the bare essentials). The locks are encapsulated in an `account`-class in this case, each account has its own lock to provide for maximum concurrency.

Adapters for other lock types are trivial to implement and we have done so as a proof-of-concept for a `pthread_mutex_t`. Their primary purpose is to adapt the different interfaces of the provided lock types to a common interface, which can be relied upon for the more advanced locks and abstractions provided by AthenaMP.

A nice side-effect of the lock adapters is that they turn the traditional C-style locks into more C++-style types. An example: it is a common mistake in OpenMP to forget to initialize a lock before using it (using `omp_init_lock`), or to forget to destroy it (using `omp_destroy_lock`) after it is needed. The *Resource Acquisition is Initialization*[1]

[a] http://www.thinkingparallel.com/2006/11/03/a-vision-for-an-openmp-pattern-library-in-c/

```
class omp_lock_ad {
public:
  omp_lock_ad();
  void set();
  void unset();
  int test();
  omp_lock_t& get_lock();
  ~omp_lock_ad();
};
```

Figure 1. Lock Adapter Interface

```
class account {
  omp_lock_ad lock;
  double balance;
};

void bank_transfer (account& send,
  account& recv, double amount)
{
  send.lock.set();
  recv.lock.set();

  send.balance -= amount;
  recv.balance += amount;

  recv.lock.unset();
  send.lock.unset();
}
```

Figure 2. A Bank Transfer with Lock Adapters

(*RAII*) idiom is employed to avoid the mistake: the locks are initialized in the constructor of the lock adapter and destroyed in the destructor. This ensures that locks are always properly initialized and destroyed without intervention from the programmer, even in the presence of exceptions. Should an exception be thrown and the thread leaves the area where the lock is defined, the lock adapter will go out of scope. As soon as this happens, its destructor is called automatically by the C++ runtime library, properly destroying the lock in the process. Thus, our lock adapters are exception-safe.

2.2 Scoped Locking with Guard Objects

The first generic lock type that builds on the lock adapters is called `guard`. It employs the *RAII*-idiom once again. This special case is so common that it has its own name: *scoped locking*[4]. A local, private guard object is passed a lock as a parameter in its constructor. The guard object sets the lock there and releases it when it goes out of scope (in its destructor). This way, it is impossible to forget to unset the lock (another common mistake when dealing with locks), even in the presence of multiple exit points or exceptions (as described in Sect. 2.1).

It is also possible to unset the lock directly (using `release`) and to acquire again later (using `acquire`), or to extract the lock out of the guard object (using `get_lock`). The interface of the guard objects is presented in Fig 3, along with a short example (a bank transfer again) in Fig. 4.

A guard object as implemented in AthenaMP can be instantiated with any locking class that provides the basic locking methods (i.e. `set`, `unset`, `test`). It does not depend on OpenMP in any way.

3 Deadlock Detection and Prevention

Deadlocks are an important and all-too common problem in multi-threaded code today. Consider the example code sketched in Fig. 2 again that shows how a bank transfer can

```
template <class LockType>
class guard {
public:
  guard (LockType& lock);
  void acquire();
  void release();
  LockType& get_lock();
  ~guard();
};
```

Figure 3. Guard Interface

```
void bank_transfer (account& send,
  account& recv, double amount)
{
  guard<omp_lock_ad> send_guard(send.lock);
  guard<omp_lock_ad> recv_guard(recv.lock);

  send.balance -= amount;
  recv.balance += amount;
}
```

Figure 4. A Bank Transfer with Guard Objects

be implemented. A deadlock might occur in this code, as soon as one thread performs a transfer from one account (let's call it account no. 1) to a different account (account no. 2), while another thread does a transfer the other way round (from account no. 2 to account no. 1) at the same time. Both threads will lock the sender's lock first and stall waiting for the receiver's lock, which will never become available because the other thread already owns it. More subtle deadlocks might occur, as soon as more accounts are involved, creating circular dependencies.

Sect. 3.1 describes a way to detect deadlocks semi-automatically, along with a corresponding implementation. Sect. 3.2 shows how to avoid deadlocks for an important subproblem.

3.1 Deadlock Detection using Levelled Locks

A common idiom to prevent deadlocks are lock hierarchies. If you always lock your resources in a predefined, absolute order, no deadlocks are possible. For our example, this means e.g. always locking account no. 2 before account no. 1. It can sometimes be hard to define an absolute order, though, a possible solution for part of this problem is described in Sect. 3.2.

Once a lock hierarchy for a project is defined, it may be documented in the project guidelines and developers are expected to obey it. Of course, there are no guarantees they will actually do so or even read the guidelines, therefore a more automated solution may be in order.

This solution is provided in the form of our second generic lock type: the `levelled_lock`. It encapsulates a lock adapter and adds a `_lock_level` to it, which is passed into the constructor of the class, associating a level with each lock. If a thread already holds a lock, it can only set locks with a lower (or the same - to allow for nested locks) level than the ones it already acquired. If it tries to set a lock with a higher level, a runtime exception of type `lock_level_error` is thrown, alerting the programmer (or quality assurance) that the lock hierarchy has been violated and deadlocks are possible.

The interface of the levelled locks and the bank transfer example are skipped here for brevity, as they are both very similar to the ones shown for the lock adapters, except of course for the additional `_lock_level`-parameter.

Like guard objects, a levelled lock as implemented in AthenaMP can be instantiated with any locking class that provides the basic locking methods (i.e. `set`, `unset`, `test`). Since the levelled locks provide these methods as well, guard objects can also be instantiated with them, thereby combining their advantages.

Our levelled locks have a configurable backend (via a template parameter) that actually stores the locks presently held for each thread. The first version we implemented depended on OpenMP, since their implementation uses threadprivate memory internally. This has also been a major implementation problem, because we found no OpenMP-compiler that could handle static, threadprivate containers — although judging from the OpenMP–specification this should be possible.

To fix this, a more generic backend was implemented. This version does not use threadprivate memory and does not depend on OpenMP. Instead it uses another generic component implemented in AthenaMP called `thread_storage`. The component stores all data in a vector that is indexed by a user-supplied thread-id. For OpenMP, this can be the number returned by `omp_get_thread_num`, for all other threading systems its the users responsibility to supply this id.

Both versions of locks with level checking induce a performance penalty. Furthermore, the use of throwing runtime exceptions in production code is limited. For this reason, a third backend for the levelled lock was implemented. It has the same interface as the other backends, but does no level checking. Of course, the basic locking functionality is provided. This class enables the programmer to switch between the expensive, checked version and the cheap, unchecked version with a simple `typedef`-command. This version of the levelled lock is called `dummy_levelled_lock` for the rest of this paper.

3.2 Deadlock Prevention using Dual-Guards / n-Guards

It has been shown in Sect. 3.1 that lock hierarchies are a very powerful measure against deadlocks. An argument that has been used against them in the past is that you may not be able to assign a unique number to all resources throughout the program. This is easy in our example of bank transfers, because every account most likely has a number and therefore this number can be used. It becomes more difficult when data from multiple sources (e.g. vectors, tables or even databases) need to share a single hierarchy.

While the general problem is difficult to solve, there is a solution for an important subclass: in our bank transfer example, two locks are needed at the same time for a short period. Even if there was no account number to order our locks into a hierarchy, there is another choice: although not every resource may have a unique number associated with it, every lock in our application does, since the lock's address remains constant during its lifetime.

One possible way to use this knowledge is to tell the user of our library to set their locks according to this implied hierarchy. But there is a better way: As has been described in Sect. 2.2, guard objects provide a convenient way to utilize exception-safe locking. Merging the idea presented above with guard objects results in a new generic locking type: the `dual_guard`. Its interface is sketched in Fig. 5 and our bank transfer example is adapted to it in Fig. 6.

No deadlocks are possible with this implementation, because the dual-guard will check the locks' addresses internally and make sure they are always locked in a predefined order (the lock with the higher address before the lock with the lower address to make them similar to the levelled locks as introduced in Sect. 3.1).

It is also obvious from this example, how the generic high-level lock types provided by AthenaMP raise the level of abstraction when compared to the more traditional locks

```
template <class LockType>
class dual_guard {
public:
  dual_guard(LockType&
    lock1, LockType& lock2);
  void acquire();
  void release();
  LockType& get_lock1();
  LockType& get_lock2();
  ~dual_guard();
};
```

Figure 5. Dual-Guard Interface

```
void bank_transfer (account& send,
  account& recv, double amount)
{
  dual_guard<omp_lock_ad>
    sr_guard(send.lock, recv.lock);

  send.balance -= amount;
  recv.balance += amount;
}
```

Figure 6. A Bank Transfer with Dual-Guards

offered by the common threading systems: to get the functionality that is provided with the one line declaration of the dual-guard, combined with an `omp_lock_ad`, two calls to initialize locks, two calls to destroy locks, two calls to set the locks and two calls to unset the locks are necessary, resulting in a total amount of eight lines of code. These eight lines of code are not exception-safe and possibly suffer from deadlocks, where the dual-guards are guaranteed to not have these problems.

As always, a dual-guard object as implemented in AthenaMP can be instantiated with any locking class that provides the basic locking methods (i.e. `set`, `unset`, `test`). It does not depend on OpenMP in any way. This also includes all variants of the levelled locks described in Sect. 3.1. When instantiated with a `levelled_lock`, using the lock's address to order them into a hierarchy is unnecessary, as there is a user-provided lock level inherent in these locks. In this case, the lock level is used for comparing two locks by the dual-guards automatically.

Generalizing the ideas presented in this section, we go one step further in AthenaMP and also provide an `n_guard` that takes an arbitrary number of locks. Since variadic functions are not well-supported in C++ and it is generally not recommended to use `varargs`, we have decided to let it take a `std::vector` of locks as argument. Apart from this, the functionality provided by these guard objects is equivalent to the dual-guards described above.

It should be noted again, that the dual-guards and n_guards presented in this section are only able to solve a subset of the general deadlock problem: only when two (or more) locks have to be set at the same point in the program, they can be employed. The solution is not applicable as soon as multiple locks have to be set at different times, possibly even in different methods, because the guards must be able to choose which lock to set first.

4 Performance

We carried out some really simple benchmarks to evaluate, how much the added features and safety impact the performance of locking. Table 1 shows how long it took to carry out a pair of set / unset operations for the respective locks. Table 2 shows the same data for the guard objects. All numbers are normalized over the course of 500.000 operations. To make the numbers for the dual-guards and n-guards comparable, we have also normalized those to one pair of operations by dividing by two or n respectively ($n = 20$ for the benchmark shown here).

Platform (Threads)	omp_lock_t	critical	omp_lock_ad	dummy_ll	ll
AMD (4)	0.40	0.38	0.56	0.67	1.48
SPARC (8)	1.02	1.47	1.09	1.56	4.96
IBM (8)	0.41	0.63	0.41	0.44	1.23

Table 1. Comparison of wall-clock times (in milliseconds) for one pair of lock/unlock operations.

Platform (Threads)	guard	dual_guard	n_guard (20 locks)
AMD (4)	0.65	0.43	0.26
SPARC (8)	1.51	0.87	0.37
IBM (8)	0.42	0.42	0.41

Table 2. Wall-clock times (in milliseconds) for one pair of lock/unlock operations for AthenaMP guards.

These tables show that there is a performance penalty associated with the added functionality. The levelled locks (shown as ll in the table) are the worst offenders and therefore the implementation of the dummy levelled lock (dummy_ll) does make a lot of sense.

5 Implementation Problems

The single most important problem when implementing the functionality described here is the present state of the compilers. Although we have tried various versions of different compilers, we have found not a single one that was able to compile all of our code and testcases, although we are quite sure they are valid according to both the C++ and OpenMP specifications. To name just a few of the problems we have encountered:

- Many compilers do not allow containers to be declared `threadprivate`.
- Local static variables cause problems with many compilers when declared `threadprivate`.
- Others have more or less subtle problems with exceptions when OpenMP is enabled (details can be found in a weblog entry of one of the authors[b]).

The functionality described here was implemented in merely about one thousand lines of code – yet, to convince ourselves that it is actually correct took a disproportionate amount of time. This convinced us that C++ and OpenMP is obviously not a combination that is wide-spread, at least judging from the state of these compilers.

6 Related Work and Contributions

An implementation of guard objects in C++ can be found in the `Boost.Threads` library[5], where they are called `boost::mutex::scoped_lock`. Their approach to the

[b] http://www.thinkingparallel.com/2007/03/02/exceptions-and-openmp-an-experiment-with-current-compiler s/

problem is different, though: Boost tries to be portable by providing different mutex implementations for different platforms. Our guard objects and high-level locks work on any platform, where a lock adapter can be implemented. Beyond that, this approach allows us to provide guard objects and advanced locking constructs on top of different lock variants, e.g. mutex variables, spinlocks, nested locks or others.

ZThreads[6] provides portable implementations for different lock types with C++, as well. The library includes guard objects that are instantiable with different lock types, but does not include our more advanced abstractions (e.g. levelled locks or dual-guards). It is a portable threading library, with focus on low level abstractions like condition variables and locks. AthenaMP, on the other hand, builds on OpenMP (which is already portable) and can therefore focus on higher-level components and patterns.

The idea of using lock hierarchies to prevent deadlocks is well-known[7]. The idea to automatically check the hierarchies has been described by Duffy for C#[8]. There is also a dynamic lock order checker called Witness available for the locks in the FreeBSD kernel[9].

As far as we know, none of the functionality described in this paper has been implemented with C++ and OpenMP. The idea of using memory addresses of locks to create a consistent lock hierarchy is also a novel contribution of this paper, as is the use of dual-guards (and n-guards) to hide the complexity of enforcing the lock hierarchy from the user.

7 Summary

Locks are still an important building block for concurrent programming, but the locks provided by most parallel programming systems are rather basic and error prone. In this paper, we have presented higher-level classes that encapsulate the locking functionality and provide additional value, in particular automatic lock hierarchy checking, automatic setting of multiple locks in a safe order, as well as exception-safety. We implemented our ideas in a generic way, decoupled from the parallel programming system used. In the future, we plan to implement other patterns with C++ and OpenMP, in the AthenaMP project.

References

1. B. Stroustrup, *The C++ Programming Language, Third Edition*, (Addison-Wesley 1997).
2. M. Suess, *AthenaMP*, `http://athenamp.sourceforge.net/`, (2006).
3. S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd edition, (Addison-Wesley, 2005).
4. D. C. Schmidt, M. Stal, H. Rohnert and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2 – Networked and Concurrent Objects*, (Wiley, 2000).
5. W. E. Kempf, *The boost.threads library*, `http://www.boost.org/doc/html/threads.html`, (2001).
6. E. Crahen, *Zthreads*, `http://zthread.sourceforge.net/`, (2000).
7. A. S. Tanenbaum, *Modern Operating Systems*, 2nd edition, (Prentice Hall, 2001).
8. J. Duffy, *No more hangs*, MSDN Magazine, **21**, (2006).
9. J. H. Baldwin, *Locking in the Multithreaded FreeBSD Kernel*, in: Proc. BSDCon, pp. 11–14, (2002).

Parallelizing a Real-Time Steering Simulation for Computer Games with OpenMP

Bjoern Knafla and Claudia Leopold

University of Kassel, Research Group Programming Languages / Methodologies
Wilhelmshoeher Allee 73, 34121 Kassel, Germany
E-mail: {bknafla, leopold}@uni-kassel.de

Future computer games need parallel programming to meet their ever growing hunger for performance. We report on our experiences in parallelizing the game-like C++ application OpenSteerDemo with OpenMP. To enable deterministic data-parallel processing of real-time agent steering behaviour, we had to change the high-level design, and refactor interfaces for explicit shared resource access. Our experience is summarized in a set of guidelines to help parallelizing legacy game code.

1 Introduction

Computer games have typically been written as sequential programs. To exploit multi-core game consoles and PCs, games need to be parallelized.

While parallelization has been actively researched and deployed for scientific applications for decades, computer games are a new area for this programming paradigm. Instead of writing parallel code from scratch, many game developers will lever their existing sequential code infrastructure by refactoring and rewriting the parts that profit most from parallelization.

We chose *OpenSteerDemo*[1] and the *OpenMP*[2] programming system to gain first insights into the parallelization of legacy game-like applications, with focus on artificial intelligence. OpenSteerDemo is a testbed for the C++ open-source library *OpenSteer* written by Reynolds[1]. It simulates and graphically displays the steering of autonomous computer-controlled characters, called agents, in real-time. Simple local environment-aware movement of individual agents emerges to a complex group behaviour.

Internally, OpenSteerDemo runs a main loop as it is typical for games. Each iteration simulates one time step, which cycles through the agents, and thereby simulates the movement and updates the state of each agent (position, velocity, direction), as well as graphically displays the agents. Agent positions are stored in a spatial data structure that supports queries for a given agent's neighbors (which influence the steering of the agent). Thus, in the sequential version, an agent sees the updated states of the agents processed before it, but the old states of the agents processed after it.

Our OpenMP-parallelized version of OpenSteerDemo treats agents as independent entities and processes them in a data-parallel fashion. Each cycle through the agents (update stage) is divided into a simulation sub-stage, followed by a modification sub-stage. During the simulation sub-stage, agents only read their local environment information and store their individual state change wish (called steering vector), independent of each other. In the modification sub-stage, all agents independently write their state based on the steering vector.

This high-level design, with its clear separation of read- and write-accesses to shared resources, results in a deterministic simulation with no need for locks. Details of the high-level design and further techniques for parallelization are outlined below. Measured speedups are 1.92 for two OpenMP threads, and 3.54 for four threads, for the update stage.

The paper starts with an introduction of OpenSteer and OpenSteerDemo, in Sect. 2. Then, Sect. 3 outlines problems with parallelization, and describes how we refactored and parallelized the code. The lessons we learnt are summarized in a set of guidelines in Sect. 4. Next, Sect. 5 contains time measurements, while Sect. 6 overviews related work. Section 7 finishes with conclusions.

2 OpenSteer and OpenSteerDemo

OpenSteer is an open-source C++ library that implements artificial intelligence (AI) steering behaviours for computer-controlled characters (agents) in games. Agents in OpenSteerDemo react to their surrounding only by means of these steering behaviours. Each steering behaviour, such as "walk along a given path" or "avoid an obstacle", corresponds to a steering vector that is computed from the current agent's state (position, orientation, velocity) and its local environment (nearby obstacles and neighboring agents, etc.). A steering vector describes an agent's "wish" to change its state through movement to react to the current situation. Simple behaviours may be combined into complex ones, by combining the corresponding vectors. Moreover, the movement of individual agents results in complex group behaviour.

OpenSteerDemo is a sandbox for rapid prototyping of combined steering behaviours. It runs the typical main loop of real-time games[3]. In each iteration, it passes through several stages, among them the update stage and the graphics stage:

update stage In this stage, each agent calls a method that computes and combines the steering vectors. Then, the agent writes its state and updates its entry in the spatial data structure used for fast neighbour search. Thus, when an agent is updated, it sees the new states of the agents processed before it, and the old states of the agents processed after it.

graphics stage After all agents have been updated, the graphics stage uses *OpenGL* to render the new world state (which comprises the states of all agents) into a picture called *frame*. The number of frames that can be drawn per second is called *frame rate*.

Typically, a game would pass the world state produced by the update stage into a separate physics stage to detect and resolve collisions and to establish a physically correct game world state. This is not done in OpenSteerDemo and hence not treated in this paper.

3 Refactoring and Parallelization

Parallelizing OpenSteerDemo required to refactor it. This section starts with a review of our first parallelization attempt without refactoring, and identifies problems in the code that made this approach fail. The rest of the section describes the new approach. Subsection 3.2 contains some general remarks. Then, subsections 3.3 and 3.4 are devoted to specific techniques. One of them, the separation of the update stage into a simulation and a modification sub-stage, is described in detail. Other techniques are only briefly sketched.

3.1 Problems for Parallelization

Our first approach to parallelize OpenSteerDemo was to incrementally insert OpenMP directives into the sequential code. Unfortunately, the program became messy this way, making reliability and correctness questionable. Moreover, a lot of critical sections were needed, which effectively serialized the program. We searched for reasons why the sequential code was hard to parallelize, and identified the following:

- Excessive use of global variables, deep inheritance hierarchies and strongly interdependent classes make the control and data flow hard to follow.
- As already explained, there are data dependencies between the updates of different agents within an iteration. Concurrent processing of the agents leads to race conditions.
- For some steering behaviours, a random number generator is called. The order of calls is deterministic in the sequential program, but may vary for different executions of the parallel program. Therefore, the same seed does not guarantee identical results, which complicates typical requirements of games, such as recording a simulation run for replay or the synchronization of game sessions over a network.
- OpenGL is not thread-safe. Therefore, concurrent OpenGL calls from different threads may lead to undefined behaviour.

3.2 Overall Approach

We based refactoring and parallelization of the program on the following considerations:

1. The steering vectors of different agents are computed independently. This naturally gives rise to data parallelism, which is the major source of parallelism in the program.
2. The order of simulating agents must not influence the outcome, otherwise race conditions would occur in a parallel environment. Data dependencies between agents were eliminated by modifying the program so that the whole update stage refers to the old world state, i.e., the state at the beginning of the time step. While this modification changed the outcome as compared to the sequential version, all agents are treated the same now, unrelated to their order of simulation.
3. For higher speedups, synchronization between agents should be reduced to a minimum.
4. For every frame, all agents need to be updated, otherwise agent movement looks choppy. Thus, all parallel processing must be finished by the end of the main loop iteration. Because of thread-safety problems with OpenGL, we restricted parallelization to the update stage, decoupled the update and graphics stage, and left the graphics stage for future work.
5. For simplicity, we do not use a spatial hash for the spatial data structure (as the sequential version does), but just a spatial hash interface around a C++ STL vector. Thus, query and access operations require time $O(n^2)$. We expect the parallel version of OpenSteerDemo to have further performance potential, if a more efficient, real spatial data structure is used.

3.3 Separation of Simulation and Update

We divided the update stage into a simulation sub-stage and a modification sub-stage. The simulation sub-stage reads in data and computes the combined steering vectors for all agents, whereas the modification sub-stage accomplishes the actual updates. Thus, there is no need for critical sections; a barrier after the simulation sub-stage and one after the modification sub-stage are sufficient. This modification was not as easy as it may appear, because accesses to global variables are difficult to recognize in the legacy code, especially where pointers are involved. To make shared resource usage explicit, we eliminated all global variables and now pass data into C++ functions and methods as arguments. C++'s *const* keyword helps in enforcing that data can only be read, and that methods can not modify the state of their object. The update stage uses a parallel region, inside which:

1. A parallel *for*-loop runs through all agents and computes their steering vectors (simulation sub-stage). Steering vectors are stored in a C++ *STL vector*, where each element corresponds to one agent and is only accessed by this agent's thread. We use the *dynamic* scheduling parameter of OpenMP because the computational expense of different agents may vary depending on their local environment, for example when they have to react to nearby obstacles.

2. Another parallel *for*-loop accomplishes the updates (modification sub-stage). Each agent changes its own state only, and therefore no race conditions may occur. *Static* scheduling is sufficient here.

3.4 Other Refactoring Techniques

Refactoring of OpenSteerDemo involved further techniques, which for brevity are only sketched here:

render feeder All agents get a reference to a so-called render feeder. The render feeder defers calls to the graphics subsystem, by first collecting graphics primitives to be rendered in a *thread storage* (see below), and later feeding them to the graphics renderer in a purely sequential stage. Within the update stage, threads can only access the graphics subsystem via the render feeder. Thus, the developer of steering behaviours is shielded from unintentional indirect non-thread-safe OpenGL calls.

thread storage Each of our OpenMP threads owns a data slot within a thread storage object. We use it to store graphics primitives that are created during the simulation and modification sub-stages. When accessing the storage, the thread is identified by its thread number, and only gets access to its associated slot. Therefore, no race conditions can occur. In sequential regions, all contained slots can be read. As an aside, this feature is difficult to use with nested parallelism, because OpenMP enumerates threads at different levels with the same numbers. Direct OpenMP support for truly unique thread numbers would be desirable.

random number generators Every agent uses its own random number generator, to guarantee deterministic behaviour. All functions and methods that need random numbers have been changed to get a reference to a generator as a parameter, instead of invoking a random number function with non-deterministic outcome somewhere in the code.

4 Lessons Learned

Based on our experiences with OpenSteerDemo, we developed the following guidelines, which we expect to be useful for parallelizing legacy game code in general:

- Refactor code for simplicity. Parallelization is error-prone and hard[4]. The more complex the code (e.g. deep inheritance hierarchies with interwoven method calls), the harder it is to find and prevent race conditions and non-thread-safe function calls.
- Keep the parallel structure simple, especially in the first version. If complex code is incrementally cluttered with OpenMP pragmas, it becomes hard to understand, and race conditions may easily slip in.
- Identify shared resources and make access to them (read, write, read/write) explicit in function and method interfaces. Use the `const` keyword of C++ where appropriate, to clearly indicate whether a function may give rise to race conditions.
- Change the high-level design (architecture) of the code to reduce the need for low-level coordination and synchronization.

Additionally, common and valuable software development knowledge suggests to:

- Profile first. Identify the section(s) of code that benefit most from optimization.
- Use tools that help analyzing, debugging, and profiling parallel programs. See reference[5] for more information.
- Measure performance to see if the parallel code scales. Deploy data parallelism where applicable, because compared to task parallelism it typically has higher scalability potential[9].

5 Performance

Performance was measured on a dual-processor dual-core 2 GHz AMD Opteron machine with 2 GB RAM and two Nvidia 7800 GTX graphics cards in SLI mode. The machine is running Linux.

Figure 1 depicts the average frame rates in frames per second (fps) for a simulation of pedestrians on a pathway, and Fig. 2 for a flock of birds. On the x-axis, different versions of OpenSteerDemo are arranged:

- the original version using a spatial hash
- the original version using a simple vector instead
- the parallel version with OpenMP support disabled in the compiler
- the parallel version run with 1, 2, and 4 threads (on 1, 2, and 4 processors, respectively).

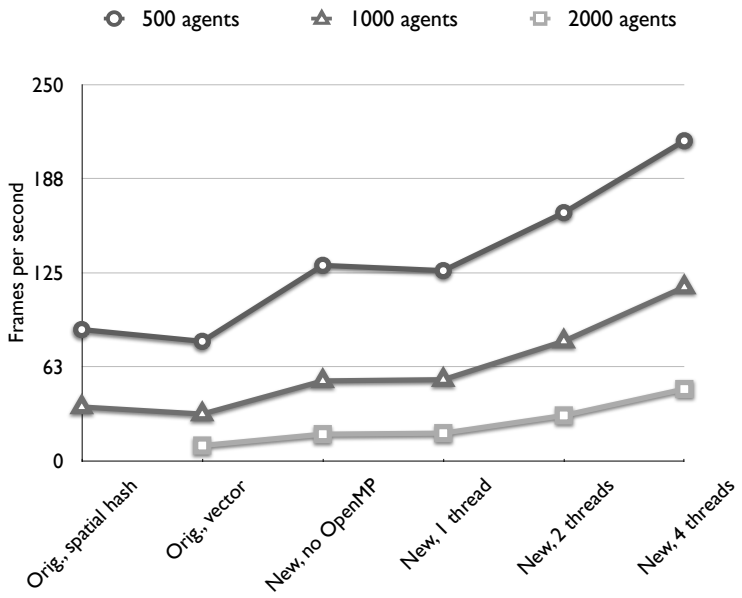


Figure 1. Simulation of pedestrians

The y-axis represents the frame rate. Curves marked with different symbols correspond to distinct numbers of simulated agents (see legend).

The figure refers to measurements for a whole iteration of the main loop, i.e., including both the parallel update stage and other, non-parallel stages such as graphics. For the pedestrian example, one can compute a speedup of about 2.65 for four threads (based on running time, not frame rate as in the figure). Not shown in the figure, we also measured the speedup of the update stage in separation, which is about 3.48 for four threads. For the birds example, the speedups are 2.84 for the whole iteration, and 3.54 for the update stage, respectively. All numbers have been computed taking the OpenMP-disabled version as the sequential base.

It can be observed that the speedup increases with the number of simulated agents. This may be partly due to a lower synchronization overhead per agent on the two OpenMP barriers, and chiefly to a higher computation-to-communication ratio in the case of many agents. More agents lead to a higher agent density since in our experiments they are located in a fixed-sized area or volume. Consequently, an agent needs to inspect more neighbors to evaluate its new position, and thus carries out more arithmetic operations that profit from parallelization.

Agents of the pedestrian example try to stay on a pathway, therefore crowding along the path, while the bird agents spread over the whole simulation world. Most pedestrian agents are surrounded by many other agents that they need to inspect to determine the ones influencing them. As the analysis of neighbors dominates the running time, the spatial hash does not show any advantages over the simple vector. In contrast, the birds spread freely in their simulation world, and therefore the spatial hash quickly excludes many agents from

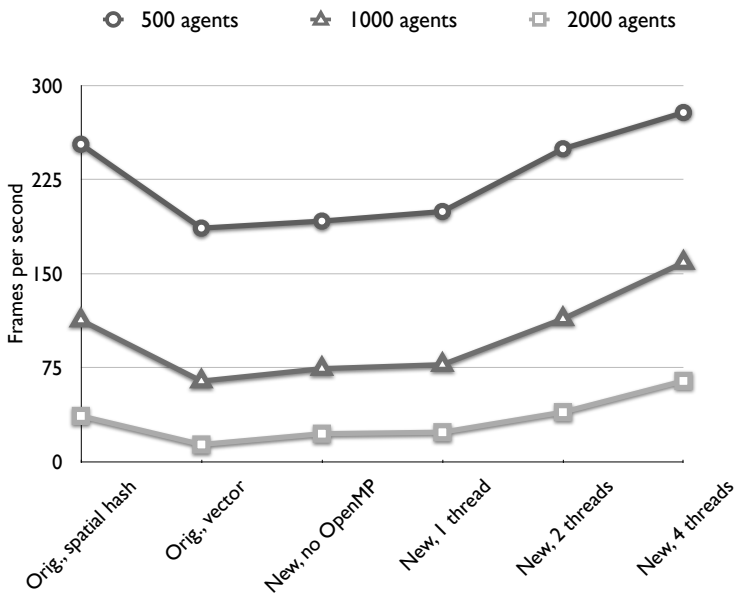


Figure 2. Simulation of a flock of birds

further inspection. With increasing bird density this effect decreases.

6 Related Work

Closest to our work is the *PSCrowd* system of Reynolds[6], which simulates crowds or flocks of agents on the Playstation 3. That platform is based on the *Cell* processor, which contains multiple heterogeneous cores and is programmed with specific libraries and tools. Quinn[7] simulates the behaviour of pedestrians in evacuation scenarios using *MPI*.

Unlike these works, we target homogeneous shared-memory computers, and deploy OpenMP. OpenMP is easier to use than other parallel programming systems, and may thus be a more frequent choice for the parallelization of legacy code.

As another difference, both Reynolds and Quinn subdivide space and map partitions (or more precisely the simulation of the agents inside these partitions) onto processors or processor cores. Quinn's system may experience load imbalances; PSCrowd balances load by using fine-grained partitions. In our approach, agents are mapped to processors with the help of OpenMP's static and dynamic scheduling parameters instead of using a spatial partitioning scheme. Therefore no explicit load balancing is needed.

Valve, an entertainment software and technology company, presented a parallelization approach for games in a press event in 2006[8]. Their approach does not use OpenMP, but is based on low-level synchronization primitives among threads, such as spin-locks and non-locking queues. Other work on parallelizing games has been presented at the last two years' Game Developers Conferences[10].

7 Conclusions

Parallelizing OpenSteerDemo required a lot of code refactorization. Among that was a high-level design change to split the update stage into a simulation sub-stage that reads data, and a modification sub-stage that writes them. Moreover, we changed interfaces by e.g. providing references to a render feeder and a random number generator. With these changes, we derived a simple data-parallel program with no need for critical sections. We are optimistic to gain even better performance with improved spatial data structures, and by introducing parallelism into parts of the graphics subsystem in the future.

An outcome of our work is the observation that it is worthwhile to first refactor complicated code before starting with parallelization. This and other observations have been formulated as a set of guidelines that may be useful for parallelizing game legacy code in general.

References

1. C. W. Reynolds, *OpenSteer Website*, http://opensteer.sourceforge.net, (2004)
2. OpenMP Architecture Review Board, *OpenMP Application Program Interface, Version 2.5 May 2005*, http://www.openmp.org/drupal/mp-documents/spec25.pdf (2005)
3. D. Sánchez-Crespo, *Core Techniques and Algorithms in Game Programming*, New Riders, (2003).
4. E. A. Lee, *The Problem with Threads*, Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, (2006). The published version of this paper is in IEEE Computer, **39**, 33–42, (2006).
5. M. Süß, C. Leopold, *Common Mistakes in OpenMP and How To Avoid Them*, in: Proc. International Workshop on OpenMP - IWOMP'06, (2006).
6. C. W. Reynolds, *Big Fast Crowds on PS3*, in: Proc. 2006 ACM SIGGRAPH symposium on Videogames, (2006).
7. M. J. Quinn, R. A. Metoyer and K. Hunter-Zaworski, *Parallel Implementation of the Social Forces Model*, in: Pedestrian and Evacuation Dynamics 2003, E. R. Galea, (ed.), (2003).
8. J. Reimer, *Valve goes multicore*, http://arstechnica.com/articles/paedia/cpu/valve-multicore.ars (2006).
9. M. Voss, *De-mystify Scalable Parallelism with Intel Threading Building Block's Generic Parallel Algorithms*, http://www.devx.com/cplus/Article/32935 (2006).
10. Game Developers Conference, http://www.gdconf.com/ (2007).

A Framework for Performance-Aware Composition of Explicitly Parallel Components

Christoph W. Kessler[1] **and Welf Löwe**[2]

[1] PELAB, IDA, Linköpings universitet, S-58183 Linköping, Sweden
E-mail: chrke@ida.liu.se

[2] MSI, Växjö universitet, Växjö, Sweden
E-mail: welf.lowe@msi.vxu.se

We describe the principles of a novel framework for performance-aware composition of explicitly parallel software components with implementation variants. Automatic composition results in a table-driven implementation that, for each parallel call of a performance-aware component, looks up the expected best implementation variant, processor allocation and schedule given the current problem and processor group sizes. The dispatch tables are computed off-line at component deployment time by interleaved dynamic programming algorithm from time-prediction metacode provided by the component supplier.

1 Introduction

The multicore revolution in microprocessor architecture has just begun. Programmers will soon be faced with hundreds of hardware threads on a single-processor chip. Exploiting them efficiently is the only way to keep up with the performance potential that still follows the exponential development predicted by Moore's Law. This makes the introduction of parallel computing inevitable even in mainstream computing. Automatic parallelization is often not applicable due to the lack of static information or not efficient due to the principle overhead. Hence, explicit parallel programming is likely to become a dominating programming paradigm for the foreseeable future. This constitutes a challenge for programmers: in addition to the ever growing complexity of software, they now ought to master parallelism in an efficient and effective way, which creates a whole new problem dimension.

Current component technology fits well the domain of sequential and concurrent object-oriented programming. However, existing component models and composition techniques are poorly suited for the performance-aware composition of components for massively parallel computing. Classical component systems allow composition of functionality beyond language and platform boundaries but disregard the performance aspect. The HPC domain (and, as argued, very soon even mainstream computing) requires composition systems with explicitly parallel components. In particular, a performance-aware component model and composition technique are necessary to build complex yet efficient and scalable software for highly parallel target platforms.

In this paper, we propose a new approach for performance-aware composition of explicitly parallel components. We require that all parallel and sequential components subject to performance-aware composition adhere to a specific *performance interface* recognized by a special *performance-aware composition tool*. Performance-aware composition requires the component provider to supply metacode for each performance-aware method f. We focus on terminating methods f; the metacode is used to predict the execution time

of f. The metacode includes a `float`-valued function `time_`f depending on the number p of processors available and some selected parameters of f; `time_`f approximates the expected runtime of f used for local scheduling purposes. It is supplied by the component provider instead of computed by static analysis of f. In practice, it may interpolate entries in a precomputed table or use a closed form function but rather not simulated execution.

Functional and performance signatures are exposed by the component provider. Different substitutable component variants implementing the same functionality f inherit from the same interface. In general, different variants have different `time_`f functions. Dynamic composition chooses, for each call, the variant expected to be the fastest .

Parallel implementations may additionally exploit nested parallelism, marked explicitly by a parallel composition operator, putting together independent subtasks that could be executed in parallel or serially. Different schedules and processor allocations are precomputed, depending on static information like the characteristics of the hardware platform and the variants of subtasks available. They are then selected dynamically for different runtime configurations of problem sizes and processor group sizes.

2 Example: Parallel and Sequential Sorting Components

In the following example, we use pseudocode with object-oriented syntax and a shared-memory programming model similar to Fork[4] where statements and calls are executed in SPMD style, i.e., by a group of processors synchronized at certain program points. Note that our framework is by no means restricted to SPMD or object-oriented paradigms. In particular, it should work equally well with modular languages that provide the interface concept. Moreover, it would work equally well with a message passing programming model such as MPI, as we exemplified for parallel composition in earlier work[3].

We consider *sorting* as an example of functionality supported by several parallel and/or sequential components. We picked sorting since alternative sequential and parallel sorting algorithms such as quicksort, mergesort, or bitonic sort are well-known. Moreover, there are variants highly depending on the actual input (quicksort) and others that are less input-independent (mergesort). Hence, sorting comprises properties of a broad range of potential applications. All component variants conform to the same interface:

```
/*@performance_aware*/ interface Sort {
  /*@performance_aware*/ void sort( float *arr, int n ); }
```

The `performance_aware` qualifier marks the interface and its method `sort` for the composition tool. It expects a performance-aware implementation variant of `sort` and a time function `time_sort` with a subset of the parameters of `sort` in all implementation variants, e.g., a parallel quicksort:
The operator `compose_parallel` marks independent calls to the performance-aware method `sort`; these may be executed in parallel or serially, in any order. The composition tool will replace this construct with a dynamic dispatch selecting the approximated optimum schedule (serialization of calls, parallel execution by splitting the current group of processors into subgroups, or a combination of thereof) and implementation variant.

Within a `time_sort` function, the component designer specifies a closed formula, a table lookup, or a recurrence equation for the expected runtime of this variant. This meta-

```
/*@performance_aware*/ component ParQS variantOf Sort {
  float find_pivot( float *arr, int n ) { ... }
  int partition( float *arr, int n, float pivot) {...}

  /*@performance_aware*/ void sort( float *arr, int n ) {
      if (n==1) return;
      float pivot = find_pivot( arr, n );
      int n1 = partition( arr, n, pivot );
      /*@compose_parallel*/
          /*@1*/  sort( arr, n1 );
          /*@2*/  sort( arr+n1, n-n1 );
      /*@end_compose_parallel*/
  }

  /*@time_metadata*/    // parsed and used by composition tool
    // aux. for time_sort, found empirically at deployment time
    const float T_test_1 = ...       // time for recursion end
    const float T_find_pivot = ...   // time for find_pivot
    const float[][] T_partition =... // table of partition times
    ...                              // benchmark metacode omitted
    float time_sort( int p, int n ) {
      if (n == 1) return T_test_1;
      float acctime=0.0;
      for (int n1=1; n1<n; n1++)
          acctime += TIMEpar( sort@1, n1, sort@2, n-n1, p );
      float exptime = acctime/(float)(n-1);
      return T_test_1 + T_find_pivot + T_partition[n][p] + exptime;
    }
  /*@end_time_metadata */
}
```

code is used to generate the dynamic dispatch tables. The operator `TIMEpar` computes the time for a parallel composition referring to independent calls, e.g., `sort@1` and `sort@2`, and the number `p` of processors available. It is the approximation of the makespan of the schedule found in optimization, cf. Section 3.

Below the `sort` and `time_sort` functions of a performance-aware sequential sorting component `SeqQS`, which simply wraps a (non-performance-aware) sequential quicksort library routine. For brevity, we omit the code of a parallel merge sort variant `ParMS`, which is also used in our example implementation, cf. Section 4.

```
  /*@performance_aware*/ void sort( float *arr, int n ) {
     seq qsort( arr, n );  // done on 1 processor only
  }
  /*@time_metadata */    // used by composition tool
    const float[] T_qsort = ... // Table of seq. sorting times
    float time_sort( int P, int n ) { return T_qsort[n]; }
  /*@end_time_metadata */
```

3 Performance-Aware Composition

The optimization problem for composition is to determine (i) for each call to a performance-aware function, the (expected) best implementation variant, (ii) for each parallel composition operator in a performance-aware function, the number of processors to spend on the calls of each subtask, and a schedule for these subtasks.

The optimization problem may be reduced to the *independent malleable task scheduling* problem. A malleable task can be executed on $p = 1 \ldots P$ processors. Its execution

time is described by a non-increasing function τ in the number of processors p actually used. A malleable task t_f corresponds to a call to a performance-aware interface function f, resp. its implementation variants. For each malleable task t_f, the τ_f-function is approximated using the `time_`f functions. Then, `compose_parallel` is replaced by selecting a schedule of independent malleable task.

Even without the choice between different variants, this scheduling problem is known to be NP-hard in the general case, but good approximations exist: for instance, k independent malleable tasks can be scheduled in time $O(P \cdot k^2)$ on P processors such that the completion time of the resulting schedule is at most $\sqrt{3}$ times the optimum[6]. Moreover, k *identical* malleable tasks can be scheduled optimally to P processors in time $O(\max(\log(k) \cdot t^{3P}, k \cdot (2t)^P)$ where t is the sequential execution time of a task[2].

Dispatch table generation The composition tool does not directly create the customized code. Instead, it generates (i) a variant dispatch table V_f for each interface function f and (ii) a schedule lookup table S for each parallel composition operator, listing the (expected) best processor allocation and the corresponding schedule. The table lists entries with the best decision for a range of problem sizes (ranging from 1 to some maximum tabulated problem size, suitably compacted) and a number of processors (ranging from 1 to the maximum number of processors available in the machine, suitably compacted). For our example above, $V_{\texttt{sort}}(n, p)$ contains a pointer to the expected best `sort` function for problem size n and processor group size p, – see Fig. 1 (right). $S(n_1, n_2, p)$ for the parallel composition operator in `ParQS` yields a processor allocation (p_1, p_2), where $p_1 + p_2 \leq p$, and a pointer to the expected best schedule variant, here only one of two variants: parallel or serial execution of the two independent calls to `sort`.

The tables V_f and S are computed by an *interleaved dynamic programming algorithm* as follows. Together with V_f and S, we will construct a table $\tau_f(n, p)$ containing the (expected) best execution times for p processors.

For a base problem size, e.g. $n = 1$, we assume the problems to be trivial and the functions not to contain recursive malleable tasks, i.e., no recursive calls to performance-aware functions. Hence, $\tau_f(1, p)$ can be directly retrieved from the corresponding `time`$_f$ functions and $V_f(1, p)$ selected accordingly as the variant with minimum `time`$_f$.

For $p = 1$, parallel composition leads to a sequential schedule, i.e., the entries $S(..., 1)$ point to sequential schedules where each task uses 1 processor. Hence, no performance-aware function contains alternative schedules for $p = 1$ and `TIMEpar` reduces to a simple addition of execution times of the subtasks. Accordingly, the $\tau_f(n, 1)$ and $V_f(n, 1)$ for $n = 1, 2, ...$ can be derived iteratively from the `time`$_f$ functions: $\tau_f(n, 1)$ is set to the minimum `time`$_f$ of all variants of f and $V_f(n, 1)$ is set to the variant with minimum `time`$_f$. Usually, the sequential variants (not containing a parallel composition at all) outperform the serialized parallel variants.

Then, we calculate the remaining table entries stepwise for $p = 2, 3, ...$ For each p, we consider successive $n = 1, 2, ...$ For each such n, we determine $\tau_f(n, p)$, $V_f(n, p)$, and the schedules of sub-problems $S(n_1, n_2, \ldots, n_k, p)$. Hence, for $n > 1, p > 1$, we have already computed $\tau_f(n', p')$, $V_f(n', p')$ and $S(n_1, n_2, \ldots, n_k, p')$ with $n', n_1, n_2, \ldots, n_k < n$ and $p' \leq p$.

First, we calculate the schedules $S(n_1, n_2, \ldots, n_k, p)$ for the `compose_parallel` constructs: Since τ_f is defined for each call contained, we simply apply an approximation to the independent malleable task problem leading to a schedule and processor allocations

$(p_1, p_2, \ldots, p_k)$, where all $p_i \leq p$. In our example, we do not even need to approximate the optimum since there is only the parallel schedule with finitely many parallel allocations $(p_1, p - p_1)$ and the sequential one.

Second, we compute $\tau_f(n,p)$ and $V_f(n,p)$: We replace the `TIMEpar` construct with the makespan of the schedule derived and evaluate the $\texttt{time}_f$ functions for each variant. Again, $\tau_f(n,p)$ is set to the minimum $\texttt{time}_f$ of all variants of f and $V_f(n,p)$ is set to the variant with minimum $\texttt{time}_f$.

Composition Auxiliary performance functions and tables as needed for the $\texttt{time}_f$ functions are determined at component deployment time. Accordingly, a component variant provider needs to provide benchmark metacode executed before optimization.

All performance-aware components for the same interface should be deployed together to get meaningful table entries. Encapsulation is still preserved as third-party component providers need not know about other performance-aware components that co-exist.

After optimization, the composition tool patches each call to f in all component implementations with special dispatch code that looks up the variant to call by inspecting the V_f table at runtime, and generates dispatch code at each parallel composition operator looking up its S table at runtime with the current subproblem and group size to adopt the (expected) best schedule.

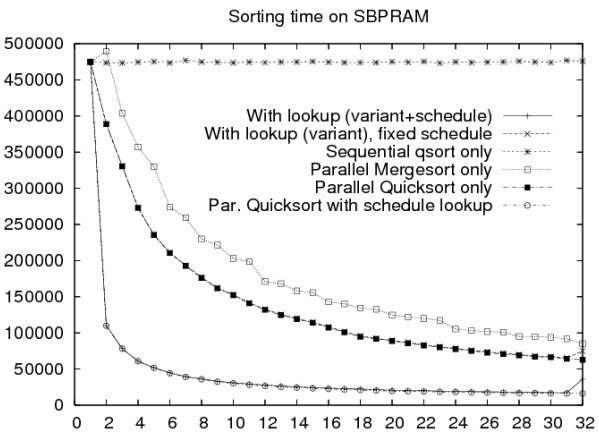
The example code for `ParQS` after composition is sketched below:

```
component ParQS {
  float find_pivot( float *arr, int n ) { ... }
  int partition( float *arr, int n, float pivot) {...}

  extern const int[][] V_sort;      // The V table
  const int[][][][] S_sort = ...    // The S table
  const void (*Sched_sort[2])(float *, int) = { s1_sort, s2_sort };

  void sort( float *arr, int n ) {
      if (n==1) return;
      float pivot = find_pivot( arr, n );
      int n1 = partition( arr, n, pivot );
      //Schedule dispatch - look up function pointer in S and call:
      int p = groupsize();  // number of executing processors
      Sched_sort[ S[n1][n-n1][p][0] ]
         ( arr, n1, n-n1, S[n1][n-n1][p][1], S[n1][n-n1][p][2] );
  }
  //serialized schedule:
  void s1_sort( float *arr, int n1, int n2, int p1, int p2 ) {
      V_sort[n1][p1] ( arr, n1, p1 );
      V_sort[n-n1][p2] ( arr+n1, n2, p2 );
  }
  //parallel schedule:
  void s2_sort( float *arr, int n1, int n2, int p1, int p2 ) {
      split_group(p1,p2) { V_sort[n1][p1](arr, n1); }
                         { V_sort[n2][p2](arr+n1, n2);}
  }
}
```

The table entry `S_sort[n1][n2][p][0]` contains the precomputed schedule variant, the entry `S_sort[n1][n2][p][i]` the processor allocation for the `i`-th call. `groupsize`() returns the number of processors in the executing group. Any other call to `sort(a,n)` would be patched to `V_sort[n][groupsize()](a,n)`. Technically, the composition tool could be based on COMPOST[1] or a similar tool for static meta-




```
Table V[N][P]:
      P=1 2 3 4 ...                                 ...
 N= 1:   3 3 3 3  3 3 3 3  3 3 3 3  3 3 3 3
 N= 2:   3 3 3 3  3 3 3 3  3 3 3 3  3 3 3 3
 N= 3:   3 3 3 3  3 3 3 3  3 3 3 3  3 3 3 3
 N= 4:   3 3 3 3  3 3 3 3  3 3 3 3  3 3 3 3
 N= 5:   3 3 3 3  1 1 1 1  1 1 1 1  1 1 1 1
 N= 6:   3 3 3 3  3 1 1 1  1 1 1 1  1 1 1 1
 N= 7:   3 3 3 3  3 3 3 3  3 3 3 3  3 3 3 3
 N= 8:   3 3 3 2  2 2 2 2  2 2 2 2  2 2 2 2
 N= 9:   3 3 3 3  3 3 3 3  1 1 1 1  1 1 1 1
 N=10:   3 3 3 3  2 2 2 2  2 2 2 2  2 2 2 2
 N=11:   3 3 1 1  1 1 1 1  2 2 1 1  1 1 1 1
 N=12:   3 3 1 1  2 2 2 2  2 2 2 2  2 2 2 2
 N=13:   3 1 1 1  2 2 2 2  2 2 2 2  2 2 2 2
 N=14:   3 3 1 2  2 2 2 2  2 2 2 2  2 2 2 2
 N=15:   3 1 1 1  2 2 2 2  2 2 2 2  2 2 2 2
 N=16:   3 3 1 2  2 2 2 2  2 2 2 2  2 2 2 2
 N=17:   3 1 1 1  1 1 1 1  1 2 2 2  1 1 1 1
 N=18:   3 1 1 1  2 2 2 1  2 2 2 2  2 2 2 2
 N=19:   3 1 1 1  1 2 1 1  2 2 2 2  2 2 2 2
 N=20:   3 1 1 1  1 1 1 1  1 2 2 2  2 2 2 2
 N=21:   3 1 1 1  1 1 1 1  1 1 1 2  2 1 1 1
 N=22:   3 1 1 1  1 1 1 1  1 1 1 2  2 2 2 2
 N=23:   3 1 1 1  1 1 1 1  1 1 1 2  2 2 2 2
 N=24:   3 1 1 1  1 1 1 1  1 1 1 2  2 2 2 2
 N=25:   3 1 1 1  1 1 1 1  1 1 1 1  2 2 2 2
 N=26:   ...
```

Figure 1. Execution times (in SBPRAM clock cycles) of the composed sorting program, applied to $N = 500$ numbers on up to $P = 32$ SBPRAM processors, compared to the times needed when using each component exclusively. The dispatch tables were precomputed by dynamic programming for $N = 1...64$ and $P = 1...32$. — Right-hand side: The top-left-most entries ($1..25 \times 1..16$) in the variant dispatch table V indicating the expected best variant, where 1 denotes a call to ParQS, 2 to ParMS, and 3 to SeqQS.

programming that enables fully programmable program restructuring transformations.

4 Implementation and First Results

A proof-of-concept implementation uses the C-based parallel programming language Fork[4] for three sorting components, parallel recursive quicksort, parallel recursive mergesort, and sequential quicksort. The time parameters for the functions `find_pivot`, `partition`, `qsort` etc. used in `time_sort` were obtained by measuring times for example instances. Times depending on input data were averaged over several instances. To record the best schedules for parallel composition in the schedule table, a brute-force enumeration approach was chosen, as the best constellation of subgroup sizes can be determined in linear time for a parallel divide degree of 2, as in `ParQS` and `ParMS`.

As a fully automatic composition tool interpreting metacode syntax is not yet available, we simulated the effect of composition by injecting the the dispatch tables and lookups by hand into the appropriate places in the source code. For evaluation purposes, the resulting Fork source code can be configured to either use the schedule and variants as given in the computed dispatch tables or the original component implementations without modification. The code is compiled to the SBPRAM[4] and executed on its cycle-accurate simulator, run on a SUN Solaris server.

Figure 1 shows average times for sorting 500 numbers on up to 32 PRAM processors (left) and a section of the variant dispatch table V (right). We can observe that all parallel variants outperform sequential quicksort. Among the former, adaptive parallel quicksort and our performance-aware composition perform best. That these two variants perform almost equivalent comes at no surprise when looking the V table entries: except for small problem sizes, quicksort is selected.

The performance improvements of the composed function (up to a factor of 10 compared to sequential sorting and up to a factor of 4 and 5 compared to the parallel quicksort and mergesort only, resp.) are due to schedule lookup. The gain increases with N because the general case of two recursive subtasks gets more common.

5 Related Work

Various static scheduling frameworks for malleable parallel tasks and task graphs of modular SPMD computations with parallel composition have been considered in the literature[7,8,9]. Most of them require a formal, machine-independent specification of the algorithm that allows prediction of execution time by abstract interpretation. In our work, we separated the actual implementation from the model of its execution time. Scheduling methods for distributed memory systems also need to optimize communication for data redistribution at module boundaries. This can be added to our framework as well. To the best of our knowledge, none of them considers the automatic composition of different algorithm variants at deployment time and their automatic selection at runtime.

As we do not consider heterogeneous or distributed systems here, additional interoperability support by parallel CORBA-like systems is not required. However, such an extension would be orthogonal to our approach.

Our earlier work explores the parallel composition operator for dynamic local load balancing in irregular parallel divide-and-conquer SPMD computations[3]. We balanced the trade-off between group splitting for parallel execution of subtasks and serialization, computing—off-line by dynamic programming—tables of the expected values of task size ratios, indexed by n and p, where scheduling should switch between group splitting and serialization. Our schedule lookup table above can be seen as a generalization of this.

6 Conclusions and Future Work

The paper proposes a composition framework for SPMD parallel components. Components are specified independently of the specific runtime environment. They are equipped with metacode allowing to derive their performance in a particular runtime (hardware) environment at deployment time. Based on this information, a composition tool automatically approximates optimal partial schedules for the different component variants and processor and problem sizes and injects dynamic composition code. Whenever the component is called at runtime, the implementation variant actually executed is selected dynamically, based on the actual problem size and the number of processors available for this component. Experiments with two parallel and one sequential sorting component prototypically demonstrate the speed-up compared to statically composed parallel solutions.

Static agglomeration of dynamic composition units is an optimization of our approach. We could consider the trade-off between the overhead of dynamic composition vs. the (expected) performance improvement due to choosing the (expected) fastest variant and schedule. We could consider units for dynamic composition that have a larger granularity than individual performance-aware function calls. A possible approach could be to virtually in-line composition operator “expressions” (which may span across function calls, i.e., define contiguous subtrees of the call graph) that will be treated as atomic units for dynamic composition. The composition tool would compose these units statically including

a static composition of the `time` functions. This will usually somewhat decrease accuracy of predictions and miss some better choices of variants within these units but also saves some dynamic composition overhead.

Table compression techniques need to be investigated. For instance, regions in the V or S tables with equal behaviour could be approximated by polyhedra bounded by linear inequalities that could result in branching code instead of the table entry interpolations for dynamic dispatch and scheduling. Compression techniques for dispatch tables of object-oriented polymorphic calls could be investigated as well.

Adaptation of time data parameters In the sorting example, we used randomly generated problem instances to compute parameter tables with average execution times for `qsort`, `partition` etc. used in optimization. In certain application domains or deployment environments, other distributions of input data could be known and exploited. Moreover, expected execution times could be adjusted dynamically with new runtime data as components are executed, and in certain time intervals, a re-optimization may take place such that the dispatch tables adapt to typical workloads automatically.

Domains of application In scientific computing as well as non-numerical applications, there are many possible application scenarios for our framework. For instance, there is a great variation in parallel implementations of solvers for ODE systems that have equal numerical properties but different time behaviour[5].

Acknowledgements Research funded by Ceniit 01.06 at Linköpings universitet, Vetenskapsrådet, SSF RISE, Vinnova SafeModSim and AdaptiveGRID, and the CUGS graduate school.

References

1. U. Aßmann, *Invasive Software Composition*, (Springer, 2003).
2. T. Decker, T. Lücking and B. Monien, *A 5/4-approximation algorithm for scheduling identical malleable tasks*, Theoretical Computer Science, **361**, 226–240, (2006).
3. M. Eriksson, Ch. Kessler and M. Chalabine, *Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments*, in: Proc. 8th Workshop on Parallel Systems and Algorithms (PASA'06), GI Lecture Notes in Informatics (LNI), vol. **P-81**, pp. 313–322, (2006).
4. J. Keller, Ch. Kessler and J. Träff, *Practical PRAM Programming*, (Wiley Interscience, 2001).
5. M. Korch and Th. Rauber, *Optimizing locality and scalability of embedded Runge-Kutta solvers using block-based pipelining*, J. Par. and Distr. Computing, **66**, 444–468, (2006).
6. G. Mounie, C. Rapine and D. Trystram, *Efficient approximation algorithms for scheduling malleable tasks*, in: Proc. 11th ACM Symposium on Parallel Algorithms and Architectures (SPAA'99), ACM Press, pp. 23–32, (1999).
7. Th. Rauber and G. Rünger, *Compiler Support for Task Scheduling in Hierarchical Execution Models*, J. Systems Architecture, **45**, 483–503, (1998).
8. L. Zhao, S. Jarvis, D. Spooner and G. Nudd, *Predictive Performance Modeling of Parallel Component Composition*, in: Proc. 19th IEEE Int. Parallel and Distr. Processing Symposium (IPDPS-05), (IEEE Press, 2005).
9. W. Zimmermann and W. Löwe, *Foundations for the integration of scheduling techniques into compilers for parallel languages*, Int. J. Comp. Sci. Eng., **1**, 3/4, (2005).

A Framework for Prototyping and Reasoning about Distributed Systems

Marco Aldinucci[1], **Marco Danelutto**[1], **and Peter Kilpatrick**[2]

[1] Dept. Computer Science – University of Pisa – Italy
E-mail: {*aldinuc, marcod*}*@di.unipi.it*

[2] Dept. Computer Science – Queen's University Belfast – United Kingdom
E-mail: p.kilpatrick@qub.ac.uk

A framework supporting fast prototyping as well as tuning of distributed applications is presented. The approach is based on the adoption of a formal model that is used to describe the orchestration of distributed applications. The formal model (Orc by Misra and Cook) can be used to support semi-formal reasoning about the applications at hand. The paper describes how the framework can be used to derive and evaluate alternative orchestrations of a well know parallel/distributed computation pattern; and shows how the same formal model can be used to support generation of prototypes of distributed applications skeletons directly from the application description.

1 Introduction

The programming of large distributed systems, including grids, presents significant new challenges. For example, the "invisible grid" and "service and knowledge utility" (SOKU) concepts advocated in the EU NGG expert group documents[1,2] both identify new challenges to be addressed. In particular, an increasingly substantial programming effort is required to set up appropriate "computing structure" for a distributed application: significant efforts have to be invested in the development of a suitable, manageable and maintainable distributed application skeleton, and, ideally, this skeleton should be validated, at least informally, *before* starting actual coding to avoid spending large amounts of time coding only to discover when running application tuning experiments that one or more of the original skeleton features is inappropriate.

What is needed is a framework that allows the application programmer to develop a specification of a distributed system using a user-friendly formal notation. Existing formal tools such as the π-calculus[3] are usually perceived as being distant from the "reasoning schemas" which are typical of programmers, and, moreover, their usage requires significant formal calculus capability and experience combined with substantial effort. Therefore, an approach is required that can replace full formal reasoning about a system's properties with "lightweight" reasoning combining properties of the notation with the developer's domain expertise and experience. Typically such reasoning will allow focus on the particular case rather than the general and, in this way, significantly reduce the overhead of formal development. In addition, the availability of such a specification will afford the possibility of generating automatically a skeletal implementation that may be used for experimentation prior to full implementation.

This research is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

A *site*, the simplest form of Orc expression, either returns a *single* value or remains silent. Three operators (plus recursion) are provided for the orchestration of site calls:

1. $>$ (sequential composition): $E_1 > x > E_2(x)$ evaluates E_1, receives a result x, calls E_2 with parameter x.

   The abbreviation $E_1 \gg E_2$ is used for $E_1 > x > E_2$ when evaluation of E_2 is independent of x.

2. $|$ (parallel composition): $(E_1 \mid E_2)$ evaluates E_1 and E_2 in parallel. Evaluation of the expression returns the merged output streams of E_1 and E_2.

3. where (asymmetric parallel composition): E_1 where $x :\in E_2$ begins evaluation of both E_1 and $x :\in E_2$ in parallel. Expression E_1 may name x in some of its site calls. Evaluation of E_1 may proceed until a dependency on x is encountered; evaluation is then delayed. The first value delivered by E_2 is returned in x; evaluation of E_1 can proceed and the thread E_2 is halted.

Orc has a number of special sites, including *RTimer(t)*, that always responds after t time units (can be used for time-outs). The notation $(|i : 1 \leq i \leq 3 : w_i)$ is used as an abbreviation for $(w_1|w_2|w_3)$.
In Orc processes may be represented as expressions which, typically, name channels which are shared with other expressions. In Orc a channel is represented by a site[4]. *c.put(m)* adds m to the end of the (FIFO) channel and publishes a signal. If the channel is non-empty *c.get* publishes the value at the head and removes it; otherwise the caller of *c.get* suspends until a value is available.

Figure 1. Minimal Orc *compendium*

Earlier work by the authors identified Orc by Misra and Cook[4,5] as a suitable notation to act as a basis for the framework proposed above. There it was shown how Orc is particularly suitable for addressing *dynamic* properties of distributed systems, although in the current work, for the sake of simplicity, only static examples/properties are considered. Subsequent work demonstrated how an Orc-based framework can be developed to support large distributed application development and tuning[6,8]. Figure 1 outlines the main features of Orc, in particular those related to the Orc specifications used here. In the current work these previous results are extended by two further contributions.

First it is shown that a cost analysis technique can be described and used to evaluate properties of alternative implementations of the same distributed application. Orc models of two (functionally equivalent) implementations of a typical distributed use-case are presented. For each, the number and kind of communications performed is determined and the overall communication cost is estimated using a measure of communication cost. To evaluate the communication patterns of the two implementations metadata denoting *locations* where parallel activities have to be performed is considered. The metadata is formulated in accordance with the principles stated in earlier work presented at the CoreGRID Symposium[6]. While metadata can be used to describe several distinct aspects of a system, here only metadata items of the form $\langle site/expression, location \rangle$ representing the fact that $site/expression$ is run on the resource $location$ are considered. A methodology is introduced to deal with partial user-supplied metadata that allows labelling of all the sites and processes appearing in the Orc specification with appropriate locations. The location of parallel activities, as inferred from the metadata, is then used to cost communications occurring during the application execution.

Second a compiler tool that allows generation of the skeleton of a distributed application directly from the corresponding Orc specification is described. The user can then complete the skeleton code by providing the functional (sequential) code implementing site and process internal logic, while the general orchestration logic, mechanisms, sched-

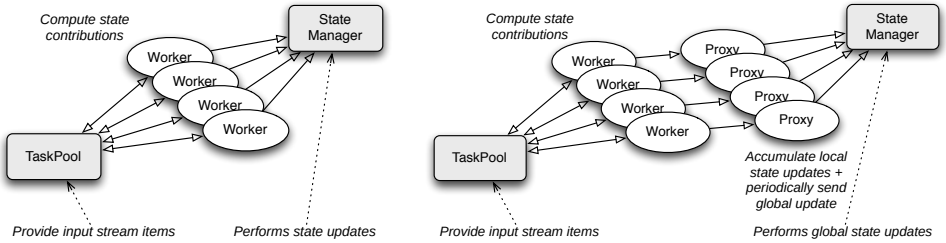


Figure 2. Logical schema corresponding to the two Orc specifications (grey boxes represent sites, white ovals represent processes)

ule, etc. are completely dealt with by the compiler generated code. Finally, results are presented which demonstrate that the code generated from the Orc specifications given here perform as expected when executed on a distributed target architecture.

These two further developments complete a framework supporting design, development and tuning of distributed grid applications.

2 Use Case: Parallel Computation of Global State Updates

A use case is now introduced as a vehicle to demonstrate how a costing mechanism can be associated with Orc, via metadata, in such a way that alternative implementations/orchestrations of the same application can be compared. A common parallel application pattern is used for illustration: data items in an input stream are processed independently, with the results being used to update a shared state (see Fig. 2). For simplicity, only the easy case where state updates are performed through an associative and commutative function $\sigma : State \times Result \rightarrow State$ is considered. This kind of computation is quite common. For example, consider a parallel ray tracer: contributions of the single rays on the scene can be computed independently and their effect "summed" onto the final scene. Adding effects of a ray on the scene can be performed in any order without affecting the final result. Two alternative designs of this particular parallelism exploitation pattern, parametric in the type of actual computation performed, are modelled in Orc.

2.1 Design 1: No Proxy

The first design is based on the classical master/worker implementation where centralized entities provide the items of the input stream and collate the resulting computations in a single state. The *system* comprises a taskpool (modelling the input stream), *TM*, a state manager, *SM* and a set of workers, W_i. The workers repeatedly take tasks from the taskpool, process them and send the results to the state manager. The taskpool and state manager are represented by Orc sites; the workers are represented by processes (expressions). This specification corresponds to the logical schema in Fig. 2 left and can be formulated as follows:

$$system(TP, SM) \triangleq workers(TP, SM)$$

$workers(TP, SM) \triangleq \mid i : 1 \leq i \leq N : W_i(TP, SM)$
$W_i(TP, SM) \triangleq TP.get >tk> compute(tk) >r> SM.update(r) \gg W_i(TP, SM)$

2.2 Design 2: With Proxy

In this design, for each worker, W_i, a proxy, *proxy*$_i$, is interposed between it and the state manager to allow accumulation of partial results before forwarding to the state manager. A proxy executes a *ctrlprocess* in parallel with a *commit* process. The control process receives from its worker, via a channel wc_i, a result and stores it in a local state manager, *LSM*$_i$. Periodically, the *commit* process stores the contents of the *LSM*$_i$ in the global state manager, *SM*. A control thread (and control process) is represented by a process; a local state manager is a site. This corresponds to the schema in Fig. 2 right.

$system(TP, SM, LSM) \triangleq proxies(SM, LSM) \mid workers(TP)$
$proxies(SM, LSM) \triangleq \mid i : 1 \leq i \leq N : proxy_i(SM, LSM_i)$
$proxy_i(SM, LSM_i) \triangleq ctrlprocess_i(LSM_i) \mid commit(LSM_i)$
$ctrlprocess_i(LSM_i) \triangleq wc_i.get >r> LSM_i.update(r) \gg ctrlprocess_i(LSM_i)$
$commit_i(LSM_i) \triangleq Rtimer(t) \gg SM.update(LSM_i) \gg commit_i(LSM_i)$
$workers(TP) \triangleq \mid i : 1 \leq i \leq N : W_i(TP)$
$W_i(TP) \triangleq TP.get >tk> compute(tk) >r> wc_i.put(r) \gg W_i(TP)$

3 Communication Cost Analysis

A procedure is now introduced to determine the cost of an Orc expression evaluation in terms of the communications performed to complete the computation modelled by the expression.

First some basic assumptions concerning communications are made. It is assumed that a site call constitutes 2 communications (one for the call, one for getting the ACK/result): no distinction is made between transfer of "real" data and the ACK. It is also assumed that an interprocess communication constitutes 2 communications. In Orc this is denoted by a *put* and a *get* on a channel. (Note: although, in Orc, this communication is represented by two complementary site (channel) calls, this exchange is not considered to constitute 4 communications.) Two cases for sequential composition with passing of a value may be identified: *site* $> x >$ *site* represents a local transfer of x (cf. a local variable) and so is assumed not to represent a communication, while *site* $> x >$ *process*, *process* $> x >$ *site* and *process* $> x >$ *process* all constitute 2 communications. Finally, care must be taken in treating communications that may overlap in a parallel computation. A simple and effective model is assumed: for Orc parallel commands the communication cost mechanism should take into account the fact that communications happening "internally" to the parallel activities overlap while those involving *shared* external sites (or processes) do not. Thus, when counting the communications occurring within an Orc expression such as:

$W_i(TP) \triangleq TP.get >tk> compute(tk) >r> wc_i.put(r) \gg W_i(TP)$

a distinction is drawn between calls to site *TP* which are calls to external, shared sites, and calls to wc_i which are related to internal sites. When assessing the calls related to the execution of N processes W_i computing M tasks, all the calls to TP are counted while it is assumed that the communications related to different wc_i are overlapped. Therefore, assuming perfect load balancing, the total will include M calls to TP and $\frac{M}{N}$ calls to wc_i.

To evaluate the number and nature of communications involved in the computation of an Orc expression two steps are performed.

First the metadata associated with the Orc specification is determined. It is assumed that the user has provided metadata stating placement of relevant sites/processes as well as strategies to derive placement metadata for the sites/processes not explicitly targeted in the supplied metadata (as discussed in previous work[6]). Thus, for the first design given above, with initial metadata $\mathcal{M} = \{\langle site,\ TP\rangle,\ \langle site,\ SM\rangle,\ \langle system,\ strategy(FullyDistributed)\rangle\}$, the metadata

$$\mathcal{M} = \{\langle site, TP\rangle, \langle site, SM\rangle, \langle TP,\ freshLoc(\mathcal{M})\rangle, \langle SM,\ freshLoc(\mathcal{M})\rangle, \langle workers,\ freshLoc(\mathcal{M})\rangle\ \langle worker_i,\ freshLoc(\mathcal{M})\rangle\}$$

can be derived, where $freshLoc$ returns a resource name not previously used (thus implementing the $strategy(FullyDistributed)$ policy). The strategy *FullyDistributed* indicates that all processes/sites should be placed on unique locations, unless otherwise stated (by explicit placement). This metadata is *ground* as all sites and all non-terminals have associated locations. For details of metadata derivation see the CoreGRID technical report[7].

The second step counts the communications involved in evaluation of the Orc specification, following the assumptions made at the beginning of this Section.

Consider the first design above. In this case, assume N worker processes, computing M tasks, with two sites providing the taskpool and the state manager (*TP* and *SM*). The communications in each of the workers involve shared, non-local sites, and therefore the total number of communications involved is $2M + 2M$, the former term representing the communications with the taskpool and the latter to those with the state manager. All these communications are remote (as the strategy is *FullyDistributed*), involving sending and receiving sites/processes located on different resources, and therefore are costed at r time units. (Had they involved sites/processes on the same processing resources the cost would have been l time units.) Therefore, the overall communication cost of the computation described by the first Orc specification with M input tasks and N workers is $4Mr$. This is independent of N, as expected, as there are only communications related to calls to global, shared sites.

For the version with proxy, there are N W_i, N $ctrlprocess_i$ processes and N LSM_i sites, plus the globally shared TP and SM sites. Assume, from the user supplied metadata, that each $\langle ctrlprocess_i, LSM_i\rangle$ pair is allocated on the same processing element, distinct from the processing element used for $worker_i$, and that all these processing elements are in turn distinct from the two used to host TP and SM. (Again, see the CoreGRID technical report[7] for details of the metadata determination.) The communications involved in the computation of M tasks are $2M$ non-local communications (those taking tasks out of the *TP*), $2\frac{M}{k}$ non-local communications (those sending partial contributions to *SM*, assuming k state updates are accumulated locally before invoking a global update on *SM*) and $2 \times 2 \times \frac{M}{N}$ local communications (those performed by the $ctrlprocess$es to get the result of W computations and perform local LSM updates, assuming an even distribution of tasks to workers). Therefore the cost of the computation described by the second Orc specification with M input tasks and N workers is $2Mr + \frac{2Mr}{k} + \frac{4Ml}{N}$.

Now the two designs can be compared with respect to communications. Simple algebraic manipulation shows that the second will "perform better", that is will lead to a communication profile costing fewer time units, if and only if $k > \frac{rN}{rN-2l}$. That is, if and only if the number of local state updates accumulated at the LSM_is before sending update

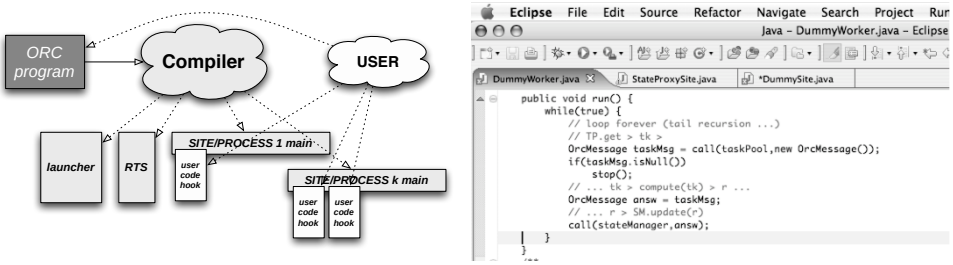


Figure 3. Orc2Java compiler tool (left) and user code implementing a dummy worker process (right)

messages to SM is larger than one (typically $l \ll r$). In Section 4 experimental results that validate this statement are presented.

4 Automatic Distributed Code Framework Generation

To support experimentation with alternative orchestrations derived using the methodology discussed above a tool for the generation of distributed Java code from an Orc specification was developed (Fig. 3 (left)). Being a compiler producing actual Java distributed code, the tool is fundamentally different from the Orc simulator provided by the Orc designers on the Orc web page[9].

In particular, the tool takes as input an Orc program (that is a set of Orc expression definitions plus a target expression to be evaluated) and produces a set of Java code files that completely implement the "parallel structure" of the application modelled by the Orc program. That is, a `main` Java code is produced for each of the parallel/distributed entities in the Orc code, suitable communication code is produced to implement the interactions among Orc parallel/distributed entities, and so on. Orc sites and processes are implemented by distinct JVMs running on either the same or on different machines and communications are implemented using plain TCP/IP sockets. A "script" program is also produced that takes as input a description of the target architecture (names of the nodes available, features of each node, interconnection framework information, and so on) and deploys the appropriate Java code to the target nodes and finally starts the execution of the resulting distributed application.

The Java code produced provides hooks to programmers to insert the needed "functional code" (i.e. the code actually implementing the sites for the computation of the application results). The system can be used to perform fast prototyping of grid applications, and the parallel application skeleton is automatically generated without the need to spend time programming all of the cumbersome details related to distributed application development (process decomposition, mapping and scheduling, communication and synchronization implementation, etc.) that usually takes such a long time. Also, the tool can be used to run and compare several different skeletons, such that users may evaluate "in the field" which is the "best" implementation.

Using the preliminary version of the Java code generator[a], the performances of

[a] The authors are indebted to Antonio Palladino who has been in charge of developing the prototype compiler

the alternative designs of the application described in Section 2 were evaluated. Figure 3 (right) shows the code supplied by the user to implement a dummy worker process. The method `run` of the class is called by the framework when the corresponding process is started. The user has an `OrcMessage call(String sitename, OrcMessage message)` call available to interact with other sites, as well as one-way call mechanisms such as `void send(String sitename, OrcMessage msg)` and `OrcMessage receive()`. If a site is to be implemented, rather than a process, the user must only subclass the `Site` framework class providing an appropriate `OrcMessage react(OrcMessage callMsg)` method.

Fig. 4 shows the results obtained when running several versions of the application, automatically derived from the alternative Orc specifications previously discussed. The version labelled "no proxy" corresponds to the version with a single, centralized state manager site which is called by all the workers, while the version labelled "with proxy" corresponds to the version where workers pass results to proxies for temporary storage, and they, in turn, periodically call the global state manager to commit the local updates. The relative placement of processes and sites is determined by appropriate metadata as explained in previous sections. The code has been run on a network of Pentium based Linux workstations interconnected by a Fast Ethernet network and running Java 1.5.0_06-b05.

As expected, the versions with proxy perform better than the one without, as part of the overhead deriving from state updates is distributed (and therefore parallelized) among the proxies.

This is consistent with what was predicted in Section 3. On the target architecture considered, values of $r = 653\mu secs$ and $l = 35\mu secs$ were obtained and so the proxy implementation is to be preferred when k is larger than 1.12 to 1.02 depending on the number of workers considered (1 to 4, in this case). In this experiment, LSM_i performed an average of 2.5 local updates before calling the SM site to perform a global state update and therefore the constraint above was satisfied.

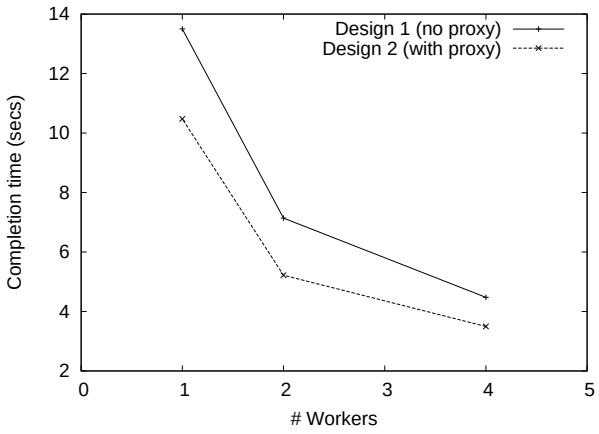


Figure 4. Comparison among alternative versions of the grid application described in Section 3

5 Conclusions

The work described here shows how alternative designs of a distributed application (schema) can be assessed by describing them in a formal notation and associating with

this specification appropriate metadata to characterise the non-functional aspect of interest, in this case, communication cost. Cost estimations were derived that show that the second implementation, the one including the proxy design pattern, should perform better than the first one, in most cases. Then, using the prototype Orc compiler two versions of the distributed application were "fast prototyped" and run on a distributed set of Linux workstations. The times measured confirmed the predictions. Overall the whole procedure demonstrates that 1) under the assumptions made here, one can, to a certain degree, evaluate alternative implementations of the same application using metadata-augmented specifications only, and in particular, without writing a single line of code; and 2) that the Orc to Java compiler can be used to generate rapid prototypes that can be used to evaluate applications directly on the target architecture.

Future work will involve 1) (semi-)automating the derivation of metadata from user-specified input (currently a manual process) and 2) investigating the use of the framework with a wider range of skeletons[10]: experience suggests that skeletons potentially provide a restriction from the general that may prove to be fertile ground for the approach.

References

1. Next Generation GRIDs Expert Group, *NGG2, Requirements and options for European grids research 2005–2010 and beyond*, (2004).
2. Next Generation GRIDs Expert Group, *NGG3, Future for European grids: GRIDs and service oriented knowledge utilities. Vision and research directions 2010 and beyond*, (2006).
3. R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*, (Cambridge University Press, 1999).
4. J. Misra and W. R. Cook, *Computation orchestration: a basis for wide-area computing*, Software and Systems Modeling, DOI 10.1007/s10270-006-0012-1, (2006).
5. D. Kitchin, W. R. Cook and J. Misra, *A language for task orchestration and its semantic properties*, CONCUR, in LNCS **4137**, pp. 477–491. (Springer, 2006).
6. M. Aldinucci, M. Danelutto and P. Kilpatrick, *Adding metadata to Orc to support reasoning about grid programs*, in: *Proc. CoreGRID Symposium 2007*, pp. 205–214, Rennes (F), Springer, 2007. ISBN: 978-0-387-72497-3
7. M. Aldinucci, M. Danelutto and P. Kilpatrick, *Prototyping and reasoning about distributed systems: an Orc based framework*, CoreGRID TR-0102, available at `www.coregrid.net`, (2007).
8. M. Aldinucci, M. Danelutto and P. Kilpatrick, *Management in distributed systems: a semi-formal approach*, in: Proc. Euro-Par 2007 – Parallel Processing 13th Intl. Euro-Par Conference, LNCS **4641**, Rennes (F), Springer, 2007.
9. W. R. Cook and J. Misra, *Orc web page*, (2007).`http://www.cs.utexas.edu/users/wcook/projects/orc/`.
10. M. Cole, *Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming*, Parallel Computing, **30**, 389–406, (2004).

Formal Semantics Applied to the Implementation of a Skeleton-Based Parallel Programming Library

Joel Falcou and Jocelyn Sérot

LASMEA, UMR6602 UBP/CNRS, Campus des Cézeaux, 63177 Aubière, France
E-mail: {joel.falcou, jocelyn.serot}@lasmea.univ-bpclermont.fr

1 Introduction

In a previous paper[1], we described QUAFF, a skeleton-based parallel programming library which main originality is to rely on C++ *template* meta-programming[2,3] techniques to significantly reduce the overhead traditionally associated to object-oriented implementations of such libraries. The basic idea is to use the C++ *template* mechanism so that skeleton-based programs are actually run at compile-time and generate a new C+MPI code to be compiled and executed at run-time. The implementation mechanism supporting this compile-time approach to skeleton-based parallel programming was only sketched mainly because the operational semantics of the skeletons was not stated in a formal way, but "hardwired" in a set of complex meta-programs. As a result, changing this semantics or adding a new skeleton was difficult. In this paper, we give a formal model for the QUAFF skeleton system, describe how this model can efficiently be implemented using C++ meta-programming techniques and show how this helps overcoming the aforementioned difficulties. It relies on three formally defined stages. First, the C++ compiler generates an abstract syntax tree representing the parallel structure of the application, from the high-level C++ skeletal program source. Then, this tree is turned into an *abstract process network* by means of a set of *production rules*; this process network encodes, in a platform-independent way, the communication topology and, for each node, the scheduling of communications and computations. Finally the process network is translated into C+MPI code. By contrast to the previous QUAFF implementation, the process network now plays the role of an *explicit intermediate representation*. Adding a new skeleton now only requires giving the set of production rules for expanding the corresponding tree node into a process sub-network. The paper is organized as follows. Section 2 briefly recalls the main features of the QUAFF programming model. Section 3 presents the formal model we defined to turn a skeleton abstract syntax tree into a process network. Section 4 shows how *template* meta-programming is used to implement this model. We conclude with experimental results for this new implementation (Section 5) and a brief review of related work (Section 6).

2 The QUAFF Library

The programming model of QUAFF is a classical skeleton-based one. Skeletons are defined as follows:

$$\begin{array}{lll} \Sigma & ::= & \mathsf{Seq}\ f \mid \mathsf{Pipe}\ \Sigma_1 \ldots \Sigma_n \mid \mathsf{Farm}\ n\ \Sigma \mid \mathsf{Scm}\ n\ f_s\ \Sigma\ f_m \mid \mathsf{Pardo}\ \Sigma_1\ \ldots \Sigma_n \\ f, f_s, f_m & ::= & \textit{sequential, application-specific user-defined C++ functions} \\ n & ::= & integer \geq 1 \end{array}$$

All user-defined functions take at most one argument and return at most one result. The skeleton set is classical. Intuitively, Seq encapsulates sequential user-defined functions in such a way they can be used as parameters to other skeletons; Pipe and Farm are the usual task-parallel skeletons (with computations performed in stages and under a master/workers scheme respectively); Scm models data-parallel computations: f_s decomposes the input data into a set of (possibly) overlapping data subsets, the inner skeleton processes each subset in parallel and the f_m function merges the sub-results; Pardo models parallel, independent computations, where n distinct tasks are run on n distinct processors. The parallel structure of an application can then be represented by a tree with nodes corresponding to skeletons and leaves to user-defined sequential functions. A distinctive feature of QUAFF – compared to other skeleton-based parallel programming libraries[4,5,6] – is that this structure is completely described by means of type definitions. This, of course, is the key point allowing optimized message-passing code to be produced *at compile-time*, as will be explained in Section 4. Considering a simple application like:

$$\mathcal{A} = \mathsf{Pipe}(\mathsf{Seq}\ f_1,\ \mathsf{Farm}(4,\ \mathsf{Seq}\ w),\ \mathsf{Seq}\ f_2)$$

It's implemented via the the following code using QUAFF:

Listing 28.1. Sample QUAFF application

```
typedef task<f1, void_, int   >        F1;                          1
typedef task<w , int  , double>         W;                          2
typedef task<f2, double, void_>        F2;                          3
                                                                    4
run( pipeline( seq(F1), farm<4>(seq(W)), seq(F2)) );                5
```

Lines 1–3 register user-defined C functions as *tasks* used into skeleton definitions. A QUAFF *task* is defined by specifying a function and a pair of input/output types. The function itself can be either a C-style function or a C++ functor. On line 5, the skeleton structure is defined using the `pipeline` and `farm` skeleton constructors and executed through the `run` function.

With QUAFF , the *same* language is used for describing the parallel structure of the application, writing application-specific sequential functions and as the target implementation language. This method has two advantages. First, programmers do not have to learn a separate language for describing this structure (as is the case with several existing skeleton-based parallel programming systems such as P3L[5] or Skipper[7]). Second, it makes insertion of existing sequential functions into skeletons easier and more efficient since no special foreign function interface is required: they just need to conform to the generic `t_result f(t_arg)` prototype.

3 Formal Model

The implementation model of QUAFF is CSP-based. A parallel program is described as a *process network*, *i.e.* a set of processes communicating by channels and executing each a sequence of instructions. In this section, we describe how such a process network can be built from the skeleton tree describing an application by means of a simple process algebra formalized by a set of *production rules*.

3.1 Process Network Description

Formally, a *process network* (PN) is a triple $\pi = \langle P, I, O \rangle$ where

- P is a set of labeled processes, i.e. pairs (pid, σ) where pid is a (unique) process id and σ a triple containing: a list[a] of *predecessors* (pids of processes p for which a communication channel exists from p to the current process), a list of *successors* (pids of processes p for which a communication channel exists from the current process to p) and a descriptor Δ. We note $\mathcal{L}(\pi)$ the set of pids of a process network π. For a process p, its predecessors, successors and descriptor will be denoted $\mathcal{I}(p)$, $\mathcal{O}(p)$ et $\delta(p)$ respectively.
- $I(\pi) \subseteq \mathcal{L}(\pi)$ denotes the set of *source* processes for the network π (i.e. the set of processes p for which $\mathcal{I}(p) = \emptyset$)
- $O(\pi) \subseteq \mathcal{L}(\pi)$ denotes the set of *sink* processes for the network π (i.e. the set of processes p for which $\mathcal{O}(p) = \emptyset$)

The process descriptor Δ is a pair $(instrs, kind)$ where $instrs$ is a sequence of (abstract) instructions and $kind$ a flag (the meaning of the $kind$ flag will be explained in Section 3.2).

$$\begin{array}{lll} \Delta & ::= & \langle \textit{instrs}, \textit{kind} \rangle \\ instrs & ::= & \textit{instr}_1, \ldots, \textit{instr}_n \\ \textit{kind} & ::= & \mathsf{Regular} \mid \mathsf{FarmM} \end{array}$$

The sequence of instructions describing the process behaviour is implicitly iterated (processes never terminate). Instructions use *implicit addressing*, with each process holding four variables named vi, vo, q and iws. The instruction set is given below. In the subsequent explanations, p designates the process executing the instruction.

$$\begin{array}{lll} \textit{instr} & ::= & \mathsf{SendTo} \mid \mathsf{RecvFrom} \mid \mathsf{CallFn\ fid} \mid \mathsf{RecvFromAny} \mid \mathsf{SendToQ} \mid \\ & & \mathsf{Ifq}\ \textit{instrs}_1\ \textit{instrs}_2 \mid \mathsf{GetIdleW} \mid \mathsf{UpdateWs} \end{array}$$

The SendTo instruction sends the contents of variable vo to the process whose *pid* is given in $\mathcal{O}(p)$. The RecvFrom instruction receives data from the process whose *pid* is given in $\mathcal{O}(p)$ and puts it in the variable vi. The CallFn instruction performs a computation by calling a sequential function. This function takes one argument (in vi) and produces one result (in vo). The RecvFromAny instruction waits (non-deterministically) data from the set of processes whose pids are given in $\mathcal{I}(p)$. The received data is placed in variable vi and the pid of the actual sending process in the variable q. The SendToQ instructions sends the contents of variable vo to the process whose pid is given by variable q. The Ifq instruction compares the value contained in variable q to the first pid listed in $\mathcal{I}(p)$. If case of equality, the instruction sequence $\textit{instrs}_1$ is executed; else $\textit{instrs}_2$ is executed. The UpdateWs instruction reads variable q and updates the variable iws accordingly. The variable iws maintains the list of idle workers for FARM master processes. The GetIdleW retrieves a process id from the iws list and places it in the variable q. Together, these two instructions encapsulate the policy used in a FARM skeleton to allocate data to workers. They are not detailed further here.

[a] Note that this is really a list, and not a set, since the order is relevant.

3.2 A Basic Process Network Algebra

The following notation will be used. If $\mathcal{E}$ is a set, we denote by $\mathcal{E}[e \leftarrow e']$ the set obtained by replacing e by e' (assuming $\mathcal{E}[e \leftarrow e'] = \mathcal{E}$ if $e \notin \mathcal{E}$). This notation is left-associative: $\mathcal{E}[e \leftarrow e'][f \leftarrow f']$ means $(\mathcal{E}[e \leftarrow e'])[f \leftarrow f']$. If $e_1, \ldots, e_m$ is an indexed subset of $\mathcal{E}$ and $\phi : \mathcal{E} \rightarrow \mathcal{E}$ a function, we will note $\mathcal{E}[e_i \leftarrow \phi(e_i)]_{i=1..m}$ the set $(\ldots((\mathcal{E}[e_1 \leftarrow \phi(e_1)])[e_2 \leftarrow \phi(e_2)])\ldots)[e_m \leftarrow \phi(e_m)]$. Except when explicitly indicated, we will note $I(\pi_k) = \{i_k^1, \ldots, i_k^m\}$ and $O(\pi_k) = \{o_k^1, \ldots, o_k^n\}$. For concision, the lists $\mathcal{I}(o_k^j)$ et $\mathcal{O}(i_k^j)$ will be noted s_k^j et d_k^j respectively. For lists, we define the a concatenation operation ++ as usual : if $l_1 = [e_1^1, \ldots, e_1^m]$ and $l_2 = [e_2^1, \ldots, e_2^n]$ then l_1++$l_2 = [e_1^1, \ldots, e_1^m, e_2^1, \ldots; e_2^n]$. The empty list is noted []. The length of list l (resp. cardinal of a set l) is noted $|l|$.

The $\lceil . \rceil$ operator creates a process network containing a single process from a process descriptor, using the function NEW() to provide "fresh" process ids :

$$\frac{\delta \in \Delta \qquad l = \text{NEW}()}{\lceil \delta \rceil = \langle \{(l, \langle [], [], \delta \rangle)\}, \{l\}, \{l\} \rangle} \qquad \text{(SINGL)}$$

The $\bullet$ operation "serializes" two process networks, by connecting outputs of the first to the inputs of the second :

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2) \qquad |O_1| = |I_2| = m}{\pi_1 \bullet \pi_2 = \langle (P_1 \cup P_2)[(o_1^j, \sigma) \leftarrow \phi_d((o_1^j, \sigma), i_2^j)]_{j=1\ldots m}[(i_2^j, \sigma) \leftarrow \phi_s((i_2^j, \sigma), o_1^j)]_{j=1\ldots m}, I_1, O_2 \rangle} \qquad \text{(SERIAL)}$$

This rule uses two auxiliary functions ϕ_s and ϕ_d defined as follows :

$\phi_s((p, \langle s, d, \langle \delta, \mathsf{Regular} \rangle \rangle), p') = (p, \langle [p']$++$s, d, \langle [\mathsf{RecvFrom}]$++$\delta, \mathsf{Regular} \rangle \rangle)$
$\phi_d((p, \langle s, d, \langle \delta, \mathsf{Regular} \rangle \rangle), p') = (p, \langle s, d$++$[p'], \langle \delta$++$[\mathsf{SendTo}], \mathsf{Regular} \rangle \rangle)$
$\phi_s((p, \langle s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle), p') = (p, \langle [p']$++$s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle)$
$\phi_d((p, \langle s, d, \langle \delta, \mathsf{FarmM} \rangle \rangle), p') = (p, \langle s, d$++$[p'], \langle \delta, \mathsf{FarmM} \rangle \rangle)$

The function ϕ_s (resp. ϕ_d) adds a process p' as a predecessor (resp. successor) to process p and updates accordingly its instruction list. This involves prepending (resp. appending) a RecvFrom (resp. SendTo) instruction) to this instruction list, *except for* FARM masters (identified by the FarmM kind flag), for which the instruction list is not modified.

The $\|$ operation puts two process networks in parallel, merging their inputs and outputs respectively.

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2)}{\pi_1 \parallel \pi_2 = \langle P_1 \cup P_2, I_1 \cup I_2, O_1 \cup O_2 \rangle} \qquad \text{(PAR)}$$

The $\bowtie$ operation merges two process networks by connecting each input and output of the second to the output of the first :

$$\frac{\pi_i = \langle P_i, I_i, O_i\rangle\ (i=1,2) \qquad |O_1| = 1 \qquad |I_2| = m \qquad |O_2| = n}{\begin{array}{c}\pi_1 \bowtie \pi_2 = \langle (P_1 \cup P_2)[(o_1,\sigma) \leftarrow \Phi((o_1,\sigma), I(\pi_2), O(\pi_2))][(i_2^j,\sigma) \leftarrow \phi_s((i_2^j,\sigma), o_1)]_{j=1\ldots m} \\ [(o_2^j,\sigma) \leftarrow \phi_d((i_2^j,\sigma), o_1)]_{j=1\ldots n}, \\ I_1, O_1\rangle\end{array}}$$

(JOIN)

where $\Phi(p, ps_s, ps_d) = \Phi_s(\Phi_d(p, ps_d), ps_s)$ and Φ_s (resp. Φ_d) generalizes the function ϕ_s (resp. ϕ_d) to a list of processes :

$$\Phi_s(p, [p_1, \ldots, p_n]) = \phi_s(\ldots, \phi_s(\phi_s(p, p_1), p_2), \ldots, p_n)$$
$$\Phi_d(p, [p_1, \ldots, p_n]) = \phi_d(\ldots, \phi_d(\phi_d(p, p_1), p_2), \ldots, p_n)$$

Skeletons can now be defined in terms of the operations defined above, using the following conversion function $\mathcal{C}^b$:

$$\begin{aligned}
\mathcal{C}[[\mathsf{Seq}\ f]] &= \lceil f \rceil \\
\mathcal{C}[[\mathsf{Pipe}\ \Sigma_1 \ldots \Sigma_n]] &= \mathcal{C}[[\Sigma_1]] \bullet \ldots \bullet \mathcal{C}[[\Sigma_n]] \\
\mathcal{C}[[\mathsf{Farm}\ n\ \Sigma]] &= \lceil \mathsf{FarmM} \rceil \bowtie (\mathcal{C}[[\Sigma]]_1 \parallel \ldots \parallel \mathcal{C}[[\Sigma]]_n) \\
\mathcal{C}[[\mathsf{Scm}\ m\ f_s\ \Sigma\ f_m]] &= \mathcal{C}[[\mathsf{Seq}\ f_s]] \triangleleft (\mathcal{C}[[\Sigma]]_1 \parallel \ldots \parallel \mathcal{C}[[\Sigma]]_m) \triangleright \mathcal{C}[[\mathsf{Seq}\ f_m]] \\
\mathcal{C}[[\mathsf{Pardo}\ \Sigma_1\ \ldots \Sigma_n]] &= \mathcal{C}[[\Sigma_1]] \parallel \ldots \parallel \mathcal{C}[[\Sigma_n]]
\end{aligned}$$

where FarmM is a process descriptor predefined as :

$$\Delta(\mathsf{FarmM}) = \langle [\mathsf{RecvFromAny}; \mathsf{Ifq}\ [\mathsf{GetIdleW}; \mathsf{SendToQ}]\ [\mathsf{UpdateWs}; \mathsf{SendTo}]], \mathsf{FarmM}\rangle$$

4 Implementation

We now explain how the production rules and the conversion function $\mathcal{C}$ introduced in the previous section can be implemented as a compile-time process. The process itself is sketched on Fig. 1.

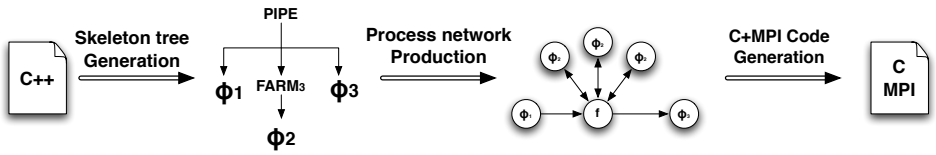


Figure 1. QUAFF code generation process

It consists of three steps: generating the skeleton tree, turning this structure into a process network and producing the C+MPI target code. These three steps are carried out at

[b] The production rules for the operators $\triangleleft$ and $\triangleright$, used in the definition of the Scm skeleton have been omitted due to space constraints.

compile-time. The key idea is that each *object* of the formal semantics defined in Section 3 is encoded as a *type* in the implementation language. Production rules, in particular, are encoded as *meta-programs* taking arguments and producing results as C++ types. The whole process is illustrated with a very simple application consisting of a two-stages pipeline. Using QUAFF, this application is written:

```
run( pipeline(seq(F1),seq(F2)) );
```

where `F1` and `F2` are registered sequential functions, as illustrated in Listing 28.1.

4.1 Generating the Skeleton Tree

For each skeleton, the corresponding function at the API level returns a value which type is a compile-time representation of the skeleton. Here's, for example, the definition of the `seq` and `farm` functions:

Listing 28.2. Skeleton constructors for SEQ and FARM

```
template<class F>
Seq<F> seq(const F&) { return Seq<F>(); }

template<int N, class W>
Farm<N,W> farm(const W&) { return Farm<N,W>(); }
```

In the two-stages pipeline example, the call `run()` at the API level need to call the `pipeline` function, and therefore the `seq` function. This will generate the following residual code, in which the argument to the `run` function is an instance of a type encoding of the skeleton tree:

```
run( Serial< Seq<F1>, Seq<F2> >() );
```

This *template* now carries informations about the skeleton tree in a form usable by our meta-functions.

4.2 Producing the Process Network

We first give, in Listing 28.3, the type encoding of *process network*, *labeled process* and *process descriptor* objects. Each of these types is defined as a simple template container, with arguments statically encoding the aggregated objects. In the `process_network` type, the `P`, `I` and `O` fields are compile-time lists of process `ID`s. Technically speaking, process `ID`s themselves are encoded as type-embedded integral constants and type lists are built and manipulated using the `BOOST::MPL` library[3]. In the `process` type, the `input_type` and `output_type` encode the argument and return type of the associated user-defined function. In the `descriptor` type, the `i_pids` and `o_pids` fields encode the list of successors and predecessors respectively and the `instrs` field encodes the list of instructions executed by the process.

Listing 28.3. process_network, process and descriptor related data types

```
template<class P,class I,class O> struct process_network
{
  typedef P    process;
  typedef I    inputs;
  typedef O    outputs;
};

template<class ID,class DESC, class IT, class OT> struct process
{
  typedef ID       pid;
  typedef DESC     descriptor;
  typedef IT       input_type;
  typedef OT       output_type;
};

template<class IPID,class OPID,class CODE, class KIND> struct descriptor
{
  typedef IPID  i_pids;
  typedef OPID  o_pids;
  typedef CODE  instrs;
  typedef KIND  kind;
};
```

Listing 28.4. The `run` function

```
template<class SKL> void run( const SKL& )
{
  typedef typename convert<SKL>::type p_net;
  p_net::Run();
}
```

The `run` function now has to convert the type describing the skeleton tree produced by the previous step into a type describing the corresponding process network (*i.e.* to implement the $\mathcal{C}$ function specified in Section 3.2).

This code show that `run` simply extracts type informations from its *template* parameter and pass it through the `convert` meta-function. This function is statically overloaded for each skeleton constructor. Listing 28.5 shows the overloaded meta-function for the `pipeline` skeleton.

Listing 28.5. `pipeline` *template* conversion

```
template<class S0,class S1,class ID> struct convert<Serial<S0,S1>,ID>
{
  typedef Serial<S1,mpl::void_>                    tail;
  typedef typename convert<S0,ID>::type            proc1;
  typedef typename convert<S0,ID>::new_id          next_id;
  typedef typename convert<tail,next_id>::new_id   new_id;
  typedef typename convert<tail,next_id>::type     proc2;
  typedef typename rule_serial<proc1,proc2>::type  type;
};
```

The `convert` meta-function extracts the skeleton sub-trees from `S0` and `S1`, converts them into process networks, computes a new process ID and apply the appropriate production rule (SERIAL) to generate a new process network embedded in the `type` *typedef*.

The production rules are also implemented as meta-programs. The *template* equivalent of the rule SERIAL defined in Section 3.2, for example, is given in Listing 28.6. This template takes as parameters the types encoding the two process networks to serialize. The type encoding the resulting process network is then built incrementally, by means of successive type definitions, each type definition precisely encoding a value definition of the formal production rule and by using MPL meta-function like `transform` which is a meta-programmed iterative function application or `copy` which is used in conjunction with the `back_inserter` manipulator to concatenates two lists of process networks.

Listing 28.6. The meta-programmed (SERIAL) rule

```
template<class P1, class P2> struct rule_serial
{
  // Get list of processes and I/O from P1 and P2
  typedef typename P1::process                                          proc1;
  typedef typename P2::process                                          proc2;
  typedef typename P1::inputs                                           i1;
  typedef typename P2::inputs                                           i2;
  typedef typename P1::outputs                                          o1;
  typedef typename P2::outputs                                          o2;

  // Add new process graph into the new process network
  typedef typename mpl::transform< proc1, phi_d<_1,o1,i2> >::type       np1;
  typedef typename mpl::transform< proc2, phi_s<_1,i2,o1> >::type       np2;
  typedef typename mpl::copy<np2, mpl::back_inserter<np1> >::type       process;

  // Process network definition
  typedef process_network<process,i1,o2>                                type;
};
```

4.3 Generating Parallel Application Code

The last step consists in turning the process network representation into C+MPI code. This transformation is triggered at the end of the `run` function. The `Run` method of the `process_network` class created by the application of `convert` sequentially instantiates and executes each macro-instructions of its descriptor. The actual process of turning the macro-instructions list into an C+MPI code is based on tuple generation similar to the one used in the previous QUAFF implementation[1]. Each instance is then able to check if its PID matches the actual process rank and executes its code. For our two-stages pipeline, the residual code looks like as shown in Listing 28.7

5 Experimental Results

We have assessed the impact of this implementation technique by measuring the overhead ρ introduced by QUAFF on the *completion time* over hand-written C+MPI code for both

Listing 28.7. Generated code for the two stage pipeline

```
if( Rank() == 0 )
{
  do {
    out = F1();
    MPI_Send(&out,1,MPI_INT,1,0,MPI_COMM_WORLD);
  } while( isValid(out) )
}
else if( Rank() == 1 )
{
  do {
    MPI_Recv(&in,1,MPI_INT,0,0,MPI_COMM_WORLD,&s);
    F2(in);
  } while( isValid(in) )
}
```

single skeleton application and when skeletons are nested at arbitrary level. For single skeleton tests, we observe the effect of two parameters: τ, the execution time of the inner sequential function and N, the "size" of the skeleton (number of stages for PIPELINE, number of workers for FARM and SCM). The test case for nesting skeletons involved nesting P FARM skeletons, each having ω workers. Results were obtained on a PowerPC G5 cluster with 30 processors and for $N = 2 - 30$ and $\tau = 1ms, 10ms, 100ms, 1s$.

For PIPELINE, ρ stays under 2%. For FARM and SCM, ρ is no greater than 3% and becomes negligible for $N > 8$ or $\tau > 10ms$. For the nesting test, worst case is obtained with $P = 4$ and $\omega = 2$. In this case, ρ decreases from 7% to 3% when τ increases from $10^{-3}s$ to $1s$.

6 Related Work

The idea of relying on a process network as an intermediate representation for skeletal programs is not new; in fact, several implementations of skeleton-based parallel programming libraries, such as P3L[5], implicitly rely on such a representation. But, the process of translating the skeleton tree into such a network has never been formalized before. Aldinucci[8] proposed a formal operational semantics for skeleton-based programs but, contrary to QUAFF , the actual implementation relies on a dynamic runtime. Thus, to our best knowledge, our work is the first to both rely on a formal approach to skeleton compilation while offering performances on par with hand-coded C+MPI implementations.

On the other hand, using generative programming and meta-programming for implementing parallel applications and libraries is currently an upcoming trend. Works by Czarnecki and al.[9],Puschel and al.[10], Hammond[11], Langhammer and Hermann[12] uses meta-programming in `MetaOCaml`[13] or `Template Haskell` to generate parallel *domain specific languages* for solving problem like signal processing optimizations or parallel processes scheduling on MIMD machines thus making generative programming a valid technique to solve realistic problems. Our work can be viewed as a specific application of these general techniques.

7 Conclusion

In this paper, we have the shown how generative and meta-programming techniques can be applied to the implementation of a skeleton-based parallel programming library . The resulting library, QUAFF , both offers a high level of abstraction and produces high performance code by performing most of the high to low-level transformations at compile-time rather than run-time. The implementation is derived directly from a set of explicit production rules, in a semantic-oriented style. It is therefore formally sounded and much more amenable to proofs or extensions.

References

1. J. Falcou, J. Sérot, T. Chateau and J.-T. Lapresté, *QUAFF: Efficient C++ Design for Parallel Skeletons*, Parallel Computing, **32**, 604–615, (2006).
2. T. Veldhuizen, *Using C++ template metaprograms*, *C++ Report*, **7**, 36–43, (1995). Reprinted in *C++ Gems*, ed. Stanley Lippman.
3. D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond* C++ in Depth Series, (Addison-Wesley Professional, 2004).
4. H. Kuchen, *A skeleton library*, in: *Euro-Par '02: Proc. 8th International Euro-Par Conference on Parallel Processing*, pp. 620–629, London, UK, (Springer-Verlag, 2002).
5. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti and M. Vanneschi, *P3l: A structured high level programming language and its structured support*, Concurrency: Practice and Experience, **7**, 225–255, (1995).
6. M. Cole, *Research Directions in Parallel Functional Programming*, Chapter 13, Algorithmic skeletons, (Springer, 1999).
7. J. Sérot and D. Ginhac, *Skeletons for parallel image processing: an overview of the skipper project*, Parallel Computing, **28**, 1685–1708, (2002).
8. M. Aldinucci and M .Danelutto, *Skeleton based parallel programming: functional and parallel semantic in a single shot*, in: Computer Languages, Systems and Structures, (2006).
9. K. Czarnecki, J.T. O'Donnell, J. Striegnitz and W. Taha, *Dsl implementation in metaocaml, template haskell and C++*, in: C. Lengauer, D. Batory, C. Consel, and M. Odersky, eds., Domain-Specific Program Generation, Lecture Notes in Computer Science, vol. **3016**, pp. 51–72, (Springer-Verlag, 2004).
10. M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson and N. Rizzolo, *SPIRAL: Code Generation for DSP Transforms*, in: Proc. IEEE Special Issue on Program Generation, Optimization, and Adaptation, (2005).
11. K. Hammond, R. Loogen and J. Berhold, *Automatic Skeletons in Template Haskell*, in: Proc. 2003 Workshop on High Level Parallel Programming, Paris, France, (2003).
12. Ch. A. Herrmann and T. Langhammer, *Combining partial evaluation and staged interpretation in the implementation of domain-specific languages*, Sci. Comput. Program., **62**, 47–65, (2006).
13. MetaOCaml. A compiled, type-safe multi-stage programming language. Available online from http://www.metaocaml.org/, (2003).

Numerical Algorithms and Automatic Differentiation

Strategies for Parallelizing the Solution of Rational Matrix Equations

José M. Badía[1], Peter Benner[2], Maribel Castillo[1], Heike Faßbender[3], Rafael Mayo[1], Enrique S. Quintana-Ortí[1], and Gregorio Quintana-Ortí[1]

[1] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain
E-mail: {badia, castillo, mayo, quintana, gquintan}@icc.uji.es

[2] Fakultät für Mathematik, Technische Universität Chemnitz, 09107 Chemnitz, Germany
E-mail: benner@mathematik.tu-chemnitz.de

[3] Technische Universität Braunschweig, Institut Computational Mathematics, 38106 Braunschweig, Germany
E-mail: h.fassbender@tu-bs.de

In this paper we apply different strategies to parallelize a structure-preserving doubling method for the rational matrix equation $X = Q + LX^{-1}L^T$. Several levels of parallelism are exploited to enhance performance, and standard sequential and parallel linear algebra libraries are used to obtain portable and efficient implementations of the algorithms. Experimental results on a shared-memory multiprocessor show that a coarse-grain approach which combines two MPI processes with a multithreaded implementation of BLAS in general yields the highest performance.

1 Introduction

The nonlinear matrix equation

$$X = Q + LX^{-1}L^T, \tag{1.1}$$

where $Q \in \mathbb{R}^{n\times n}$ is symmetric positive definite, $L \in \mathbb{R}^{n\times n}$ is nonsingular, and $X \in \mathbb{R}^{n\times n}$ is the sought-after solution, arises in the analysis of stationary Gaussian reciprocal processes over a finite interval. The solution of certain 1-D stochastic boundary-value systems are reciprocal processes. For instance, the steady-state distribution of the temperature along a heated ring or beam subjected to random loads along its length can be modeled in terms of such reciprocal processes[1].

The problem considered here is to find the (unique) largest positive definite symmetric solution X_+ of (1.1) when n is of large dimension ($n \approx O(1,000) - O(10,000)$). Given the cubic computational cost of numerically reliable solvers[2], the use of parallel computers is necessary for large-scale rational matrix equations (RMEs). Several methods have been proposed for solving (1.1) in the past[2,3,4]. Among these, we select the method based on the solution of a discrete-time algebraic Riccati equation (DARE) via a structure-preserving doubling algorithm (SDA), because of its parallel properties.

In this paper we compare different strategies to parallelize the solution of the RME 1.1 via the SDA. The strategies attack the problem at three levels of granularity, or combinations of those, and rely on different programming models and standard libraries. In many cases, the additional effort required to analyze the parallelism of the problem from different

perspectives, so that combining two or more levels of granularity becomes feasible, yields a higher parallel efficiency of the algorithm.

The paper is structured as follows. In Section 2 we review the SDA for the RME (1.1). In Section 3 we describe several different approaches for the parallelization of the method. The results in Section 4 report the performance of these strategies on a shared-memory multiprocessor consisting of 16 Intel Itanium2 processors. Finally, some concluding remarks follow in Section 5.

2 Numerical Solution of Rational Matrix Equations via a Structure-Preserving Doubling Algorithm

The solution X of

$$X = Q + LX^{-1}L^T$$

satisfies[3]

$$G \begin{bmatrix} I \\ X \end{bmatrix} = H \begin{bmatrix} I \\ X \end{bmatrix} W \tag{2.1}$$

for some matrix $W \in \mathbb{R}^{n\times n}$, where

$$G = \begin{bmatrix} L^T & 0 \\ -Q & I \end{bmatrix}, \qquad H = \begin{bmatrix} 0 & I \\ L & 0 \end{bmatrix}.$$

Hence, the desired solution X can be computed via an appropriate deflating subspace of $G - \lambda H$. The following idea[4] delivers a fast solver based on these ideas.

Assume that X is the unique symmetric positive definite solution of (1.1). Then it satisfies (2.1) with $W = X^{-1}L^T$. Let

$$\widehat{L} = LQ^{-1}L, \quad \widehat{Q} = Q + LQ^{-1}L^T, \quad \widehat{P} = L^TQ^{-1}L,$$

and

$$\widehat{X} = X + \widehat{P}.$$

Then it follows that

$$\widehat{G} \begin{bmatrix} I \\ \widehat{X} \end{bmatrix} = \widehat{H} \begin{bmatrix} I \\ \widehat{X} \end{bmatrix} W^2, \tag{2.2}$$

where

$$\widehat{G} = \begin{bmatrix} \widehat{L}^T & 0 \\ \widehat{Q} + \widehat{P} & -I \end{bmatrix}, \qquad \widehat{H} = \begin{bmatrix} 0 & I \\ \widehat{L} & 0 \end{bmatrix}.$$

It is easy to see that $\widehat{X}$ satisfies (2.2) if and only if the equation

$$\widehat{X} = (\widehat{Q} + \widehat{P}) - \widehat{L}\widehat{X}^{-1}\widehat{L}^T \tag{2.3}$$

has a symmetric positive definite solution $\widehat{X}$.

The doubling algorithm[4] was recently introduced to compute the solution $\widehat{X}$ of (2.3) using an appropriate doubling transformation for the symplectic pencil (2.2). Applying this special doubling transformation repeatedly, the SDA in Fig. 1 is derived[4].

$L_0 = \widehat{L}\ ,\ Q_0 = \widehat{Q} + \widehat{P}\ ,\ P_0 = 0$
for $i = 0, 1, 2, \ldots$
1. $Q_i - P_i = C_i^T C_i$
2. $A_C = L_i^T C_i^{-1}$
3. $C_A = C_i^{-T} L_i^T$
4. $Q_{i+1} = Q_i - C_A^T C_A$
5. $P_{i+1} = P_i + A_C A_C^T$
6. $L_{i+1} = A_C C_A$

until convergence

Figure 1. SDA

As the matrices $Q_i - P_i$, $i = 0, 1, 2, \ldots$ are positive definite[4], the iterations are all well-defined and the sequence Q_{i+1} will converge to $\widehat{X}$. Thus, the unique symmetric positive definite solution to (1.1) can be obtained by computing

$$X_+ = \widehat{X} - \widehat{P}. \tag{2.4}$$

The SDA has nice numerical behaviour, with a quadratic convergence rate, low computational cost, and fair numerical stability. The algorithm requires about $6.3n^3$ floating-point arithmetic (flops) operations per iteration step when implemented as follows: first a Cholesky factorization $(Q_i - P_i) = C_i^T C_i$ is computed ($\frac{n^3}{3}$ flops), then $L_i^T C_i^{-1}$ and $C_i^{-T} L_i^T$ are solved (each triangular linear system requires n^3 flops), and finally $L_{i+1},\ Q_{i+1},\ P_{i+1}$ are computed using these solutions ($4n^3$ flops if the symmetry of Q_{i+1} and P_{i+1} is exploited). Hence, one iteration step requires $\frac{19}{3} n^3 \approx 6.3n^3$ flops.

The iteration for the SDA is basically composed of traditional dense linear algebra operations such as the Cholesky factorization, solution of triangular linear systems, and matrix products. On serial computers these operations can be efficiently performed using (basic) kernels in BLAS and (more advanced) routines in LAPACK[5].

3 Parallelization

We can parallelize the SDA at different levels (see Figure 2):

- At the highest level, several operations of the sequential algorithm can be executed concurrently. Thus, e.g., the computations of A_C and C_A are independent operations which can be performed simultaneously.
- At an intermediate level, parallel implementations of linear algebra kernels and routines in ScaLAPACK[6] can be used to extract parallelism from each operation. As an example, the parallel routine `pdpotrf` in ScaLAPACK can be used to compute the Cholesky factorization $Q_i - P_i = C_i^T C_i$.
- At the lowest level, multithreaded implementations of BLAS can be used to extract parallelism from the calls to BLAS performed from within each operation, as for example, in invocations to BLAS from the routine for the Cholesky factorization.

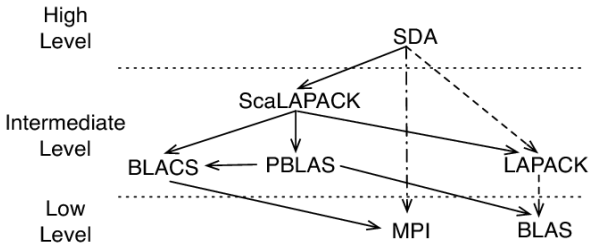


Figure 2. Parallel levels and libraries exploited by the different strategies for the parallelization of the SDA.

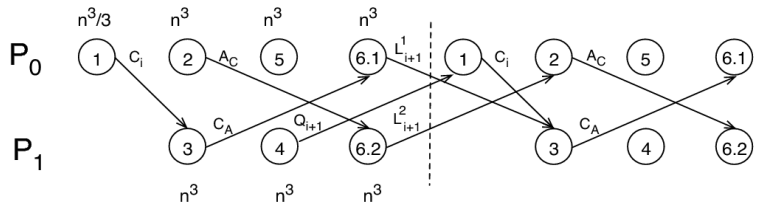


Figure 3. Execution of two consecutive iterations of the 2P parallelization of the SDA by two processes, P_0 and P_1.

Exploiting the two highest levels of parallelism is more natural on platforms with distributed-memory, including multicomputers and clusters. On the other hand, multi-threaded implementations of BLAS are only suited for architectures with shared address space as, e.g., SMPs or multicore processors.

Following these possibilities, we have developed several parallel implementations of the SDA:

- **Message-passing, two processes (2P)**. An analysis of the data dependency graph of the algorithm shows that there are two groups of operations which can be performed concurrently. In this first approach, one MPI process is employed per group.

  Figure 3 illustrates the execution of two consecutive iterations of the SDA (see Algorithm 1) in this approach. The nodes are labeled with their corresponding number of the operation of the SDA, and the computational cost of each operation is included. The arrows represent communication of intermediate results. In order to balance the computational load, we divide the matrix product in step 6 between the two processes. Thus, each process computes half of the columns of the matrix L_{i+1}, which needs then to be communicated to the other process.

  A serial implementation of BLAS and LAPACK is used on each process to perform the different operations of the algorithm.

- **Message-passing multiprocesses (MP)**. We have also developed a message-passing implementation that computes all operations in the SDA using the parallel routines in ScaLAPACK.

- **Multithreading (MT)**. In this case the *parallel* algorithm is reduced to a "serial" code that uses BLAS and LAPACK, and extracts all parallelism from a multithreaded implementation of BLAS.
- **Hybrid (HB)**. Different hybrid versions of the parallel algorithm can be obtained by combining two of the levels of parallelism described earlier. In particular, we have implemented and tested two hybrid algorithms:
  - **Two-process hybrid algorithm (HB2P)**. Each of the two processes in the 2P implementation invokes the kernels from a multithreaded version of BLAS to take advantage of a system with more than two processors.
  - **Multiprocess hybrid algorithm (HBMP)**. Each of the MPI processes executing the ScaLAPACK routines invokes kernels from a multithreaded implementation of BLAS.

4 Numerical Experiments

All the experiments presented in this section were performed on a SGI Altix 350. This is a (CC-NUMA) shared-memory multiprocessor composed of 16 Intel Itanium2 processors running at 1.5 GHz. The processors share 32 GBytes of RAM via a *SGI NUMAlink* interconnect. Two different (multithreaded) implementations of BLAS were used on this platform:

- The SGI *Scientific Computing Software library* (SCSL).
- The *Math Kernel Library* (MKL) 8.1 from Intel.

We have also used the implementation of ScaLAPACK provided by SGI. For simplicity, we only report results for the parallel implementations combined with the optimal implementation of BLAS for each case. All the speed-ups reported next have been measured with respect to an optimized sequential implementation of the SDA.

Figures 4 and 5 show the speed-up of the parallel implementations of the SDA. The results of the HB2P parallel implementation for two processors correspond to those of the 2P algorithm, while the results of the HB2P algorithm for more than two processors correspond to the hybrid version that combines two MPI processes with different numbers of threads. Thus, for example, the results of HB2P on 10 processors involve two processes, each running five threads to exploit a multithreaded implementation of BLAS. Although we have also tested the HBMP parallel implementation, results are not reported for this option as the performance was always lower than those of the MP and MT parallel implementations.

The figures show that, as could be expected, the efficiency (speed-up divided by the number or processors) decreases with the number of processors while the speed-up increases with the equation size. For equations of "small" size, algorithms MP and HB2P obtain similar speed-ups but, as the equation size is increased, the hybrid algorithm outperforms the message-passing version. The difference between these two algorithms remains almost constant regardless of the number of processors. In general, the highest speed-ups are delivered by the algorithm HB2P. On the other hand, the MT algorithm exhibits scalability problems with the number of processors (Figure 4) and, when more than 4 processors

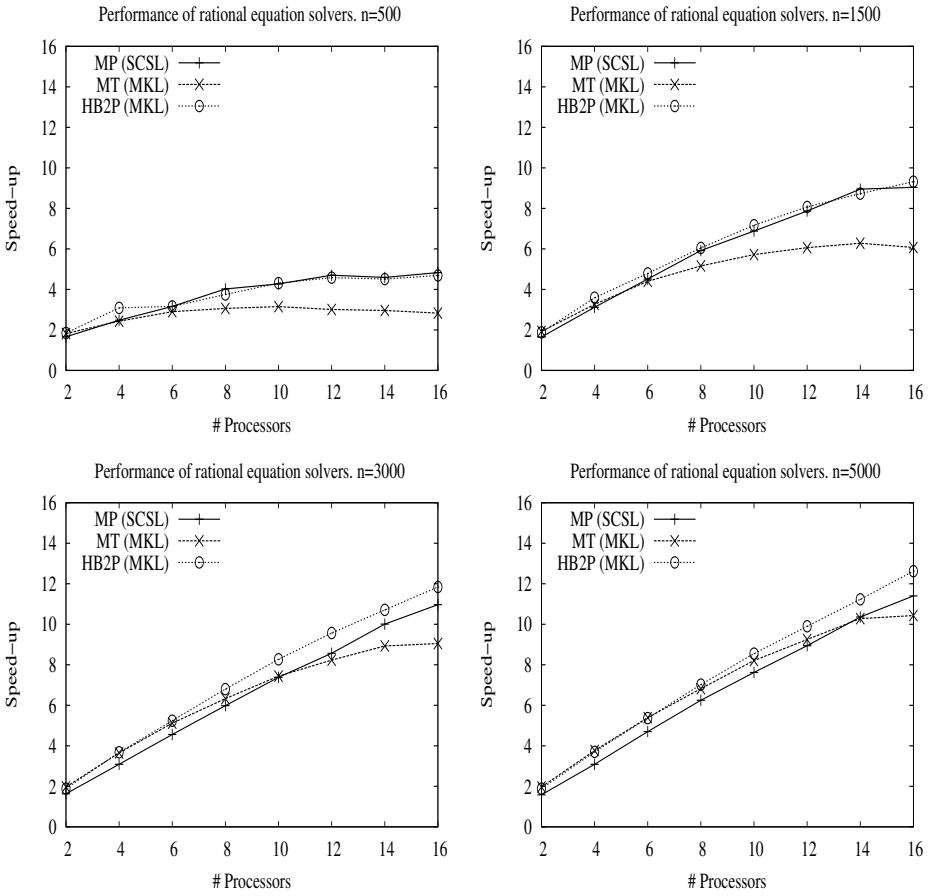


Figure 4. Speed-ups of the parallel implementations MP, MT, 2P (=HB2P on 2 processors), and HB2P of the SDA for n=500, 1500, 3000, 5000, and varying number of processors.

are employed, requires equations of large size to yield a performance comparable to those of the other two approaches (Figure 5).

Possible causes of these results are the following:

- In general, exploiting the parallelism at a higher level identifies more work to be "shared" among the computational resources and should deliver higher performances.

- In the MT algorithm, all threads/processors must access a unique copy of the data (matrices and vectors) in the main memory. Thus, the coherence hardware of this architecture is likely to be stressed to keep track of data that is shared. The overall cost of data transference between cache memories becomes higher as the number of processors is increased, explaining the lower performance of the MT algorithm when the number of processors is large.

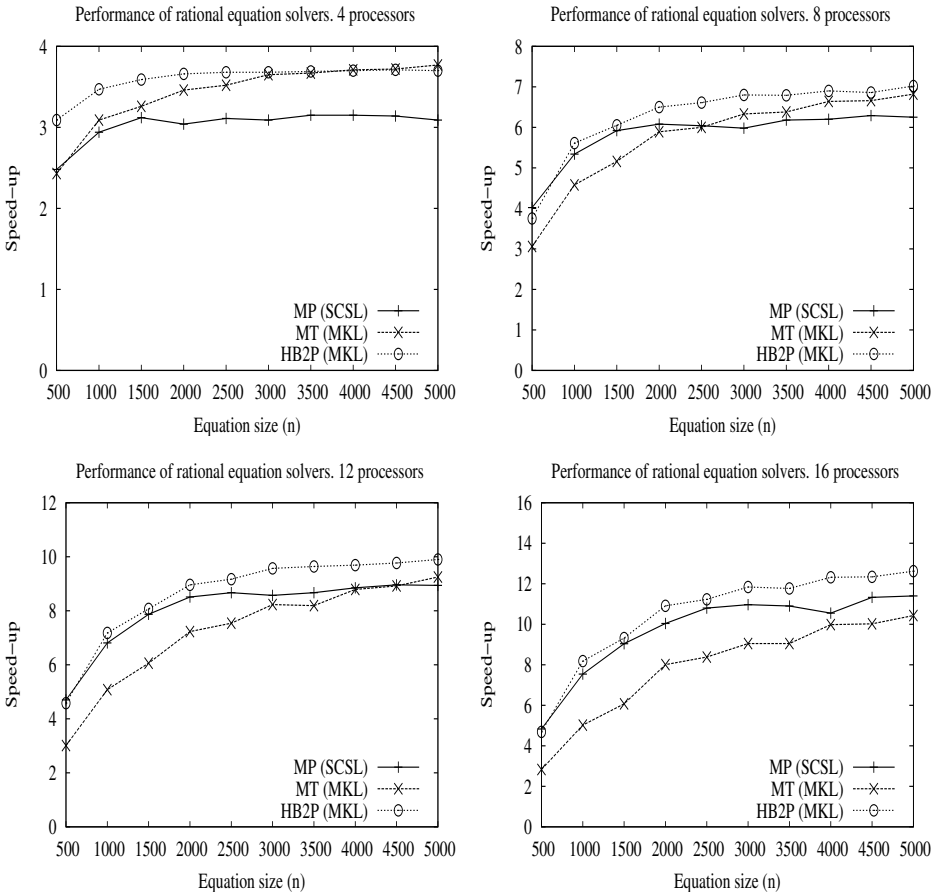


Figure 5. Speed-ups of the parallel implementations MP, MT, 2P (=HB2P on 2 processors), and HB2P of the SDA on 4, 8, 12, 16 processors and varying dimension of the equation.

- In the MP algorithm, the data is partitioned among the processes. As the SGI Altix 350 is a CC-NUMA platform, the data is also likely to be distributed so that data accessed from one process lies in the "local" memory of the corresponding processor. Thus, as most of the accesses performed by the algorithm are satisfied by the local memory, this solution incurs a lower memory access overhead than the MT algorithm when the number of processors is large. On the other hand, when the number of processors is small, sharing data between threads, as in the MT algorithm, is more efficient than doing so via processes, as in the MP algorithm.

- The hybrid algorithm increases the data locality of memory accesses performed by the threads since, in this approach, the amount of data shared by each thread group is roughly halved with respect to the MT algorithm.

5 Conclusions

In this paper we have implemented five different parallel strategies to solve the RME (1.1) via a structure-preserving doubling algorithm. The algorithms pursue the solution of the equation at different levels of parallelization: at the highest level we have used MPI processes to execute the different operations of the sequential algorithm concurrently. At the lowest level we have exploited a multithreaded version of the BLAS library to execute each basic linear algebra kernel in parallel.

Experiments on a shared-memory multiprocessor show that the best results are obtained with an hybrid parallel algorithm that combines MPI processes with threads. This algorithm obtains high speed-ups and scales linearly up to 16 processors. High performance is also obtained with the multiprocess version of the algorithm which employs only routines from ScaLAPACK. These results confirm that parallelizing the solution of the problem at different levels is worth the effort.

All the parallel algorithms are implemented using standard sequential and parallel linear algebra libraries and MPI, ensuring the portability of the codes.

Acknowledgements

José M. Badía, Maribel Castillo, Rafael Mayo, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí were supported by the CICYT project TIN2005-09037-C02-02 and FEDER. These authors and Peter Benner were also supported by the DAAD Acciones Integradas project HA2005-0081 (Spain), D/05/25675 (Germany).

References

1. B. C. Levy, R. Frezza and A. J. Kerner, *Modeling and estimation of discrete-time Gaussian reciprocal processes*, IEEE Trans. Automat. Control, **90**, 1013–1023, (1990).
2. P. Benner and H. Faßbender, *On the solution of the rational matrix equation* $X = Q + LX^{-1}L^T$, EURASIP J. Adv. Signal Proc., **2007**, Article ID 21850, (2007).
3. A. Ferrante and B. B. Levy, *Hermitian Solutions of the Equation* $X = Q+NX^{-1}N^*$, Linear Algebra Appl., **247**, 359–373, (1996).
4. W.-W Lin and S.-F Xu, *Convergence analysis of structure-preserving doubling algorithms for Riccati-type matrix equations*, SIAM J. Matrix Anal. Appl., **28**, 26–39, (2006).
5. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, *LAPACK Users' Guide*, Third edition, (SIAM, Philadelphia, 1999).
6. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker and R. C. Whaley, *ScaLAPACK Users' Guide*, (SIAM, Philadelphia, 1997).

A Heterogeneous Pipelined Parallel Algorithm for Minimum Mean Squared Error Estimation with Ordered Successive Interference Cancellation

Francisco-Jose Martínez-Zaldívar[1], **Antonio. M. Vidal-Maciá**[2], **and Alberto González**[1]

[1] Departamento de Comunicaciones, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
E-mail: {fjmartin, agonzal}@dcom.upv.es

[2] Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
E-mail: avidal@dsic.upv.es

1 Introduction

Multiple Input Multiple Output (MIMO) systems have been extensively studied in recent years in the context of wireless communications. The original proposal by Foschini[1], known as BLAST (Bell Labs Layered Space-Time), has generated a family of architectures that uses multiple antenna arrays to transmit and receive information – with the aim of increasing the capacity and reliability of links. One of these architectures is V-BLAST (Vertical-BLAST). In this architecture, we can use linear decoders such as: Zero Forcing; MMSE; and the ordered version OSIC[2] (Ordered Successive Interference Cancellation), to be used in applications such as multicarrier systems[3] (i.e. OFDM —Orthogonal Frequency Division Multiplex— in Digital Video Broadcasting-Terrestrial or DVB-T) where the dimension of the problem may be around several thousands.

This paper describes a novel algorithm to solve the OSIC decoding problem and its parallelization — with better performance than those reported in the literature. Firstly, the basic OSIC decoding procedure is shown and cost comparisons made using two sequential algorithms. A pipelined parallelization of the sequential algorithm is derived: showing details of load balancing in homogeneous or heterogeneous networks; communications in message passing; shared memory architectures; and scalability. Finally, some experimental results are depicted.

2 OSIC Decoding Procedure

In a basic approach, it is necessary to solve the typical perturbed system $\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{v}$: where the known full rank matrix $\mathbf{H} = (\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_n) \in \mathbb{C}^{m \times n}$ represents the channel matrix; $\mathbf{y}$ is the observation vector; $\mathbf{x}$ is a vector whose components belong to a discrete symbol set; and $\mathbf{v}$ is process noise. The use of MMSE (Minimum Mean Square Error) estimation yields:

$$\hat{\mathbf{x}} = \left\lfloor \begin{pmatrix} \mathbf{H} \\ \sqrt{\alpha}\mathbf{I}_n \end{pmatrix}^{\dagger} \begin{pmatrix} \mathbf{y} \\ \mathbf{0} \end{pmatrix} \right\rceil = \lfloor \mathbf{H}_{\alpha}^{\dagger}\mathbf{y} \rceil \tag{2.1}$$

where $\lfloor \cdot \rceil$ denotes the mapping of the result in the symbol set, $\mathbf{H}_\alpha^\dagger$ (whose rows are named the *nulling vectors*) denotes the first m columns of the pseudoinverse of the *augmented* channel matrix $(\mathbf{H}^*, \sqrt{\alpha}\mathbf{I}_n)^*$, α^{-1} denotes a signal-to-noise ratio, and the asterisk superscript $(\cdot)^*$ denotes the complex conjugate. In OSIC, the signal components x_i, $i = 1, \ldots, n$, are decoded from the *strongest* (with the highest signal-to-noise ratio) to the *weakest*, cancelling out the contribution of the decoded signal component into the received signal and then repeating the process with the remaining signal components. Let $\mathbf{P}$ be the symmetric positive definite error estimation covariance matrix, $\mathbf{P} = \mathrm{E}\{(\mathbf{x} - \hat{\mathbf{x}})(\mathbf{x} - \hat{\mathbf{x}})^*\} = (\alpha\mathbf{I}_n + \mathbf{H}^*\mathbf{H})^{-1}$ that can be factorized as: $\mathbf{P} = \mathbf{P}^{1/2}\mathbf{P}^{*/2}$. These square roots factors are not unique, but it is advantageous if they have a triangular shape (i.e. Cholesky triangles). The index of the highest signal-to-noise ratio component of $\hat{\mathbf{x}}$ is the index of the lowest diagonal entry of $\mathbf{P}$: or the least Euclidean norm row index of $\mathbf{P}^{1/2}$. Let $(\mathbf{H}^*, \sqrt{\alpha}\mathbf{I}_n)^* = \mathbf{QL}$ be the QL factorization of the augmented channel matrix, where $\mathbf{L} \in \mathbb{C}^{n \times n}$ is a lower triangular matrix and the columns of $\mathbf{Q} \in \mathbb{C}^{(m+n) \times n}$ are orthogonal. Let us define $\mathbf{Q} = (\mathbf{Q}_\alpha^*, \mathbf{Q}_\beta^*)^*$, where $\mathbf{Q}_\alpha = (\mathbf{q}_{\alpha,1}, \mathbf{q}_{\alpha,2}, \ldots \mathbf{q}_{\alpha,n})$ are the first m rows of $\mathbf{Q}$. It is easy to verify that $\mathbf{L}^{-1} = \mathbf{P}^{1/2}$, and

$$\mathbf{H}_\alpha^\dagger = \mathbf{P}^{1/2}\mathbf{Q}_\alpha^*. \tag{2.2}$$

Let us suppose that $\mathbf{P}^{1/2}$ is a lower triangular matrix with $p_i^{1/2}, i = 1, \ldots, n$, as their diagonal entries and, for simplicity, with their rows sorted in increasing Euclidean norm order (otherwise, we obtain this using a permutation matrix $\mathbf{\Pi}$ and a unitary transformation $\mathbf{\Sigma}$ in $\mathbf{\Pi}\mathbf{H}_\alpha^\dagger = \mathbf{\Pi}\mathbf{P}^{1/2}\mathbf{\Sigma}\mathbf{\Sigma}^*\mathbf{Q}_\alpha^*$, while preserving the lower triangular structure in $\mathbf{\Pi}\mathbf{P}^{1/2}\mathbf{\Sigma}$). Using (2.2), the first *nulling vector* is $\mathbf{H}_{\alpha,1}^\dagger = p_1^{1/2}\mathbf{q}_{\alpha,1}^*$, and $\hat{x}_1 = \left\lfloor \mathbf{H}_{\alpha,1}^\dagger \mathbf{y} \right\rceil$. Now, in order to obtain $\hat{x}_2$ the process must be repeated with the *deflated* channel matrix $\mathbf{H}' = (\mathbf{h}_2, \mathbf{h}_3, \ldots, \mathbf{h}_n)$ and cancelling out the contribution of $\hat{x}_1$ in the received signal $\mathbf{y}' = \mathbf{y} - \hat{x}_1\mathbf{h}_1$. So, $\mathbf{P}'^{1/2}$ must be recomputed (to obtain the index of the least Euclidean norm of $\mathbf{P}'^{1/2}$, - again let us suppose that it is the first) and the QL factorization of the augmented *deflated* channel matrix. Hence, $\mathbf{H}_{\alpha,1}'^\dagger = p_1'^{1/2}\mathbf{q}_{\alpha,1}'^*$, and $\hat{x}_2 = \left\lfloor \mathbf{H}_{\alpha,1}'^\dagger \mathbf{y}' \right\rceil$.

With the previous assumptions, $\mathbf{P}'^{1/2}$ can be obtained directly[2] from the last $n-1$ rows and columns of $\mathbf{P}^{1/2}$, and $\mathbf{Q}_\alpha'$ from the last $n-1$ columns of $\mathbf{Q}_\alpha$, saving an order of magnitude computational cost. Therefore, only the $\mathbf{P}^{1/2}$ (matrix diagonal elements) and the $\mathbf{Q}_\alpha$ matrix are necessary to solve the OSIC problem. $\mathbf{P}^{1/2}$ and $\mathbf{Q}_\alpha$ or $\mathbf{H}_\alpha^\dagger$ can be computed solving a Recursive Least Squares (RLS) problem in a special way[2]. A subtle improvement in the execution time can be achieved[4] by obtaining the nulling vectors with $\mathbf{P}^{1/2}$ and $\mathbf{H}$. In both cases, the $\mathbf{P}^{1/2}$ matrix is needed. The algorithms will be developed for the ideas reported in Hassibi's paper[2]; because the results are extrapolable to the implementation proposed in Hufei Zhu's report[4]. To simplify the description of the algorithm, the details of the permutations and transformations due to the signal-to-noise ratio order will be omitted.

2.1 The Square Root Kalman Filter for OSIC

From (2.2), $\mathbf{Q}_\alpha = \mathbf{H}_\alpha^{\dagger *}\mathbf{P}^{-*/2}$. This matrix is propagated along the iterations of the square root Kalman Filter, which was initially devised to solve a Recursive Least Squares (RLS)

problem[2]. Below, a block version of the algorithm for OSIC[2] called SRKF-OSIC is reproduced.

Input: $\mathbf{H} = \left(\mathbf{H}_0^*, \mathbf{H}_1^*, \ldots, \mathbf{H}_{m/q-1}^*\right)^*$, $\mathbf{P}_{(0)}^{1/2} = \frac{1}{\sqrt{\alpha}}\mathbf{I}_n$ and $\mathbf{Q}_{\alpha,(0)} = \mathbf{0}$

Output: $\mathbf{Q}_\alpha = \mathbf{Q}_{\alpha,(m/q)}$, $\mathbf{P}^{1/2} = \mathbf{P}_{(m/q)}^{1/2}$

for $i = 0, \ldots, m/q - 1$ **do**

Calculate $\mathbf{\Theta}_{(i)}$ and applied in such a way that:

$$\mathbf{E}_{(i)}\mathbf{\Theta}_{(i)} = \begin{pmatrix} \mathbf{I}_q & \mathbf{H}_i\mathbf{P}_{(i)}^{1/2} \\ \mathbf{0} & \mathbf{P}_{(i)}^{1/2} \\ -\mathbf{\Gamma}_{(i+1)} & \mathbf{Q}_{\alpha,(i)} \end{pmatrix} \mathbf{\Theta}_{(i)} = \begin{pmatrix} \mathbf{R}_{e,(i)}^{1/2} & \mathbf{0} \\ \overline{\mathbf{K}}_{p,(i)} & \mathbf{P}_{(i+1)}^{1/2} \\ \mathbf{Z} & \mathbf{Q}_{\alpha,(i+1)} \end{pmatrix} = \mathbf{F}_{(i)}$$

end for

---

where $\mathbf{Z} = -\left(\mathbf{\Gamma}_{(i+1)}^* - \mathbf{H}_i\mathbf{H}_{\alpha,(i+1)}^\dagger\right)^* \mathbf{R}_{e,(i)}^{-*/2}$, q is the number of consecutive rows of $\mathbf{H}$ processed in a block (block size), so $\mathbf{H}_i \in \mathbb{C}^{q\times n}$. The iteration index subscript enclosed between parenthesis denotes that the variable is updated iteratively: $\mathbf{Q}_\alpha$ and $\mathbf{P}^{1/2}$ are the values $\mathbf{Q}_{\alpha,(i+1)}$ and $\mathbf{P}_{(i+1)}^{1/2}$ in the last iteration $i = m/q - 1$. $\mathbf{\Gamma}_{(i+1)} = \left(\mathbf{0}_{iq\times q}^T, \mathbf{I}_q, \mathbf{0}_{(m-q(i+1))\times q}^T\right)^T \in \mathbb{R}^{m\times q}$. $\mathbf{R}_{e,i}$ and $\overline{\mathbf{K}}_{p,(i)}$ are variables of the Kalman Filter whose meaning can be found in Sayed's paper[5], and $\mathbf{H}_{\alpha,(i+1)}^\dagger$ appears implicitly in $\mathbf{Z}$. The cost of one iteration is a matrix multiplication $\mathbf{H}_i\mathbf{P}_{(i)}^{1/2}$ and the application of a sequence of Givens rotations $\mathbf{\Theta}_{(i)}$, exploiting and preserving the triangular structure of $\mathbf{P}_{(i)}^{1/2}$ along the iterations. If we use a QR-like factorization algorithm we do not exploit the submatrix structure and the number of computations increase unnecessarily. Let $\mathrm{w}_{\mathrm{TRMM}}(q,n)$ denote the cost of the $\mathbf{H}_i\mathbf{P}_{(i)}^{1/2}$ matrix multiplication (qn^2 flops[6]), where q and n are the dimensions of the result, and $\mathrm{w}_{\mathrm{ROT}}(z)$, the cost of applying a Givens rotation to a pair of vectors of z components ($6z$ flops[6]). The cost can be approximated as:

$$\sum_{i=0}^{m/q-1}\left\{\mathrm{w}_{\mathrm{TRMM}}(q,n) + \sum_{c=1}^{n}\sum_{r=1}^{q}\left[\mathrm{w}_{\mathrm{ROT}}(q-r+1) + \mathrm{w}_{\mathrm{ROT}}(n-c+1+[i+1]q)\right]\right\} =$$

$$= \sum_{i=1}^{m/q}\mathrm{w}_{\mathrm{TRMM}}(q,n) + q\sum_{i=1}^{m/q}\sum_{c=1}^{n}\mathrm{w}_{\mathrm{ROT}}(c+iq) + n\sum_{i=1}^{m/q}\sum_{r=1}^{q}\mathrm{w}_{\mathrm{ROT}}(r) \approx 4n^2m + 3nm^2 \quad (2.3)$$

2.2 The Square Root Information Filter for OSIC

The *square root information filter* algorithm[5] variation to solve this problem, called SRIF-OSIC, is shown below. Let $\mathbf{V}_{(i)}^{(a)}$ and $\mathbf{W}_{(i)}^{(a)}$ be:

$$\mathbf{V}_{(i)}^{(a)} = \begin{pmatrix} \mathbf{P}_{(i)}^{-*/2} & \mathbf{H}_i^* \\ \mathbf{P}_{(i)}^{1/2} & \mathbf{0} \end{pmatrix}, \qquad \mathbf{W}_{(i)}^{(a)} = \begin{pmatrix} \mathbf{P}_{(i+1)}^{-*/2} & \mathbf{0} \\ \mathbf{P}_{(i+1)}^{1/2} & -\overline{\mathbf{K}}_{p,(i)} \end{pmatrix}$$

If $\mathbf{\Theta}_{(i)}$ is a unitary matrix such that $\mathbf{V}_{(i)}^{(a)}\mathbf{\Theta}_{(i)} = \mathbf{W}_{(i)}^{(a)}$, then $\mathbf{V}_{(i)}^{(a)}\mathbf{V}_{(i)}^{(a)*} = \mathbf{W}_{(i)}^{(a)}\mathbf{W}_{(i)}^{(a)*}$. Let the following augmented matrices be:

$$\mathbf{V}_{(i)}^{(b)} = \begin{pmatrix} \mathbf{P}_{(i)}^{-*/2} & \mathbf{H}_i^* \\ \mathbf{P}_{(i)}^{1/2} & \mathbf{0} \\ \mathbf{A} & \mathbf{B} \end{pmatrix}, \qquad \mathbf{W}_{(i)}^{(b)} = \begin{pmatrix} \mathbf{P}_{(i+1)}^{-*/2} & \mathbf{0} \\ \mathbf{P}_{(i+1)}^{1/2} & -\overline{\mathbf{K}}_{p,(i)} \\ \mathbf{L} & \mathbf{M} \end{pmatrix};$$

To propagate $\mathbf{Q}_{\alpha,(i)}$ along the iterations, let us force $\mathbf{A} = \mathbf{Q}_{\alpha,(i)}$ and $\mathbf{L} = \mathbf{Q}_{\alpha,(i+1)}$, and evaluate $\mathbf{V}_{(i)}^{(b)}\mathbf{V}_{(i)}^{(b)*} = \mathbf{W}_{(i)}^{(b)}\mathbf{W}_{(i)}^{(b)*}$. Hence, a solution can be obtained for $\mathbf{B} = \mathbf{\Gamma}_{(i+1)}$ and $\mathbf{M} = (\mathbf{\Gamma}_{(i+1)} - \mathbf{H}_{\alpha,(i)}^{\dagger *}\mathbf{H}_i^*)\mathbf{R}_{e,(i)}^{-*/2}$. The necessary ordering information can be obtained from $\mathbf{P}^{-*/2}$ without its (total) inversion, so avoiding the propagation of the second row of matrices in the algorithm. Accordingly, they will be deleted from $\mathbf{V}_{(i)}^{(b)}$ and $\mathbf{W}_{(i)}^{(b)}$ and the new matrices as will be denoted as $\mathbf{V}_{(i)}$ and $\mathbf{W}_{(i)}$.

Input: $\mathbf{H}^* = (\mathbf{H}_0^*, \mathbf{H}_1^*, \ldots, \mathbf{H}_{m/q-1}^*)$, $\mathbf{P}_{(0)}^{-1/2} = \sqrt{\alpha}\mathbf{I}$, $\mathbf{Q}_{\alpha,(0)} = \mathbf{0}$
Output: $\mathbf{Q}_\alpha = \mathbf{Q}_{\alpha,(m/q)}$, $\mathbf{P}^{-*/2} = \mathbf{P}_{(m/q)}^{-*/2}$
for $i = 0, \ldots, m/q - 1$ **do**
Compute $\mathbf{\Theta}_{(i)}$ in such a way that:

$$\mathbf{V}_{(i)}\mathbf{\Theta}_{(i)} = \begin{pmatrix} \mathbf{P}_{(i)}^{-*/2} & \mathbf{H}_i^* \\ \mathbf{Q}_{\alpha,(i)} & \mathbf{\Gamma}_{(i+1)} \end{pmatrix}\mathbf{\Theta}_{(i)} = \begin{pmatrix} \mathbf{P}_{(i+1)}^{-*/2} & \mathbf{0} \\ \mathbf{Q}_{\alpha,(i+1)} & \mathbf{M} \end{pmatrix} = \mathbf{W}_{(i)}$$

end for

Zeroes must be placed in the positions of the submatrix $\mathbf{H}_i^*$ in $\mathbf{V}_{(i)}$. This can be achieved by using Householder transformation applications or Givens rotation applications, or both, and right applied to $\mathbf{V}_{(i)}$. Let us suppose that Givens rotations are used, then for every row $r = 1, \ldots, n$ of $\mathbf{H}_i^*$, q Givens rotations need to be applied to a pair of vectors of $(n - r + 1) + [i + 1]q$ components, so the cost of the i^{th} iteration and the total iterations is:

$$W_{\text{sec},i} = \sum_{r=1}^{n} q\mathrm{w}_{\text{ROT}}((n - r + 1) + [i + 1]q) \; = \; 3qn^2 + 6q^2n[i + 1] + 3qn \quad (2.4)$$

$$W_{\text{sec}} = \sum_{i=0}^{m/q-1} W_{\text{sec},i} \; \approx \; 3n^2m + 3nm^2 \quad (2.5)$$

flops respectively. As a result, this version can be about 16% faster than the square root Kalman Filter based on (2.3) when $n \approx m$, because neither matrix multiplication, nor rotation applications, are necessary. This speedup could be even greater if Householder transformations are used (the higher the value of q, the greater the speedup, asymptotically to 25% when $n \approx m$).

3 Parallel Algorithm

For reasons of clarity, let us suppose as an example that we have $p = 2$ processors, P_0 and P_1. A matrix enclosed within square brackets with a processor subscript will denote

that part of the matrix belonging to such a processor. If it is enclosed within parenthesis, it denotes that the entire matrix is in such a processor. Let $\mathbf{C}_{(i)}$, $\mathbf{D}_{(i)}$ and $\mathbf{V}_{(i)}$ be:

$$\mathbf{C}_{(i)} = \begin{pmatrix} \mathbf{P}_{(i)}^{-*/2} \\ \mathbf{Q}_{\alpha,(i)} \end{pmatrix}, \ \mathbf{D}_{(i)} = \begin{pmatrix} \mathbf{H}_i^* \\ \mathbf{\Gamma}_{(i+1)} \end{pmatrix}, \ \mathbf{V}_{(i)} = \left(\mathbf{C}_{(i)}, \mathbf{D}_{(i)} \right)$$

The n columns of $\mathbf{C}_{(i)}$ will be distributed among the processors (n_0 columns belong to P_0 and n_1 columns to P_1, with $n_0 + n_1 = n$) and this assignment will not change during the parallel algorithm execution (it could change in an adaptive load balance algorithm version). $\mathbf{D}_{(i)}$ will be manipulated in a pipelined way by all the processors, so the subscript will change accordingly (it will be initially in P_0). We will divide $\mathbf{H}_i^*$ in $\left(\mathbf{D}_{(i)}\right)_{P_j}$ in p groups of n_j consecutive rows denoted by a left superscript. The initial data partition will be given by:

$$\mathbf{V}_{(i)} = \left(\begin{bmatrix} \mathbf{P}_{(i)}^{-*/2} \\ \mathbf{Q}_{\alpha,(i)} \end{bmatrix}_{P_0} \begin{bmatrix} \mathbf{P}_{(i)}^{-*/2} \\ \mathbf{Q}_{\alpha,(i)} \end{bmatrix}_{P_1} \begin{pmatrix} {}^{n_0}[\mathbf{H}_i^*] \\ {}^{n_1}[\mathbf{H}_i^*] \\ \mathbf{\Gamma}_{(i+1)} \end{pmatrix}_{P_0} \right) = \left(\left[\mathbf{C}_{(i)}\right]_{P_0} \left[\mathbf{C}_{(i)}\right]_{P_1} \left(\mathbf{D}_{(i)}\right)_{P_0} \right)$$

3.1 Processor Tasks

Let us suppose that P_0 gets zeroes in the n_0 rows of ${}^{n_0}[\mathbf{H}_i^*]$ by applying a sequence of unitary transformations $\mathbf{\Theta}_{(i),P_0}$ (the apostrophe $(\cdot)'$ will denote the updating of a matrix):

$$\mathbf{V}'_{(i)} = \mathbf{V}_{(i)}\mathbf{\Theta}_{(i),P_0} = \left(\begin{bmatrix} \mathbf{P}_{(i+1)}^{-*/2} \\ \mathbf{Q}_{\alpha,(i+1)} \end{bmatrix}_{P_0} \begin{bmatrix} \mathbf{P}_{(i)}^{-*/2} \\ \mathbf{Q}_{\alpha,(i)} \end{bmatrix}_{P_1} \begin{pmatrix} {}^{n_0}[\mathbf{0}] \\ {}^{n_1}[\mathbf{H}_i^*]' \\ \mathbf{\Gamma}'_{(i+1)} \end{pmatrix}_{P_0} \right)$$

It can be observed that the data not belonging to P_0 are not involved in the computations. $[\mathbf{P}_{(i)}^{-*/2}]_{P_0}$ is converted in $[\mathbf{P}_{(i+1)}^{-*/2}]_{P_0}$ and only the first $(i+1)q$ rows of the matrices $\mathbf{\Gamma}_{(i+1)}$ and $[\mathbf{Q}_{\alpha,(i)}]_{P_0}$ are updated due to the structure of $\mathbf{\Gamma}_{(i+1)}$ and the zero initial value of $\mathbf{Q}_{\alpha,(0)}$. It is also important to note that $\left[\mathbf{C}_{(i+1)}\right]_{P_0}$ (the first n_0 columns of the result $\mathbf{V}'_{(i)}$) are the first n_0 columns of the matrix $\mathbf{V}_{(i+1)}$. This is useful to obtain a pipelined behaviour in the processing work with minimum data movement from one iteration to the next. Now, if P_0 transfers ${}^{n_1}[\mathbf{H}_i^*]'$ and the nonzero part of $\mathbf{\Gamma}'_{(i+1)}$ to P_1, then P_1 can obtain zeroes in the n_1 rows of ${}^{n_1}[\mathbf{H}_i^*]'$ with the application of the unitary transformation sequence $\mathbf{\Theta}_{(i),P_1}$. Simultaneously, P_0 could work with $\left[\mathbf{C}_{(i+1)}\right]_{P_0}$ provided that new input data is available (pipelined behaviour):

$$\mathbf{V}''_{(i)} = \mathbf{V}'_{(i)}\mathbf{\Theta}_{(i),P_1} = \left(\begin{bmatrix} \mathbf{P}_{(i+1)}^{-*/2} \\ \mathbf{Q}_{\alpha,(i+1)} \end{bmatrix}_{P_0} \begin{bmatrix} \mathbf{P}_{(i+1)}^{-*/2} \\ \mathbf{Q}_{\alpha,(i+1)} \end{bmatrix}_{P_1} \begin{pmatrix} {}^{n_0}[\mathbf{0}] \\ {}^{n_1}[\mathbf{0}] \\ \mathbf{M} \end{pmatrix}_{P_1} \right) = \mathbf{W}_{(i)}$$

Then, the final result is obtained in a pipelined way by means of $\mathbf{V}_{(i)}\mathbf{\Theta}_{(i)} = \mathbf{V}_{(i)}\mathbf{\Theta}_{(i),P_0}\mathbf{\Theta}_{(i),P_1} = \mathbf{W}_{(i)}$. Let n_j be the number of columns of $\mathbf{C}_{(i)}$ assigned to P_j and r_{0_j}, the index of the first of them. It can be verified that the arithmetic overhead due to

parallelization is zero. The arithmetic cost in the P_j processor for the i^{th} iteration is:

$$W_{P_j,i} = \sum_{r=1}^{n_j} \text{w}_{\text{ROT}}(n - r_{0_j} + 2 - r + [i+1]q) = [6q(n - r_{0_j} + 2 + [i+1]q) - 3q]n_j - 3qn_j^2 \tag{3.1}$$

3.2 Load Balance, Heterogeneous Systems and Maximum Speedup

If the same number of columns of $\mathbf{C}_{(i)}$ are assigned to each processor, the parallel algorithm is clearly unbalanced due to to the lower triangular structure of $\mathbf{P}_{(i)}^{-*/2}$ in $\mathbf{C}_{(i)}$. The workload should be balanced at iteration level in order to minimize processor waiting time. The optimum number of columns n_j assigned to the processor P_j can be calculated solving the following second order equation:

$$W_{P_j,i} t_{w_j} = W_{\text{seq},i} t_{w_{\text{seq}}} / S_{\max}, \; \forall\, 0 \leq j \leq p-1 \tag{3.2}$$

beginning with n_0 and $r_{0_0} = 1$, up to n_{p-1}. $S_{\max}$ is the maximum speedup attainable in the heterogeneous or homogeneous system (in this case $S_{\max} = p$) and t_{w_j} and $t_{w_{\text{seq}}}$ are the time per flop in the P_j processor and the sequential processor, respectively. This result depends on the iteration index i, so if a static load balance scheme is desired then the load can balanced for the worst case, $i = m/q - 1$. An alternative way to balance the workload is to symmetrically assign columns to a processor, therefore the processor workload now depends linearly on the number of columns assigned to it, although the number of transfers nearly doubles. The maximum speedup in the heterogeneous network depends on the time per flop t_{w_j} of each processor. Let us define s_j as the normalized relative speed of the processor P_j (dimensionless): $s_j = \left(\sum_{r=0}^{p-1} \frac{tw_j}{t_{w_r}} \right)^{-1}$. It can be verified that $\sum_{j=0}^{p-1} s_j = 1$, and $t_{w_j} s_j = t_{w_k} s_k$, and if P_j is u times faster than P_k, then $s_j = u s_k$. Let us suppose that the sequential algorithm is run on the fastest processor P_f ($t_{w_{\text{seq}}} = t_{w_f}$). The maximum speedup can be obtained from (3.2) when a perfect load balance is obtained and there is no parallel arithmetic overhead. Hence, $S_{\max}$ can be obtained from (3.2) with $j = f$:

$$S_{\max} = \frac{W_{\text{seq},i} t_{w_f}}{W_{P_f,i} t_{w_f}} = \frac{\sum_{j=0}^{p-1} W_{P_j,i}}{W_{P_f,i}} = \frac{\sum_{j=0}^{p-1} W_{P_f,i} \frac{t_{w_f}}{t_{w_j}}}{W_{P_f,i}} = \frac{1}{s_f}$$

3.3 Communications, Shared Memory Implementation and Scalability

The parallel algorithm organization requires that processor P_j transfers the nonzero part of the updated $(\mathbf{D}_{(i)})_{P_j}$ matrix —$(i+1)q + \sum_{k=j+1}^{p-1} q n_k$ elements to P_{j+1}, $0 \leq j < p-1$. Let us suppose that the time for transferring this information from P_j to P_{j+1} for the i^{th} iteration can be modeled as a linear function of the number of elements to transfer. The worst case takes place when there is no computation and communication overlap, and all the transfers must be handled serially. For this worst case, the communication time is:

$$T_C = \sum_{i=0}^{m/q-1} \sum_{j=0}^{p-1} T_{C,P_j,i} = \mathbf{\Theta}\left(mnp\right) + \mathbf{\Theta}\left(mp^2\right) + \mathbf{\Theta}\left(\frac{pm^2}{q}\right)$$

If we use a shared memory multiprocessor system, and each process can be mapped on a processor. In this case, data transfer consists in copying this data from the memory space of P_j to the memory space of P_{j+1} and controlling access to it with a typical producer-consumer strategy. The copying of data to the next processor memory space can be a time consuming operation. We avoid this copying time by using a shared circular buffer whose elements are arrays of the same size as the data to transfer, so the symbolic copy can consist in a pointer to an element buffer update. These ideas can be extrapolated to shared and distributed memory parallel systems with minimum change in the code.

The scalability of the parallel system based on the isoefficiency function[7] can be evaluated by comparing the sequential time (2.5) with the total parallel overhead time. In our case, the only theoretical source of overhead is the communication time, so the total parallel overhead time is pT_C. If $W_{\text{sec}} t_{w_{\text{sec}}} = pT_C$, the workload must be increased as in the worst case of $n = \mathbf{\Theta}(p^2)$, $nm = \mathbf{\Theta}(p^3)$ or $\frac{n^2 q}{m} = \mathbf{\Theta}(p^2)$. If the transfers can be handled simultaneously, then $m, n = \mathbf{\Theta}(p)$.

3.4 Experimental Results

The tests have been run in a ccNUMA architecture multiprocessor running a 64 bit Linux operating system with up to 16 processors available to one user. Each processor is a 1.3 GHz Itanium 2. The programs have been coded in Fortran using a proprietary MPI communications library. Figure 1 shows the sequential execution time ratio of the SRKF-OSIC and SRIF-OSIC (Givens) versions for several values of q, where we can observe that the proposed SRIF-OSIC algorithm is faster than the SRKF-OSIC. We observed in the SRIF-OSIC algorithm sequential execution time an optimum value for q that gave a minimum execution time. There is no algorithmic reason for obtaining such minimum execution times, (2.5), so this behaviour is caused by the processor architecture. Figure 2 shows the efficiency of the proposed parallel algorithm in a message passing architecture for $q = 20$ and $m = 6000$, depicting a good efficiency in the results for $p = 2, 4, 8$ and 16 processors.

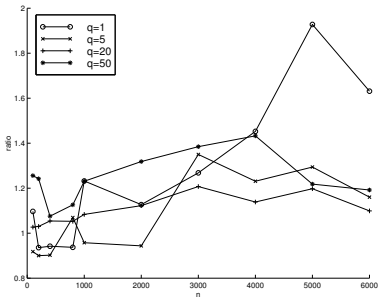


Figure 1. Execution time ratio of SRKF-OSIC and SRIF-OSIC (Givens) versions for $m = 6000$.

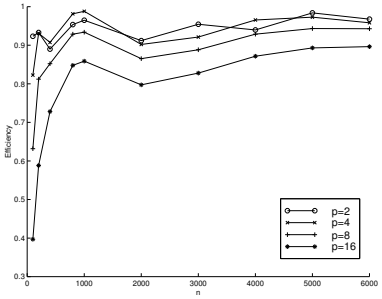


Figure 2. Efficiency of the SRIF-OSIC (Givens) parallel algorithm for $q = 20$ and $m = 6000$.

4 Conclusions

A novel algorithm is proposed to solve the OSIC decoding problem based on the square root information filter algorithm — and it offers a better performance than the reference based on the square root Kalman Filter algorithm. The improvement lies in the fact that a matrix multiplication, and the application of some rotations, are unnecessary in the new algorithm. A parallelization with a high degree of efficiency and scalability was obtained with a simple load balancing criterion for heterogeneous networks; and a simple extrapolation to shared memory, or distributed and shared memory systems.

Acknowledgements.

This work has been supported by the Spanish Government MEC and FEDER under grant TIC 2003-08238-C02.

References

1. G. J. Foschini, *Layered space-time architecture for wireless communications in a fading environment when using multiple antennas*, Bell Labs Tech. J., **1**, 41–59, (1996).
2. B. Hassibi, *An efficient square-root algorithm for BLAST*, IEEE International Conference on Acoustics, Speech and Signal Processing, **2**, II-737–II740, (2000).
3. Y.-S. Choi, P. J. Voltz and F. A. Cassara, *On channel estimation and detection for multicarrier signals in fast and selective Rayleigh fading channels*, IEEE Transactions on Communications, **49**, , (2001).
4. H. Zhu, Z. Lei and F. P. S. Chin, *An improved square-root algorithm for BLAST*, IEEE Signal Processing Letters, **11**, , (2004).
5. A. H. Sayed and Th. Kailath, *A state-space approach to adaptive RLS filtering*, IEEE Signal Processing Magazine, **11**, 18–60, (1994).
6. G. H. Golub and C. F. Van Loan, *Matrix Computations*, (Johns Hopkins University Press, Baltimore, 1996).
7. V. Kumar, A. Gram, A. Gupta and G. Karypis, *An Introduction to Parallel Computing: Design and Analysis of Algorithms*, (Addison-Wesley, 2003).

OpenMP Implementation of the Householder Reduction for Large Complex Hermitian Eigenvalue Problems

Andreas Honecker[1] **and Josef Schüle**[2]

[1] Institut für Theoretische Physik, Georg-August-Universität Göttingen
37077 Göttingen, Germany
E-mail: ahoneck@uni-goettingen.de

[2] Gauss-IT-Zentrum, Technische Universität Braunschweig, 38106 Braunschweig, Germany
E-mail: j.schuele@tu-bs.de

The computation of the complete spectrum of a complex Hermitian matrix typically proceeds through a Householder step. If only eigenvalues are needed, this Householder step needs almost the complete CPU time. Here we report our own parallel implementation of this Householder step using different variants of C and OpenMP. As far as we are aware, this is the only existing parallel implementation of the Householder reduction for complex Hermitian matrices which supports packed storage mode. As an additional feature we have implemented checkpoints which allow us to go to dimensions beyond 100 000. We perform runtime measurements and show firstly that even in serial mode the performance of our code is comparable to commercial libraries and that secondly we can obtain good parallel speedup.

1 Introduction

There are problems which require the complete diagonalization of big complex Hermitian matrices. For example, in theoretical solid-state physics, the computation of thermodynamic properties of certain quantum many-body problems boils down to the computation of the complete spectrum of eigenvalues of the Hermitian matrix representing the Hamiltonian[1,2,3] (eigenvectors are not needed for this specific purpose, but would be required for the computation of other properties). In this application, the original problem is split into smaller pieces using symmetries[4,5]. In particular, the aforementioned problems are lattice problems and we can use Fourier transformation to exploit translational symmetry. This yields a range of matrix eigensystems where the matrices of largest dimension turn out to be complex. Thus, the diagonalization of Hermitian matrices constitutes the bottleneck of such computations. Although the individual matrices are originally sparse, they are treated as dense matrices since all eigenvalues are needed, and the standard algorithms for solving this problem[6,7,8] proceed through a stage where the matrix becomes densely populated.

Libraries like LAPACK[9] provide diagonalization routines such as `zheev` for dense complex Hermitian matrices. Such routines typically perform first a Householder reduction to tridiagonal form and then use QR-/QL-iteration to solve the tridiagonal problem[6,10]. The applications mentioned above[1,2,3] do not need eigenvectors. Therefore, almost all CPU time and memory is spent in the Householder step. Furthermore, if eigenvectors are not needed, one can store the matrix in a so-called packed form which exploits symmetry of the matrix and stores only one triangular half. Available library routines work well in low dimension n, but memory requirements grow as n^2 and CPU time grows as n^3 for larger dimensions. More precisely, approximately $8\,n^2$ bytes are needed to store a complex double-precision matrix in packed form and the Householder reduction of such a matrix

requires approximately $16\,n^3/3$ floating-point operations. For example, in dimension $n = 40\,000$ we need 12 GByte to store the complex matrix in packed form, while full storage would already require 24 GByte main memory. Thus, for complex dimensions $n \gtrsim 40\,000$ substantial fractions of the main memory even of current high-performance computers are going to be used. Consequently, a parallelized code should be used to utilize the CPUs attached to this memory and reduce runtime.

There are different parallel diagonalization routines supporting distributed memory and several comparative studies of such libraries exist (see, e.g., Refs. 11, 12, 13). We will not attempt yet another comparative study, but just mention a few relevant features. ScaLAPACK[14] and variants like PESSL[15] do unfortunately not support packed matrices and even require two copies for the computation of eigenvectors. In addition, these libraries may have problems with numerical accuracy[11]. PeIGS supports packed matrices, but it does not provide diagonalization routines for complex matrices[16]; the necessary conversion of a complex Hermitian to a real symmetric eigenvalue problem wastes CPU-time and a factor 2 of main memory. There are further parallel matrix diagonalization routines (see, e.g., Refs. 17, 18, 19, 20), but we do not know any parallel implementation of an eigensolver for complex Hermitian matrices which contends itself with the minimal memory requirements.

On shared memory parallel computers, the diagonalization routines contained e.g. in LAPACK may call a parallel implementation of the basic linear algebra package BLAS. However, this does not yield good parallel performance since parallelization should set in at a higher level. While inspection of the Householder algorithm[6,7,8,21] shows that it is possible to parallelize the middle of three nested loops, we are not aware of any such parallelization for a shared-memory parallel machine.

Here we report our own efforts at a parallel implementation of the Householder reduction in C using OpenMP[22]. In Section 2 we first discuss our parallelization strategy. Then we present performance measurements both of the serial and the parallel code in Section 3. Finally, Section 4 summarizes the current status and discusses future plans.

2 Implementation of the Householder Algorithm

We start from a serial implementation of the standard Householder algorithm[6,7,8] in C99. A complete version of the code is available from Ref. 23. This code has been inspired by the routine `tred2` of Ref. 21, but is has been generalized to complex numbers, thoroughly optimized for modern CPUs, and prepared for parallelization.

Let us discuss our parallelization strategy using a part of the code as an example. The fragment which performs a reduction of the Hermitian matrix `m` in C99 is shown in Listing 31.1.

First, we observe that dependencies inside the loop over `j` can be eliminated by performing the operations on `p` in a separate loop. It is then possible to parallelize the loop over `j`, i.e. the middle of three nested loops, with OpenMP by adding a `#pragma omp parallel for` in front of it. One problem with such a naïve approach is that the length of the innermost loop depends on `j` such that we run the risk that the work is not distributed equally across the different threads. It is possible to ensure load balancing by introducing an auxiliary variable which explicitly runs over the threads. However, a second problem remains. Namely, the innermost loop over `k` should be able to reuse the vector `p` and the row `i` of the matrix `m` from the Level 2 cache of the CPU. For dimensions `dim` around

Listing 31.1. Householder algorithm: Hermitian matrix reduction

```
for (i=dim-1; i>=1; i--) {
  :
    for (j=0; j<i; j++) {
      f = conj(m[i][j]);
      g = p[j] = p[j]-hh*f;
      cptr1 = p; cptr2 = m[j]; cptr3 = m[i];
      for (k=0; k<=j; k++)
        *cptr2++ -= (f * conj(*cptr1++) + g * *cptr3++);
    }
  :
}
```

33 000 the memory requirements for these data start to exceed 1 MByte such that they fail to fit into the Level 2 cache for larger dimensions, leading to a substantial degradation of the performance of such a code for big matrices.

Therefore, we proceed differently. Namely, we split the innermost loop over `k` into sufficiently small chunks and pull the loop over the chunks outside the loop over `j`. A parallel version of the above code fragment then looks as follows:

Listing 31.2. Parallel matrix reduction

```
for (i=dim-1; i>=1; i--) {
  :
    for (j=0; j<i; j++) {
      p[j] -= hh*conj(m[i][j]);
      nchunks = compute_chunks(sizeof(complex double), i-1);
#pragma omp parallel for private(chunk,j,k,f,g,cbegin,cend,
                  cendt,cptr1,cptr2,cptr3,pptr) if(nchunks>1)
      for (chunk=0; chunk<nchunks; chunk++) {
        cbegin = chunks[chunk].begin;
        cendt = chunks[chunk].end;
        pptr = p+cbegin;
        for (j=0; j<i; j++) {
          f = conj(m[i][j]);
          g = p[j];
          cptr1 = pptr;
          cptr2 = m[j]+cbegin;
          cptr3 = m[i]+cbegin;
          cend = (j<cendt)?j:cendt;
          for (k=cbegin; k<=cend; k++)
            *cptr2++ -= (f * conj(*cptr1++) + g * *cptr3++);
        }
      }
    }
  :
}
```

The administrative routine `compute_chunks` has the task of splitting the segment of size `i` into chunks such that two constraints are obeyed. Firstly, each chunk should fit into Level 2 CPU cache. For this reason the size of an individual element is passed as an argument. Secondly, the load is to be balanced knowing that the inner loop is restricted to $k \leq j$. There are two further variants of this routine: `compute_eq_chunks` performs the same task for the situation where the inner loop also runs to `i-1` and

	2.4 GHz Intel Core2		1.9 GHz IBM Power5		1.6 GHz Itanium2	
variant	gcc 4.1.1	icc 10.0	gcc 4.0.2	xlc 8.0	gcc 4.1.0	icc 9.1
C99	5.18	4.60	7.05	17.03	20.42	8.04*
C++	5.11	44.23	7.05	10.18	26.70	16.87
plain C	5.48	5.46	7.94	6.43*	28.74	14.68
SSE3	4.39*	4.41				

Table 1. Runtimes in seconds for the serial computation of eigenvalues for a "small" complex matrix in dimension $n = 1184$. Different rows are for different variants of our code, different columns for different CPUs and compilers (GNU gcc, Intel icc and IBM xlc). Note that runtimes include overhead, e.g., for initialization and diagonalization of the resulting tridiagonal matrix.

`compute_min_chunks` just computes chunks such that they fit into Level 2 CPU cache. With these administrative routines we can then follow the same strategy as above and parallelize a total of three loops which are needed for the reduction to tridiagonal form and two additional loops which compute the transformation matrix which is needed if eigenvectors are also desired.

The following additional features of our implementation may be worthwhile mentioning. Firstly, at the beginning of each outermost loop over `i` it is possible to checkpoint the computation by writing the updated part of the matrix `m` and some additional data onto hard disc. We have implemented such checkpoints with a second set of administrative routines. Secondly, the matrix `m` is implemented as a vector of pointers to its rows. On the one hand, this renders it unnecessary to store the complete matrix consecutively in memory and also allows convenient access to the matrix elements in a lower triangular packed storage mode. On the other hand, in combination with the checkpoint it becomes possible to return memory to the system for those parts of the matrix where the computation is completed, allowing part of the computation to run with reduced memory requirements.

3 Performance

We start with tests of the serial performance of our code. For this purpose we use a "small" complex Hermitian matrix of dimension $n = 1184$ which can be downloaded from Ref. 23. This matrix needs about 11 MByte if stored in packed form and should therefore be big enough *not* to fit completely into the Level 2 CPU cache.

First, we have compared the public domain Gnu C-compiler[24] with compilers provided by the vendors on different platforms as well as different variants of the implementation of the Householder algorithm, namely the C99 version discussed in Section 2, a C++ version using the `complex<double>` data type from the Standard Template Library, a version with complex numbers hand-coded in plain C, and a version with complex numbers hand-coded in inline-assembler for CPUs supporting SSE instructions. Results are shown in Table 1. One observes that variations of runtimes by a factor 3 on the same CPU are not uncommon depending on the version of our code and more noteworthy on the compiler. On the Intel Core2 we obtained the best performance for a version where complex numbers had been hand-coded with SSE3 assembler instructions, but the C99 variant compiled with Intel icc 10.0 is almost as fast. On the Power5, gcc 3.3.2 gave somewhat better performance for the plain C code, but did not compile the C99 version properly. Evidently,

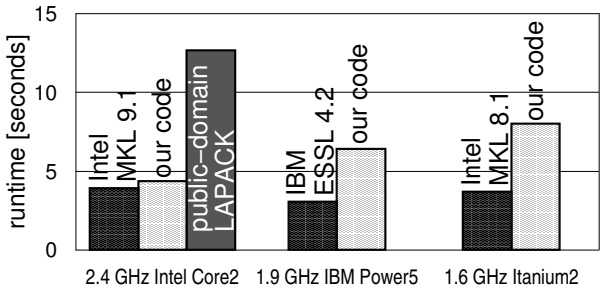


Figure 1. Comparison of runtimes for the serial computation of eigenvalues for a "small" complex matrix in dimension $n = 1184$ between our code and routines `zhpev` provided by different libraries. Note that runtimes include overhead, e.g., for initialization and diagonalization of the resulting tridiagonal matrix.

we are witnessing how complex numbers according to the C99 standard are just being properly implemented in C compilers. C++ is still lagging somewhat behind in performance, but this should also be remedied once the `complex<double>` template defaults to a properly implemented internal data type of the compiler. Overall, we hope that on each platform a C99 compiler will be available soon which yields optimal performance and supports OpenMP such that we will be able to focus on the C99 variant for future code developments. For further analysis in this paper we use the combinations marked by a star in Table 1.

Figure 1 presents a comparison of the serial performance of our code with other libraries. In all cases, the corresponding routine is called `zhpev`, although the interface of IBM ESSL differs from Intel Math Kernel Library (MKL)/LAPACK[9]. It is gratifying to see that we do not only outperform public-domain libraries (like a version of LAPACK included with a recent distribution of Mandrake Linux), but that we can also compete with the commercial Math Kernel Library on the Intel Core2. On the high-performance machines, our code is about a factor 2 slower than the commercial libraries. We can only speculate if this could be improved with better compilers (compare Table 1), but we believe that even serial performance is acceptable at the moment.

Now we move on to discuss parallel performance of our code. For this purpose we use a "large" complex Hermitian matrix of dimension $n = 41\,835$ which is also available from Ref. 23. This matrix requires about 13 GByte of main memory in packed storage mode. On the one hand, this problem is substantially bigger than the problem sizes investigated in other eigensolver performance tests[12,13]. On the other hand, this is still at the lower edge of dimensions where it becomes necessary to use a parallel eigensolver for our purposes[a]. On an IBM p575 with 8 CPUs, our code needs about 14.2 hours real time for the computation of all eigenvalues whereof 99.8% are spent on the Householder reduction[b]. This machine is the one with the Power5 CPUs on which we have previously tested single-CPU performance. So, we can can use n^3 scaling of the runtime measured on our small problem. It turns out that the diagonalization of the large problem takes less than 50% longer

[a] CPU time is too precious to carry out systematic testing for bigger production-type problems[2,3].

[b] We also tested IBM parallel ESSL 3.3[15] under the same conditions. The best performance, namely 9.3 hours real time, was obtained with the routine `pzheevx` running in SMP mode whereas variants with MPI communication were slower than our solution.

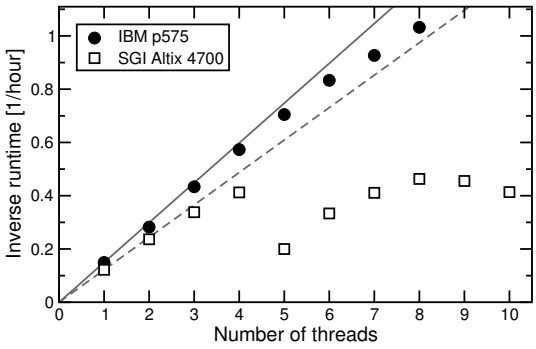


Figure 2. Inverse real runtimes on an IBM p575 and an SGI Altix 4700 for the first 1000 iterations of the Householder transformation for a "large" complex matrix in dimension $n = 41\,835$. Lines indicate inverse runtimes corresponding to perfect scaling of the single-thread case.

than this optimistic estimate, demonstrating good parallelization and scaling with problem size. More details can be seen in Fig. 2 which shows the inverse real runtime for the first 1000 steps of the Householder reduction as a function of the number of threads. On the IBM p575 we indeed observe good scaling with the number of threads. Fig. 2 also contains results for an SGI Altix 4700. The nodes of this machine consist of 4 Itanium2 CPU Cores whose single-thread performance we have discussed previously. Accordingly, here we observe reasonable scaling for up to 4 threads while the drop in performance between 4 and 5 threads can be attributed to the onset of memory access across the network.

To summarize this section, we have shown that serial performance of our code is competitive, that it scales well to big problems, and that good parallel speedups can be obtained.

4 Discussion and Conclusions

We have presented a parallel implementation of the Householder algorithm for packed complex matrices with proper load balancing and CPU-cache optimization using OpenMP[22]. Our implementation is also able to write checkpoints of the computation onto hard disc which can be used to successively reduce main memory requirements even further in later stages of the computation. A preliminary version of this code without checkpoints has been used[2] to compute the full eigenvalue spectrum of double precision complex matrices in dimension $n = 81\,752$. With checkpoints it has been possible to push this further[3] to dimension $n = 121\,968$, and very recently in one case to $n = 162\,243$. The latter diagonalization required 197 GByte of main memory (mainly for packed storage of the complex matrix) and close to 400 GByte hard-disc space for a fail-save checkpoint. This computation was executed in parallel on a node with 32 1.3 GHz Power4 CPUs where we have measured average CPU efficiencies $\gtrsim 90\%$.

The Householder reduction yields a tridiagonal matrix which can be transformed to real form using simple phase factors. Thus, a diagonalization procedure for real symmetric tridiagonal matrices is needed to finish the computation. Currently, we simply call the LAPACK[9] routine `dsterf` for a reliable computation of all eigenvalues of the tridiagonal matrix. This routine follows the traditional approach provided by the QL/QR-

algorithms[6,10,21]. If only eigenvalues are desired, the diagonalization of the tridiagonal matrix requires a negligible amount of CPU time as compared to the Householder reduction such that optimization of performance is unnecessary. If eigenvectors are also needed, basis transformations have to be computed in the QL/QR-algorithm. Inspection of the QL/QR algorithm shows that this can also be parallelized after putting the rotations into a buffer[c]. Indeed, we already have an OpenMP-parallelized implementation of the QL-transformation for the tridiagonal problem. Recent optimization efforts by other groups have focussed on this diagonalization step of the symmetric tridiagonal matrix[17,18,19,20]. Runtime measurements of our not yet optimized QL-transformation show that it requires less CPU time than the Householder step. Faster algorithms for the tridiagonal problem are therefore unnecessary and may even be detrimental for our applications if they go at the expense of reduced numerical accuracy or increased memory requirements[d].

At the moment, the numerical efficiency of our own implementation of the QL-transformation for the tridiagonal problem is still at the level of the routine `tqli` from Ref. 21. As a next step we need to bring this up to the level of LAPACK[10] and implement checkpoints during the diagonalization of the symmetric tridiagonal matrix. It will also be straightforward to derive real variants from our routines although it is not our priority to optimize performance for the real symmetric case. Finally, everything can be canned into a stand-alone package with general-purpose OpenMP-parallelized diagonalization routines which we plan to release into public domain. This package is also scheduled to be integrated in a future release of the ALPS applications suite for strongly correlated electron systems[26,27]. Furthermore, we hope that our code developments will also be useful in other fields such as quantum chemistry.

Acknowledgements

A.H. acknowledges support by the Deutsche Forschungsgemeinschaft through a Heisenberg fellowship (Project HO 2325/4-1). The biggest computations reported here have been made possible by a CPU-time grant at the HLRN Hannover.

References

1. M. E. Zhitomirsky and A. Honecker, *Magnetocaloric effect in one-dimensional antiferromagnets*, J. Stat. Mech.: Theor. Exp., P07012, (2004).
2. F. Heidrich-Meisner, A. Honecker and T. Vekua, *Frustrated ferromagnetic spin-$\frac{1}{2}$ chain in a magnetic field: the phase diagram and thermodynamic properties*, Phys. Rev. B, **74**, 020403(R), (2006).
3. O. Derzhko, A. Honecker and J. Richter, *Low-temperature thermodynamics for a flat-band ferromagnet: rigorous versus numerical results*, preprint arXiv:cond-mat/0703295 (2007).

[c] Such a buffer also helps to optimize CPU cache performance and exists in the corresponding LAPACK[9] routines.
[d] An additional feature of some of these algorithms is the computation of selected eigenvalues only[20]. However, for our purposes (see, e.g., Refs. 1,2,3) the most important eigenvectors are usually the lowest ones and we start from sparse matrices. Therefore we use completely different algorithms like the Lanczos method[5,25] which can handle much bigger problems if we are interested only in a few eigenvalues (and -vectors).

4. H. Q. Lin, *Exact diagonalization of quantum-spin models*, Phys. Rev. B, **42**, 6561–6567, (1990).
5. N. Laflorencie and D. Poilblanc, *Simulations of pure and doped low-dimensional spin-1/2 gapped systems*, Lect. Notes Phys., **645**, 227–252, (2004).
6. B. N. Parlett, *The Symmetric Eigenvalue Problem*, (Prentice-Hall, 1980).
7. A. Jennings and J. J. McKeown, *Matrix Computation*, (John Wiley, Chichester, 1992).
8. G. H. Golub and C. F. Van Loan, *Matrix Computations*, (Johns Hopkins UP, 1996).
9. E. Anderson *et al.*, *LAPACK Users' Guide, Third Edition*, (SIAM, Phil., 1999).
10. A. Greenbaum and J. Dongarra, *Experiments with QR/QL methods for the symmetric tridiagonal eigenproblem*, LAPACK Working Note 17 (1989).
11. B. Lang, *Direct solvers for symmetric eigenvalue problems*, in: Modern Methods and Algorithms of Quantum Chemistry, NIC Series, Vol. **3**, (Jülich, 2000).
12. I. Gutheil and R. Zimmermann, *Performance of software for the full symmetric eigenproblem on CRAY T3E and T90 systems*, FZJ-ZAM-IB-2000-07, (2000).
13. A. G. Sunderland and I. J. Bush, *Parallel Eigensolver Performance*, CLRC Daresbury Laboratory (http://www.cse.clrc.ac.uk/arc/diags.shtml).
14. L. S. Blackford *et al.*, *ScaLAPACK Users' Guide* (SIAM, Philadelphia, 1997).
15. *Parallel ESSL for AIX V3.3, Parallel ESSL for Linux on POWER V3.3, Guide and Reference*, Sixth edition, (IBM, Boulder, 2006).
16. G. Fann, *Parallel eigensolver for dense real symmetric generalized and standard eigensystem problems*, (http://www.emsl.pnl.gov/docs/nwchem/).
17. J. J. M. Cuppen, *A divide and conquer method for the symmetric tridiagonal eigenproblem*, Numerische Mathematik, **36**, 177–195, (1981).
18. F. Tisseur and J. Dongarra, *A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures*, SIAM J. Sci. Comput., **20**, 2223–2236, (1999).
19. I. S. Dhillon, *A New $O(n^2)$ Algorithm for the Tridiagonal Eigenvalue/Eigenvector Problem*, Ph.D. thesis, University of California, Berkeley (1997).
20. I. S. Dhillon and B. N. Parlett, *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Alg. & Appl., **387**, 1–28, (2004).
21. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, (Cambridge University Press, 1992).
22. R. Chandra *et al.*, *Parallel Programming in OpenMP*, (Morgan Kaufmann, 2000).
23. http://www.theorie.physik.uni-goettingen.de/~honecker/householder/
24. http://gcc.gnu.org/
25. J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, Vol. I (Birkhäuser, Boston, 1985).
26. A. F. Albuquerque *et al.*, *The ALPS Project Release 1.3: Open Source Software for Strongly Correlated Systems*, J. Magn. Magn. Mater., **310**, 1187–1193, (2007).
27. http://alps.comp-phys.org/

Multigrid Smoothers on Multicore Architectures

Carlos García, Manuel Prieto, and Fransisco Tirado

Dpto. de Arquitectura de Computadores y Automática
Universidad Complutense
28040 Madrid, Spain
E-mail: {*garsanca, mpmatias, ptirado*}*@dacya.ucm.es*

We have addressed in this paper the implementation of red-black multigrid smoothers on high-end microprocessors. Most of the previous work about this topic has been focused on cache memory issues due to its tremendous impact on performance. In this paper, we have extended these studies taking *Multicore processors (MCP)* into account. With the introduction of *MCP*, new possibilities arise, which makes a revision of the different alternatives highly advisable. A new strategy is proposed which tries to achieve a cooperation between the threads in a *MCP*. Performance results on an *Intel Core*^{TM}*2 Duo* based system reveal that our alternative scheme can compete with and even improve sophisticated schemes based on loop fusion and tiling transformations aimed at improving temporal locality.

1 Introduction

Multigrid methods are regarded as being the *fastest* iterative methods for the solution of the linear systems associated with elliptic partial differential equations, and as amongst the *fastest* methods for other types of integral and partial differential equations[1]. *Fastest* refers to the ability of Multigrid methods to attain the solution in a computational work which is a small multiple of the operation counts associated with discretizing the system. Such efficiency is known as *textbook multigrid efficiency* (TME)[2] and has made multigrid one of the most popular solvers on the niche of large-scale problems, where performance is critical.

Nowadays, however, the number of executed operations is only one of the factors that influences the actual performance of a given method. With the advent of parallel computers and superscalar microprocessors, other factors such as *inherent parallelism* or *data locality* (i.e. the memory access behaviour of the algorithm) have also become relevant. In fact, recent evolution of hardware has exacerbated this trend since:

- The disparity between processor and memory speeds continues to grow despite the integration of large caches.
- Parallelism is becoming the key of performance even on high-end microprocessors, where multiple cores and multiple threads per core are becoming mainstream due to clock frequency and power limitations.

In the multigrid context, these trends have prompted the development of specialized multigrid-like methods[3,4,5,6], and the adoption of new schemes that try to bridge the processor/memory gap by improving locality[7,8,9,10]. Our focus in this paper is the extension of this cache-aware schemes to a *MCP*.

As its name suggests, *MCP* architectures integrate two or more processors (cores) on a chip. Its main goal is to enhance performance and reduce power consumption, allowing the

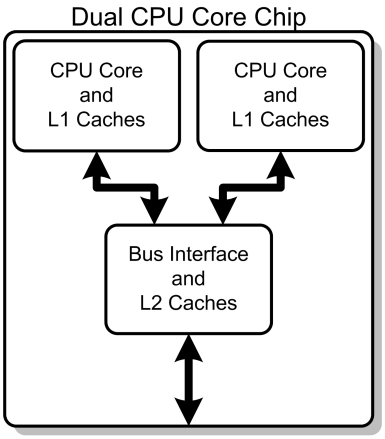


Figure 1. Intel Core^{TM}2 Duo block diagram.

simultaneous processing of multiple tasks in parallel. Technology trends indicate that the number of cores/processors on a chip will continue to grow. AMD has recently announced chips with four cores (*Native Quad technology*) and Intel has began to incorporate the Intel Core^{TM} Extreme quad-core Processor in their servers systems.

These cores can be seen as independent processors. Therefore, one may think that the optimizations targeted for *Symmetric Multiprocessors* (*SMP*) systems are also good candidates for *MCP* architectures. However, as shown in Fig. 1, most MCP architectures integrate a shared cache memory layer, which allows for a faster connection between on-chip cores. This sharing introduces an additional interaction between threads, which may translate into positive (fine-grain sharing among threads) or negative (competition for cache lines) effects. Conventional parallel schemes used in our context for SMP systems, such as block-based data distributions, may promote negative interactions and do not benefit from positive ones. Therefore, optimizations that are appropriate for these conventional machines may be inappropriate or less effective for *MCP*.

Unfortunately, *MCP* potentials are not yet fully exploited in most applications due to the relative underdevelopment of compilers, which despite many improvements still lag far behind. Due to this gap between compiler and processor technology, applications cannot exploit successfully *MCP* hardware unless they are explicitly aware of thread interactions. In this paper, we have revised the implementation of multigrid smoothers in this light. The popularity of multigrid makes this study of great practical interest. In addition, it also provides certain insights about the potential benefits of this relatively new capability and how to take advantage of it, which could ideally helps to develop more efficient compiler schemes.

The organization of this paper is as follows. We begin in Sections 2 by briefly introducing multigrid methods, in Section 3 summarizes the most relevant optimizations in Multigrid Algorithm in our context and Section 4 describes the main characteristics of our target computing platform. Afterwards, in Section 5, we discuss our *MCP*-aware implementation. Performance results are discussed in Section 6. Finally, the paper ends with some conclusions.

2 Multigrid Introduction

The fundamental idea behind Multigrid methods[1] is to capture errors by utilizing multiple length scales (multiple grids). They consist of the following complementary components:

- *Relaxation*. Also called smoother in multigrid lingo, is basically a simple (and inexpensive) iterative method like *Gauß-Seidel*, damped *Jacobi* or block *Jacobi*. It is able to reduce the high-frequency or oscillatory components of the error in relatively few steps.
- *Coarse-Grid Correction*. Smoothers are ineffectual in attenuating low-frequency content of the error, but since the error after relaxation should lack the oscillatory components, it can be well-approximated using a coarser grid. On that grid, errors appear more oscillatory and thus the smoother can be applied effectively. New values are transferred afterwards to the target grid to update the solution.

The *Coarse-Grid Correction* can be applied recursively in different ways, constructing different cycling strategies. One of the most employed cycles correspond to the V-cycle.

3 Related Work

The optimization of multigrid codes has been a popular research topic over the last years. For instance, one of the most outstanding and systematic studies is the *DIME* project (*DIME* stands for *Data Local Iterative Methods For The Efficient Solution of Partial Differential Equations*) [11,7,8,9,10]. Most optimization has been focused on the smoother, which is typically the most time consuming part of a multigrid method, and specifically on the red-black Gauß-Seidel method, which is by far one of the most popular smoothers. Our study is based on this previous research. In fact, as baseline codes we have employed the highly optimized variants of a two-dimensional red-black Gauß-Seidel relaxation algorithm developed within the *DIME* project [11]. This naïve implementation performs a complete sweep through the grid for updating all the red nodes, and then another complete sweep for updating all the black nodes. Therefore, *rb1* exhibits lower spatial locality than a lexicographic ordering. Furthermore, if the target grid is large enough, temporal locality is not fully exploited.

Alternatively, some authors have successfully improved cache reuse (locality) using loop reordering and data layout transformations that were able to improve both temporal and spatial data locality[12,7].

Following these previous studies, in this paper we have used as baseline codes the different red-black smoothers developed within the framework of the *DIME* project. To simplify matters, these codes are restricted to 5-point as well as 9-point discretization of the Laplacian operator. Figures 2-4 illustrate some of them which tries to fuse the *red* and *black* sweeps.

On the other hand, to the author's knowledge, the study of multigrid smoothers on Multicore Architectures has been hardly researched previously. In[13], authors addressed the alternative parallelization of a 3D-Multigrid Smoother in a *MCP* architectures which is based on temporal blocking scheme. The main idea consists in a temporal division of each slice (group of points) into blocks, which are assigned in the proper order to the

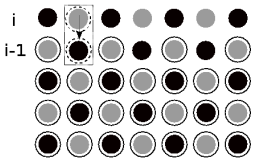


Figure 2. *DIME's* rb2. The update of red and black nodes is fused to improve temporal locality.

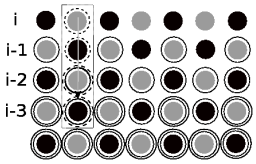


Figure 3. *DIME's* rb5. Data within a tile is reused as much as possible before moving to the next tile.

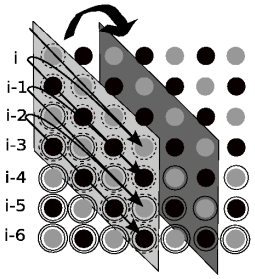


Figure 4. *DIME's* rb9. Data within a tile is reused as much as possible before moving to the next tile.

— — — —

Table 1. Main features of the target computing platform.

Processor	Intel Core^{TM}2 Duo 2.4 GHz	
	L1 DataCache	32+32 KB (data+instruction) 8-way set associative
	L2 Unified Cache	4 MB 8-way set associative
Memory	2048 MBytes DDR2-533 MHz SDRAM	
Operating System	GNU Debian Linux kernel 2.6.20-SMP for 32 bits	
Intel Fortran and C/C++ Compiler Switches(v9.1)	-static -O3 -tpp7 -xT -ip -ipo -no-prec-div Parallelization with OpenMP: -qsmp=omp	

different threads as it was a pipeline approach. Each block is marked with the numbers of relaxations which allows the multiple relaxations in each sweep. However, despite the relative simplicity of the proposed scheme, it is not compatible with some of the most successfully sweeps developed in the *DIME* project. Therefore, it makes highly necessary the revision of the exploitation in a *MCP* architectures in conjunction with better memory usage exposed in the *DIME* project.

4 Experimental Platform

Our experimental platform consists in an *Intel Core^{TM}2 Duo* processor running under Linux, the main features of which are summarized in Table 1.

With this design, this dual-core includes two independent microprocessors that share some resources such as L2 memory cache and the main memory access.

Finally, it is worth to mention that the exploitation of *MCP* has been performed in this work by means of *OpenMP* directives, which are directly supported by the Intel FORTRAN/C native compilers[14].

5 MCP-aware Red-Black Smoothers

The availability of *MCP* introduces a new scenario in which thread-level parallelism can be exploited by means of the execution of several threads in the different cores. This logical view suggests the application of the general principles of data partitioning to get the multithreaded versions of the different *DIME* variants of the red-black Gauß-Seidel smoother. This strategy, which can be easily expressed with *OpenMP* directives, is suitable for shared memory multiprocessor. However, in a *MCP*, the similarities amongst the different threads (they execute the same code with just a different input dataset) may cause contention since they have to compete for the shared L2-cache and memory accesses.

```
#pragma omp parallel private(task,more_tasks) shared(control_variables)
more_tasks = true
while more_tasks do

  #pragma omp critical
  Scheduler.next_task(&task);

  if (task.type == RED) then
    Relax_RED_line(task);
  end if

  if (task.type == BLACK) then
    Relax_BLACK_line(task);
  end if

  #pragma omp critical
  more_task=Scheduler.commit(task);

end while
```

Algorithm 4. Interleaved implementation of a red-black Gauß-Seidel

Alternatively, we have employed a dynamic partitioning where computations are broken down into different tasks with are assigned to the pool of available threads. Intuitively, the smoothing of the different colours is interleaved by assigning the relaxation of each colour to a different thread. This interleaving is controlled by a scheduler, which avoids race conditions and guarantees a deterministic ordering.

Algorithm 1 shows a pseudo-code of this approach for red-black smoothing. Our actual implementation is based on the *OpenMP's parallel* and *critical* directives. The critical sections introduce some overhead but are necessary to avoid race-conditions. However, the interleaving prompted by the scheduling allows the *black thread* to take advantage of some sort of data prefetching since it processes grid nodes that have just been processed by the *red thread*, i.e. the *red thread* acts as a helper thread that performs data prefetching for the *black* one.

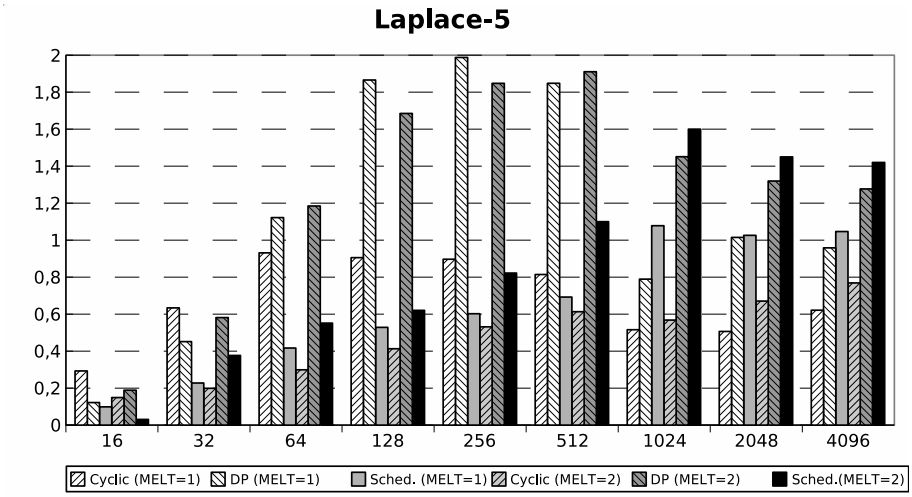


Figure 5. Speedup achieved by different parallel implementations of a red-black Gauß-Seidel smoother for a 5-point Laplace stencil. Sched denotes our strategy, whereas DP and Cyclic denote the best block and a cyclic distribution of the smoother's outer loop respectively. MELT is the number of successive relaxations that have been applied.

6 Results

Figure 5 shows the speedup achieved by the different parallel strategies over the baseline code (with the best DIME's transformation) for the the 5-point stencil. Our strategy improves inter-thread locality taking advantage of fine-grain thread sharing especially in large grid sizes.

As can be noticed, the election of the most suitable strategy depends on the grid size:

- For small and medium grid sizes block and cyclic distributions outperforms our approach, although for the smallest sizes none of them is able to improve performance as consequence of the costs involved in the creation/destruction of thread which is not compensated by the parallel execution. For these working sets, memory bandwidth and data cache exploitation are not a key issue and traditional strategies beats our approach on performance due to the overheads introduced by the dynamic task scheduling.

- However, for large sizes we observe the opposite behaviour given that the overheads involved in task scheduling become negligible, whereas the competition for memory resources becomes a bottleneck in the other versions. In fact, we should highlight that the block and cyclic distributions become less efficient for large grids.

- The break-even point between the static distributions and our interleaved approach is a relative large grid and corresponds to 4 MB (L2 shared cache of the *Intel Core^{TM}2 Duo*).

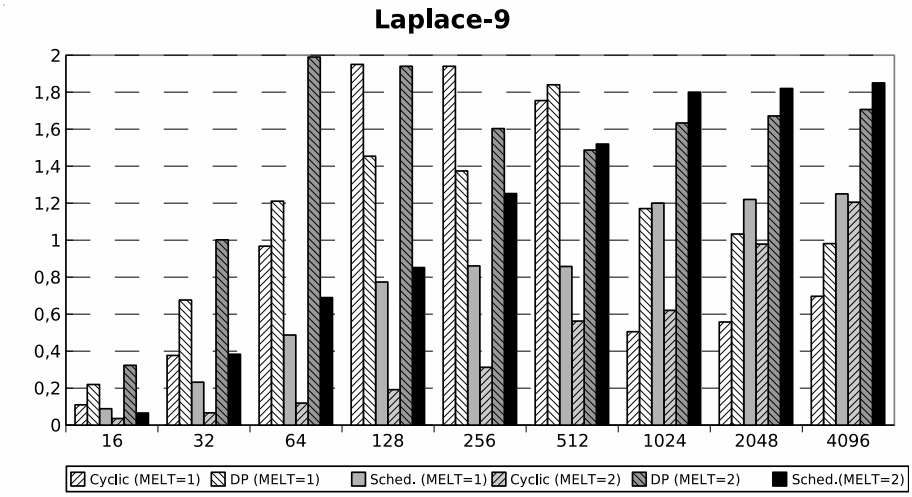


Figure 6. Speedup achieved by different parallel implementations of a red-black Gauß-Seidel smoother for a 9-point Laplace stencil. Sched denotes our strategy, whereas DP and Cyclic denote the best block and a cyclic distribution of the smoother's outer loop respectively. MELT is the number of successive relaxations that have been applied.

Figure 6 confirms some of these observations for the the 9-point stencil. Furthermore, the improvements over *DIME's* variants are higher in this case, since this is a more demanding problem.

7 Conclusions

In this paper, we have introduced a new implementation of red-black Gauß-Seidel Smoothers, which on *MCP* processors fits better than other traditional strategies. From the results presented above, we can draw the following conclusions:

- Our alternative strategy, which implicitly introduce some sort of tiling amongst threads, provide noticeable speedups that match or even outperform the results obtained with the different *DIME's rb2-9* variants for large grid sizes. Notice that instead of improving *intra-thread locality*, our strategy improves locality taking advantage of fine-grain thread sharing, expecially successful in the caches shared as in most of *MCPs*.

- For large grid sizes, competition amongst threads for memory bandwidth and data cache works against traditional block distributions. Our interleaved approach performs better in this case, but suffers important penalties for small grids, since its scheduling overheads does not compensate its better exploitation of the temporal locality. Given that multigrid solvers process multiple scales, we advocate hybrid approaches.

Acknowledgements

This work has been supported by the Spanish research grants TIC 2002-750, TIN 2005-5619 and the Hipeac European Network of Excellence.

References

1. U. Trottenberg, C. Oosterlee, and A. Schüller, *Multigrid*, (Academic Press, 2000).
2. J. L. Thomas, B. Diskin and Achi Brandt, *Textbook multigrid efficiency for fluid simulations*, Annual Review of Fluid Mechanics, **35**, 317–340, (2003).
3. M. F. Adams, M. Brezina, J. J. Hu, and R. S. Tuminaro, *Parallel multigrid smoothing: polynomial versus Gauss-Seidel*, J. Comp. Phys., **188**, 593–610, (2003).
4. E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro and U. M. Yang, *A Survey of parallelization techniques for multigrid solvers*, Tech. Rep., (2004).
5. W. Mitchell, *Parallel adaptive multilevel methods with full domain partitions*, App. Num. Anal. and Comp. Math, **1**, 36–48, (2004).
6. F. Hülsemann, M. Kowarschik, M. Mohr and U. Rüde, *Parallel Geometric Multigrid*, Lecture Notes in Computer Science and Engineering, **51**, 165–208, (2005).
7. C. Weiß, W. Karl, M. Kowarschik and U. Rüde, *Memory characteristics of iterative methods*, in: Proc. ACM/IEEE Supercomputing Conf. (SC99), Portland, Oregon, USA, (1999).
8. M. Kowarschik, U. Rüde, C. Weißand W. Karl, *Cache-aware multigrid methods for solving Poisson's equation in two dimensions*, Computing, **64**, 381–399, (2000).
9. C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde and C. Weiß, *Cache optimization for structured and unstructured grid multigrid*, Electronic Transactions on Numerical Analysis (ETNA), **10**, 21–40, (2000).
10. M. Kowarschik, C. Weißand U. Rüde, *Data layout optimizations for variable coefficient multigrid*, in: Proc. 2002 Int. Conf. on Computational Science (ICCS 2002), Part III, P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, (Eds.), of *Lecture Notes in Computer Science*, vol. **2331**, pp. 642–651, (Springer, Amsterdam, 2002).
11. Friedrich-Alexander University Erlangen-Nuremberg. Department of Computer Science 10, DIME project, Available at http://www10.informatik.uni-erlangen.de/Research/Projects/DiME-new.
12. D. Quinlan, F. Bassetti and D. Keyes, *Temporal locality optimizations for stencil operations within parallel object-oriented scientific frameworks on cache-based architectures*, in: Proc. PDCS'98 Conference, (1998).
13. D. Wallin, H. Löf, E. Hagersten and S. Holmgren, *Multigrid and Gauss-Seidel smoothers revisited: parallelization on chip multiprocessors*, in: ICS '06: Proc. 20th Annual International Conference on Supercomputing, pp. 145–155, (ACM Press, New York, 2006).
14. Intel Corpation, em Intel C/C++ and Intel Fortran Compilers for Linux, Available at http://www.intel.com/software/products/compilers.

Parallelization of Multilevel Preconditioners Constructed from Inverse-Based ILUs on Shared-Memory Multiprocessors

José I. Aliaga[1], **Matthias Bollhöfer**[2], **Alberto F. Martín**[1], **and Enrique S. Quintana-Ortí**[1]

[1] Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain
E-mail: {*aliaga, martina, quintana*}*@icc.uji.es*

[2] Institute of Computational Mathematics, TU-Braunschweig, D-38106 Braunschweig, Germany
E-mail: m.bollhoefer@tu-braunschweig.de

In this paper, we present an OpenMP parallel preconditioner based on ILUPACK. We employ the METIS library to locate independent tasks which are dynamically scheduled to a pool of threads to attain a better load balance. Experimental results on a shared-memory platform consisting of 16 processors report the performance of our parallel algorithm.

1 Introduction

The solution of linear systems and eigenvalue problems is ubiquitous in chemistry, physics, and engineering applications. When the matrix involved in these problems is large and sparse, iterative methods as, e.g., those based on Krylov subspaces, are traditionally employed in the solution of these problems[1]. Among these methods, ILUPACK[a] (Incomplete LU decomposition PACKage) is a novel software package based on approximate factorizations which enhances the performance of the process in terms of a more accurate solution and a lower execution time.

In order to reduce the time that is needed to compute the preconditioner or the execution time per iteration of a linear system solver, we can use high-performance computing techniques to better exploit the platform where the problem is to be solved. In this paper we pursue the parallelization using OpenMP[2] of the computation of a preconditioner, based on ILUPACK, for the solution of linear systems with symmetric positive definite (s.p.d.) coefficient matrix. The target architecture, shared-memory multiprocessors (SMMs), includes traditional parallel platforms in scientific computing, such as symmetric multiprocessors (SMPs), as well as the novel multicore processors. OpenMP provides a natural, simple, and flexible application programming interface for developing parallel applications for parallel architectures with shared-memory (and we assume that it will continue to do so for future multicore systems).

The paper is structured as follows. In Section 2 we briefly review ILUPACK. Next, in Section 3, we offer some details on the parallel preconditioner. Section 4 then gathers data from our numerical experiments with the parallel algorithm, and a few concluding remarks and future research goals follow in Section 5.

[a] http://www.math.tu-berlin.de/ilupack.

2 An Overview over ILUPACK

ILUPACK includes C and Fortran routines to solve linear systems of the form $Ax = b$ via (iterative) Krylov subspace methods: the package can be used to both compute a preconditioner and apply it to the system. We will focus on the computation of the preconditioner since this is the most challenging task from the parallelization viewpoint. The routines that perform this task in ILUPACK sum more than 4000 lines of code. Hereafter we will consider the coefficient matrix A to be s.p.d.

The rationale behind the computation of the preconditioner is to obtain an incomplete LU decomposition of A, while avoiding computations with "numerically-difficult" diagonal pivots, which are moved to the last rows by applying a sequence of permutations, P. To do that, the Crout variant[1] of the LU decomposition is used, so that the following computations are performed in each step of the procedure:

1. Apply the transformations corresponding to the part of the matrix that is already factored to the current row and column of the matrix.
2. If the current pivot is "numerically dubious", move the current row and column to the last positions of the matrix and accumulate this permutation on P.
3. Otherwise, proceed with the factorization of the current row and column of the matrix and apply certain "dropping techniques".

When this process is finished a partial ILU decomposition of $P^T AP$ is obtained, and a Schur complement must be computed for that part of the permuted matrix that was not factored. The process is recursively repeated on the Schur complement until the matrix is fully factored, yielding a *multilevel* ILU decomposition of the permuted matrix. The dropping techniques and the computation of the Schur complement are designed to bound the elements of the inverses of the triangular factors in magnitude. This property improves the numerical performance of the method, as the application of the preconditioner involves those inverses[3,4,5,6].

3 An OpenMP Parallel Preconditioner

In this section we describe our parallel preconditioner. It is important to realize that this is not a parallel implementation of the serial preconditioner in ILUPACK, but a parallel algorithm for the computation of a preconditioner that employs the serial routines in ILUPACK. The computations (stages and operations) as well as the results (preconditioners) obtained by ILUPACK and our parallel algorithm are, in general, different.

The first step in the development of a parallel preconditioner consists in splitting the process into tasks and identifying the dependencies among these. After that, tasks are mapped to threads deploying task pools[7], in an attempt to achieve dynamic load balancing while fulfilling the dependencies.

Frequently, the initial ordering of the sparse coefficient matrix is not suitable to parallel factorization, because it is not possible to identify a number of independent tasks sufficiently large or because the costs of the tasks that would result are highly unbalanced. In these cases, a different ordering (permutation) of the matrix may help to expose more parallelism. In particular, the MLND (Multilevel Nested Dissection) algorithm included

in METIS[8] usually leads to balanced elimination trees which exhibit a high degree of concurrency. Figure 1 illustrates the original sparsity pattern of the *Apache1* matrix from the University of Florida (UF) sparse matrix collection[9], and the pattern of the matrix ordered using MLND.

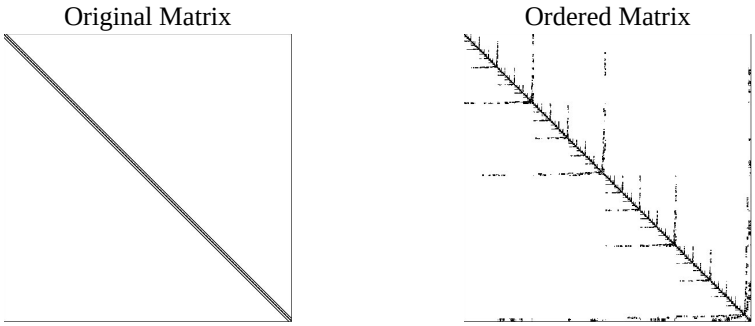


Figure 1. Sparsity pattern of the *Apache1* matrix. Left: original. Right: matrix ordered using MLND.

Tasks for the parallel algorithm and their dependencies can be identified by manipulating the elimination tree[10] of the ordered matrix: If we condense each elimination subtree rooted at height $log_2(p)$, where p is the number of processors (threads), we obtain a set of tasks which is organized in a tree-like structure, with nodes representing tasks, and the ancestor-descendant relationship representing dependencies among them; see Fig. 2. This task tree defines a partition of the ordered matrix which identifies matrix blocks that can be factorized in parallel and matrix blocks whose factorization depends on other computations. Figure 3 represents the partition defined by a task tree of height 1. The factorization of the leading blocks A_{11}, A_{22}, and their corresponding subdiagonal blocks, can be performed in parallel. However, the factorization of A_{33} depends on the results produced by the factorizations of the two leading diagonal blocks.

In order to perform the parallel factorization of the matrix, we assign a submatrix to each leaf of the tree; see the bottom of Fig. 3. There, A^1_{33} and A^2_{33} comply with $A_{33} = A^1_{33} + A^2_{33}$. Driven by the multilevel ILU scheme, the parallel algorithm computes the factorization of the leading diagonal block of each leaf submatrix, and then obtains the corresponding Schur complements. To do this computations, the parallel algorithms employs the serial routines in ILUPACK. Upon completion of this process, the root task appropriately combines the Schur complements of each factorization, creating a submatrix which can be fully factorized. This process is easily generalized for task trees of height larger than 1.

Due to the properties of the MLND ordering, the major part of the computation is concentrated on the leaves of the task tree; therefore a good load balance in the computational costs of the tasks associated with the leaves is mandatory to achieve high parallel performance. Figure 2 shows the "estimated" cost (in terms of the number of nonzero elements of the corresponding submatrix) of the leaves for the MLND-ordered *Apache1* matrix. Although MLND ordering performs a best-effort work, there are a few leaves that concentrate the major part of the computation. In order to attain a better load balance, our parallel al-

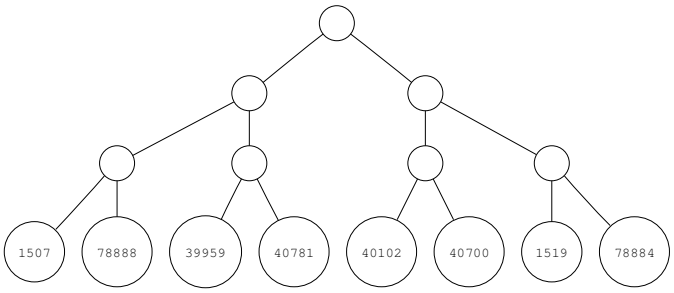


Figure 2. Non-split task tree for the MLND-ordered *Apache1* matrix. The labels in the nodes represent the number of nonzero elements in the corresponding submatrix. These values are used as an estimation of the cost of the associated task.

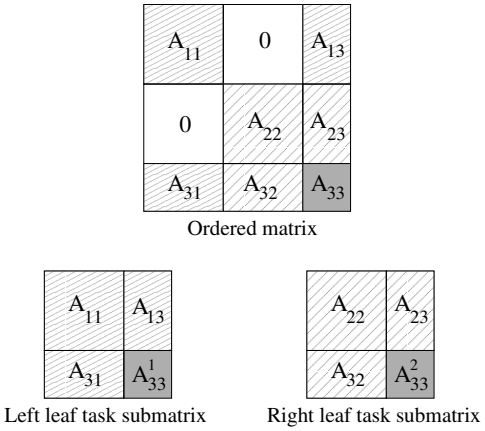


Figure 3. Top: ordered matrix partition defined by an example task tree of height 1. Bottom: submatrices created on leaf tasks.

gorithm splits those tasks with cost higher than a given threshold into finer-grain tasks; see the split task tree for the MLND-ordered *Apache1* matrix in Fig. 4. MLND ordering is a must in our current implementation of the parallel algorithm as it forms the basis for the identification of concurrent tasks; on the other hand, this ordering is optional for the serial algorithm in ILUPACK. Task splitting in the parallel algorithm generally forces a different number of levels compared with those of the serial algorithm; therefore, the stages and operations that are performed by these two algorithms also differ.

The task tree is constructed sequentially, before the (true) computation of the preconditioner commences. All leaf tasks in this tree are initially inserted in the *ready queue* which, at any moment, contains those tasks with all dependencies fulfilled. Tasks are dequeued from the head and enqueued at the tail of this structure. In order to balance the load, the execution of the leaf tasks with higher computational cost is priorized by inserting them first in the queue. The execution of tasks is scheduled dynamically: as threads become idle, they monitor the queue for work (pending tasks). When a thread completes execution

of a task, all tasks dependent on it are examined and those with their dependencies fulfilled are enqueued at the ready queue by this thread. Idle threads continue to dequeue tasks until all tasks have been executed. Similar mechanisms have been proposed for irregular codes as part of the Cilk project[11] and for dense linear algebra codes in the FLAME project[12].

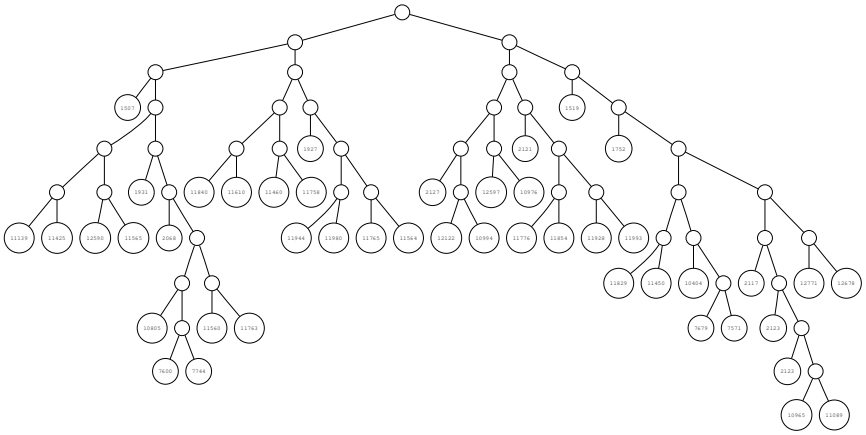


Figure 4. Split task tree for the MLND-ordered *Apache1* matrix. The labels in the nodes represent the number of nonzero elements in the corresponding submatrix. These values are used as an estimation of the cost of the associated task.

4 Experimental Results

All experiments in this section were obtained on a SGI Altix 350 CC-NUMA multiprocessor consisting of 16 Intel Itanium2@1.5 GHz processors sharing 32 GBytes of RAM via a SGI NUMAlink interconnect. No attempt is made to exploit the data locality on this CC-NUMA architecture. IEEE double-precision arithmetic was employed in all the experiments, and one thread was scheduled per processor. In both the serial and parallel algorithms we used ILUPACK default values for the condition estimator (`condest`=100), and the factor and Schur tolerances (`tol1`=10^{-2}, `tol2`=10^{-2}) for the dropping thresholds in the triangular factors and their complements.

Table 1 characterizes the benchmark matrices from the UF sparse matrix collection[9] employed in the evaluation, and reports the results obtained from the execution of the serial algorithm in ILUPACK: serial execution time and fill-in factor (ratio between the number of nonzero elements in the triangular factors produced by the algorithm and the matrix) for the matrix preprocessed using MLND ordering.

In the parallel algorithm, a task is further subdivided into two subtasks when the ratio between the number of nonzero entries of the complete matrix and the submatrix associated with the task was larger than p (option A) or $2p$ (option B), with p the number of processors/threads. Table 2 reports the number of leaves in the task trees that these values produce. Clearly option A yields a smaller number of leaves, and thus a smaller degree of

Code	Matrix name	Rows/Cols.	Nonzeros	Time (s)	Fill-in factor
M1	*GHS_psdef/Apache1*	80800	542184	2.81	4.8
M2	*Schmid/Thermal1*	82654	574458	1.32	4.2
M3	*Schenk_AFE/Af_0_k101*	503625	17550675	24.5	2.7
M4	*GHS_psdef/Inline_1*	503712	36816342	143	4.8
M5	*GHS_psdef/Apache2*	715176	4817870	36.9	5.8
M6	*GHS_psdef/Audikw_1*	943695	77651847	320	4.1

Table 1. Matrices selected to test the parallel multilevel ILU algorithm (left) and results (execution time and fill-in factor) from the execution of the serial algorithm in ILUPACK (right).

Matrix	Option A: #Leaf tasks					c_w (%)					Fill-in factor				
M1	2	4	11	23	24	0	0	57	58	58	4.8	4.8	4.8	4.7	4.7
M2	3	5	9	17	19	70	49	35	24	31	4.1	4.1	4.2	4.2	4.2
M3	2	4	9	16	18	0	0	4	4	20	2.7	2.6	2.6	2.5	2.5
M4	3	6	12	16	21	42	30	29	14	32	4.4	4.2	3.8	3.6	3.5
M5	2	7	11	27	27	0	85	60	78	78	5.8	5.8	5.9	6.0	6.0
M6	2	4	8	16	**17**	0	2	4	6	**15**	3.7	3.5	3.4	3.4	3.4
#Procs.	2	4	8	12	16	2	4	8	12	16	2	4	8	12	16

Matrix	Option B: #Leaf tasks					c_w (%)					Fill-in factor				
M1	4	11	24	43	45	0	58	58	47	46	4.8	4.8	4.7	4.3	4.2
M2	5	9	19	33	37	50	35	31	18	25	4.1	4.2	4.2	4.1	4.0
M3	4	9	18	32	37	0	5	20	07	19	2.6	2.6	2.5	2.3	2.2
M4	6	12	21	33	42	30	29	32	16	28	4.2	3.8	3.5	3.2	3.1
M5	7	11	27	50	**50**	85	60	78	67	**67**	5.8	5.9	6.0	6.1	6.1
M6	4	8	**17**	32	32	2	4	**15**	7	7	3.5	3.4	3.4	3.4	3.4
#Procs.	2	4	8	12	16	2	4	8	12	16	2	4	8	12	16

Table 2. Number of leaf tasks, coefficient of variation, and fill-in generated by the parallel algorithm using 2, 4, 8, 12, and 16 processors.

parallelism, but the tasks present higher granularity. As a measure of how similar the estimated costs of the leaves are, the table also shows the coefficient of variation $c_w = \sigma_w/\bar{w}$, with σ_w the standard deviation and $\bar{w}$ the average of these costs. A ratio close to 0 (e.g., in Option A/M1/4 processors) indicates that all leaves have very similar costs, while a ratio closer to 100% (e.g., Option A/M5/4 processors) indicates a high variability of the costs that could be the source for an unbalanced distribution of the computational load. Finally, the last column of the table reports the fill-in factor produced by the parallel algorithm, which are similar to those attained by the serial algorithm in ILUPACK. As the number of processors or the number of tasks are increased, the fill-in factor tends to be reduced.

Table 3 reports the execution time and the speed-up of the parallel algorithm. The speed-up is computed with respect to the parallel algorithm executed using the same task

Matrix	Option A Time (secs.)					Speed-up				
M1	1.36	0.66	0.38	0.32	0.23	1.97	3.87	6.37	7.19	10.18
M2	0.64	0.32	0.17	0.16	0.01	1.99	3.81	7.20	7.60	12.53
M3	12.2	6.13	3.24	2.13	1.8	1.96	3.89	7.26	11.09	13.02
M4	64.9	36.7	16.9	15.4	9.28	2.01	3.41	6.69	7.13	11.35
M5	18.2	9.44	7.80	3.40	2.61	1.99	3.84	4.54	10.43	13.61
M6	152	89.4	68.5	51.8	**51.2**	1.88	3.25	5.38	9.69	**9.92**
#Procs.	2	4	8	12	16	2	4	8	12	16

Matrix	Option B Time (secs.)					Speed-up				
M1	1.29	0.64	0.35	0.27	0.18	1.97	3.79	6.56	8.96	11.13
M2	0.62	0.31	0.17	0.12	0.01	1.99	3.81	7.08	9.70	15.91
M3	12.2	6.38	3.21	2.97	2.13	1.96	3.69	7.28	7.46	10.32
M4	64.9	30.6	15.1	9.56	7.07	1.93	3.70	6.99	10.45	13.77
M5	18.3	9.04	4.61	3.40	**2.46**	1.97	3.92	7.70	10.17	**14.08**
M6	150	110	**70.2**	64.9	51.3	1.94	3.35	**4.07**	10.55	13.38
#Procs.	2	4	8	12	16	2	4	8	12	16

Table 3. Performance of the parallel algorithm using 2, 4, 8, 12, and 16 processors.

tree on a single processor (notice that the number of leaves in the task tree depends on the number of processors). A comparison against the serial algorithm in ILUPACK offers superlinear speed-ups in many cases (see execution times in Table 1) and has little meaning here: the serial and the parallel algorithms compute different preconditioners and to do so, perform different operations. Remarkable speed-ups are attained for Option A/M3 and Option B/M4, M5 and other combinations of option/matrix/number of processors. On the other hand, using 16 processors, a speed-up of only 9.92 is obtained for Option A/M6. The coefficient c_w for this case (see Table 2) is 15%, indicating that all tasks are similar in cost, but the splitting mechanism yields 17 leaf tasks (which concentrate the major part of the computational load) and are to be mapped on 16 processors. As a consequence the distribution of the computational load is unbalanced. A similar case occurs for Option B/M6 on 8 processors. Surprisingly, another similar case, the combination of Option A/M2 on 8 processors, which presents 9 leaves in the task tree, delivers a high speed-up. A closer inspection revealed 8 leaves with very similar costs in the corresponding task tree and a single leaf with of much smaller cost. Due to the dynamic scheduling mechanism, one of the processors receives two tasks, one of them the task of much smaller cost, so that the computational load is not significantly unbalanced.

An analysis of the scalability of the parallel algorithm, though desirable, is difficult. First, in many benchmarks the coefficient matrix is associated with a physical problem, and its dimension (size/sparsity degree) cannot be increased at will (at least easily). Second, even in those cases where the dimension is a parameter that can be adjusted, there may not be a direct relation between the dimension and the cost of the computation of the preconditioner. Finally, in general it is hard to predict the fill-in that will occur during the computation of the preconditioner.

5 Conclusions and Future Work

We have presented a parallel multilevel preconditioner for the iterative solution of sparse linear systems using Krylov subspace methods. The algorithm internally employs the serial routines in ILUPACK. MLND ordering, task splitting, and dynamic scheduling of tasks are used to enhance the degree of parallelism of the computational procedure. Experimental results on a SMM with 16 Itanium2 processors report the performance of our parallel algorithm.

Future work includes:

- To compare and contrast the numerical properties of the preconditioner in ILUPACK and our parallel preconditioner.
- To parallelize the application of the preconditioner to the system.
- To exploit data locality on CC-NUMA architectures, evaluating policies which map tasks to threads taking into consideration the latest tasks assigned to the threads.
- To develop an MPI parallel preconditioner.

References

1. Y. Saad, *Iterative Methods for Sparse Linear Systems*, (SIAM Publications, 2003).
2. OpenMP Arch. Review Board: OpenMP specifications", http://www.openmp.org.
3. M. Bollhoefer, *A robust ILU based on monitoring the growth of the inverse factors*, Linear Algebra Appl., **338**, 201–218, (2001).
4. O. Schenk, M. Bollhöfer and R. A. Römer, *On large scale diagonalization techniques for the Anderson model of localization*, SIAM J. Sci. Comput., **28**, 963–983, (2006).
5. M. Bollhoefer and Y. Saad, *On the relations between ILUs and factored approximate inverses*, SIAM J. Matrix Anal. Appl., **24**, 219–237, (2002).
6. M. Bollhoefer, *A robust and efficient ILU that incorporates the growth of the inverse triangular factors*, SIAM J. Sci. Comput., **25**, 86–103, (2003).
7. M. Korch and Th. Rauber, *A comparison of task pools for dynamic load balancing of irregular algorithms*, Concurrency and Computation: Practice and Experience, **16**, 1–47, (2004).
8. G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., **20**, 359–392, (1998).
9. T. Davis, *University of Florida sparse matrix collection*, http://www.cise.ufl.edu-/research/sparse/matrices.
10. T. Davis, *Direct methods for sparse linear systems*, (SIAM Publications, 2006).
11. C. Leiserson and A. Plaat, *Programming parallel applications in Cilk*, SINEWS: SIAM News, (1998).
12. E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí and R. van de Geijn, *SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures*, in: Proc. 19th ACM SPAA'07, pp. 116–125, (2007).

Parallelism in Structured Newton Computations

Thomas F. Coleman and Wei Xu

Department of Combinatorics and Optimization
University of Waterloo
Waterloo, Ontario, Canada. N2L 3G1
E-mail: tfcoleman@uwaterloo.ca
E-mail: wdxu@math.uwaterloo.ca

Many vector-valued functions, representing expensive computation, are also structured computations. A structured Newton step computation can expose useful parallelism in many cases. This parallelism can be used to further speed up the overall computation of the Newton step.

1 Introduction

A fundamental computational procedure in practically all areas of scientific computing is the calculation of the Newton step (in n-dimensions). In many cases this computation represents the dominant cost in the overall computing task. Typically the Newton step computation breaks down into two separable subtasks: calculation of the Jacobian (or Hessian) matrix along with the right-hand-side, and then the solution of a linear system (which, in turn, may involve a matrix factorization). Both subtasks can be expensive though in many problems it is the first, calculation of the function and derivative matrices, that dominates.

In most cases when the Newton step computation is relatively expensive, the function that yields the Newton system is itself a 'structured' computation. A structured computation is one that breaks down into a (partially ordered) straight-line sequence of (accessible) macro computational subtasks. For example, if F is a function that is computed by evaluating the sequence F_1, F_2, F_3, in order, then F is a structured computation. The general structured situation can be described as follows: F is a structured computation, $z = F(x)$, if F is evaluated by computing a (partially-ordered) sequence of intermediate vectors y defined below:

$$\left.\begin{array}{rc} \text{Solve for } y_1 & : F_1^E(x, y_1) = 0 \\ \text{Solve for } y_2 & : F_2^E(x, y_1, y_2) = 0 \\ \vdots & \vdots \\ \text{Solve for } y_p & : F_p^E(x, y_1, y_2, \cdots, y_p) = 0 \\ \text{"Solve" for output } z & : z - F_{p+1}^E(x, y_1, y_2, \cdots, y_p) = 0 \end{array}\right\}. \tag{1.1}$$

For convenience define

$$F^E(x, y_1, \cdots, y_p) = \begin{pmatrix} F_1^E(x, y_1) \\ F_2^E(x, y_1, y_2) \\ \vdots \\ F_p^E(x, y_1, \cdots, y_p) \\ F_{p+1}^E(x, y_1, \cdots, y_p) \end{pmatrix}.$$

The Newton process for (1.1) can be written,

$$J^E \begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix} = -F^E = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ -F \end{pmatrix}, \tag{1.2}$$

where the square Jacobian matrix J^E is a block lower-Hessenberg matrix:

$$J^E = \left(\begin{array}{c|cccc} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y_1} & & & \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y_1} & \frac{\partial F_2}{\partial y_2} & & \\ \vdots & \vdots & \vdots & \ddots & \\ \frac{\partial F_p}{\partial x} & \frac{\partial F_p}{\partial y_1} & \cdots & \cdots & \frac{\partial F_p}{\partial y_p} \\ \hline \frac{\partial F_{p+1}}{\partial x} & \frac{\partial F_{p+1}}{\partial y_1} & \cdots & \cdots & \frac{\partial F_{p+1}}{\partial y_p} \end{array}\right). \tag{1.3}$$

It has been illustrated[2,3] that by exploiting the structure illustrated in (1.1), it is possible to compute the Newton step, at any point x, significantly faster than by following the standard 2-step procedure: form the Jacobian (Hessian) matrix of F, and then solve a linear system. The key insight is that the Jacobian matrix of the larger system illustrated in (1.1) is typically sparse and *thus can be computed much more cheaply than the (possibly) dense Jacobian matrix of* F. Moreover, given that the Jacobian matrix has been computed, it is possible to compute the Newton step to the original system $F(x) = 0$, by working directly with the large (sparse) matrix, and possibly avoiding the formulation of $J(x)$, the Jacobian matrix of F.

In this paper we show that these structural ideas also expose parallelism which can be used to further speed up the computation of the Newton step. The rest of the paper is organized as follows. Two extremal examples for exposing parallelism are studied in Section 2. One is the generalized partially separable problem, which is the best case for the parallelism. The other example is the composite function of a dynamic system, which is the worst case. Generally, most problems are somewhere between the above two cases. In Section 3, we construct an example for the structured general case and expose the parallelism to speed up the Newton step. Finally, conclusions are given in Section 4.

2 Two Extremal Cases

Many practical problems can be covered by the structural notions presented in Section 1. Here, we describe two examples. These two examples represent the two extreme cases. The first example, a generalized partially separable problem, yields a set of decoupled computations, which is the best for parallelism. The second example is a composite function of a dynamic system, representing a recursive computation.

A square generalized partial separable (GPS) function is a vector mapping $F : \mathbb{R}^n \to \mathbb{R}^n$. The evaluation of $z = F(x)$ may involve the following steps:

$$\left.\begin{array}{l} \text{Solve for } y_i : \ y_i - T_i(x) = 0 \quad i = 1, \cdots, p \\ \text{Solve for } z : \ z - \bar{F}(y_1, \cdots, y_p) = 0. \end{array}\right\}, \tag{2.1}$$

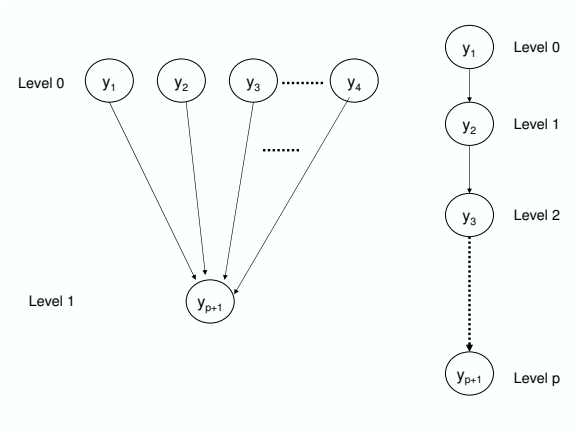


Figure 1. Directed acyclic graphs corresponding to the generalized partially separable function and the composite dynamic system.

where $T_i(i = 1, \cdots, p)$ and $\bar{F}$ are nonlinear functions. Clearly, the GPS case is the best one can hope for from the point of view of (macro-) parallelism since each computation of intermediate variable y_i is independent. The structured computation approach often allows for the identification of less obvious (macro-) parallelism. However, the worst case is the composite function of a dynamic system which involves heavy recursion:

$$\left.\begin{array}{l} \text{Solve for } y_i: \ y_i - T_i(y_{i-1}) = 0, i = 1, \cdots, p \\ \text{Solve for } z: \ z - \bar{F}(y_p) = 0, \end{array}\right\}. \tag{2.2}$$

In this case there is no obvious (macro) parallelism. The component functions T_i and corresponding Jacobian matrices J_i must be computed sequentially. The expanded Jacobian matrices of generalized partially separable function in (2.1) and the composite dynamic system (2.2) are, respectively,

$$J^E_{GPS} = \begin{pmatrix} -J_1 & I & & \\ -J_2 & & I & \\ \vdots & & & \ddots \\ -J_p & & & I \\ 0 & \bar{J}_1 & \bar{J}_2 \cdots & \bar{J}_p \end{pmatrix} \quad \text{and} \quad J^E_{DS} = \begin{pmatrix} -J_1 & I & & \\ & -J_2 & I & \\ & & \ddots & \ddots \\ & & & -J_p \ \ I \\ & & & \bar{J} \end{pmatrix}.$$

Typically, as illustrated above, many of the block entries of J^E in (1.3) are zero-blocks, and we can associate a directed acyclic graph, $\overrightarrow{G}(J^E)$, to represent this block structure. Specifically $\overrightarrow{G}(J^E)$ has $p+1$ nodes, $y_1, \cdots, y_p, y_{p+1} = z$ and there is a directed edge y_j from y_i iff $\frac{\partial F_i}{\partial y_j} \neq 0 (i = 1 : p+1, j = 1 : p, i \neq j)$. Thus, the corresponding directed acyclic graphs for the GPS and composite functions are shown in Fig. 1. It illustrates that the generalized partially separable case is the most 'parallelism-friendly' since there are only two levels and p of total $p+1$ nodes can be computed concurrently. Figure 1 also illustrates that the composite dynamic system case is the worst with respect to parallelism

since there are p levels for a total of $p+1$ nodes and this sequence of nodes has to be computed sequentially [a].

To illustrate the benefit of parallelism, consider the following experiment on a GPS problem. We define a composite function $T(x) = \hat{F}(A^{-1}\tilde{F}(x))$, where $\hat{F}$ and $\tilde{F}$ are Broyden functions[1], $y = B(x)$, (their Jacobian matrices are tridiagonal), which is in the following form

$$\begin{aligned} y_1 &= (3-2x_1)x_1 - 2x_2 + 1, \\ y_i &= (3-2x_i)x_i - x_{i-1} - 2x_{i+1} + 1, \quad i = 2, 3, \cdots, n-1 \\ y_n &= (3-2x_n)x_n - x_{n-1} + 1, \end{aligned}$$

where n is the size of the vector variable x. The structure of A is based on 5-point Laplacian defined on a square $(\sqrt{n}+2)$-by-$(\sqrt{n}+2)$ grid. For each nonzero element of A, A_{ij} is defined as the function of x, that is $A_{ij} = r_j x_j$ where r_j is a random variable, e.g. $r_j = N(1,0)$, that is r_j is normally distributed around 1 with variance 0. So, the GPS function can be defined as follows

$$G(x) = \frac{MB^{-1}}{K}[T_1(x) + T_2(x) + \cdots + T_K(x)],$$

where $T_i(x)$ is same as $T(x)$ except the uncertainties in A, B is sparse symmetric positive definite tridiagonal, M is tridiagonal and K is a scalar. The explicit form of Jacobian matrix of $G(x)$ is

$$J = \frac{MB^{-1}}{K}(J_1 + J_2 + \cdots + J_K),$$

where where $J_i = \hat{J}A^{-1}[\tilde{J} - A_x y_2],\ i = 1, 2, \cdots, K$, $\hat{J}$ and $\tilde{J}$ are Jacobian matrices of $\hat{F}(\cdot)$ and $\tilde{F}(\cdot)$, respectively. It is clear that parallelism can be used to concurrently evaluate each pair $(y_i,\ J_i),\ i = 1, \cdots, p$. A simple master-slave scheduling mechanism can be used to assign tasks to processors and collect results. The MatlabMPI package[5] developed by Kepner at MIT is used to implement the parallel computation. The experiments were carried out on a SGI Altix 3700 system with 64 1.4 GHz Itanium2 processors and 192 GB RAM running under SuSE Linux Enterprise System (SLES) 10 with SGI's ProPack 5 added on. Matlab 7.0.1 (R14) does not support the 64-bit Itanium architecture, so we use "Matlab -glnx86" to run Matlab in 32-bit emulation mode. Twenty-four processors, one for master and the other twenty three for slaves, were employed to solve nonlinear equations of generalized partially separable problems with vector variable sizes ranged from 625 to 2500 and $K = 240$. In the parallel computation of the Newton method exploiting the structure, each processor computes the summation of 10 J_i's independently. Then, the master collects the summation from slaves to compute the Jacobian J, and solves the dense Newton system by '\' in Matlab. We do not construct the expanded Jacobian matrix, J^E_{GPS}, in this experiment because forming the explicit form of J is quite efficient. The implementation of parallelism without exploiting the structure is as follows. Each processor computes some columns of J, that is each processor computes the corresponding columns of $J_1,\ J_2, \cdots, J_K$ and sums them and then sends the columns to a master processor, which constructs J explicitly and solves the Newton system using '\'. The package

[a] In this paper, we talk a worst case view since some of the computations in node i depend on results from node $i-1$. We assume, for simplicity, that there is no concurrency between these nodes.

ADMAT-2.0, a new version of ADMAT[4], is employed to implement the structured and the forward mode AD. Table 1 displays the results of the running time of the standard Newton method exploiting and without exploiting the structure in sequential and parallel computation, respectively.

	Newton method exploiting the structure			Newton method without exploiting the structure		
n	Sequential comp.	Parallel comp.	Speedup	Sequential comp.	Parallel comp.	Speedup
625	121.92	8.52	14.31	268.87	73.97	3.63
900	310.47	19.41	16.00	564.07	172.11	3.28
1024	469.37	26.71	17.57	735.09	266.47	2.83
1600	1526.17	81.16	18.80	2817.22	824.97	3.41
2500	5556.49	265.46	20.93	27685.07	8556.37	3.19

Table 1. Running times of the Newton method for solving a GPS problem in sequential and parallel computation implemented on 24 processors in seconds.

As the problem size increases, the computation of the Jacobian matrices J_i becomes increasingly expensive, and this dominates the computation time of a single Newton step. The speedup due to parallelism approaches 20 exploiting the structure and is less than 4 without exploiting it, using 24 processors. Without exploiting the structure, when computing columns of J_i, we need to compute $A^{-1}[\tilde{J} - A_x y_2]_{col}$, where $[\tilde{J} - A_x y_2]_{col}$ consists of some columns of $[\tilde{J} - A_x y_2]$. Instead of computing the inverse of A, we solve a linear system with right-hand sides $[\tilde{J} - A_x y_2]_{col}$. However, the running time of solving a multiple right-hand sides linear system is not proportional to the size of right-hand sides in Matlab using '\'. Suppose $[\tilde{J} - A_x y_2]$ has 625 columns. $[\tilde{J} - A_x y_2]_{col}$ only has 25 columns, but computing $A^{-1}[\tilde{J} - A_x y_2]_{col}$ is only 3 or 4 times faster than computing $A^{-1}[\tilde{J} - A_x y_2]$, rather than 25 times as we might expect. It turns out that the product, $A^{-1}[\tilde{J} - A_x y_2]_{col}$ dominates the speedup of parallelism of forward mode AD on solving the GPS problem. In our experiment, we employ 24 processors to implement the parallelism of forward mode, but because of the product $A^{-1}[\tilde{J} - A_x y_2]_{col}$, the speedup is limited between 3 and 4.

MatlabMPI was implemented through the file system in Matlab, rather than using "real message passing ". In other words, message passing between the master and slave processors can be quite time-consuming. However, in our program, we minimize the communications among processors. In a single Newton step, only two communications are required. One is sending the result back to the master from slaves. The other is distributing the updated x from the master to slaves for the next Newton step. Thus, the message passing in MatlabMPI does not slow down the parallel computation on the generalized partially separable problem. The time spent on communication is less than 5% running time in our experiments.

The other example is a dynamic system computation. Consider the autonomous ODE,

$$y' = F(y),$$

where $F(\cdot)$ is a Broyden function and suppose $y(0) = x^0$, we use an explicit one-step Euler method to compute an approximation y_k to a desired final state $y(T)$. Thus, we

n	Newton method exploiting the structure	Newton method without exploiting the structure	Speedup
200	0.3120	0.0940	3.3191
400	3.0620	0.2970	10.3098
800	26.6100	1.0790	24.6846
1000	52.3460	1.6090	32.6576

Table 2. Running times of one Newton step through two approaches and the speedup of exploiting the structure AD on a dynamical system.

obtain a recursive function in following form,

$$\begin{aligned} &y_0 = x \\ &\text{for } i = 1, \cdots, p \\ &\qquad \text{Solve for } y_i : \ y_i - F(y_{i-1}) = 0 \\ &\text{Solve for } z : \ z - y_p = 0, \end{aligned}$$

where we take $p = 5$ in the experiment. This experiment was carried out on a laptop with Intel Duo 1.66 GHz processor and 1 GB RAM running Matlab 6.5. The Jacobian matrix J is treated as full when the structure is ignored. Table 2 records the running times and the speedup in sequential computation since there is no apparent parallelism in this example. Subsequently, Table 2 illustrates that the cost of Newton step with structured AD is linear while the cost of the unstructured is cubic. In other words, exploiting the structure accelerates the computation of the Newton step.

3 General Case

Of course the generalized partially separable case represents the best situation with respect to parallelism whereas the right-hand of Fig. 1, the composite function, represents the worst - there is no apparent parallelism at this level. In general, this structured approach will reveal some easy parallelism which can be used to further accelerate the Newton step computation. In this section, we will look at an "in-between" example, to illustrate the more general case.

We consider the evaluation of $z = F(x)$ in following steps:

$$\left.\begin{aligned} &\text{Solve for } y_i : \ y_i - T_i(x) = 0 \quad i = 1, \cdots, 6 \\ &\text{Solve for } y_7 : \ y_7 - T_7((y_1 + y_2)/2) = 0 \\ &\text{Solve for } y_8 : \ y_8 - T_8((y_2 + y_3 + y_4)/3) = 0 \\ &\text{Solve for } y_9 : \ y_9 - T_9((y_5 + y_6)/2) = 0 \\ &\text{Solve for } y_{10} : \ y_{10} - T_{10}((y_7 + y_8)/2) = 0 \\ &\text{Solve for } y_{11} : \ y_{11} - T_{11}((y_8 + y_9)/2) = 0 \\ &\text{Solve for } z : \ z - 0.4(y_{10} + y_{11}) - 0.2y_5 = 0. \end{aligned}\right\}, \tag{3.1}$$

where $T_i(x)(i = 1, \cdots, 6)$ is same as the function $G(x)$ in Section 2 except have $K = 3000$, $T_j(x)(j = 7, \cdots, 11)$ is the same as the function $T(x)$ in Section 2, but with different uncertainties. It is clear that the computation of $T_i(x)$ $(i = 1, \cdots, 6)$ dominates

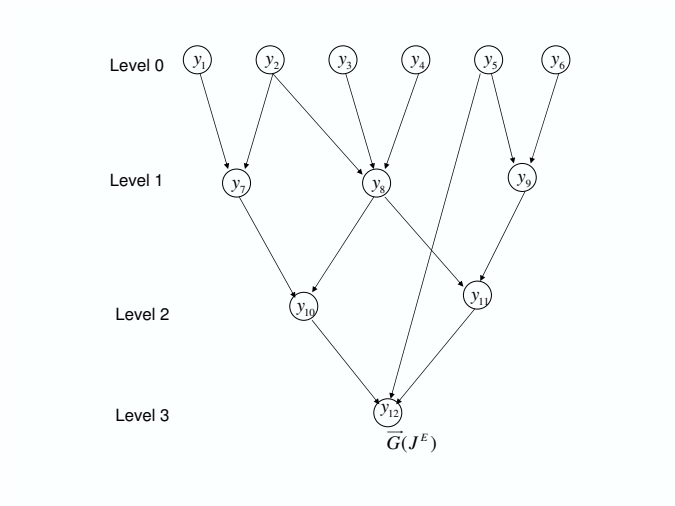


Figure 2. Directed acyclic graph corresponding to 12-by-12 Jacobian matrix J^E.

the running time of evaluating $z = F(x)$. The structure of the corresponding expanded Jacobian matrix is illustrated as follows,

$$J^E = \begin{bmatrix} X & X & & & & & & & & & & \\ X & & X & & & & & & & & & \\ X & & & X & & & & & & & & \\ X & & & & X & & & & & & & \\ X & & & & & X & & & & & & \\ X & & & & & & X & & & & & \\ & X & X & & & & & X & & & & \\ & & X & X & X & & & & X & & & \\ & & & & & X & X & & & X & & \\ & & & & & & & X & X & & X & \\ & & & & & & & & X & X & & X \\ & & & & & X & & & & & X & X \end{bmatrix},$$

and the corresponding directed acyclic graph $\overrightarrow{G}(J^E)$ is given in Fig. 2. The level sets in $\overrightarrow{G}(J^E)$ (Fig. 2.) can be used to identify the (macro-) parallelism in the structured Newton computation. All the nodes on level i can be computed concurrently. Then, after all the nodes on level i are computed, we can start to compute all the nodes on level $i + 1$. In the experiment for exploiting the structure, we divide the six nodes on level zero into three groups. Each group employed 8 CPUs, one for master and others for slaves. The master processor is in charge of collecting results from 'slaves' in its own group and sending the results to the master processors in other groups if necessary. There are only three nodes on level 1, so only master processors do the computation and communication. On level two, one of the master processor is idled while only one master processor is working on level

	Newton method exploiting the structure			Newton method without exploiting the structure		
n	Sequential comp.	Parallel comp.	Speedup	Sequential comp.	Parallel comp.	Speedup
100	401.60	19.24	20.87	428.77	201.32	2.13
169	807.31	38.41	21.02	1002.07	240.45	4.17
225	1286.73	61.25	21.01	1694.07	335.18	5.05
400	3795.80	180.21	20.93	5373.92	602.02	3.19
625	8878.24	481.71	18.43	12954.22	1017.71	12.73
900	19681.72	1045.37	18.83	26883.98	2105.77	13.34

Table 3. Running times of the Newton method in sequential and parallel computation implemented on 24 processors.

3. Table 3 records the results of the Newton method exploiting and without exploiting the structure. When exploiting the structure, the time spent on communication is about 10% of running time much more than the percentage without exploiting it, which is 2%. In Fig. 2, it shows that communications are required between slaves and master in each group on level zero and among master processors on other levels. Thus, the structured AD approach spent more time on communication than the forward mode, but it is still significantly faster than the forward mode case.

4 Conclusions

In this paper, we have illustrated how a standard Newton step also exposes useful parallelism in most cases. This parallelism can be used to further speed up the computation of the Newton step. We studied two extreme cases. One is the general partially separable case, the best case for (macro) parallelism. The other is the composite function of a dynamic system as shown in (2.2), where there is no exposed parallelism (though in this case the structural techniques proposed in (2.1) conveniently work particularly well). Generally, most cases are somewhere between these two extreme cases.

References

1. C. G. Broyden, *A class of methods for solving nonlinear simulations equations*, Mathematics of Computations, **19**, 577–593, (1965).
2. T. F. Coleman and W. Xu, *Fast Newton computations*, submitted to SIAM on Scientific Computing (2007).
3. T. F. Coleman and A. Verma, *Structured and efficient Jacobian calculation*, in: M. Berz, C.Bischof, G. Corliss and A. Griewank, editors, Computational Differentiation: Techniques, Applications and Tools, , 149–159, (SIAM, Phil., 1996)
4. T. F. Coleman and A. Verma, *ADMIT-1: Automatic differentiation and MATLAB interface Toolbox*, ACM Transactions on Mathematical Software, **16**, 150–175, (2000).
5. J. Kepner, *Parallel programming with MatlabMPI*, `http://www.ll.mit.edu/MatlabMPI`.

Automatic Computation of Sensitivities for a Parallel Aerodynamic Simulation

Arno Rasch, H. Martin Bücker, and Christian H. Bischof

Institute for Scientific Computing
RWTH Aachen University, D–52056 Aachen, Germany
E-mail: {*rasch, buecker, bischof*}*@sc.rwth-aachen.de*

Derivatives of functions given in the form of large-scale simulation codes are frequently used in computational science and engineering. Examples include design optimization, parameter estimation, solution of nonlinear systems, and inverse problems. In this note we address the computation of derivatives of a parallel computational fluid dynamics code by automatic differentiation. More precisely, we are interested in the derivatives of the flow field around a three-dimensional airfoil with respect to the angle of attack and the yaw angle. We discuss strategies for transforming MPI commands by the forward and reverse modes of automatic differentiation and report performance results on a Sun Fire E2900.

1 Introduction

An interdisciplinary team of engineers, mathematicians, and computer scientists at RWTH Aachen University is currently bringing together computational fluid dynamics with experimental data for high-lift and cruise configurations of civil aircraft in the full range of Mach and Reynolds numbers. The aim of the work carried out within the context of the collaborative research center SFB 401 “Modulation of flow and fluid–structure interaction at airplane wings” is to better understand the dynamics of the vortex system which is generated by large aircraft during take-off and landing. Such a detailed knowledge of the wake flow field is essential to estimate safe-separation distances between aircraft in take-off and landing.

Developed at the Institute of Aerodynamics, the TFS package[1,2] solves the Navier–Stokes equations of a compressible ideal gas in two or three space dimensions with a finite volume formulation. The spatial discretization of the convective terms follows a variant of the advective upstream splitting method (AUSM$^+$) proposed by Liou.[3,4] The viscous stresses are centrally discretized to second-order accuracy. The implicit method uses a relaxation-type linear solver whereas the explicit method relies on a multigrid scheme. A serial as well as a block-oriented MPI-parallelized version of TFS based on domain decomposition are available.

Given the TFS code, we are interested in a qualitative and quantified analysis of the dependences of certain program outputs on certain input parameters. Such information is invaluable for optimizing objective functions with respect to suitable decision variables or for the identification of model parameters with respect to given experimental data. For instance, aircraft design[5] or analysis of turbulence parameters[6] require the sensitivities of functions given in the form of large-scale Navier–Stokes solvers. A recent trend in computational science and engineering is to compute such sensitivities by a program transformation known as automatic differentiation (AD).[7,8]

In the present work, we are interested in the derivatives of the flow field with respect to the angle of attack and the yaw angle, making it necessary to transform a parallel program. No current software implementing the AD transformation is capable of fully handling MPI or OpenMP parallel code. So, it is no surprise that there is hardly any publication in the open literature describing the application of AD to a nontrivial parallel code, an exception being an article by Heimbach et al.[9] The contribution of this work is to sketch the underlying ideas of transforming a parallel code by AD and to demonstrate the feasibility of that approach on a concrete real-world problem arising from aerodynamics.

In Section 2, we briefly explain the automatic differentiation technique and its application to the (serial) TFS code. The transformation of the constructs involving MPI-based parallelism is addressed in Section 3. Performance results of the AD-generated parallel program are reported in Section 4.

2 Automatic Differentiation of TFS

Automatic differentiation (AD) refers to a set of techniques for transforming a given computer program P into a new program P' capable of computing derivatives in addition to the original output. More precisely, if P implements some function

$$f : \mathbb{R}^n \to \mathbb{R}^m, \tag{2.1}$$

mapping an input x to an output $y = f(x)$, the transformed program P' computes not only the function value y but also the $m \times n$ Jacobian matrix $J = \partial y / \partial x$ at the same point x. The AD-generated program P' is called the *differentiated* program of the *original* program P.

Throughout this note, we consider the computation of the flow field for a given angle of attack and yaw angle using TFS. From a conceptual point of view, this computation is carried out by a routine

```
TFS(alpha,beta,vel,pre,...)
```

where the scalar input variables `alpha` and `beta` represent the angle of attack and the yaw angle, respectively. The arrays `vel(1:3*L)` and `pre(1:L)` denote the velocity and pressure at L grid points in three space dimensions. So, we consider the TFS code to implement some function (2.1) with $n = 2$ and $m = 4L$ computing velocity and pressure from two angles.

Specifying that one is interested in the derivatives of the variables `vel` and `pre` with respect to the variables `alpha` and `beta`, automatic differentiation with the ADIFOR[10] tool then transforms the original `TFS` code into a new code

```
g_TFS(g_alpha,alpha,g_beta,beta,g_vel,vel,g_pre,pre,...)
```

where, in addition to all variables occurring in the parameter list of `TFS`, new derivative variables with the prefix `g_` are introduced. These new derivative variables are designed to store two scalar derivatives for every scalar value occurring in the original program. For instance, `g_alpha(1:2)` is associated with the scalar variable `alpha` while `g_vel(1:2,1:3*L)` is associated with `vel(1:3*L)`.

The basic idea behind AD is the fact that the execution of a code is nothing but a sequence of elementary arithmetic operations such as binary addition or intrinsic functions.

The partial derivatives of these elementary functions are known and, following the chain rule of differential calculus, can be combined in a step-wise manner to yield the overall derivative of the entire program. In contrast to approximations obtained from numerical differentiation like divided differences, derivatives computed by AD are free from truncation error. To illustrate AD, consider the following simple code fragment

```
u = z(1) + z(2)
v = v*u
```

that could appear somewhere inside of TFS, and keep in mind that the technique is also applicable to larger codes. We introduce a straightforward AD strategy called the *forward mode* by studying the transformation of the sample code fragment. In the forward mode, a gradient object g_w is associated to every variable w appearing in the original code. This gradient object stores the partial derivatives of the variable w with respect to the input variables of interest. In the TFS example, the variable g_w stores $\partial \mathtt{w}/\partial(\mathtt{alpha},\mathtt{beta})$. Setting a variable that stores the number of scalar variables with respect to which derivatives are computed to n=2, the resulting AD-generated code is given by:

```
do i = 1,n
  g_u(i) = g_z(i,1) + g_z(i,2)
enddo
u = z(1) + z(2)
do i = 1,n
  g_v(i) = g_v(i)*u + v*g_u(i)
enddo
v = v*u
```

This code propagates derivatives of intermediate variables with respect to input variables following the control flow of the original program. If the differentiated code g_TFS is called with the initializations

```
g_alpha(1)=1 , g_alpha(2)=0 , g_beta(1)=0 , g_beta(2)=1
```

it computes the desired derivatives

$$\begin{aligned}
\mathtt{g_vel(1,1:3*k)} &= \partial \mathtt{vel(1:3*k)}/\partial \mathtt{alpha} \\
\mathtt{g_vel(2,1:3*k)} &= \partial \mathtt{vel(1:3*k)}/\partial \mathtt{beta} \\
\mathtt{g_pre(1,1:k)} &= \partial \mathtt{pre(1:k)}/\partial \mathtt{alpha} \\
\mathtt{g_pre(2,1:k)} &= \partial \mathtt{pre(1:k)}/\partial \mathtt{beta}
\end{aligned}$$

In addition to the forward mode, automatic differentiation provides the *reverse mode* which propagates derivatives of the output variables with respect to intermediate variables by reversing the control flow of the original program.

3 Automatic Differentiation of MPI Functions

Automatic differentiation of the serial 2D TFS code employing version 2 of the AD-tool ADIFOR[10] is reported in previous publications.[11,12,13] In the present study, we report on automatic differentiation of the MPI-parallelized version of 3D TFS. Besides the extension from two to three space dimensions, the main achievement compared to our previous work is the semi-automatic transformation of the parallel constructs. Previous work

on automatic differentiation of message-passing parallel programs is published in several contributions.[14,15,16,17,18,19,20]

The basic point-to-point communication of the MPI interface is a pair of send and receive commands operating between two processes P_1 and P_2. Assume an operation `send(u,P2)` executed on P_1 that sends a variable `u` from P_1 to process P_2. Assume further that `receive(v,P1)` is the corresponding command on P_2 that receives the data obtained from P_1 in a variable `v`. This pair of communication commands can be thought of as copy to and from a virtual data structure `buffer` that is accessible from both processes. That is, in the notation

```
P1:   send( u, P2 )          P2:   receive( v, P1 )
   // buffer = u                   // v = buffer
```

the statements on the left carried out on P_1 are equivalent as are the statements on the right on P_2. Using the interpretation of a global buffer, the differentiated statements in the forward mode can be derived as

```
P1:// g_buffer = g_u         P2:// g_v = g_buffer
      send( g_u, P2 )              receive( g_v, P1 )
```

By reversing the control flow of the original program, the differentiated statements in the reverse mode are given by

```
P1:// a_u = a_u + a_buffer   P2:// a_buffer = a_buffer + a_v
      receive( tmp, P2 )           send( a_v, P1 )
      a_u = a_u + tmp              a_v = 0
```

Reduction operations combine communication and computation. As an example, consider the case where each process P_i contains a scalar value u_i in a variable `u`. Let `reduce(u,v,'sum')` denote the reduction computing the sum $v = \sum_{i=1}^{p} u_i$ over p processes, where the result is stored in a variable `v` on process, say, P_1. In the forward mode, the derivatives are reduced in the same way. That is, the result `g_v` on P_1 is computed by `reduce(g_u,g_v,'sum')` from given distributed values stored in `g_u`. In the reverse mode, a broadcast of `a_v` from P_1 to all other processes is followed by an update of the form `a_u = a_u + a_v` executed on each process. A recipe for AD of a reduction operation computing the product of distributed values is given in a paper by Hovland and Bischof.[15]

Another issue to be specifically addressed by AD for message-passing parallel programs is activity analysis. The purpose of this data-flow analysis is to find out if a variable is active or not. A variable is called active if it depends on the input variables with respect to which derivatives are computed and if it also influences the output variable whose derivatives are computed. In addition to activity analysis of serial programs, there is need to track variable dependencies that pass through send and receive operations. Typically, activity analysis of message-passing parallel programs is carried out conservatively,[19] overestimating the set of active variables which leads to AD-generated programs whose efficiency in terms of storage and computation could be improved. Only recently, a study is concerned with investigating activity analysis for MPI programs more rigorously.[20]

Since ADIFOR 2 is not capable of transforming MPI library calls, the corresponding MPI routines are excluded from the AD process. That is, using ADIFOR 2 the TFS code is automatically differentiated with respect to the angle of attack and the yaw angle, where

the MPI process communication is ignored. However, in order to ensure correct overall forward propagation of the derivatives, the derivative information computed by the MPI processes needs to be exchanged whenever the original function values are exchanged. In essence, our approach combines AD of the serial TFS code in an automated fashion with a manual activity analysis of MPI library calls along the lines discussed in Strout et al.[20]

In TFS the original code contains the following non-blocking *send* operation

```
  call mpi_isend(A, k, mpi_double_precision, dest,
 &               tag, mpi_comm_world, request, ierror)
```

which asynchronously sends the data stored in the buffer `A` to a destination process `dest`. This buffer consists of `k` consecutive entries of type `double precision`. Then, the differentiated code produced by the forward mode of AD requires a similar operation for sending the derivative values `g_A` of `A`, where the number of data entries is increased by the number, n, of directional derivatives propagated through the code. The corresponding non-blocking *send* operation reads as follows:

```
  call mpi_isend(g_A, k*n, mpi_double_precision, dest,
 &               tag, mpi_comm_world, request, ierror)
```

Recall from the previous section that the dimension of the derivative `g_A` is increased by one, compared to its associated original variable `A`. The leading dimension of `g_A` is therefore given by `n`. In this study we manually inserted the additional MPI calls for sending the derivative values, although in principle they could be automatically generated by the AD tool. In a completely analogous fashion, for every blocking *receive* operation occurring in the original TFS code, receiving `A` from the MPI process `source`

```
  call mpi_recv(A, k, mpi_double_precision, source,
 &              tag, mpi_comm_world, status, ierror)
```

we inserted a corresponding MPI call to receive derivative information from other MPI processes:

```
  call mpi_recv(g_A, k*n, mpi_double_precision, source,
 &              tag, mpi_comm_world, status, ierror)
```

Finally, we sketch the corresponding AD transformation for the reverse mode. In the reverse mode of AD, for each scalar program variable in the original code, an adjoint of length m is propagated through the differentiated code. Moreover during the reversal of the control flow of the original program the blocking *receive* operation is replaced by a blocking *send* for the adjoint computation:

```
  call mpi_send(a_A, k*m, mpi_double_precision, dest,
 &              tag, mpi_comm_world, ierror)
```

Here, the destination process `dest` is the sending process `source` of the message in the corresponding *receive* operation in the original code.

4 Performance Experiments

In the sequel we present results from numerical experiments with TFS and its differentiated version. The corresponding execution time measurements are carried out on

a Sun Fire E2900 equipped with 12 dual-core UltraSPARC IV processors running at 1.2 GHz clock speed. We compute an inviscid three-dimensional flow around the BAC 3-11/RES/30/21 transonic airfoil which is reported in the AGARD advisory report no. 303[21] and used as reference airfoil for the collaborative research center SFB 401. The Mach and Reynolds numbers are $M_\infty = 0.689$ and $Re = 1.969 \times 10^6$, respectively. The angle of attack is -0.335° and the yaw angle is 0.0°. The computational grid for TFS consists of approximately 77,000 grid points and is divided into three different blocks.

The execution times for the parallel code running on one and three processors is given in Table 1. The table also shows the resulting speedup, taking the execution time of the parallel code with one MPI task as reference. The second column refers to the original TFS code while the third column contains the data of its differentiated version denoted by TFS$'$. The ratio of the third to second column is displayed in the fourth column of this table.

Metric	TFS	TFS$'$	Ratio
Serial execution time [s]	338	2197	6.50
Parallel execution time [s]	157	969	6.17
Speedup	2.15	2.27	1.05

Table 1. Execution times required for performing 100 iterations with TFS and its differentiated version TFS$'$. The serial and parallel execution time is measured on a Sun Fire E2900 using one and three processors, respectively.

While the original TFS code achieves a speedup of 2.15 employing three MPI tasks, TFS$'$ yields a slightly better speedup of 2.27 for the same configuration. However, optimal speedup cannot be achieved because the three processors are assigned to blocks of different sizes. While the largest of the three blocks consists of 35,640 grid points, the remaining two blocks comprise 24,687 and 17,091 grid points, respectively. This leads to a work load imbalance where one task is assigned about 46% of the total amount of computational work, while the other two tasks perform only 32% and 22% of the computational work, respectively. As a consequence two of the three MPI processes spend a certain amount of time waiting for the process with the higher work load, assuming identical processor types. A run-time analysis of the execution trace using the VAMPIR[22] performance analysis and visualization tool reveals that about 23.7% of the overall execution time is spent in MPI communication.

5 Concluding Remarks

Sensitivities of selected output variables with respect to selected input variables of large-scale simulation codes are ubiquitous in various areas of computational science and engineering. Examples include optimization, solution of nonlinear systems of equations, or inverse problems. Rather than relying on numerical differentiation based on some form of divided differencing that inherently involves truncation error, we suggest the use of techniques of automatic differentiation. While this program transformation applied to serial programs has proven to work in a robust and stable fashion under a wide range of

circumstances, no current automatic differentiation tool is capable of handling message-passing parallel programs in a fully automated way. In fact, the number of successful transformation for non-trivial parallel codes published in the open literature is small. For a concrete three-dimensional computational fluid dynamics application arising from aerodynamics, we successfully applied automatic differentiation to an MPI-parallelized finite volume code solving the Navier–Stokes equations. More precisely, we computed the sensitivities of the flow field around a transonic airfoil with respect to the angle of attack and the yaw angle. Though most of the program transformation is handled automatically, the differentiation of the MPI functions is carried out manually.

Acknowledgements

We appreciate the comments by the anonymous reviewers and would like to thank the Institute of Aerodynamics for making available the source code of the flow solver TFS. This research is partially supported by the Deutsche Forschungsgemeinschaft (DFG) within SFB 401 “Modulation of flow and fluid–structure interaction at airplane wings,” RWTH Aachen University, Germany. The Aachen Institute for Advanced Study in Computational Engineering Science (AICES) provides a stimulating research environment for our work.

References

1. D. Hänel, M. Meinke, and W. Schröder, *Application of the multigrid method in solutions of the compressible Navier–Stokes equations*, in: Proc. 4th Copper Mountain Conference on Multigrid Methods, J. Mandel, (Ed.), pp. 234–254, SIAM, Philadelphia, PA, (1989).
2. M. Meinke and E. Krause, *Simulation of incompressible and compressible flows on vector-parallel computers*, in: Proceedings of the 1998 Parallel CFD Conference, Hsinchu, Taiwan, May 11–14, 1998, pp. 25–34, (1999).
3. M. S. Liou and Ch. J. Steffen, *A new flux splitting scheme*, J. Comp. Phys., **107**, 23–39, (1993).
4. M. S. Liou, *A sequel to AUSM: AUSM*$^+$, J. Comp. Phys., **129**, 164–182, (1996).
5. C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and E. Slusanschi, *Efficient and accurate derivatives for a software process chain in airfoil shape optimization*, Future Generation Computer Systems, **21**, 1333–1344, (2005).
6. C. H. Bischof, H. M. Bücker, and A. Rasch, *Sensitivity analysis of turbulence models using automatic differentiation*, SIAM Journal on Scientific Computing, **26**, 510–522, (2005).
7. L. B. Rall, *Automatic differentiation: techniques and applications*, vol. **120** of *Lecture Notes in Computer Science*, (Springer Verlag, Berlin, 1981).
8. A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, (SIAM, Philadelphia, 2000).
9. P. Heimbach and C. Hill and R. Giering, *An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation*, Future Generation Computer Systems, **21**, 1356–1371, (2005).

10. C. Bischof, A. Carle, P. Khademi, and A. Mauer, *Adifor 2.0: Automatic differentiation of Fortran 77 programs*, IEEE Computational Science & Engineering, **3**, 18–32, (1996).
11. H. M. Bücker, B. Lang, A. Rasch, and C. H. Bischof, *Sensitivity analysis of an airfoil simulation using automatic differentiation*, in: Proc. IASTED International Conference on Modelling, Identification, and Control, Innsbruck, Austria, February 18–21, 2002, M. H. Hamza, (Ed.), pp. 73–76, (ACTA Press, Anaheim, CA, 2002).
12. H. M. Bücker, B. Lang, A. Rasch, and C. H. Bischof, *Computation of sensitivity information for aircraft design by automatic differentiation*, in: Computational Science – ICCS 2002, Proc. International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II, P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra, (Eds.), vol. **2330** of *Lecture Notes in Computer Science*, pp. 1069–1076, (Springer, Berlin, 2002).
13. C. H. Bischof, H. M. Bücker, B. Lang, and A. Rasch, "Automated gradient calculation", in: Flow Modulation and Fluid-Structure Interaction at Airplane Wings, J. Ballmann, (Ed.), number 84 in Notes on Numerical Fluid Mechanics and Multidisciplinary Design, pp. 205–224, (Springer, Berlin, 2003).
14. P. D. Hovland, *Automatic differentiation of parallel programs*, PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, (1997).
15. P. D. Hovland and C. H. Bischof, *Automatic differentiation for message-passing parallel programs*, in: Proc. First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, pp. 98–104, IEEE Computer Society Press, Los Alamitos, CA, USA, (1998).
16. C. Faure and P. Dutto, *Extension of Odyssée to the MPI library – Direct mode*, Rapport de recherche 3715, INRIA, Sophia Antipolis, (1999).
17. C. Faure and P. Dutto, *Extension of Odyssée to the MPI library – Reverse mode*, Rapport de recherche 3774, INRIA, Sophia Antipolis, (1999).
18. A. Carle and M. Fagan, *Automatically differentiating MPI-1 datatypes: The complete story*, in: Automatic Differentiation of Algorithms: From Simulation to Optimization, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, (Eds.), pp. 215–222, (Springer, New York, 2002).
19. A. Carle, *Automatic Differentiation*, in: Sourcebook of Parallel Computing, J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennnedy, L. Torczon, and A. White, (Eds.), pp. 701–719, (Morgan Kaufmann Publishers, San Francisco, 2003).
20. M. Mills Strout, B. Kreaseck, and P. D. Hovland, *Data-Flow Analysis for MPI Programs*, in: Proceedings of the International Conference on Parallel Processing (ICPP-06), Columbus, Ohio, USA, August 14–18, 2006, pp. 175–184, IEEE Computer Society, Los Alamitos, CA, USA, (2006).
21. I. R. M. Moir, *Measurements on a two-dimensional aerofoil with high-lift devices*, in: AGARD–AR–303: A Selection of Experimental Test Cases for the Validation of CFD Codes, vol. 1 and 2. Advisory Group for Aerospace Research & Development, Neuilly-sur-Seine, France, (1994).
22. W. E. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer, **12**, no. 1, 69–80, (1996).

Parallel Jacobian Accumulation

Ebadollah Varnik and Uwe Naumann

Department of Computer Science
RWTH Aachen University, D-52056 Aachen, Germany
E-mail: {varnik, naumann}@stce.rwth-aachen.de

The accumulation of the Jacobian matrix F' of a vector function $F : \mathbb{R}^n \to \mathbb{R}^m$ implemented as a computer program can be regarded as a transformation of its linearized computational graph into a subgraph of the directed complete bipartite graph $K_{n,m}$. This transformation can be performed by applying a vertex elimination technique. We report on first results of a parallel approach to Jacobian accumulation.

1 Introduction

In *automatic differentiation*[1,2,3] we consider implementations of vector functions

$$F : \mathbb{R}^n \supseteq D \to \mathbb{R}^m, \quad \mathbf{y} = F(\mathbf{x}) \quad ,$$

as computer programs written in some imperative programming language, that map a vector $\mathbf{x} \equiv (x_i)_{i=1,\ldots,n}$ of *independent* variables onto a vector $\mathbf{y} \equiv (y_j)_{j=1,\ldots,m}$ of *dependent* variables. We assume that F has been implemented as a computer program. Following the notation in Griewank's book[4] we assume that F can be decomposed into a sequence of p single assignments of the value of scalar *elemental* functions φ_i to unique *intermediate* variables v_j. The *code list* of F is given as

$$(\mathbb{R} \ni)\ v_j = \varphi_j(v_i)_{i \prec j} \quad , \tag{1.1}$$

where $j = n+1, \ldots, q$ and $q = n + p + m$. The binary relation $i \prec j$ denotes a direct dependence of v_j on v_i. The variables $\mathbf{v} = (v_i)_{i=1,\ldots,q}$ are partitioned into the sets X containing the *independent* variables $(\mathbf{v}_i)_{i=1,\ldots,n}$, Y containing the *dependent* variables $(v_i)_{i=n+p+1,\ldots,q}$, and Z containing the intermediate variables $(v_i)_{i=n+1,\ldots,n+p}$. The code list of F can be represented as a directed acyclic *computational graph* $G = G(F) = (V, E)$ with integer vertices $V = \{i : i \in \{1, \ldots, q\}\}$ and edges $(i, j) \in E$ if and only if $i \prec j$. Moreover, $V = X \cup Z \cup Y$, where $X = \{1, \ldots, n\}$, $Z = \{n+1, \ldots, n+p\}$, and $Y = \{n+p+1, \ldots, q\}$. Hence, X, Y, and Z are mutually disjoint. We distinguish between *independent* ($i \in X$), *intermediate* ($i \in Z$), and *dependent* ($i \in Y$) vertices. Under the assumption that all elemental functions are continuously differentiable in some neighbourhood of their arguments all edges (i, j) can be labeled with the partial derivatives $c_{j,i} \equiv \frac{\partial v_j}{\partial v_i}$ of v_j w.r.t. v_i. This labelling yields the *linearized* computational graph G of F. Eq. (1.1) can be written as a system of nonlinear equation $C(\mathbf{v})$ as follows[4].

$$\varphi_j(v_i)_{i \prec j} - v_j = 0 \quad \text{for } j = n+1, \ldots, q \quad .$$

Differentiation with respect to $\mathbf{v}$ leads to

$$C' = C'(\mathbf{v}) \equiv (c'_{j,i})_{i,j=1,\ldots,q} = \begin{cases} c_{j,i} & \text{if } i \prec j \\ -1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \quad .$$

The *extended Jacobian* C' is lower triangular. Its rows and columns are enumerated as $j, i = 1, \ldots, q$. Row j of C' corresponds to vertex j of G and contains the partial derivatives $c_{j,k}$ of vertex j w.r.t. all of its predecessors. In the following we refer to a row i as *independent* for $i \in \{1, \ldots, n\}$, as *intermediate* for $i \in \{n+1, \ldots, n+p\}$, and as *dependent* if $i \in \{n+p+1, \ldots, q\}$.

The Jacobian matrix $F' = F'(\mathbf{x}) = \left(\frac{\partial y_j}{\partial x_i}(\mathbf{x})\right)_{i=1,\ldots,n}^{j=1,\ldots,m} \in I\!R^{m \times n}$ of F at point $\mathbf{x}$ can be computed on the linearized computational graph by elimination[5,6] of all intermediates vertices as introduced in Griewank et al.[7] resulting in a bipartite graph $G' = (\{X, \emptyset, Y\}, E')$ with labels on the edges in E' representing exactly the nonzero entries of F'. Following the chain rule an intermediate vertex $j \in Z$ can be eliminated by multiplying the edge labels over all paths connecting pairwise the predecessors i and successors k followed by adding these products. The result is the label of the edge (i, j).

The elimination of intermediate vertex j can be also understood as elimination of all non-zero entries in row/column j of C'[8]. Therefore one has to find all those rows k with $j \prec k$ and eliminate this dependency according to the chain rule. Our implementation is based on a *Compressed Row Storage* (CRS)[9] representation of C'. However we believe that an explanation in terms of the computational graph will be easier to follow. References to the CRS representation are included where necessary or useful.

Example: Consider a vector function $f : I\!R^2 \to I\!R^2$ with the code list given in Fig. 1 (a). The corresponding G and C' are shown in Fig. 1 (b) and (c), respectively. The vertices [rows] 1 and 2 represent independent, 6 and 7 dependent, and 3, 4, and 5 intermediate vertices [rows]. Consider row 4 in Fig. 1 (c) containing the partials $c_{5,3}$ and $c_{5,4}$. These are labels of incoming edges $(3, 5)$ and $(4, 5)$ of vertex 5 in Fig. 1 (b). Column 5 contains the partial derivatives $c_{6,5}$ and $c_{7,5}$ that are the labels of the outgoing edges $(5, 6)$ and $(5, 7)$ of vertex 5.

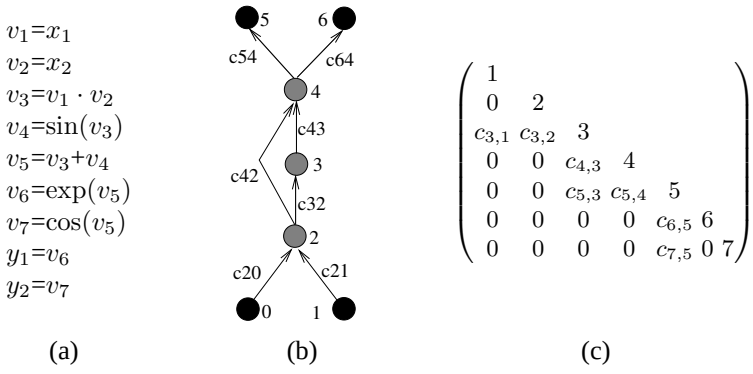
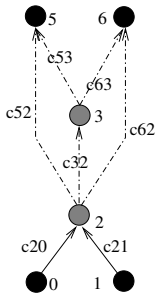


Figure 1. Code list (a), linearized computational graph G (b) and extended Jacobian C' (c) of f. The diagonal entries of C' mark the row index.

Eliminating $c_{6,5}$ in C' as shown in Fig. 2 (b) is equivalent to *back-elimination*[10] of $(5, 6)$ as shown in Fig. 2 (a). *Fill-in* is generated as $c_{6,3}$ $[(3, 6)]$ and $c_{6,4}$ $[(4, 6)]$ since row

[vertex] 6 has a non-zero [incoming edge] in [from] column [vertex] 5. The elimination of row [vertex] 5 on C' [G] can be considered as elimination [back-elimination] of all partial derivatives [outedges] $c_{k,5}$ with $5 \prec k$ [$(5,k)$]. Further elimination of intermediate rows [vertices] 4 and 3 on C' [G] as shown in Fig. 3 (b) [(a)] yield the Jacobian entries in [as] the first two column's [labels of the incoming edges] of the dependent rows [vertices] 6 and 7.



(a)

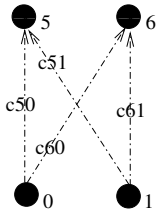
$$\begin{pmatrix} 1 & & & & & \\ 0 & 2 & & & & \\ c_{3,1} & c_{3,2} & 3 & & & \\ 0 & 0 & c_{4,2} & 4 & & \\ 0 & 0 & \mathbf{0} & \mathbf{0} & 5 & \\ 0 & 0 & \mathbf{c_{6,3}} & \mathbf{c_{6,4}} & \mathbf{0} & 6 \\ 0 & 0 & \mathbf{c_{7,3}} & \mathbf{c_{7,4}} & \mathbf{0} & 0 \; 7 \end{pmatrix}$$

(b)

$$\begin{aligned} c_{6,3} &= c_{6,5} \cdot c_{5,3} \\ c_{6,4} &= c_{6,5} \cdot c_{5,4} \\ c_{7,3} &= c_{7,5} \cdot c_{5,3} \\ c_{7,4} &= c_{7,5} \cdot c_{5,4} \end{aligned}$$

(c)

Figure 2. Computational graph G (a) and extended Jacobian C' (b) of f after elimination of vertex and row 5, respectively.



(a)

$$\begin{pmatrix} 1 & & & & & \\ 0 & 2 & & & & \\ \mathbf{0} & \mathbf{0} & 3 & & & \\ 0 & 0 & \mathbf{0} & 4 & & \\ 0 & 0 & \mathbf{0} & \mathbf{0} & 5 & \\ \mathbf{c_{6,1}} & \mathbf{c_{6,2}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 6 \\ \mathbf{c_{7,1}} & \mathbf{c_{7,2}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & 0 \; 7 \end{pmatrix}$$

(b)

$$\begin{aligned} c_{6,3} &+= c_{6,4} \cdot c_{4,3} \\ c_{7,3} &+= c_{7,4} \cdot c_{4,3} \\ c_{6,1} &= c_{6,3} \cdot c_{3,1} \\ c_{6,2} &= c_{6,3} \cdot c_{3,2} \\ c_{7,1} &= c_{7,3} \cdot c_{3,1} \\ c_{7,2} &= c_{7,3} \cdot c_{3,2} \end{aligned}$$

(c)

Figure 3. Bipartite graph G (a) and extended Jacobian C' after elimination of all intermediate vertices and rows, respectively.

2 Parallel Algorithms

In order to parallelize the Jacobian accumulation by vertex elimination, we decompose the computational graph G of F into k subgraphs $G_p = (V_p, E_p)$ with $p \in \{1 \cdots k\}$, $V_p \in V$ and $E_p = \{(i,j) \in E | v_i, v_j \in V_p\}$, where $V = \cup_{p=1}^{k} V_p$ and $E = \cup_{p=1}^{k} E_p$ with $E_p \cap E_q = \emptyset$ for $p, q \in \{1, \cdots, k\}$. Moreover, $V_p = X_p \cup Y_p \cup Z_p$, with vertex cuts X_p and Y_p representing the *local independent* and *local dependent* vertices of the subgraph G_p, respectively. Z_p represents the set of *local intermediate* vertices k, which lie on a path from a vertex $i \in X_p$ to a vertex $j \in Y_p$, with $k \in V_p - \{X_p \cup Y_p\}$. Hence, X_p,

Y_p, and Z_p are mutually disjoint. We call subgraphs G_i and G_j neighbors, if Y_i=X_j with $i, j \in 1, k-1$ and this is only the case, if $j = i + 1$. Whenever we talk about *interface* vertices, we mean the common vertices of two neighbors. We refer to subgraphs G_p as *atomic* subgraphs in the sense that all edges $(i, j) \in E_p$ are between vertices $i, j \in V_p$ of the same subgraph.

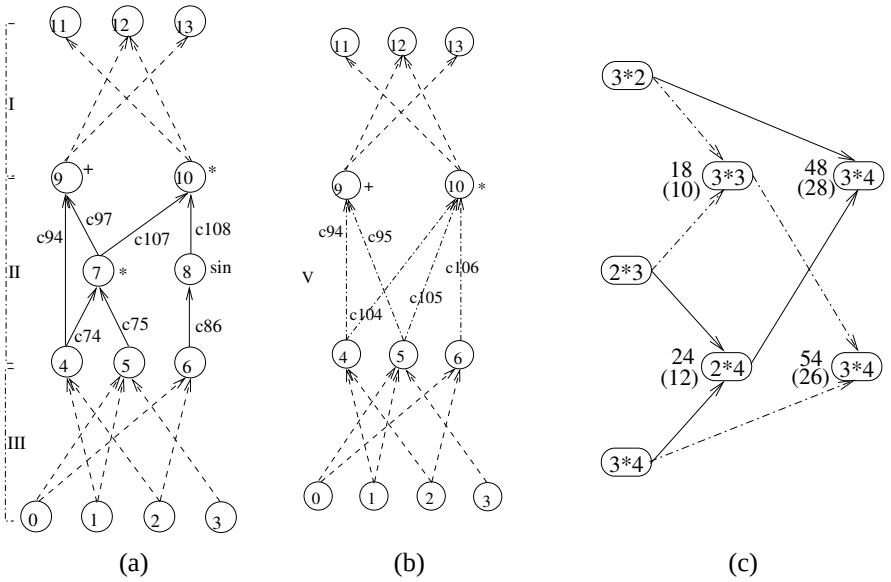


Figure 4. Graph decomposition (a), local Jacobians by vertex elimination (b), optimal matrix product (c).

In this step the main focus is on having balanced subgraphs to optimize the computational and communication effort in concurrent processes. Applying the vertex elimination technique to a subgraph G_p yields the local Jacobian matrix F'_p. Considering the subgraph G_2 in Fig. 4 (a) the elimination of vertices 8 and 9 yields the corresponding bipartite graph G'_2 as shown in Fig. 4 (b). The edge labels of G'_2 correspond to the entries of the local Jacobian

$$F'_2 = \begin{pmatrix} c_{10,5} & c_{10,6} & 0 \\ c_{11,5} & c_{11,6} & c_{11,7} \end{pmatrix} \quad .$$

The reduction to the Jacobian matrix $F' = \prod_{p=1}^{k} F'_p$ can be considered as the chained product of k local Jacobian matrices. For the decomposed computational graph in Fig. 4 (b) with $k = 3$ the Jacobian $F' = {F'_3}^{3*2} \times {F'_2}^{2*3} \times {F'_1}^{3*4}$ is the chained product of local Jacobians F'_3,F'_2, and F'_1. Dynamic programming can be used to optimize the number of floating point multiplications (FLOPS) arising in the chained matrix product as shown in Fig. 4 (c), and described in Griewank et al.[11].

Our implementation uses OpenMP[12,13]. It runs on the shared memory system Solaris Sun Fire E6900 with 16 Ultra Sparc VI 1.2 GHz Dual Core processors and 96 GByte

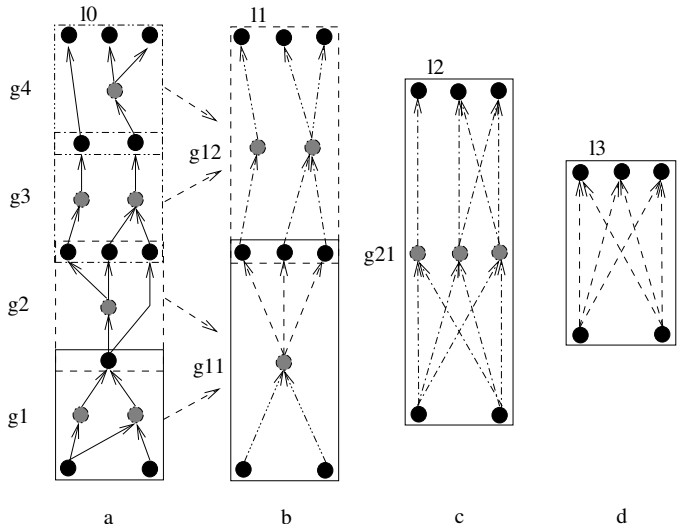


Figure 5. Parallel Jacobian accumulation by vertex elimination using Pyramid strategy.

Memory. In the following we present two ideas for parallelizing the process of Jacobian accumulation by vertex elimination on the computational graph.

2.1 Pyramid Approach

The pyramid approach realizes a level-based parallel vertex elimination illustrated with an example in Fig. 5. At the lowest level $l = 0$ (Fig. 5 (a)) the computational graph G consists of 4 atomic subgraphs. The gray coloured vertices represent the local intermediate vertices of subgraphs. For example the subgraph G_3^0 has 2 local intermediate, 3 local independent, and 2 local dependent vertices, respectively. Applying the vertex elimination on all 4 subgraphs in parallel yields the computational graph as shown in Fig. 5 (b) at level $l = 1$. G gets decomposed into 2 subgraphs, namely G_1^1 and G_2^1. The decomposition at level $l > 0$ is nothing else than merging two neighboring subgraphs into one. The interface vertices of two neighbors become local intermediates of the current subgraph. For instance G_1^1 Fig. 5 (b) at level $l = 1$ results from merging G_1^0 and G_2^0 after elimination of their local intermediate vertices. Repeating this process for G_1^1 and G_2^1 yields G^2 as shown in Fig. 5 (c). After elimination of 3 local intermediates of G_1^2 we get the bipartite graph G_3 shown in Fig. 5 (d). It is obvious that the elimination process at level $l = 2$ proceeds serially.

2.2 Master-Slave Approach

The master-slave approach[14] consists of two steps: *elimination* and *reduction*. The former is the same as the vertex elimination on subgraph G_p yielding the local bipartite graph that corresponds to local Jacobian F'_p. This step is performed by the slaves in parallel. The latter multiplies the local Jacobians and is performed by the master. In graphical terms the

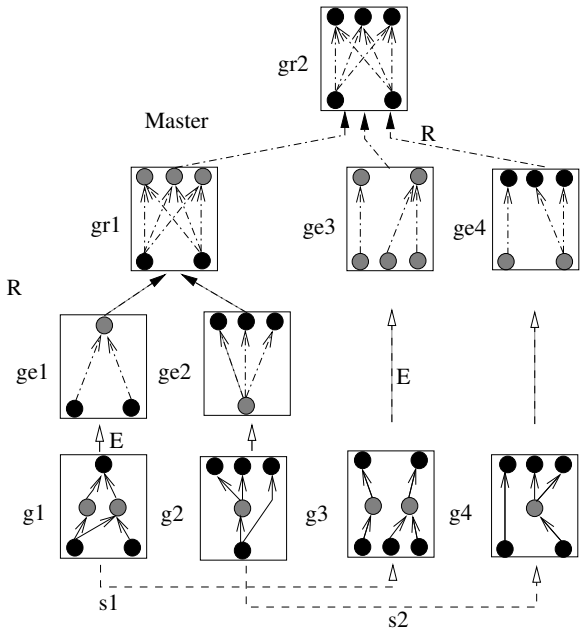


Figure 6. Parallel Jacobian accumulation by vertex elimination using Master-Slave strategy.

reduction step can be understood as the elimination of interface vertices of two neighboring local bipartite graphs. The underlying computational graph and its decomposition as shown in Fig. 6 are the same as in Fig. 5 (a). The example in Fig. 6 illustrates the master-slave idea using 2 slaves s_1 and s_2. The slaves s_1 and s_2 apply vertex elimination to the subgraph G_1 and G_1, respectively. Each slave, for instance s_1, gets the next job (subgraph) G_3 immediately after termination of the previous job. The local bipartite graphs G_1^e and G_2^e shown in Fig. 6 are the result of previous elimination. They are reduced by the master to G_1^r.

Conclusion and Numerical Results

Our implementation consists of two main steps: *symbolic* and *accumulation*. The symbolic step proceeds on a bit pattern[8] and detects the fill-in generated during the accumulation process on CRS. Different elimination techniques can yield different fill-in schemes, which have a big impact on both memory and runtime requirement of the Jacobian accumulation. Fig. 7 presents first numerical results of parallel Jacobian accumulation on CRS of the extended Jacobian of a $2D$ discretization of the Laplace equation comparing the pyramid (PYRAMID) and master-slave (MASTERSLAVE) approaches with the serial (SERIAL) version. The serial version computes the Jacobian by applying reverse elimination to the entire CRS of the underlying problem. Using two threads our parallel approaches are roughly three times faster than the serial version for large problem sizes. Partly this speedup is caused by the difference in the generated fill-in as shown in Fig. 8,

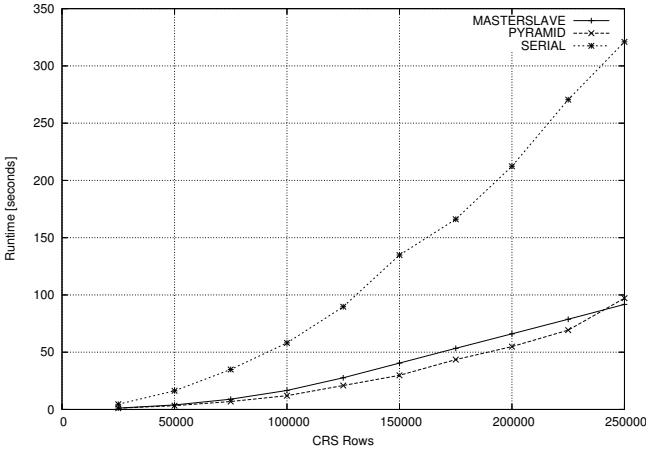


Figure 7. Runtime analysis of parallel Jacobian accumulation on CRS in reverse order against the serial version.

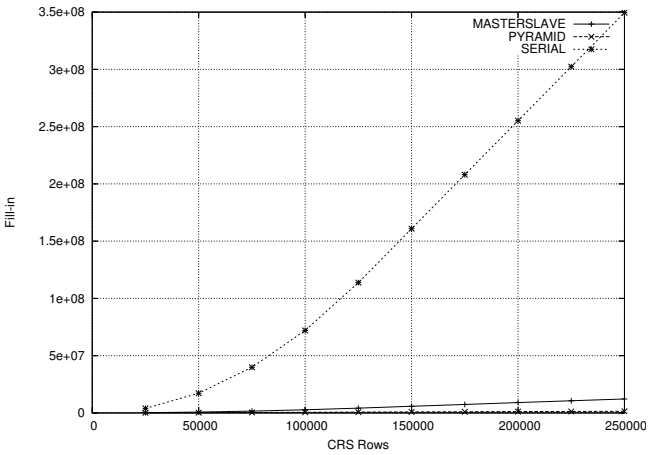


Figure 8. Fill-in comparison between SERIAL, MASTERSLAVE, and PYRAMID approaches.

which has a big impact on the runtime and memory requirement of the elimination process. The real runtime contribution of the fill-in is the subject of ongoing research. Our first results show that for parallelizing the Jacobian accumulation on CRS fill-in has to be taken into account, which makes the parallelization an even more complex task. Further work will focus on optimizing both fill-in generation and parallelization approaches thus aiming for better memory and runtime behaviour.

References

1. M. Berz, C. Bischof, G. Corliss, and A. Griewank, (Eds.), *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series, (SIAM, 1996).
2. G. Corliss and A. Griewank, (Eds.), *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series, Philadelphia, (SIAM, 1991).
3. G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, (Eds.), *Automatic Differentiation of Algorithms – From Simulation to Optimization*, (Springer, New York, 2002).
4. A. Griewank, *Evaluating Derivatives. Principles and Techniques of Algorithmic Differentiation*, Number 19 in Frontiers in Applied Mathematics, (SIAM, Philadelphia, 2000).
5. A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markowitz rule*, in: Automatic Differentiation of Algorithms: Theory, Implementation, and Application, pp. 126–135, (SIAM, Philadelphia, PA, 1991).
6. U. Naumann, *Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph*, Math. Prog., **99**, 399–421, (2004).
7. A. Griewank and S. Reese, *On the calculation of Jacobian matrices by the Markovitz rule*, in: Automatic Differentiation of Algorithms: Theory, Implementation, and Application, A. Griewank and G. F. Corliss, (Eds.), pp. 126–135, (1991).
8. E. Varnik, U. Naumann, and A. Lyons, *Toward low static memory Jacobian accumulation*, WSEAS Transactions on Mathematics, **5**, 909–917, (2006).
9. I. Duff, A. Erisman, and J. Reid, *Direct Methods for Sparse Matrices*, (Clarendon Press, Oxford, 1986).
10. U. Naumann, *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*, PhD thesis, Technical University of Dresden, (1999).
11. A. Griewank and U. Naumann, *Accumulating Jacobians as chained sparse matrix products*, Math. Prog., **3**, 555–571, (2003).
12. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*, (Morgan Kaufmann, San Francisco, 2001).
13. M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, (McGraw-Hill Education, ISE Editions, 2003).
14. C. H. Bischof, H. M. Bücker, and P.T. Wu, *Time-parallel computation of pseudo-adjoints for a leapfrog scheme*, International Journal of High Speed Computing, **12**, 1–27, (2004).

Scheduling

Layer-Based Scheduling Algorithms for Multiprocessor-Tasks with Precedence Constraints

Jörg Dümmler, Raphael Kunis, and Gudula Rünger

Chemnitz University of Technology
Department of Computer Science, 09107 Chemnitz, Germany
E-mail: {djo, krap, ruenger}@cs.tu-chemnitz.de

A current challenge in the development of parallel applications for distributed memory platforms is the achievement of a good scalability even for a high number of processors. The scalability is impacted by the use of communication operations, e.g. broadcast operations, whose runtime exhibits a logarithmic or linear dependence on the number of utilized processors. The multiprocessor-task programming model can help to reduce the communication overhead, but requires an appropriate schedule for an efficient execution. Many heuristics and approximation algorithms are available for this scheduling task. The choice of a suitable scheduling algorithm is an important factor in the development of multiprocessor-task applications. In this paper, we consider *layer-based* scheduling algorithms and compare their runtimes for large task graphs consisting of up to 1000 nodes and target systems with up to 256 processors. Furthermore, we introduce an extension methodology to enable scheduling algorithms for independent multiprocessor-tasks to handle precedence constraints.

1 Introduction

Modular applications from scientific computing can be implemented using the multiprocessor-task (M-Task) programming model with precedence constraints, which has been shown to yield better results than a pure data parallel or a pure task parallel execution, especially for distributed memory platforms with a large number of processors. In the M-Task programming model, a parallel application is defined as a set of M-Tasks where each M-Task can be executed on an arbitrary subset of the available processors of the target machine. Dependencies arise from data and control dependencies between M-Tasks. Independent M-Tasks can be executed in parallel on disjoint subsets of the available processors.

The execution of an M-Task application is based on a schedule that assigns each M-Task a subset of the available processors and fixes the execution order of the M-Tasks. In general, for a given M-Task program many different schedules are possible. Which schedule achieves the best results, i.e. leads to a minimum parallel runtime of the application, depends on the application itself and on the target platform. Therefore, for target platforms with different computation and communication behaviour, different schedules may lead to a minimum runtime. As determining the optimal schedule is an NP-hard problem, many scheduling heuristics and approximation algorithms have been proposed to get a near optimal solution to this problem.

We define two main classes of scheduling algorithms for M-Tasks with precedence constraints: *allocation-and-scheduling-based* and *layer-based* algorithms. In this paper, we examine *layer-based* scheduling algorithms. *Allocation-and-scheduling-based* algorithms have been considered in [1]. Algorithms of both categories were implemented in a scheduling toolkit that supports application developers in scheduling M-task applications.

The structure of this toolkit is described in [2]. Much theoretical and practical research has been done for scheduling sets of independent M-Tasks but most scientific applications can only be modeled with precedence constraints. Therefore, we present an extension strategy for the problem of scheduling tasks without precedence constraints to the scheduling problem with dependencies and apply this strategy to several scheduling algorithms. Our extension strategy enables the extension of any scheduling algorithm for independent M-Tasks to support precedence constraints.

The paper is structured as follows: Section 2 explains the M-Task programming model with dependencies. Section 3 outlines our extension methodology and gives an overview of the considered *layer-based* scheduling algorithms. The obtained benchmark results are discussed in Section 4. Section 5 concludes the paper.

2 Multiprocessor-Task Programming Model

In the M-Task programming model a parallel application is represented by an annotated directed acyclic graph (M-Task dag) $G = (V, E)$. An example of a small dag is given in Fig. 1. A node $v \in V$ corresponds to the execution of an M-Task, which is a parallel program part implemented using an SPMD programming style. An M-Task can be executed on any nonempty subset $g_v \subseteq \{1, \ldots, P\}$ of the available processors of a P-processor target platform. The size of a processor group $|g_v|$ is denoted as the allocation of the task v.

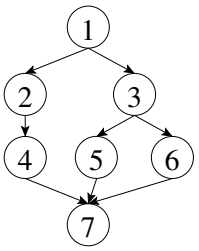


Figure 1. Example of a small M-Task dag.

A directed edge $e = (u, v) \in E$ represents precedence constraints between two M-Tasks u and v, i.e. u produces output data that v requires as input data. Therefore, u and v have to be executed one after another. Edges may lead to a data re-distribution if the processor group changes, i.e. $g_u \neq g_v$ or if u and v require different data distributions. M-Tasks that are not connected by a path in the M-Task dag can be executed concurrently on disjoint subsets of the available processors. Each node $v \in V$ is assigned a computation cost $T_v : [1, \ldots, P] \rightarrow \mathbb{R}^+$ and each edge $e = (u, v) \in E$ is assigned a communication cost $T_{comm}(u, v)$.

The execution of an M-Task application is based on a schedule S, which assigns each M-Task $v \in V$ a processor group g_v and a starting time T_{S_v}, i.e. $S(v) = (g_v, T_{S_v})$. A feasible schedule has to assure that all required input data are available before starting an M-Task, meaning that all predecessor tasks have finished their execution and all necessary data redistributions have been carried out, i.e.

$$\forall\, u, v \in V, (u, v) \in E \quad T_{S_u} + T_u(|g_u|) + T_{comm}(u, v) \leq T_{S_v}.$$

Furthermore, M-Tasks whose execution time interval overlaps have to run on disjoint processor groups, i.e.

$$\forall\, u, v \in V \quad [T_{S_u}, T_{S_u} + T_u(|g_u|)] \cap [T_{S_v}, T_{S_v} + T_v(|g_v|)] \neq \emptyset \implies g_u \cap g_v = \emptyset$$

The makespan $C_{max}(S)$ of a schedule S is defined as the point in time at which all M-Tasks of the application have finished their execution, i.e.

$$C_{max}(S) = \max_{v \in V}(T_{S_v} + T_v(|g_v|).$$

In this paper we do not take data re-distribution costs into account. This is feasible because the communication costs in scientific applications are usually a magnitude lower compared to the computational costs of the M-Tasks. Furthermore, it is often possible to hide at least parts of these costs by overlapping of computation and communication.

3 Layer-Based Scheduling Algorithms

There has been a lot of research regarding scheduling algorithms for independent M-Tasks. However, these scheduling algorithms cannot cope with precedence constraints between M-Tasks. This limitation can be avoided using layer-based scheduling algorithms[3] for M-Tasks with precedence constraints. These algorithms utilize a *shrinking phase* and a *layering phase* to decompose an M-Task dag into sets of independent M-Tasks, called layers. The subsequent *layer scheduling phase* computes a schedule for each layer in isolation. Additionally, we introduce an explicit fourth phase, the *assembling phase* constructing the output schedule for the M-Task dag. Our extension strategy enables the combination of the shrinking phase, the layering phase and the assembling phase with a scheduling algorithm for independent M-Tasks in the layer scheduling phase. Therefore, any scheduling algorithm for independent M-Tasks can be extended to support M-Task dags. The phases are illustrated in Fig. 2 utilizing the small example M-Task dag from Fig. 1.

This extension method has been applied to the following algorithms for independent M-Tasks described in Section 3.3: The scheduling algorithm with approximation factor two[4] was extended to the *Ext-Approx-2* algorithm. The *Ext-Dual-*$\sqrt{3}$ algorithm was derived from the dual-$\sqrt{3}$ approximation algorithm[5] and the dual-3/2 approximation algorithm[6] was transformed into the *Ext-Dual-*$3/2$ algorithm. A similar approach was utilized for malleable tasks[7]. This approach also executes phases to derive independent tasks, but does not include a shrinking phase and the malleable tasks consist of single processor tasks.

3.1 Shrinking Phase

The shrinking phase reduces the solution space of the scheduling problem by replacing subgraphs of the M-Task dag that should be executed on the same set of processors by a single node leading to a new dag $G' = (V', E'), |V'| \leq |V|, |E'| \leq |E|$. This phase is implemented as follows: At first we try to find all maximum linear chains in the dag G. A linear chain is a subgraph G_c of G with nodes $\{v_1, .., v_n\} \subset V$ and edges $\{e_1, .., e_{n-1}\} \subset E$, with $e_i = (v_i, v_{i+1}),\ i = 1, .., n-1$, and v_i is the only predecessor of v_{i+1} and v_{i+1} is the only successor of v_i in G. A maximum linear chain G_{c_m} is a chain which is no subgraph of another linear chain. We then aggregate the nodes and edges of each G_{c_m} to a new node v_{c_m} having the sum of the computation costs of all nodes plus the sum of the communication costs of all edges as computation cost. This aggregation implies that each G_{c_m} in G is replaced by the aggregated node v_{c_m} connecting all incoming edges of v_1 and all outgoing edges of v_n of the maximum chain G_{c_m} with the new node v_{c_m}.

3.2 Layering Phase

In the layering phase the nodes of the shrunk dag G' are partitioned into l disjoint subsets of nodes where each subset contains only independent nodes. Such a subset without precedence constraints is called a layer. In the following the nodes of a layer $i, i = 1, \ldots, l$,

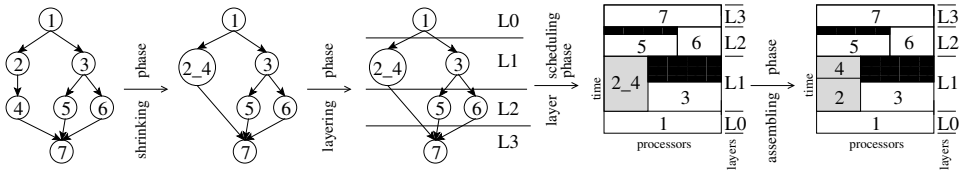


Figure 2. Illustration of the phases of the *layer-based* scheduling algorithms.

are denoted as $V_{L_i}, V_{L_i} \subseteq V'$ with $\bigcup_{i \in 1..l} V_{L_i} = V'$. Additionally, there has to exist an ordering between the layers i and k, $i, k = 1, \ldots, l$ given by:

$$\forall u \in V_{L_i}, \forall v \in V_{L_k} \text{ if there is a path from } u \text{ to } v \text{ in } G' \implies i < k.$$

The scheduling decision in the next phase becomes more flexible when the number of layers gets smaller and the number of nodes in a layer gets larger. Therefore a greedy approach is used, which performs a breadth first search and puts as many nodes as possible into a layer, to realize this phase.

3.3 Layer Scheduling Phase

In this phase an M-Task schedule is computed for each constructed layer $V_{L_i}, i = 1, \ldots, l$ in isolation. In the following we omit the index i and use V_L for the layer to be scheduled.

TwoL-Level determines the total execution time for each possible partitioning of the set of available processors into $\kappa, \kappa = 1, \ldots, \min(P, |V_L|)$ subgroups $\hat{g}_{\kappa,1}, \ldots \hat{g}_{\kappa,\kappa}$ of about equal size[3]. The schedule for each of these partitionings is computed by adopting a list scheduling heuristic. In each step of this heuristic the M-Task $v \in V_L$ is assigned to group $\hat{g}^* \in \{\hat{g}_{\kappa,1}, \ldots \hat{g}_{\kappa,\kappa}\}$, where $\hat{g}^*$ is the first subgroup becoming available and v is the M-Task with the largest execution time. The final processor groups $g_1, \ldots, g_{\kappa^*}$ are computed by a subsequent group adjustment step from the groups $\hat{g}_{\kappa^*,1}, \ldots, \hat{g}_{\kappa^*,\kappa^*}$, where κ^* denotes the partitioning resulting in a minimum runtime.

TwoL-Tree starts by constructing a tree for each M-Task $v \in V_L$ consisting of a single node[8]. A dynamic programming approach is used to find all unordered pairs of trees $\{t_1, t_2\}$ with an equal depth and disjoint sets of M-Tasks. For each pair $\{t_1, t_2\}$ a new tree t with a new root node and children t_1 and t_2 is created. Each tree represents a schedule of the contained M-Tasks. The inner nodes of the trees are annotated with a cost table containing the execution time of the whole subtree for all possible processor group sizes $gs = 1, \ldots, P$. A second annotation defines whether the schedules represented by the children of the node should be executed one after another or in parallel on disjoint processor groups. Finally, a set of trees each containing all nodes of the current layer is constructed, where each such tree symbolizes a different schedule. The output schedule of the algorithm is constructed from the tree which admits the minimum execution time.

Approx-2 is a 2-approximation algorithm for a set of independent M-Tasks[4]. It partitions the M-Tasks of a layer into the sets $P(\tau)$ and $S(\tau)$. $P(\tau)$ contains the parallel M-Tasks that are assigned a number of processors such that their execution time is smaller than τ and

$S(\tau)$ is a set of M-Tasks that are assigned a single processor. The schedule is constructed by starting all M-Tasks of $P(\tau)$ at time 0 on disjoint processor groups and by using a list scheduling heuristic to assign the M-Tasks of $S(\tau)$ to the remaining processors. The optimal value of τ is selected by using binary search on an array of candidate values.

Dual-$\sqrt{3}$ belongs to the class of dual approximation algorithms[5]. A dual-θ approximation scheduling algorithm takes a real number d as an input and either delivers a schedule with a makespan of at most $\theta * d$ or outputs that there is no schedule with a makespan $\leq d$. Dual-$\sqrt{3}$ is a 2-shelve approximation algorithm that uses a canonical list algorithm or a knapsack solver to construct the schedule. At first the algorithm determines the minimal number of processors p_v for each task $v \in V_L$ such that the execution time does not exceed d. If $\sum_{v \in V_L} p_v < P$ a canonical list algorithm is used to pack all tasks in the first shelve starting at time 0. Otherwise three subsets of the tasks are created. S_1 consists of the tasks with $T_v(p_v) > \lambda, \lambda = \sqrt{3}d - d$. In S_2 tasks with $\frac{1}{2}d \leq T_v(p_v) \leq \lambda$ and in S_3 tasks with $T_v(p_v) < \frac{1}{2}d$ are stored. S_2 and S_3 are packed into the second shelve that starts at time d. The first shelve is filled with tasks of S_1 until the sum of the needed processors gets larger than P. All remaining tasks of S_1 are packed into the second shelve with a processor number of $q_v > p_v$ determined by a knapsack algorithm.

Dual-3/2 is a dual approximation algorithm with $\theta = \frac{3}{2}$ that constructs schedules consisting of two shelves[6]. The algorithm starts by removing the set T_S of small M-Tasks from the layer and defining a knapsack problem to partition the remaining M-Tasks into the sets T_1 and T_2. The M-Tasks of T_1 and T_2 are assigned the minimal number of processors such that their execution time is less than d or $\frac{d}{2}$, respectively. The initial schedule is obtained by first inserting the M-Tasks of T_1 at start time 0 in the first shelve, the M-Tasks of T_2 at start time d in the second shelve and by assigning the small M-Tasks of T_S to idle processors. The output schedule is calculated by repeatedly applying one of three possible transformations to the schedule until it becomes feasible. These transformations include moving tasks between sets and stacking tasks on top of each other.

3.4 Assembling Phase

This phase combines the layer schedules and inserts necessary data re-distribution operations between the layers resulting in a global schedule for the M-Task dag. The layer schedules are combined following the ordering of the layers defined in the layering phase. Furthermore, the shrunk nodes, aggregated nodes of maximum linear chains G_{c_m}, are expanded, i.e. all nodes $v \in V$ of a shrunk node $v_{c_m} \in V'$ are executed one after another on the same processor group. This is realized for each G_{c_m} with nodes $\{v_1, ..v_n\} \subset V$ and edges $\{e_1, ..e_{n-1}\} \subset E$ as follows: Schedule node v_1 of the chain on processor group $g_{v_{c_m}}$ at time $T_{S_{v_{c_m}}}$. The following nodes $v_i, i = 2, \ldots, n$ of G_{c_m} are scheduled on $g_{v_{c_m}}$ at time $T_{S_{v_{i-1}}} + T_{v_{i-1}}(|g_{v_{c_m}}|) + T_{comm}(v_{i-1}, v_i)$ including data re-distributions between two nodes v_{i-1} and v_i.

4 Simulation Results

The benchmark tests presented in this section are obtained by running the scheduling toolkit on an Intel Xeon 5140 ("Woodcrest") system clocked at 2.33 GHz. For each number of nodes we constructed a *test set* consisting of 100 different synthetic M-Task dags

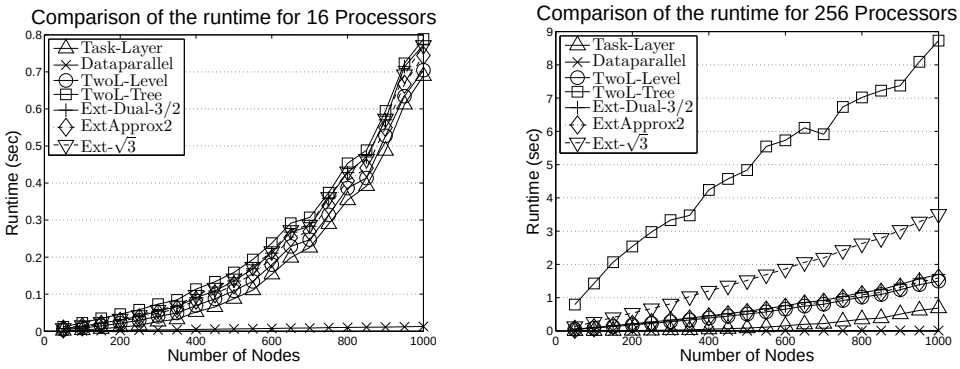


Figure 3. Comparison of the average runtime of the *layer-based* scheduling algorithms for task graphs with 50 to 1000 nodes and 16 (left) and 256 (right) available processors.

belonging to the class of series-parallel-graphs (SP-graphs). SP-graphs reflect the regular structure of most scientific applications. All nodes of the graphs are annotated by a runtime estimation formula according to Amdahl's law ($T_{par} = (f + (1-f)/p) * T_{seq}$), which describes the parallel runtime T_{par} on p processors for a problem with an inherent sequential fraction of computation $f (0 \leq f \leq 1)$ and a runtime on a single processor $T_{seq} (T_{seq} > 0)$.

First we present the runtime of the implemented scheduling algorithm averaged over each test set. For a comparison with a pure data or task parallel execution we use *Data-parallel* and *Task-Layer* respectively. *Task-Layer* uses a list scheduling algorithm in the layer scheduling phase that assigns each node of the shrunk dag one processor and schedules the tasks in parallel. Figure 3 shows the runtime of the scheduling algorithms for 16 (left) and 256 (right) processors depending on the number of nodes. In both cases *Data-parallel* achieves the lowest runtimes and *TwoL-Tree* achieves the highest runtimes. For 16 available processors the schedulers, except *Dataparallel*, achieve nearly the same runtime that is below one second for 1000 nodes and show a similar behaviour if the number of nodes is increased. *Task-Layer* achieves the second lowest runtimes that are in average 20 times smaller than that of *Dataparallel*. All other algorithms have a nearly constant deviation from the runtimes of *Task-Layer* of at most 0.1 seconds. In the case of 256 available processors *TwoL-Tree* achieves the highest runtimes. The runtime is a factor of 3 to 9 higher compared to the other schedulers for an M-Task dag with 1000 nodes. *Ext-Dual-*$\sqrt{3}$ is the second slowest scheduler followed by *TwoL-Level*, *Ext-Approx-2* and *Ext-Dual-*$3/2$ that have significantly lower runtimes. *Dataparallel* and *Task-Layer* again achieve the lowest runtimes. The runtimes of the schedulers for 256 available processors increases linearly in the number of nodes of the dag. This behaviour results from the linear runtimes of the shrinking phase, the layering phase, and the assembling phase, which have a worst case runtime of $\mathcal{O}(V + E)$. The layer scheduling phase is only executed for small subsets V_L of the available nodes V ($|V_L| \ll |V|$). Additionally, the runtime of the layer scheduling phase is dominated by the number of processors P, because $P > |V_L|$ for many layers.

Figure 4 shows the makespan of the produced schedules for 16 (left) and 256 (right) processors depending on the number of nodes. The results for 16 available processors show that *TwoL-Tree* and *TwoL-Level* produce the lowest makespans with a deviation of 1.8%. They are followed by the three adapted approximation algorithms *Ext-Dual-*$3/2$,

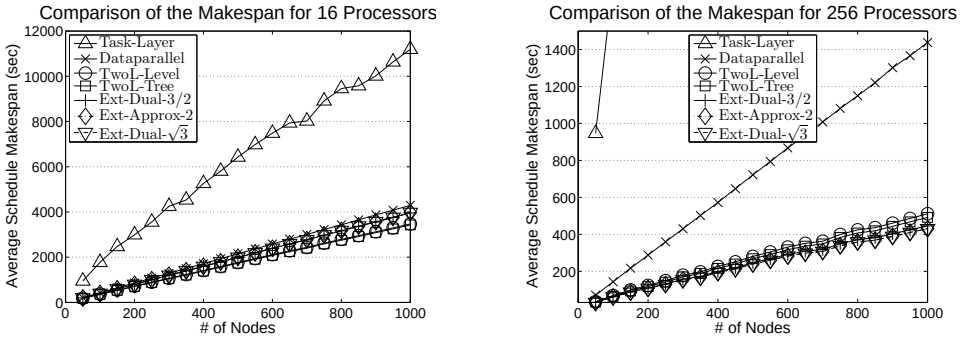


Figure 4. Comparison of the average makespan of the *layer-based* scheduling algorithms for task graphs with 50 to 1000 nodes and 16 (left) and 256 (right) available processors.

Ext-Dual-$\sqrt{3}$, and *Ext-Approx-2* with 14.5%, 14.7%, and 14.7% higher makespans. The makespan of the schedules produced by *Dataparallel* are also competitive (23.5% higher than the ones from *TwoL-Tree*). The worst schedules were produced by *Task-Layer* with a 281% higher average makespan than *TwoL-Tree*. The results show that the makespans of the produced schedules increase linearly in the number of nodes.

Considering the results for target machines with 256 processors, Fig. 4 (right) shows that the two TwoL schedulers changed their places with the adapted approximation algorithms. The best average schedules were produced by *Ext-Dual-*$\sqrt{3}$ and *Ext-Approx-2* with a very small deviation, followed by *Ext-Dual-*$3/2$ with 2.8% higher makespans. *TwoL-Tree* and *TwoL-Level* have a 11.6% and 16.9% higher average makespan. The results show again that both, a pure data parallel and a pure task parallel execution are clearly outperformed by all other scheduling algorithms. The makespan of *Task-Layer* are in average 27 times smaller than the ones from *Ext-Dual-*$\sqrt{3}$ and *Ext-Approx-2* which produce the best schedules. *Dataparallel* creates schedules with an average makespan that is 3 times worse compared to *Ext-Dual-*$\sqrt{3}$. These results also show that the makespans of the produced schedules increase linearly in the number of nodes.

Table 1 shows the speedups of the produced schedules compared to *Dataparallel*. It shows that except *Task-Layer* all scheduling algorithms construct better schedules than *Dataparallel* on average. Also we have examined that the schedule with the lowest makespans were never created by *Task-Layer* or *Dataparallel*. The best schedules were produced by *TwoL-Tree* for 16 available processors and by *Ext-Approx-2*, *Ext-Dual-*$\sqrt{3}$ or *Ext-Dual-*$3/2$ for 256 available processors. The obtained results of the runtimes of the algorithms and the quality of the schedules lead to the recommendations given in Table 2. The recommendations quote which algorithm should be utilized in which situation. A situation is determined by the attributes: size of the dag, number of available processors of the target machine, and the choice whether the best solution should be found or a good trade-off between quality and runtime should be reached.

5 Conclusion

In this paper we proposed an extension methodology for scheduling algorithms for independent M-Tasks to handle M-Tasks with precedence constraints, which is accomplished

Sched./Procs	16	64	128	256
Task Layer	0.33	0.16	0.13	0.11
TwoL-Level	1.21	1.72	2.12	2.54
TwoL-Tree	1.24	1.80	2.24	2.66
Ext-Dual$3/2$	1.08	1.73	2.31	2.90
Ext-Approx-2	1.08	1.82	2.43	2.98
Ext-Dual-$\sqrt{3}$	1.08	1.82	2.43	2.98

Table 1. Speedups of the produced schedules relative to the *Dataparallel* Scheduler.

	low number of processors	high number of processors
low number of nodes	*TwoL-Tree** *TwoL-Level***	(*Ext-Dual*-$3/2$, *Ext-Dual*-$\sqrt{3}$ *Ext-Approx2*)*,**
high number of nodes	*TwoL-Tree** *TwoL-Level***	(*Ext-Dual*-$\sqrt{3}$, *Ext-Approx2*)* (*Ext-Dual*-$3/2$, *Ext-Approx2*)**

* best quality ** good quality, reasonable runtime

Table 2. Recommended scheduling algorithms for different situations.

by a *layer-based* approach. Additionally, we presented a comparison of the extended algorithms with existing *layer-based* scheduling algorithms and derived a guideline, which algorithm developers should utilize depending on the parallel application and target platform. For this purpose we took the runtime of the scheduling algorithms as well as the quality of the generated schedules into account. The results show that a mixed task and data parallel execution derives better results than a pure data and task parallel execution in all considered cases. Especially for a large number of processors, the M-Task approach results in much higher speedups.

References

1. J. Dümmler, R. Kunis and G. Rünger, *A comparison of scheduling algorithms for multiprocessortasks with precedence constraints*, in: Proc. 2007 High Performance Computing & Simulation (HPCS'07) Conference, pp. 663–669, (2007).
2. J. Dümmler, R. Kunis and G. Rünger, *A scheduling toolkit for multiprocessor-task programming with dependencies*, in: Proc. 13th Int. European Conf. on Par. Distr. Comp. (Euro-Par 07), *LNCS*, vol. **4641**, Rennes, France, (2007).
3. T. Rauber and G. Rünger, *Compiler support for task scheduling in hierarchical execution models*, J. Syst. Archit., **45**, 483–503, (1998).
4. W. Ludwig and P. Tiwari, *Scheduling malleable and nonmalleable parallel tasks*, in: SODA '94: Proc. fifth annual ACM-SIAM symposium on Discrete algorithms, pp. 167–176, Society for Industrial and Applied Mathematics. (1994).
5. G. Mounie, C. Rapine and D. Trystram, *Efficient approximation algorithms for scheduling malleable tasks*, in: SPAA '99: Proc. eleventh annual ACM symposium on Parallel algorithms and architectures, pp. 23–32, (ACM Press, 1999).
6. G. Mounie, C. Rapine and D. Trystram, *A $\frac{3}{2}$-approximation algorithm for scheduling independent monotonic malleable tasks*, SIAM J. on Computing, **37**, 401–412, (2007).
7. W. Zimmermann and W. Löwe, *Foundations for the integration of scheduling techniques into compilers for parallel languages*, Int. J. Comp. Science & Engineering, **1**, (2005).
8. T. Rauber and G. Rünger, *Scheduling of data parallel modules for scientific computing.*, in: Proc. 9th SIAM Conf. on Parallel Processing for Scientific Computing (PPSC), SIAM(CD-ROM), San Antonio, Texas, USA, (1999).

Unified Scheduling of I/O- and Computation-Jobs for Climate Research Environments

N. Peter Drakenberg[1] **and Sven Trautmann**[2]

[1] German Climate Computing Center
Bundesstraße 55, 20146 Hamburg, Germany
E-mail: drakenberg@dkrz.de

[2] Max Planck Institute for Meteorology
Bundesstraße 53, 20146 Hamburg, Germany
E-mail: sven.trautmann@zmaw.de

Scheduling I/O- and computation jobs is a key factor to efficiently operate large cluster systems. After outlining the necessity of scheduling I/O- and computation jobs in a climate research environment, we present a methodology to schedule I/O jobs depending on the current load on a parallel file system (Lustre). Starting with a system description and some file system benchmarks, we present how to integrate an I/O load sensor into the Sun grid engine. Furthermore we exemplary present how the I/O job scheduling is realized on the *Tornado* Linux cluster system at the German Climate Computing Center (DKRZ) in Hamburg.

1 Introduction

Climate simulation is not only used to predict global warming and its consequences, just like weather prediction, climate simulation is of significant importance to many areas of human society. Notable current and future examples being prediction of outbreaks of infectious diseases such as cholera[1] and malaria[2], prediction of agricultural conditions[3], and of course prediction of "difficult" seasonal weather conditions.[4]

In order to achieve reasonable run times (and waiting times), climate simulations are typically run on high-performance computer systems (*e.g.*, vector supercomputers or large clusters), and in addition to using large amounts of processor time, climate simulations also consume and produce large volumes of data.

The cluster system recently installed at DKRZ, provides more than 1000 processor cores, a parallel file system (*Lustre*[5]) with a capacity of about 130 Terabytes, and runs the Linux operating system on all nodes. The cluster has had to be integrated into an existing environment with a shared file system (Sun QFS/SamFS) providing about 75 Terabyte of storage capacity and an underlying storage system with over 4 Petabyte capacity.

As a user front-end to the cluster we use the Sun Grid Engine (SGE)[6]. The SGE is a resource management tool, the purpose of which is to accept *jobs* submitted by users, and schedule the jobs to run on appropriate systems available to it and in accordance with defined resource management policies.

In this paper, we present and explain a sample climate model simulation run. Followed by a description of some system details, we present benchmarks to decide which version of data transfer should be used by I/O jobs. We then describe how data moving jobs can be tightly integrated into the Sun Grid Engine and be scheduled to minimize impact on computation jobs. For user convenience, we provide a set of small tools which enable

the user to copy/move data from one file system to the other. We conclude the paper by discussing future improvements to our I/O job scheduling approach.

2 Climate Research Scenarios

As in most sciences, earth-system researchers and meteorologists (*e.g.*, at national weather agencies) use partial differential equations to describe the behaviour of the earth and/or its constituent components (*e.g.*, oceans, atmosphere, *etc*). For example, the evolution of momentum in sea-water (*i.e.*, in oceans) is often expressed as:

$$\frac{\partial}{\partial t}\mathbf{v} + f(\mathbf{k}\times\mathbf{v}) + g\nabla\zeta + \mathbf{v}\cdot\nabla\mathbf{v} + w\cdot\frac{\partial}{\partial z}\mathbf{v} = -\frac{1}{\rho_0}\nabla p + \mathbf{F}_{\mathrm{H}} + \mathbf{F}_{\mathrm{V}},$$

$$\frac{\partial\zeta}{\partial t} + \nabla\left(\int_{z=-H}^{z=\zeta}\mathbf{v}\,\mathrm{d}z\right) = 0,$$

$$\frac{\partial p}{\partial z} = -g\rho$$

where $(\mathbf{v}, w)$ is the velocity vector in a spherical coordinate system (λ, θ, z), f ($f = f(\theta)$) is the Coriolis parameter, $\mathbf{k}$ is an upward vertical unit vector, g is the gravitational acceleration of the earth, ζ is the sea surface elevation, ρ_0 and ρ are the mean sea water density and deviations from that density, respectively, and p is the internal pressure.

Earth-system research and meteorology are primarily concerned with relatively large-scale phenomena. The effects of small-scale phenomena are accounted for through empirically determined parameters that express the ensemble effect of sub-grid scale phenomena in terms of the resolved grid-scale variables. The combination of a set of equations, values of associated parameters, and the method and details of discretization is referred to as a climate/ocean/atmosphere *model*. The execution of a program that numerically solves the discretized equations of some model is referred to as a *model run*.

Model runs are associated with large (for low resolutions) to vast (for high resolutions) volumes of data. To begin with, the (irregular) domains must be specified on which the unknown functions, such as $\mathbf{v}$, ζ, *etc* (*vide supra*), are defined. Additional values are associated with parameters and boundary conditions, and time-varying boundary conditions (*e.g.*, prescribed sea-surface temperatures for an atmosphere model run) in particular. During the model run, the values of unknowns must be maintained for all grid-points/mesh-cells in their domains of definition.

Finally, the values corresponding to the properties of interest are written to files at regular intervals, typically every 10th or every 20th time step. For most model-runs, the generated output is several orders of magnitude larger than all input combined, and illustrates the climate research community's need for high-speed high-capacity file systems.

As a concrete example we consider the model runs performed by the Max-Planck-Institute for Meteorology at the German Climate Computing Center (DKRZ) for the Fourth Assessment Report (AR4)[7] of the Intergovernmental Panel on Climate Change (IPCC). In agreement with the requirements of the IPCC, model runs were performed for three scenarios, denoted A1B, A2 and B1. Using the ECHAM5[8]/MPI-OM[9] coupled model at the T63L31 resolution, 18 individual experiments were realized with a total simulated time of approximately 5000 years. The corresponding model runs consumed 400,000 CPU-hours (on a NEC SX-6[10] system) and produced 400 Terabytes of data. The 192 processor

SX-6 system at DKRZ has a peak performance of 1.5 Teraflop/s and typically runs climate models with $\sim$30% efficiency. The cluster on the other hand, has a peak performance of 5.8 Tflop/s, but typically runs climate models with only $\sim$5% efficiency. Thus, we expect the cluster to be capable of a sustained data-production rate of $\sim$125 Gigabyte/hour, when performing the type of computations for which it is primarily intended.

3 System Description

The system we used for our study is the *Tornado* Linux cluster system operated by the German Climate Computing Center (DKRZ) in Hamburg, Germany. It consists of

- 256 dual core dual socket compute nodes (Sun Fire X2200),
- five dual core eight socket SMP head nodes (Sun Fire X4600) and
- a number of additional nodes providing an management infrastructure and a Lustre file system.

In this paper, we focus on the Lustre file system[5] and two dedicated data mover nodes. A Lustre file system (version 1.6) consists of a number of object storage targets and one meta data target for each of the provided file systems. The object storage targets belonging to one file system are combined to form a logical volume. The *Tornado* cluster provides two Lustre file systems with a combined capacity of approximately 130 TB. The underlying hardware system consists of eight Sun Fire X4500 devices with 48 $\times$ 500 GB hard disks each and a fail over meta data server configuration. The Lustre file systems are accessible from all compute and head nodes using a DDR-InfiniBand interconnect.

Since the Tornado cluster system had to be integrated into an existing infrastructure providing QFS file systems and an underlying HSM system, we had to establish a gateway between the cluster and HSM system. For this purpose the *Tornado* cluster system is equipped with two dedicated data mover nodes (Sun Fire X4200) with access to both the Lustre file system and the QFS (see Fig. 1).

Our first working data mover configuration was running SuSE Linux 9 with service pack 2. The read and write performance to the QFS was impressive with up to 380 Megabyte per second for writing and 325 Megabyte per second for reading operations on an multi-path QFS configuration. To our knowledge, this represented the first installation where *both* Lustre and QFS could be used on one machine with read and write access.

Due to some bugs in the old InfiniBand stack, we had to update the whole cluster system to a newer version, and the new InfiniBand stack was no longer running on SLES 9 SP 2. Since QFS is only supported on a limited number of Linux distributions, we had to update the data mover nodes to SLES 10 and QFS to version 4.6. Unfortunately, we have not yet reached the performance of the previous installation (see next sections for more details). Another drawback of the new QFS version is that we have to use a patchless Lustre client, which has a negative influence on the Lustre file system performance.

4 File System Benchmarks

From a performance point of view, the system's hardware gives an upper limit on the amount of data which can be transferred from one file system to the other in a given period

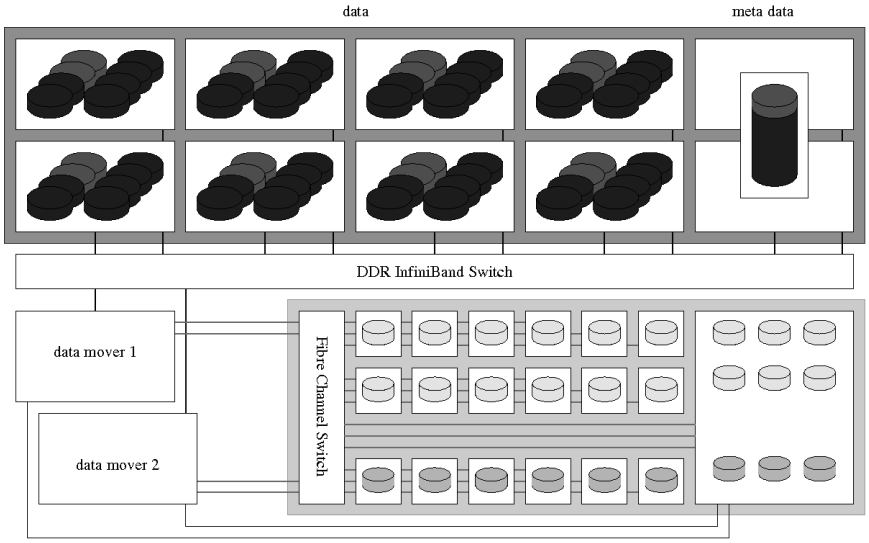


Figure 1. File systems of the *Tornado* Linux cluster system

of time. Both data mover nodes are equipped with two dual-core AMD Opteron processors, one InfiniBand host channel adapter (HCA) and one dual-port Fiber-Channel host bus adapter (HBA) providing two 2 Gigabit ports to the QFS storage network. The InfiniBand HCA limits the throughput to 2 Gigabyte per second. In contrast, the Fiber-Channel HBA allows only 400 Megabyte per second if both ports can be utilized.

To get conclusive performance results we had to study different scenarios. We started by measuring the data rates which can be achieved on each file system. Since the file systems are in production mode already (and remain so), we have always load influences. The second step was to find the best way to copy/move data from one file system to the other, depending on the number of files to transfer and their size. In the third step we studied if and how the concurrent access to both file systems influenced the throughput.

For raw performance measurements we started with the *IOZone*[11] file system benchmark tool which measures the performance for file systems at a time. In Fig. 2 the output of the IOZone benchmark is shown. Lustre delivers up to 1.2 GB/s when reading from the file system and more than 600 MB/s when writing to the file system. In contrast, reading from the QFS is limited to about 180 MB/s. Writing to the QFS can be done with up to 600 MB/s. We assume cache effects here due to hardware limits. The read and write performance of the Lustre file system depends on the file size to transfer[a].

Based on this results we study different methods to transfer data between the file systems by comparing ways of moving data on operating system level with varying parameter sets. As a starting point we use the systems `cp` command and study which transfer rates can be achieved depending on the file size. Table 1 shows the results. In comparison with

[a] File systems details are as follows: *qfs1* file systems provides 44 TB of data with currently 70% capacity utilization. *qfs2* has a size of 32 TB with 94% used. The Lustre file systems provide 32 TB and 94 TB with an utilization of 9% and 32%, respectively.

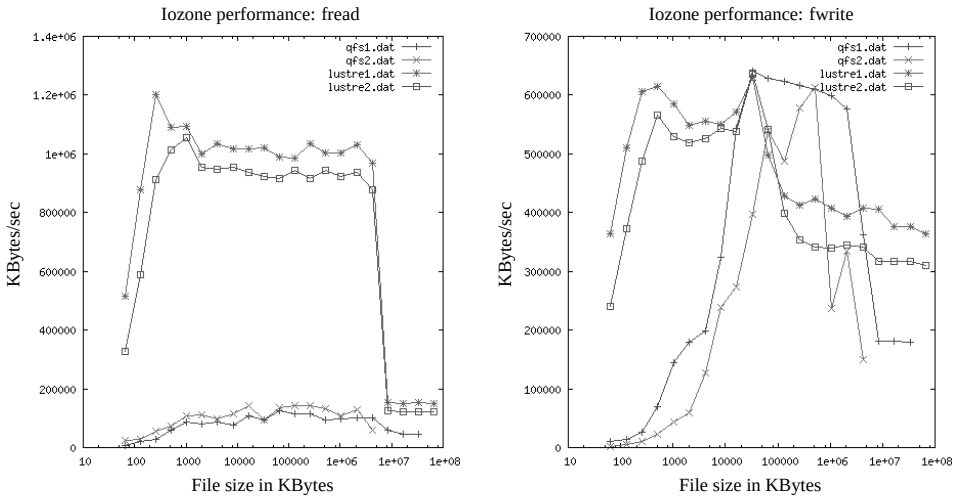


Figure 2. IOZone file system benchmark for QFS and Lustre file systems on *Tornado*

file systems used	1M	4M	16M	64M	256M	1G	4G	16G	64G
qfs1 → lustre1	21.7	30.8	37.6	38.3	56.7	79.4	74.7	65.3	66.5
qfs1 → lustre2	50.0	61.5	80.0	93.0	102.0	79.0	79.1	74.5	68.1
qfs2 → lustre1	22.2	30.1	38.3	35.1	32.4	35.0	31.5	30.0	30.6
qfs2 → lustre2	20.4	20.5	30.0	30.6	36.0	29.6	29.6	31.7	29.7
lustre1 → qfs1	9.0	28.8	58.4	76.7	81.8	80.2	81.9	87.3	93.2
lustre1 → qfs2	5.9	20.6	36.8	51.4	61.0	73.8	73.2	80.4	76.5
lustre2 → qfs1	11.4	25.8	50.6	69.3	76.5	76.9	79.0	79.6	79.6
lustre2 → qfs2	8.5	18.3	34.9	55.0	65.9	65.3	63.2	69.4	66.0

Table 1. Throughput using cp in MB/s

the IOZone benchmark the results achieved using the `cp` command are much lower. To get better performance, we study if and how using a different block sizes can improve the throughput. For this purpose we use the `dd` command which allows to specify different block size for input and output.

We studied all eight file system combinations as before for the `cp` command, five different file sizes (16M, 256M, 1G, 4G, 16G) and six different block sizes (4K, 16K, 64K, 256K, 1M, 4M) for both input and output. Table 2 shows the maximum throughput and Table 3 the corresponding values for input and output block size.

The maximum throughput was reached when copying data from the Lustre file systems to the *qfs1* file system. Since the throughput is much higher for the `dd` command we decided to use it instead of the `cp` command. Since the main purpose of the data mover nodes is to copy data from the Lustre file system to the QFS, we decided to use a block size of 4 Megabyte for reading the Lustre file system and a block size of 256k to write to the QFS.

file systems used	16M	256M	1G	4G	16G
qfs1 → lustre1	116.0	114.0	95.8	88.8	83.8
qfs1 → lustre2	119.0	113.0	112.0	91.5	87.0
qfs2 → lustre1	34.3	113.0	32.9	35.5	37.6
qfs2 → lustre2	120.0	116.0	103.0	91.9	36.9
lustre1 → qfs1	60.8	173.0	85.3	111.0	83.2
lustre1 → qfs2	47.4	111.0	68.2	87.5	74.1
lustre2 → qfs1	81.3	184.0	108.0	123.0	104.0
lustre2 → qfs2	51.0	117.0	85.9	95.7	89.5

Table 2. Maximum throughput (in MB/s) reached using the `dd` command

file systems used	16M	256M	1G	4G	16G
qfs1 → lustre1	64k/256k	16k/256k	64k/256k	16k/256k	1m/256k
qfs1 → lustre2	64k/64k	4k/256k	64k/256k	64k/64k	256k/256k
qfs2 → lustre1	256k/256k	256k/256k	64k/1m	256k/64k	64k/256k
qfs2 → lustre2	256k/256k	256k/64k	64k/64k	256k/256k	256k/256k
lustre1 → qfs1	256k/256k	4m/256k	256k/16k	256k/64k	1m/256k
lustre1 → qfs2	4m/256k	4m/64k	4m/1m	4m/64k	1m/64k
lustre2 → qfs1	256k/64k	64k/64k	256k/4k	256k/16k	256k/64k
lustre2 → qfs2	256k/64k	4m/16k	4m/64k	1m/256k	1m/64k

Table 3. Input/output buffer size used to reach maximum throughput using the `dd` command

5 Grid Engine Integration

In its standard configuration, the SGE keeps track of and manages resources such as CPU time, physical memory, process segment sizes, load averages, *etc*. The scheduling of a job is determined by factors such as the job's resource requirements, the job's waiting time, the job's deadline (if any), the job owner's assigned share entitlements of resources, *etc*.

In our configuration, the SGE also keeps track of the I/O load on the Lustre file system and uses this information to run data transfer jobs on the combined Lustre/QFS nodes (see above) at times when the I/O load on the Lustre file system (due to compute jobs) is not high. The Lustre I/O load was made visible to the SGE by defining a so-called consumable attribute, named `ioload`, in the SGE (see [12, pp. 67–86]) and by implementing a custom load sensor. Every 30 seconds, the custom load sensor gathers the volume (no. of bytes) of data read and written per second on each object storage server (OSS) of the Lustre file system (this information is made available by Lustre in `/proc/fs/lustre/ost/OSS/ost_io/stats` on each OSS). With the volume of data read and written on the ith OSS ($i \in \{1,...,8\}$) denoted by dr_i and dw_i, respectively, the load sensor calculates the currently available Lustre I/O capacity ioc as:

$$ioc = 1000.0 - \{\max_{1 \le i \le 8}(w_r dr_i + w_w dw_i)\}/2^{20} \quad (5.1)$$

and reports this value to the SGE as the current value of the attribute `ioload`. In Eq. (5.1), w_r and w_w are weight factors (at present $w_r = 1.0$, $w_w = 2.0$), and the value 1000.0 was

chosen as the available I/O capacity of an idle Lustre file system based on the file system I/O benchmark results reported in Section 4.

In addition to the matters described above, an SGE job queue named `qfs` has been defined, for which the combined Lustre/QFS nodes are the only execution hosts. The `qfs` queue is configured with `ioload` as a consumable attribute (base value: 1000.0), and with `ioload` values of 200.0 and 100.0 as load- and suspend-thresholds, respectively. As a result of this setup, the specified use of the `ioload` resource by jobs running in the `qfs` queue is deducted from the current value of the `ioload` consumable (while each job is running), and the specified thresholds prevent the `qfs` queue from overusing the Lustre file system. Furthermore, and more importantly, the load- and suspend-thresholds are applied to the minimum of the `ioload` sensor value and the `ioload` consumable, and will thereby prevent jobs in the `qfs` queue form starting or suspend running data transfer jobs, in order to keep the available Lustre I/O capacity at a sufficient level to minimize the impact on running computations.

6 Experiences

Model runs at DKRZ are typically performed as job-chains, with the last action of a running job being to submit a new instance of itself that resumes the model run where the current job left off, occasionally interleaved with (non-parallel) post-processing jobs and/or data transfer operations. For this context and environment, the commands `qfs-cp` and `qfs-mv` have been implemented to transfer data between the Lustre and QFS file systems. These commands first determine the volume of data and the expected duration of the transfer, and then create a job-script to perform the actual file transfers, that is subsequently submitted to the SGE with the consumption of the `ioload` resource and an upper limit on run-time specified. A benefit of this scheme is that it enables data transfers to be decoupled from computations, allowing both to proceed independently and in parallel (also for data transfers initiated by computational jobs).

7 Conclusion and Future Work

As mentioned in Section 5, the purpose of the work described here has been to find a viable way of transferring data away from the cluster's file-systems, while minimizing the impact on computations. In other words, our objective has been to schedule and run pure I/O jobs without disturbing compute jobs.

The primary constraint in the present context is a lack of information.[b] Ideally (given the repetitive nature of our computations), each compute job would indicate its patterns of interconnect usage (*e.g.*, MPI communication) as well as how often and how much output it will generate. The fact that information of this kind will not be provided by queued and/or running jobs, has led us to adopt the relatively simple scheme for I/O-job

[b] For pure compute jobs, detailed and accurate information about expected resource utilization, appears not to be essential for good job scheduling.[13] In part, this is due to the implicitly uniform use of the primary resource (*i.e.*, CPUs). I/O (by compute jobs), on the other hand, is often done in bursts at regular intervals that are well separated in time (*e.g.*, after every n timesteps, for some fixed n), and accurate information about such behavioural properties (of compute jobs) would clearly be useful for scheduling I/O jobs.

scheduling described above. However, despite the simplicity of our scheme, we have found the SGE to be insufficiently flexible for some of our needs, and these limitations appear to be present also in many other job-scheduling systems. In particular, persistent increases *and* decreases of resources (*e.g.*, available disk space) as consequences of scheduling and running jobs can not be properly handled by the SGE.

In our work thus far, we have introduced a Lustre I/O load sensor to the *Sun Grid engine*, developed user-level commands that handle I/O job creation and submission, and we have evaluated different means of performing the actual data transfers. As a result of the latter evaluation, the I/O jobs created by our user-level commands use the `dd` command, with optimized parameters, to perform data transfers. For the future we plan to not only monitor the file system load, but also the InfiniBand network load, to decide if an I/O job should be executed at a given point in time or not. This approach will further reduce the effect of I/O jobs on computational jobs, since the same InfiniBand network is used for both I/O and message-passing communication.

References

1. K. Koelle et al., *Refractory periods and climate forcing in cholera dynamics*, Nature, **436**, 696–700, (2005).
2. M. C. Thomson et al., *Malaria early warnings based on seasonal climate forecasts from multi-model ensembles*, Nature, **439**, 576–579, (2006).
3. F. S. Royce et al., *Model-based optimization of crop management for climate forecast applications*, Trans. Am. Soc. Agr. Eng., **44**, 1319–1327, (2001).
4. D. R. Easterling et al., *Climate extremes: observations, modelling, and impacts*, Science, **289**, 2068–2074, (2000).
5. Cluster File Systems, Lustre homepage, http://clusterfs.com.
6. Sun Microsystems, Gridengine homepage, http://gridengine.sunsource.net/.
7. S. Solomon, D. Qin, M. Manning, M. Marquis, K. B. Averyt, M. Tignor, H. L. Miller and Z. Chen, (Eds.), *Climate Change 2007: The Physical Science Basis.*, (Cambridge University Press, Cambridge, 2007).
8. E. Roeckner et al., *The atmospheric general circulation model ECHAM5: Part I, model description*, Report No. 349, Max-Planck-Institut für Meteorologie, Bundesstraße 53, Hamburg, Germany, (2003).
9. S. J. Marsland, H. Haak, J. H. Jungclaus, et al., *The Max-Planck-Institute global ocean/sea ice model with orthogonal curvilinear coordinates*, Ocean Modelling, **5**, 91–127, (2003).
10. K. Kitagawa, S. Tagaya, Y. Hagihara and Y. Kanoh, *A Hardware Overview of SX-6 and SX-7 Supercomputer*, NEC Res. & Develop., **44**, 2–7, (2003).
11. W. Norcott and D. Capps, IOZone Filesystem Benchmark, http://www.iozone.org.
12. Sun Microsystems, Inc., Santa Clara, CA 95054, *N1 Grid Engine 6 Administration Guide*, (2005), Part No: 817-5677-20.
13. A. W. Mu'alem and D. G. Feitelson, *Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*, IEEE Trans. Parall. Distrib. Sys., **12**, 529–543, (2001).

Fault Tolerance

Towards Fault Resilient Global Arrays

Vinod Tipparaju, Manoj Krishnan, Bruce Palmer, Fabrizio Petrini, and Jarek Nieplocha

Pacific Northwest National Laboratory
Richland, WA 99352, USA
E-mail: {vinod, manoj, bruce.palmer, fabrizio.petrini, jarek.nieplocha}@pnl.gov

This paper describes extensions to the Global Arrays (GA) toolkit to support user-coordinated fault tolerance through checkpoint/restart operations. GA implements a global address space programming model, is compatible with MPI, and offers bindings to multiple popular serial languages. Our approach uses a spare pool of processors to perform reconfiguration after the fault, process virtualization, incremental or full checkpoint scheme and restart capabilities. Experimental evaluation in an application context shows that the overhead introduced by checkpointing is less than 1% of the total execution time. A recovery from a single fault increased the execution time by 8%.

1 Introduction

As the number of processors for high-end systems grows to tens or hundred of thousands, hardware failures are becoming frequent and must be handled in such manner that the capability of the machines is not severely degraded. The development of scalable fault tolerance is critical to the success of future extreme-scale systems. In addition to general and fully automated approaches[1] we recognize that some classes of applications in chemistry, bioinformatics, data mining, and Monte Carlo simulations have a natural fault resiliency providing that some critical data is protected from loss and corruption due to hardware faults. For such applications, fault recovery can be much less expensive and consume less system resources than otherwise would be required in the general case.

There has been a considerable amount of work aiming at achieving fault tolerance in MPI. Using MPI programming model, the programmer must explicitly represent and manage the interaction between multiple processes, distribute and map parallel data structures, and coordinate the data exchanges through pairs of send/receive operations. For the upcoming massively parallel systems with complex memory hierarchies and heterogeneous compute nodes, this style of programming leads to daunting challenges when developing, porting, and optimizing complex multidisciplinary applications that need to demonstrate performance at a petascale level. These productivity challenges in developing complex applications with MPI have resulted in renewed interest in programming models providing a shared global-address data view such as UPC, Co-Array Fortran or Global Arrays (GA). The focus of the current paper is adding fault resiliency to the Global Arrays. The Global Array (GA) toolkit[2] enables scientific applications use distributed memory systems as a single global address space environment. The user has ability to create and communicate through data objects called global arrays in way like they were located in shared memory. GA has been adopted by multiple application areas and is supported by most hardware vendors, including IBM for the fastest system on Top-500 list, the IBM BlueGene/L[3].

We extended the GA toolkit to provide capabilities that will enable programmer to implement fault resiliency at the user level. Our fault-recovery approach is programmer

assisted and based on frequent incremental checkpoints and rollback recovery. In addition, it relies on a pool of spare nodes that are used to replace the failing nodes. This approach is consistent with that of FT-MPI[4], an MPI implementation that handles failure at the MPI communicator level and allows the application to manage the recovery by providing corrective options such as shrinking, rebuilding or aborting the communicator. We demonstrate usefulness of fault resilient Global Arrays in context of a Self Consistent Field (SCF) chemistry application. On our experimental platform, the overhead introduced by checkpointing is less than 1% of the total execution time. A time to recover from a single fault increased the execution time by only 8%.

2 Programming Model Considerations

The demand for high-end systems is driven by scientific problems of increasing size, diversity, and complexity. These applications are based on a variety of algorithms, methods, and programming models that impose differing requirements on the system resources (memory, disk, system area network, external connectivity) and may have widely differing resilience to faults or data corruption. Because of this variability it is not practical to rely on a single technique (e.g., system initiated checkpoint/restart) for addressing fault tolerance for all applications and programming models across the range of petascale systems envisioned in the future. Approaches to fault tolerance may be broadly classified as user-coordinated or user-transparent. In the latter model, tolerance to hardware failures is achieved in a manner that is completely transparent to the programmer, where as user-coordinated FT approaches rely on explicit user involvement in achieving fault tolerance.

Under the US DoE FASTOS program funding, we have been developing fault tolerance solutions for global address space (GAS) models. This includes both automatic (user-transparent)[9] as well as user-coordinated approach described in the current paper where we focus on fault tolerance for the Global Arrays (GA). Our user-transparent approach relies on Xen virtualization and supports high-speed networks. With Xen, we can checkpoint and migrate the entire OS image including the application to another node. These two technologies are complementary. However, they differ in several key respects such as portability, generality, and use of resources. The virtualization approach is most general yet potentially not as portable. For example, it relies on availability of Xen and Xen-enabled network drivers for the high speed networks. The user coordinated approach is very portable. However, since it requires modifications the application code (e.g., placing checkpoint and restart calls and possibly other restructuring of the program), it is harder to use and less general.

The GA library offers a global address space programming model implemented as a library with multiple language bindings (Fortran, C, C++, Python). It provides a portable interface through which each process in a parallel program can independently, asynchronously, and efficiently access logical block of physically distributed matrices, with no need for explicit cooperation by other processes. This feature is similar to the traditional shared-memory programming model. However, the GA model also acknowledges that remote data is slower to access than local data, and it allows data locality to be explicitly specified and used. Due to interoperability with MPI, the GA extends benefits of the global address space model to MPI applications. GA has been used in large applications in multiple science areas.

The GAS models such as GA enable the programmer to think of a single computation running across multiple processors, sharing a common address space. For discussions in the rest of the paper, we assume that each task or application process is associated with a single processor. All data is distributed among the processors and each of these processors has affinity to a part of the entire data. Each processor may operate directly on the data it contains but must use some indirect mechanism to access or update data at other processors. GAS models do not provide explicit mechanisms like send/receive for communication between processors but rather offer implicit style of communication with processors updating and accessing shared data. These characteristics lead to different design requirements for fault tolerance solutions for the GAS models than for MPI. In particular, we need to assure consistency of the global data during checkpoint/restart operations, and the tasks executed by processors affected by the failure can be transparently reassigned and the updates to the global shared data are fully completed (partial updates should either be prevented or undone).

3 Technical Approach

Our prototype fault resilient Global Arrays uses a spare pool of processors to perform reconfiguration after the fault. The prototype solution includes task virtualization, incremental or full checkpoint scheme, and restart capabilities. In addition, we rely on the parallel job's resource manager to detect and notify all the tasks in an application in case of a fault. Not all resource managers have the ability to deliver a fault notification to the application upon a fault. Quadrics is one vendor who incorporated such mechanisms into their RMS resource manager. As discussed in the previous section, to achieve fault tolerance we need to assure that the global shared data is fault tolerant, the tasks executed by processors affected by the failure can be transparently reassigned and the updates to the global shared data are fully completed (partial updates should either be prevented or undone).

3.1 Reconfiguration After the Fault

Our approach allocates a spare set of idle nodes to replace the failed node after the fault, see Fig. 1. This approach is more realistic than dynamic allocation of replacement nodes given the current operation practices of supercomputer centres. For quicker restart the spare nodes load a copy of the executable at startup time.

3.2 Processor Virtualization

To enable effective reconfiguration, virtualization of processor IDs is needed. The advantage of this approach is that once a failed node is replaced with a spare node, the array distribution details as known to the application don't have to be changed after recovery. Any prior information obtained by the application as to the array distribution or the task IDs of the tasks that hold a particular section of an array are still valid after recovery.

3.3 Data Protection and Recovery

The fault tolerance scheme for GA was designed to impose only small overhead relative to the execution of an application. Similarly to prior work like libckpt[5,6] we checkpoint the

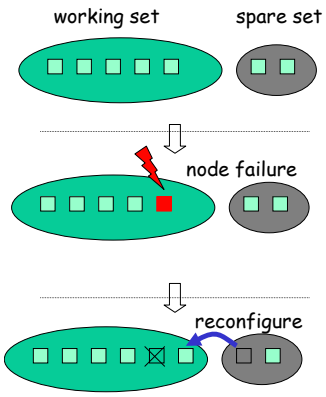


Figure 1. Fault reconfiguration using spare node pool.

global data, which is distributed among processes. Checkpointing global arrays is a based on a collective call i.e., all the processes participate in the checkpointing operation. There are two-options for checkpointing global arrays: (i) full checkpointing (ii) incremental checkpointing. With full checkpointing, each process takes a full checkpoint of its locally owned portion of the distributed global array. In this case, the entire global array is always checkpointed. The other approach, incremental checkpointing[6], reduces the amount of global array data saved at each checkpoint. Our implementation uses a page-based scheme similar to prior work[7], where only the pages modified since the last checkpoint is saved rather than the entire global data.

4 Application Example

The target application for testing fault tolerance was a chemistry code that implements a simple Hartree-Fock Self Consistent Field (HF SCF). The method obtains an approximate solution to Schrödinger's equation $H\Psi = E\Psi$. The solution is assumed to have the form of a single Slater determinant composed of one electron wavefunctions. Each single electron wavefunction $\varphi_i(r)$ is further assumed to be a linear combination of basis functions $\chi_\mu(r)$ that can be written as $\varphi_i(r) = \sum_\mu C_{i\mu}\chi_\mu(r)$. Combining the linear combination of basis functions with the assumption that the solution to Schrödinger's equation is approximated by a single Slater determinant leads to the self-consistent eigenvalue problem $F_{\mu\nu}C_{k\nu} = \epsilon D_{\mu\nu}C_{k\nu}$ where the density matrix $D_{\mu\nu} = \sum_k C_{k\mu}C_{k\nu}$ and the Fock matrix $F_{\mu\nu}$ is given by

$$F_{\mu\nu} = h_{\mu\nu} + \frac{1}{2}\sum_{\omega\lambda}\left[2(\mu\nu|\omega\lambda) - (\mu\omega|\nu\lambda)\right] D_{\omega\lambda} \tag{4.1}$$

Because the Fock matrix is quadratically dependent on the solution vectors, $C_{k\mu}$, the

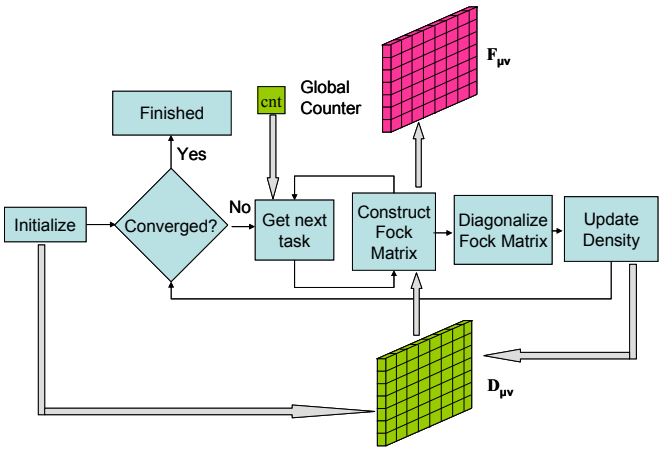


Figure 2. Block diagram of SCF. Block arrows represent data movement

solution procedure is an iterative, self-consistent procedure. An initial guess for the solution is obtained and used to create an initial guess for the density and Fock matrices. The eigenvalue problem is then solved and the solution is used to construct a better guess for the Fock matrix. This procedure is repeated until the solution vectors converge to a stable solution.

The solution procedure to the HF SCF equations is as follows, see Fig. 2. An initial guess to the density matrix is constructed at the start of the calculation and is used to construct an initial guess for the Fock matrix. The HF SCF eigenvalue equation is then solved to get an initial set of solution vectors. These can be used to obtain an improved guess for the density matrix. This process continues until a converged solution is obtained. Generally, the convergence criteria is that the total energy (equal to the sum of the lowest k eigenvalues of the HF SCF equations) approaches a minimum value. The density and Fock matrices, along with the complete set of eigenvectors all form two-dimensional arrays of dimension N, where N is the number of basis vectors being used to expand the one electron wavefunctions. These arrays are distributed over processors. The Fock matrix is constructed by copying patches of the density matrix to a local buffer, computing a contribution to the Fock matrix in another local buffer, and then accumulating the contribution back into the distributed Fock matrix. This is shown as the task loop in Fig. 2. The tasks are kept track of via a global counter that is incremented each time a processor takes up a new task. An obvious point for checkpointing in this process is to save the solution vectors at each cycle in the self-consistent field solution. The checkpoint would occur right after the "Diagonalize Fock Matrix" operation in Fig. 2. If the calculation needs to restart, this can easily be accomplished by restoring the saved solution vectors from the most recent cycle. It is fairly straightforward to reconstruct the density matrix using the most recent solution and the remainder of the HF SCF process can proceed as before. This would correspond to restarting in the "Update Density" block in Fig. 2.

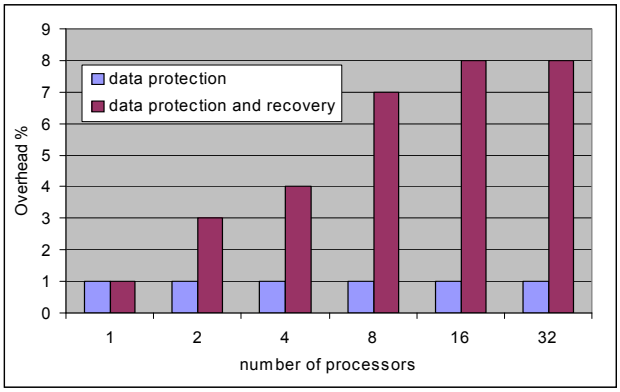


Figure 3. Overhead for data protection and overhead in SCF relative to total execution time

For very large calculations, this level of checkpointing may be too coarse and it would be desirable to add additional refinement. This could be accomplished by storing intermediate versions of the Fock matrix, as well as the solution vectors, which would correspond to adding additional checkpoints inside the task loop that constructs the Fock matrix.

4.1 Experimental Results

The experiment evaluation was performed on a Quadrics cluster with 24 dual Itanium nodes running Linux kernel 2.6.11. We run HF calculations for a molecule composed of sixteen Beryllium atoms. In addition to the checkpointing operation, we measured time for recovery from a single fault of a node. Figure 3 shows that the overhead introduced by the data protection is very low (1%). This is the overhead representative of normal program execution (without faults). Of course, in those cases the restart operation is not triggered and therefore the restart overhead also shown in Fig. 3 does not apply.

As expected the time to recover from the fault was more significant (8%) than checkpointing, as measured relative to the original application. It involves the cost of loading the restart files and reconfiguration. That level of overhead is quite reasonable since by extending the execution time by 8% we would be able avoid the costly alternative of restarting the whole application from the beginning.

These results indicate that at least for applications with characteristics similar to the HF program, our fault tolerant solution would have a very limited impact on the performance while providing a protection against faults.

5 Conclusions and Future Work

We described a prototype of a fault resilient Global Arrays toolkit that supports a global address space programming model. This capability was used for a chemistry application and shown to introduce very small overheads as compared to the baseline application without

fault resiliency. Our plans for future work include more experimental validation with other applications, comparing effectiveness and performance of the current user-directed and the user-transparent approaches. We also plan on adding our own collective fault notification mechanism as a fallback strategy when working with resource managers that do not have a mechanism to deliver the failure notification to all the tasks in an application.

References

1. F. Petrini, J. Nieplocha and V. Tipparaju, *SFT: Scalable fault tolerance*, ACM SIGOPS Operating Systems Review, **40**, 2, (2006).
2. J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease and E. Apra. *Advances, applications and performance of the Global Arrays Shared Memory Programming Toolkit*. International Journal of High Performance Computing and Applications, **20**, 2, (2006).
3. M. Blocksome, C. Archer, T. Inglett, P. McCarthy, M. Mundy, J. Ratterman, A. Sidelnik, B. Smith, G. Almasi, J. Castanos, D. Lieber, J. Moreira, S. Krishnamoorthy and V. Tipparaju, *Blue Gene system software: Design and implementation of a one-sided communication interface for the IBM eServer Blue Gene supercomputer*, Proc. SC'06, (2006).
4. G. Fagg, T. Angskun, G. Bosilca, J. Pjesivac-Grbovic and J. Dongarra, *Scalable Fault Tolerant MPI: Extending the Recovery Algorithm*, Proc. 12th Euro PVM/MPI, (2005).
5. J. S. Plank, M. Beck, G. Kingsley, K. Li, *Libckpt. Transparent Checkpointing under Unix*, in: Proc. Usenix Technical Conference, New Orleans, pp. 213–223, (1995).
6. J. S. Plank, J. Xu, R. H. Netzer, *Compressed differences: An algorithm for fast incremental checkpointing*, Tech. Rep. CS-95-302, University of Tennessee, (1995).
7. Princeton University Scalable I/O Research, *A checkpointing library for Intel Paragon*. `http://www.cs.princeton.edu/sio/CLIP`.
8. Y. Zhang, R. Xue, D. Wong, W. Zheng, *A checkpointing/recovery system for MPI applications on cluster of IA-64 computers*, International Conference on Parallel Processing Workshops (ICPPW'05), pp. 320-327, (2005).
9. D. Scarpazza, P. Mullaney, O. Villa, F. Petrini, V. Tipparaju, D. M. L. Brown, J. Nieplocha, *Transparent system-level migration of PGAs applications using Xen on Infiniband*, Proc. IEEE CLUSTER'07, (2007).

Using AOP to Automatically Provide Distribution, Fault Tolerance, and Load Balancing to the CORBA–$\mathcal{LC}$ Component Model

Diego Sevilla[1], **José M. García**[1], **and Antonio Gómez**[2]

[1] Department of Computer Engineering

[2] Department of Information and Communications Engineering
University of Murcia, Spain
E-mail: {dsevilla, jmgarcia}@ditec.um.es, skarmeta@dif.um.es

Programming abstractions, libraries and frameworks are needed to better approach the design and implementation of distributed High Performance Computing (HPC) applications, as the scale and number of distributed resources is growing. Moreover, when Quality of Service (QoS) requirements such as load balancing, efficient resource usage and fault tolerance have to be met, the resulting code is harder to develop, maintain, and reuse, as the code for providing the QoS requirements gets normally mixed with the functionality code. Component Technology, on the other hand, allows a better modularity and reusability of applications and even a better support for the development of distributed applications, as those applications can be partitioned in terms of components installed and running (deployed) in the different hosts participating in the system. Components also have requirements in forms of the aforementioned non-functional aspects. In our approach, the code for ensuring these aspects can be automatically generated based on the requirements stated by components and applications, thus leveraging the component implementer of having to deal with these non-functional aspects. In this paper we present the characteristics and the convenience of the generated code for dealing with load balancing, distribution, and fault-tolerance aspects in the context of CORBA–$\mathcal{LC}$. CORBA–$\mathcal{LC}$ is a lightweight distributed reflective component model based on CORBA that imposes a peer network model in which the whole network acts as a repository for managing and assigning the whole set of resources: components, CPU cycles, memory, etc.[a]

1 Introduction

Component-based development (CBD)[1], resembling integrated circuits (IC) connections, promises developing application connecting independently-developed self-describing binary components. These components can be developed, built and shipped independently by third parties, and allow application builders to connect and use them. This development model is very convenient for distributed applications, as components can be installed in different hosts to match the distributed nature of this kind of application. Moreover, as applications become bigger, they must be modularly designed. Components come to mitigate this need, as they impose the development of modules that are interconnected to build complete applications. Components, being binary, independent and self-described, allow:

- Modular application development, which leads to maximum code reuse, as components are not tied to the application they are integrated in.

[a] This work has been jointly supported by the Spanish MEC and European Commission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03".

- Soft application evolution and incremental enhancement, as enhanced versions of existing components can substitute previous versions seamlessly, provided that the new components offer the required functionality. New components can also be added to increase the set of services and functionality that new components can use, thus allowing applications to evolve easily.

To bring the benefits of Component-Based Development to distributed High Performance Computing (HPC), we developed CORBA *Lightweight Components* (CORBA–$\mathcal{LC}$)[2], a distributed component model based on CORBA[3]. CORBA–$\mathcal{LC}$ offers traditional component models advantages (modular applications development connecting binary interchangeable units), while performing an automatic deployment of components over the network. This deployment solves the component dependencies automatically, using the complete network of hosts to decide the placement of component instances in network nodes, intelligent component migration and load balancing, leading to maximum network resource utilization.

In order to perform this intelligent deployment, components separate the actual component functionality from the non-functional specification of Quality of Service (QoS) requirements, such as load balancing, fault tolerance, and distribution. This information is used by the CORBA–$\mathcal{LC}$ framework to generate the code that deals with those non-functional aspects of the component. In this way, the programmer can concentrate only on the component functionality, leaving to the framework the responsibility of ensuring that the actual QoS requirements are met. Moreover, separating component code from the specification of non-functional requirements allows us to apply *Aspect-Oriented Programming* (AOP)[4] techniques to the CORBA–$\mathcal{LC}$ Component Model. In this paper we show how AOP techniques can be used for automatic code generation of the aforementioned non-functional aspects code, and discuss the convenience of this approach of combining Component-Based Development (CBD) with AOP.

The paper is organized as follows: Section 2 offers an overview of CORBA–$\mathcal{LC}$. Section 3 shows how graphics applications can be used to define how to connect a set of components and how to specify non-functional aspects requirements. Section 4 shows how automatic code can be generated to seamlessly and transparently offer non-functional aspects implementation. Finally, Section 5 offers related work in the fields of component models and aspects, and Section 6 presents our conclusions.

2 The CORBA–$\mathcal{LC}$ Component Model

CORBA Lightweight Components (CORBA–$\mathcal{LC}$)[2,5] is a lightweight component model based on CORBA, sharing many features with the CORBA Component Model (CCM)[6]. The following are the main conceptual blocks of CORBA–$\mathcal{LC}$:

- *Components*. Components are the most important abstraction in CORBA–$\mathcal{LC}$. They are both a *binary package* that can be installed and managed by the system and a *component type*, which defines the characteristics of component instances (interfaces offered and needed, events produced and consumed, etc.) These are connection points with other components, called *ports*.
- *Containers and Component Framework.* Component instances are run within a runtime environment called *container*. Containers become the instances view of the

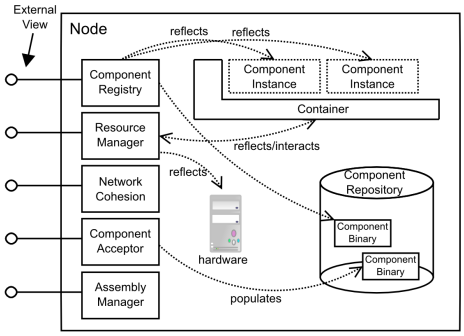


Figure 1. Logical Node Structure.

world. Instances ask the container for the required services and it in turn informs the instance of its environment (its *context*).

- *Packaging model*. The packaging allows to build self-contained binary units which can be installed and used independently. Components are packaged in ".ZIP" files containing the component itself and its description as IDL (*CORBA Interface Definition Language*) and XML files.

- *Deployment and network model*. The deployment model describes the rules a set of components must follow to be installed and run in a set of network-interconnected machines in order to cooperate to perform a task. CORBA–$\mathcal{LC}$ deployment model is supported by a set of main concepts:

  – *Nodes*. The CORBA–$\mathcal{LC}$ network model can be seen as a set of nodes (hosts) that collaborate in computations. Nodes maintain the logical network connection, encapsulate physical host information and constitute the external view of the internal properties of the host they are running on – Fig. 1. Nodes offer information about memory and CPU load, and the set of components installed.

  – *The Reflection Architecture* is composed of the meta-data given by the different node services: The *Component Registry* provides information about (a) running components, (b) component instances running in the node and the properties of each, and (c) how those instances are connected via ports (assemblies); the *Resource Manager* in the node collaborates with the *Container* implementing initial placement of instances, migration/load balancing at run-time.

  – *Network Model and The Distributed Registry*. The CORBA–$\mathcal{LC}$ deployment model is a network-centred model: The complete network is considered as a repository for resolving component requirements.

  – *Applications and Assembly*. In CORBA–$\mathcal{LC}$, *applications* are a set of rules that a set of component instances must follow to perform a given task. Applications are also called *assemblies*, as they encapsulate explicit rules to connect component instances. Application deployment is then issued by instantiating an

assembly: creating component instances and connecting them. The CORBA–$\mathcal{LC}$ deployment process is intelligent enough to select the nodes to host the component instances based on the assembly requirements. Users can create assemblies using visual building tools, as the CORBA–$\mathcal{LC}$ Assembly GUI – Fig. 2.

3 Component Non-Functional Aspects

Components are not only a way of structuring programs, but a framework in which the programmer can focus in the functionality, leaving other aspects (called *non-functional aspects*) such as reliability, fault tolerance, distribution, persistence, or load balancing to the framework. The goal of CORBA–$\mathcal{LC}$ is to allow the programmer to write the functionality of the components, then describe how the components would work in terms of those non-functional aspects in a declarative manner, and let the framework to implement that requirements.

We can use an assembly of an example *Master/Worker* application based on components to show how CORBA–$\mathcal{LC}$ can achieve this goal. Figure 2 shows this assembly in the CORBA–$\mathcal{LC}$ Assembly GUI. The upper left part of the screen shows the available components in the network, while the lower left part shows the characteristics of the selected component or connection. Each red rectangle in the right part represents a component. Used interfaces are shown in the left part of the rectangle, while provided ones are shown in the right part. You can see the connection among components.

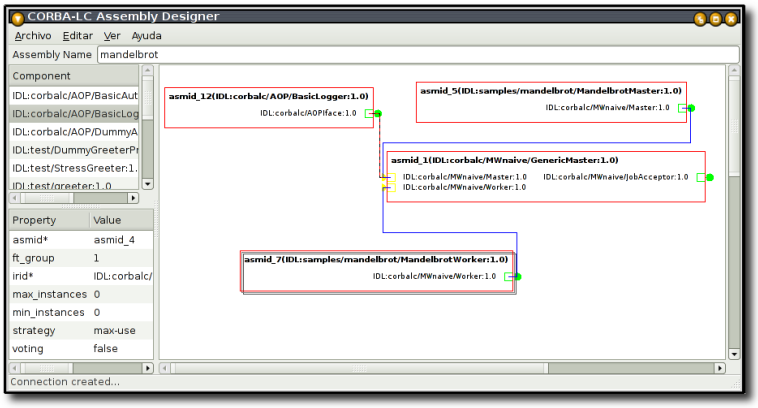


Figure 2. The CORBA–$\mathcal{LC}$ Assembler Graphical User Interface (GUI).

The `GenericMaster` component is in charge of the standard *Master/Worker* protocol. Note that the only needs of this component is to have one component that *provide* the generic `Master` interface, and a set of components that *provide* the generic `Worker` interface. This is very convenient, as allows us to plug *any* pair of components that perform that type of computation. In this case, the figure shows a pair of components that calculate a Mandelbrot fractal.

In the lower left part, the figure shows the set of properties of the `Worker` connection of the `GenericMaster` component instance (that is, the characteristics of the connection between the `GenericMaster` and the `MandelbrotWorker` component through the `Worker` interface.) The most important property is "`strategy`", which defines how this connection is made. In CORBA–*LC*, this property can have different possible values:

- `default`. This is a normal CORBA call.
- `local`. The connection is instructed to be local (both components must be instantiated in the same node).
- `remote`. The connection is instructed to be remote (components must be instantiated in different nodes).
- `fault-tolerant`. With this value, instead of one instance of the component, a set of component instances (*replicas*) are created, in different nodes. Whenever the component calls an operation on this interface, several threads are created to issue concurrent calls to all the alive replicas, performing a final voting and a signalling of dead replicas if necessary.
- `load-balancing`. Similar to the previous one, instead of one component instance, several are created. When a call to that interface is issued, the call is redirected to the less loaded node.
- `max-use`. Again, instead of one instance, the whole network will be used to create as many component instances as possible.

The `local` strategy is useful for components that need an high-speed connection, such as streaming processing. The `remote` value instead is useful for keeping a small set of components in a low-end node (such as a PDA,) while the rest of components are in remote, more powerful, nodes. For the last three values, optional "`min_instances`" and "`max_instances`" properties can also be specified. When the CORBA–*LC* Assembler builds the application, it will ensure the number of needed component instances and connections satisfy the needs of the assembly. Note that in some cases, to meet the number required instances, this step may also require sending the component for installation and instantiation to other nodes if there are not enough nodes with this component available in the first place. In this specific example, the "`strategy`" property shows the "`max-use`" value, as suggested for the *Master/Worker* functionality.

Finally, AOP connections are also possible. In Fig. 2, the stripped connection line shows an AOP connection. These connections allows a component (that provides the `AOPIface` interface) to take control each time a call is made between two components (allow it, abort it, or even modify it.) In the figure, the `BasicLogger` component is in charge of writing a log of all the calls made. Note that this behaviour goes unnoticed for both the caller and the callee, and that this functionality can be activated or deactivated at any time, even at run-time, without any specific code written for logging in the application.

Other AOP connections can be made. In our research, an authenticator component has also been created, allowing any connection to be authenticated prior to making any call.

4 Automatic Generation of Aspects Code

The CORBA–*LC Code Generator* is in charge of generating all the code to deal with the required non-functional aspects of components. It uses the information of both (1) the

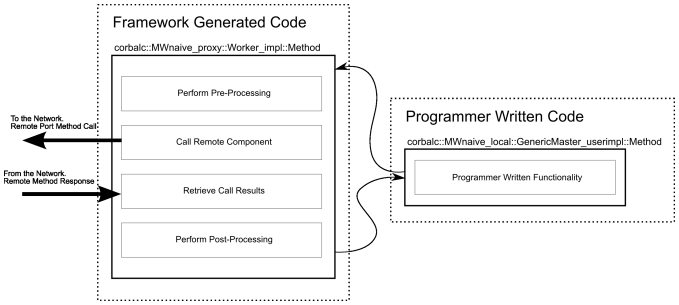


Figure 3. Proxy call sequence.

standard CORBA *Interface Repository* (IR), and (2) the Component's XML file. While the XML describes the component ports and non-functional requirements, the IR describes all the IDL interfaces used in those ports. As output, the code generator produces the needed implementation files and boilerplate that can be used by the programmer to write the functionality proper of the component.

CORBA–$\mathcal{LC}$ must have control of all the communication that happens among the component ports. This way, the framework can modify how the components communicate assuring the required load balancing, fault-tolerance, etc. Thus, for each used and provided interface, interception code must be created by the Code Generator. This is also referred to as "*point-cuts*" in Aspect-Oriented Programming (AOP)[4] terminology.

For each provided interface of a component, CORBA implementation objects (*servants*) are created. Servants are in charge of receiving the actual CORBA calls, and propagating the call to the final programmer code (called *executor*), that implements the functionality of the offered interface. The servant can also perform pre- and post-processing on the call. For instance, it can retrieve and store the executor data from a data-base, offering seamless persistence (as another aspect) to component instances.

For each required (*used*) interface of a component, *proxy objects* are created. They are in charge of delivering the local programmer code call to other component's provided interface as a normal CORBA call. At this point, the proxy code can also do some pre- and post-processing. Concretely, they are in charge of Fig. 3: (a) Calling the possibly attached AOP connections to this port, passing them all the parameters of the call, (b) Maintaining the set of active remote component instances, and (c) Depending on the *strategy*:

- Generating a pool of threads and concurrently calling all the remote component instances, retrieve their results and optionally performing voting (*fault-tolerant*.)
- Localizing the less loaded node and sending the call to the component instance running in that particular node (*load-balancing* strategy.)
- Providing to the component the set of remote component instances (*max-use*.)

5 Related Work

To date, several distributed component models have been developed. Although CORBA–$\mathcal{LC}$ shares some features with them, it also has some key differences.

Java Beans[7], Microsoft's Component Object Model (COM)[8], .NET[9] offer similar component models, but lack in some cases that are either limited to the local (non-distributed) case or do not support heterogeneous environments of mixed operating systems and programming languages as CORBA does.

In the server side, SUN's EJB[10] and the recent Object Management Group's CORBA Component Model (CCM)[11] offer a server programming framework in which server components can be installed, instantiated and run. Both are fairly similar. Both are designed to support enterprise applications, offering a container architecture with support for transactions, persistence, security, etc. They also offer the notion of components as binary units which can be installed and executed (following a fixed assembly) in Components Servers.

Although CORBA–$\mathcal{LC}$ shares many features with both models, it presents a more dynamic model in which the deployment is not fixed and is performed at run-time using the dynamic system data offered by the Reflection Architecture. Also, CORBA–$\mathcal{LC}$ is a *lightweight* model in which the main goal is the optimal network resource utilization instead of being oriented to enterprise applications. Finally, CORBA–$\mathcal{LC}$ adds AOP connections, not present in the other two models.

Applying Aspects-Oriented techniques to Component Models has also been explored in several works. In[12] the authors apply AOP techniques to the EJB component model. This work is limited to Java and the usage of AspectJ[13] to provide a finer grain of control over actual calls in EJB. A quantitative study showing the benefits of AOP for component-based applications (in terms of number of lines of code and number of places to change when a modification on the application has to be done) can be found in[14].

In[15], the authors apply aspect oriented techniques in the context of the CORBA Component Model and security policies using the Qedo framework (an implementation of the CCM). Real-Time has been treated as an aspect in a CCM component framework implementation (CIAO) in[16]. The approach of these works is similar to the one presented in this paper, but none of them treat distribution, load balancing and fault tolerance as an aspect.

In the field of High Performance Computing (HPC) and Grid Computing, Forkert et al. ([17]) present the TENT framework for wrapping applications as components. However, this wrapping is only used to better organize applications, and not to provide an integrated framework in which offer services to component implementations.

The Common Component Architecture (CCA)[18] is a component model framework also based on the idea of reusable, independent components. However, it does not offer any basic run-time support for distribution, load balancing or fault tolerance. Thus, implementing those services require of *ad-hoc* programming, which goes against reusability.

6 Conclusions

As we showed in this paper, component technology in general, and CORBA–$\mathcal{LC}$ in particular, offer a new and interesting way of approaching distributed applications. Services otherwise complicated can be offered by the framework just by specifying them in the characteristics and needs of components and applications. We showed how convenient the Aspect-Oriented approach is to seamlessly and transparently offer services such as fault tolerance, replication and load balancing to components, and the importance of being able to specify those non-functional aspects in a declarative manner, so that the required code for those aspects can be generated automatically.

References

1. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, (ACM Press, 1998).
2. D. Sevilla, J. M. García and A. Gómez, *CORBA lightweight components: a model for distributed component-based heterogeneous computation*, in: EUROPAR'2001, LNCS vol. **2150**, pp. 845–854, Manchester, UK, (2001),
3. M. Henning and S. Vinoski, *Advanced CORBA Programming with C++*, (Addison-Wesley Longman, 1999).
4. F. Duclos, J. Estublier and P. Morat, *Describing and using bon functional aspects in component-based applications*, in: International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, (2002).
5. D. Sevilla, J. M. García and A. Gómez, *Design and implementation requirements for CORBA Lightweight Components*, in: Metacomputing Systems and Applications Workshop (MSA'01), pp. 213–218, Valencia, Spain, (2001).
6. Object Management Group, *CORBA: Common Object Request Broker Architecture Specification, revision 3.0.2*, (2002), OMG Document formal/02-12-06.
7. SUN Microsystems, *Java Beans specification*, 1.0.1 edition, (1997). http://java.sun.com/beans.
8. Microsoft, *Component Object Model (COM)*, (1995). http://www.microsoft.com/com.
9. Microsoft Corporation, Microsoft .NET, http://www.microsoft.com/net
10. SUN Microsystems, *Enterprise Java Beans specification*, 3.0 edition, May 2006, http://java.sun.com/products/ejb.
11. Object Management Group, *CORBA Component Model*, 1999, OMG Document ptc/99-10-04.
12. R. Pichler, K. Ostermann and M. Mezini, *On aspectualizing component models*, Software, Practice and Experience, **33**, 957–974, (2003).
13. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, Lecture Notes in Computer Science, **2072**, 327–355, (2001).
14. O. Papapetrou and G. Papadopoulos, *Aspect oriented programming for a component based real life application: a case study*, in: Symposium on Applied Computing — Software Engineering track, (2004).
15. T. Ritter, U. Lang and R. Schreiner, *Integrating security policies via container portable interceptors*, Distributed Systems Online, (2006).
16. N. Wang, C. Gill, D. C. Schmidt and V. Subramonian, *Configuring real-time aspects in component middleware*, in: International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus, (2004).
17. T. Forkert, G. K. Kloss, C. Krause and A. Schreiber, *Techniques for wrapping scientific applications to CORBA Components.*, in: High-Level Parallel Programming Models and Supportive Environments (HIPS'04), pp. 100–108, (2004).
18. D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan and A. Slominski, *On building parallel & grid applications: component technology and distributed services.*, Cluster Computing, **8**, 271–277, (2005).

VirtuaLinux: Virtualized High-Density Clusters with no Single Point of Failure

Marco Aldinucci[1], Marco Danelutto[1], Massimo Torquati[1], Francesco Polzella[1], Gianmarco Spinatelli[1], Marco Vanneschi[1], Alessandro Gervaso[2], Manuel Cacitti[2], and Pierfrancesco Zuccato[2]

[1] Computer Science Department
University of Pisa, Largo B. Pontecorvo 3, I-56127 Pisa, Italy
E-mail: {*aldinuc, marcod, torquati, polzella, spinatel, vannesch*}*@di.unipi.it*

[2] Eurotech S.p.A.
Via Fratelli Solari 3/a, I-33020 Amaro (UD), Italy
E-mail: {*a.gervaso, m.cacitti, p.zuccato*}*@eurotech.it*

VirtuaLinux is a Linux meta-distribution that allows the creation, deployment and administration of both physical and virtualized clusters with no single point of failure. VirtuaLinux supports the creation and management of virtual clusters in seamless way: VirtuaLinux Virtual Cluster Manager enables the system administrator to create, save, restore Xen-based virtual clusters, and to map and dynamically remap them onto the nodes of the physical cluster. We introduces and discuss VirtuaLinux virtualization architecture, features, and tools. These rely on a novel disk abstraction layer, which enables the fast, space-efficient, dynamic creation of virtual clusters composed of fully independent complete virtual machines.

1 Introduction

Clusters are usually deployed to improve performance and/or availability over that provided by a single computer, while typically being much more cost-effective than a single computer of comparable speed or availability. A cluster is a network of complete computers (a.k.a. nodes), each of them running its own copy of a – possibly standard – operating system (OS). A range of solutions for the collective and/or centralised management of nodes are available in all OSes, from very low level tools (e.g. `rdist`) to complete software packages (e.g. *Sun Grid Engine*[1]).

Typically, a node of the cluster acts as the master of the cluster, while the others depend on it for several services, such as file sharing, user authentication, network routing and resolution, time synchronisation. The master is usually statically determined at the installation time for its hardware (e.g. larger disks) or software (e.g. configuration of services). In high-density clusters, disks mounted on nodes are statistically the main source of failures because of the strict constraints of size, power and temperature[2]. This is particularly true on master disk that happens also to be a critical single point of failure for cluster availability since it hosts services for cluster operation. A hardware or software crash/malfunction on the master is simply a catastrophic event for cluster stability.

In addition, rarely a single configuration or even a single OS can be adapted to supply all users' needs. Classic solutions like static cluster partitioning with multiple boots are static and not flexible enough to consolidate several user environments and require an additional configuration effort. Since cluster configuration involves the configuration

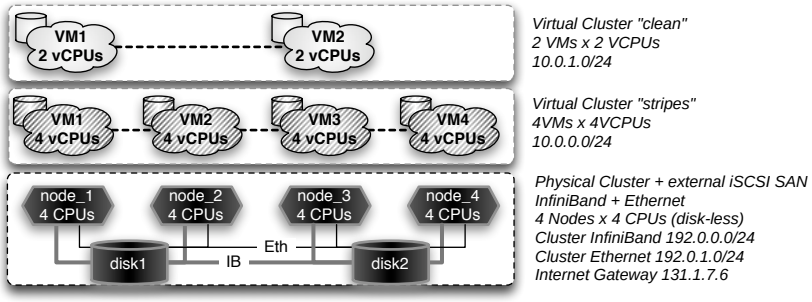


Figure 1. A physical cluster running three Virtual Clusters.

of distinct OS copies in different nodes, any configuration mistake, which may seriously impair cluster stability, is difficult to undo.

We present VirtuaLinux, an innovative Linux meta-distribution able to support the installation and the management of a disk-less, master-less cluster with a standard Linux distribution (e.g. CentOS, Ubuntu). VirtuaLinux address previously mentioned robustness problems of standard cluster installation, and natively supports Virtual Clusters (VCs). In particular, this paper presents VirtuaLinux VCs facility and the software architecture supporting them. We refer back to Aldinucci et al.[3] for a detailed description of other VirtuaLinux features, such as disk-less cluster boot, storage abstraction, and master-less services configuration.

VirtuaLinux improves the management flexibility and configuration error resilience of a cluster by means of transparent node virtualization. A physical cluster may support one or more virtual clusters (i.e. cluster of virtual nodes) that can be independently managed without affecting the underlying physical cluster configuration and stability. Virtual clusters run a guest OS (either a flavour of Linux or Microsoft Windows™) that may differ from the host OS that handles physical cluster activities.

2 Virtual Clustering

The virtualization of the physical resources of a computing system to achieve improved degrees of sharing and utilisation is a well-established concept that goes back decades[4,5]. In contrast to a non-virtualized system, full virtualization of all system resources (including processors, memory and I/O devices) makes it possible to run multiple OSes on a single physical platform. A virtualized system includes a new layer of software, called a Virtual Machine Monitor (VMM). The principal role of the VMM is to arbitrate access to the underlying physical host platform resources so that these resources can be shared among multiple OSes that are guests of the VMM. The VMM presents to each guest OS a set of virtual platform interfaces that constitute a virtual machine (VM).

By extension, a Virtual Cluster (VC) is a collection of VMs that are running onto one or more physical nodes of a cluster, and that are wired by a virtual private network. By uniformity with the physical layer, all VMs are homogeneous, i.e. each VM may access a private virtual disk and all VMs of a virtual cluster run the same OS and may access a

shared disk space. Different virtual clusters may coexist on the same physical cluster, but no direct relationship exists among them, apart from their concurrent access to the same resources (see Fig. 1). Virtual clusters bring considerable added value to the deployment of a production cluster because they ease a number of management problems, such as: *physical cluster insulation* and *cluster consolidation*. Physical cluster insulation ensures that crashes or system instability due to administration mistakes or cursoriness at the virtual layer are not propagated down to the physical layer and have no security or stability impact on the physical layer. Virtualization is used to deploy multiple VCs, each exploiting a collection of VMs running an OS and associated services and applications. Therefore, the VMs of different VCs may be targeted to exploit a different OS and applications to meet different user needs. The main drawback of virtualization is overhead, which usually grows with the extent of hardware and software layers that should be virtualized.

3 VirtuaLinux

VirtuaLinux is a Linux distribution that natively supports the dynamic creation and management of VCs on a physical cluster. VirtuaLinux implementation is arranged in a two-tier architecture: *VM implementation* layer and *VM aggregation* layer. The first one implements the single VM (currently the Xen[6] VMM). The second one aggregates many VMs in a VC, and dynamically creates and manages different VCs. This is realised by way of the *VirtuaLinux Virtual Cluster Manager* (VVCM). Overall, the VVCM enables the system administrator to dynamically create, destroy, suspend and resume from disk a number of VCs. The VCs are organised in a two-tier network: each node of a VC is connected to a private virtual network, and to the underlying physical network. The nodes of a VC are homogeneous in terms of virtualized resources (e.g. memory size, number of CPUs, private disk size, etc.) and OS. Different clusters may exploit different configurations of virtual resources and different OSes. Running VCs share the physical resources according to a creation time mapping onto the physical cluster. VCs may be reallocated by means of the run-time migration of the VM between physical nodes.

Each virtual node of a VC is implemented by a Xen VM that is configured at the VC creation time. Each virtual node includes: a virtual network interface with a private IP, a private virtual disk, a private virtual swap area and a VC-wide shared virtual storage. The virtualization of devices is realised via the standard Xen virtualization mechanisms.

3.1 VirtuaLinux Storage Architecture

VirtuaLinux uses EVMS to provide a single, unified system for handling storage management tasks, including the dynamic creation and destruction of volumes, which are an EVMS abstraction that are seen from the OS as disk devices[7,8].

The external SAN should hold a distinct copy of the OS for each node. At this end, VirtuaLinux prepares, during installation, one volume per node and a single volume for data shared among nodes. Volumes are formatted with an OS specific native file system (e.g. ext3), whereas shared volumes are formatted with a distributed file system that arbitrates concurrent reads and writes from cluster nodes, such as the Oracle Concurrent File System (OCFS2) or the Global File System (GFS).

Volumes are obtained by using the EVMS snapshot facility. A snapshot represents a frozen image of a volume of an original source. When a snapshot is created, it looks exactly like the original at that point in time. As changes are made to the original, the snapshot remains the same and looks exactly like the original at the time the snapshot was created. A file on a snapshot is a reference, at the level of disk block, to its original copy.

3.1.1 Snapshots Usage in VirtuaLinux

EVMS snapshots can be managed as real volumes that can be activated and deactivated, i.e. mapped and unmapped onto Unix device drivers. However, despite being standard volumes, snapshots have a subtle semantics regarding activation due to their *copy-on-write* behaviour[9]. In fact, the system cannot write on an inactive snapshot since it is not mapped to any device, thus may lose the correct alignment with its original during the deactivation period. EVMS solves the problem by logically marking a snapshot for reset at deactivation time, and resetting it to the current original status at activation time. Since snapshots cannot be deactivated without losing snapshot private data, they all should always be kept active in all nodes, even if each node will access only one of them. Snapshots on Linux OS (either created via EVMS, LVM, or other software) are managed as UNIX devices via the *device mapper* kernel functionality.

Although EVMS does not fix any limit on the number of snapshots that can be created or activated, current Linux kernels establish a hardwired limit on the number of snapshots that can be currently active on the same node. This indirectly constrains the number of snapshots that can be activated at the same time, and thus the number of nodes that VirtuaLinux can support. Raising this limit is possible, but requires a non-trivial intervention on the standard Linux kernel code. VirtuaLinux overcomes the limitation with a different approach, which does not require modifications to the kernel code. Since each snapshot is used as private disk, each snapshot is required to be accessible in the corresponding node only. In this way, each node can map onto a device just one snapshot. The status of an EVMS snapshot is kept on the permanent storage. This information is also maintained in a lazy consistent way in the main memory of each node. Status information is read at EVMS initialisation time *(evms_activate)*, and committed out at any EVMS command (e.g. create, destroy, activate, and deactivate a snapshot). While each snapshot can have just a single global state for all nodes (on the permanent storage), it may have different status on the local memory of nodes (e.g. it can be mapped onto a device on a node, while not appearing on another). Snapshot deactivation consists in unmapping a snapshot device from the system, then logically marking it for reset on permanent storage.

VirtuaLinux extends EVMS features with the option to disable EVMS snapshot reset-on-activate feature via a special flag in the standard EVMS configuration file. In the presence of this flag, the extended version of EVMS will proceed to unmap the snapshot without marking it for reset. VirtuaLinux EVMS extension preserves snapshot correctness since the original volume is accessed in read-only mode by all nodes, and thus no snapshot can lose alignment with the original. One exception exists: major system upgrades, which are performed directly on the original copy of the file system, and that trigger the reset of all snapshots. At the implementation level, the VirtuaLinux EVMS extension requires the patching of EVMS user space source code (actually just few lines of C code).

3.2 VC Networking

Xen supports VM networking through *virtualized Ethernet interfaces*. These interfaces can be connected to underlying physical network devices either via *bridged* (OSI model layer 2) or *routed* (OSI model layer 3) networking. Bridging requires less setup complexity and connection tracking overhead as compared to the routing method. On the other hand, bridging impairs insulation among different networks on the same bridge, and it lacks flexibility since it can hardly be dynamically configured to reflect the dynamic creation and destruction of VC-private networks. For this, VirtuaLinux currently adopts the routed networking.

VirtuaLinux sets up VC-private networks in a simple manner: all nodes in the VC are assigned addresses from a private network chosen at creation time, and the VC does not share the same subnet as the physical cluster. In this way, the communications among physical and virtual clusters are handled by setting up appropriated routing policies on each physical node, which acts as a router for all the VMs running on it. Routing policies are dynamically set up at the deployment time of the VM. All VMs of all VCs can be reached from all physical nodes of the cluster and each VC can access to the underlying physical network without any master gateway node. Virtual nodes of a VC are simply VMs on the same virtual subnet. However, each virtual network is insulated from the others. The routing configuration is dynamic, and has a VC lifespan. The configuration is dynamically updated in the case virtual nodes are remapped onto the physical cluster.

3.3 VC Disk Virtualization

Typically, VM-private disks are provided through either disk partitions or disk image files. The former method usually provides a speed edge while the latter guarantees a greater flexibility for dynamic creation of VMs. Actually, both methods require the whole root file system of the host OS as many times as the number of nodes in the VC. This leads to a very high data replication on the physical disk, a very long VC creation time. VirtuaLinux copes with these issues by means of the EVMS snapshotting technique described in Section 3.1. All private disks of a VC are obtained as snapshots of a single image including the VC guest OS. As discussed in Section 3.1.1, this leads to a VC creation time that is independent of the number of nodes in the VC (in the range of seconds) and all benefit discussed in Section 3.1. Once created, EVMS volumes are dynamically mounted on physical nodes according to the virtual-to-physical mapping policy chosen for the given VC.

As for the physical cluster, each VC comes with its own VC-private shared storage, which relies on OCFS2 distributed file system to arbitrate concurrent read and write accesses from virtual cluster nodes. However, since Xen does not currently enable the sharing of disks between VMs on the same physical nodes, the VC shared disk cannot be directly accessed from within virtual nodes. VirtuaLinux currently overcomes the problem by wrapping the shared storage with a NFS file system. At VC deployment time, each physical node involved in the deployment mounts the VC shared storage, which is in turn virtualized and make available to virtual nodes.

3.4 VC Management

VirtuaLinux provides two strategies for virtual-to-physical mapping of VMs: *Block* and *Cyclic*. The first one aims to minimise the spread of VMs on the physical nodes. This is

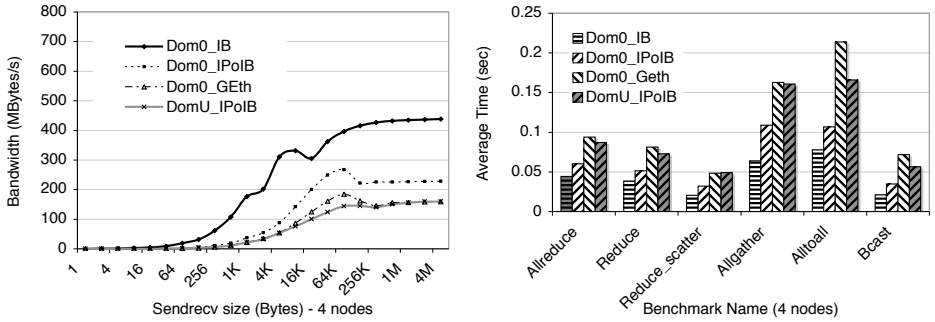


Figure 2. VirtuaLinux: evaluation of network bandwidth (left) and collective communication performance (right) with the Intel MBI Benchmarks[10]. Legend: Dom0_IB: Ubuntu Dom0, Infiniband user-space verbs (MPI-gen2); Dom0_IPoIB: Ubuntu Dom0, Infiniband IPoverIB (MPI-TCP); Dom0_Geth: Ubuntu Dom0, Giga-Ethernet (MPI-TCP); DomU_IPoIB: Ubuntu DomU, virtual net on top of Infiniband IPoverIB (MPI-TCP).

achieved by allocating on the physical node the maximum allowed number of VMs. The second one tries to spread the cluster's VM across all the cluster's physical nodes. The two strategies discussed can be coupled with two modifiers: *Strict* and *Free*. With the first modifier the deployment can be done only if there are enough free cores, with the second the constraint between the number of VM processors and physical cores is not taken into account at all. Notice that the mapping strategy of a VC can be changed after the first deployment provided it is the suspended state.

The VVCM (VirtuaLinux Virtual Cluster Manager) consists of a collection of Python scripts to create and manage the VCs. All the information about the virtual nodes such as the mapping between physical and virtual nodes and the state of each virtual machine are stored in a *database*. The information is maintained consistent between the launch of different clusters. A simple *command-line library* for the creation (*VC_Create* command), the activation (*VC_Control* command) and the destruction of the VCs (*VC_Destroy* command) is provided to the administrator. All the communications used for the staging and the execution of the VMs are implemented on top of the Secure Shell support (ssh). The *VC_Control* command relies on a simple *virtual cluster start-time support* to dynamically configure the network topology and the routing policies on the physical nodes for each virtual cluster.

4 Experiments

Experimental data presented have been collected on a 4U-case Eurotech cluster hosting 4 high-density blades, each of them equipped with a two dual-core AMD Opteron@2.2 GHz and 8 GBytes of memory. Each blade has two Giga-Ethernets and one 10 Gbits/s Infiniband NIC (Mellanox InfiniBand HCA). The blades are connected with a Infiniband switch. Experimental data has been collected on two installations of VirtuaLinux: i) a testbed installation Ubuntu Edgy 6.10 with Xen 3.0.1 VMM, Linux kernel 2.6.16 Dom0 *(Ub-Dom0)* and DomU *(Ub-DomU)*; ii) a reference installation CentOS 4.4, no VMM, Linux kernel 2.6.9 *(CentOS)*.

Two sets of microbenchmarks have been used: the *LMbench* benchmark suite[11], which has been used to evaluate the OS performance, and the *Intel MBI Benchmarks*[10] with *MVAPICH MPI* toolkit (mvapich2-0.9.8)[12], which has been used to evaluate networking performance. According to LMbench, as expected, the virtualization of system calls has a non negligible cost: within both the privileged domain (Ub-Dom0) and user domain (Ub-DomU) a simple syscall pays a consistent overhead ($\sim +700\%$) with respect to the non-virtualized OS (CentOS) on the same hardware (while the difference between the privileged and the user domain is negligible). Other typical OS operations, such as fork+execve, exhibit a limited slowdown due to virtualization ($\sim +120\%$). However, as expected in a para-virtualized system, processor instructions exhibit almost no slowdown. Overall, the OS virtualization overhead is likely to be largely amortised in a real business code.

The second class of experiments is related to networking. Figure 2 left and right report an evaluation of the network bandwidth and collective communications, respectively. Experiments highlight that the only configuration able to exploit Infiniband potentiality is the one using user space Infiniband verbs (that are native drivers). In this case, experiment figures are compliant with state-of-the-art performances reported in literature (and with CentOS installation, not reported here). Since native drivers bypass the VMM, virtualization introduces no overheads. As mentioned in Section 3.2, these drivers cannot be currently used within the VM (DomU), as they cannot be used to deploy standard Linux services, which are based on the TCP/IP protocol. At this aim, VirtuaLinux provides the TCP/IP stack on top of the Infiniband network (through the *IPoverIB*, or *IPoIB* kernel module). Experiments show that this additional layer is a major source of overhead (irrespectively of the virtualization layer): the TCP/IP stack on top of the 10 Gigabit Infiniband (*Dom0_IPoIB*) behaves as a 2 Gigabit network. The performance of a standard Gigabit network is given as reference testbed (*Dom0_GEth*). Network performance is further slowed down by user domain driver decoupling that require data copy between front-end and back-end network drivers. As result, as shown by *DomU_IPoIB* figures, VC virtual networks on top of a 10 Gigabit network, exhibits a Giga-Ethernet-like performances. Results of extensive testing of VirtuaLinux can be found in Aldinucci et al.[3]

5 Conclusions

VirtuaLinux is a novel Linux meta-distribution aiming at installation and the management of robust disk-less high density clusters. Among all features of VirtuaLinux, this paper has introduced virtual clustering architecture, features, and tools. These enable the dynamic and seamless creation and management of ready-to-use VCs on top of Xen VMM. Both the physical cluster and the VCs can be installed with a number of predefined OSes (e.g. Ubuntu Edgy 6.10 and CentOS 4.4) or easily extended to other Linux distributions. VCs managing tools can be easily extended to manage almost any guest Linux distribution by providing VC tools with a tarball of the OS and a simple configuration file.

VirtuaLinux introduces a novel disk abstraction layer, which is the cornerstone of several VirtuaLinux features, such as the time and space efficient implementation of virtual clustering. Preliminary experiments show that VirtuaLinux exhibits a reasonable efficiency, which will naturally improve with virtualization technology evolution. VirtuaLinux is currently distributed with Eurotech HPC platforms. In this regard, Eurotech laboratory

experienced a tenfold drop of clusters installation and configuration time. To the best of our knowledge, few existing OS distributions achieve the described goals, and none achieve all of them.

Acknowledgements and Credits

VirtuaLinux has been developed at the Computer Science Department of the University of Pisa and Eurotech S.p.A. with the partial support of the initiatives of the LITBIO Consortium, founded within FIRB 2003 grant by MIUR, Italy. VirtuaLinux is an open source software under GPL available at http://virtualinux.sourceforge.net/. We are grateful to Peter Kilpatrick for his help in improving the presentation.

References

1. Sun Microsystems, *Sun Grid Engine*, (2007). http://gridengine.sunsource.net
2. E. Pinheiro, W.-D. Weber and L. A. Barroso, *Failure trends in a large disk drive population*, in: Proc. 5th USENIX Conference on File and Storage Technologies (FAST'07), pp. 17–28, San Jose, CA, (2007).
3. M. Aldinucci, M. Torquati, M. Vanneschi, M. Cacitti, A. Gervaso and P. Zuccato, *VirtuaLinux Design Principles*, Tech. Rep. TR-07-13, Università di Pisa, Dipartimento di Informatica, Italy, (2007).
4. R. P. Goldberg, *Survey of Virtual Machine Research*, Computer, pp. 34–45, June, (1974).
5. M. Rosenblum, *The Reincarnation of Virtual Machines*, Queue, **2**, 34–40, (2005).
6. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt and A. Warfield, *Xen and the art of virtualization*, in: Proc. of the 9th ACM Symposium on Operating Systems Principles (SOSP'03), pp. 164–177, (ACM Press, 2003).
7. EVMS website, *Enterprise Volume Management System*, (2007). http://evms.sourceforge.net
8. S. Pratt, *EVMS: A common framework for volume management*, in: Ottawa Linux Symposium, (2002). http://evms.sourceforge.net/presentations/evms-ols-2002.pdf
9. IBM, *Understanding and exploiting snapshot technology for data protection*, (2007). http://www-128.ibm.com/developerworks/tivoli/library/t-snaptsm1/index.html
10. Intel Corporation, *Intel MPI Benchmarks: Users Guide and Methodology Description*, ver. 3.0 edition, (2007). http://www.intel.com/cd/software/products/asmo-na/eng/cluster/clustertoolkit/219848.htm.
11. L. McVoy and C. Staelin, *LMbench: Tools for Performance Analysis*, ver. 3.0 edition, (2007). http://sourceforge.net/projects/lmbench/
12. The Ohio State University, *MVAPICH: MPI over InfiniBand and iWARP*, (2007) http://mvapich.cse.ohio-state.edu/overview/mvapich2/

Performance Analysis

Analyzing Cache Bandwidth on the Intel Core 2 Architecture

Robert Schöne, Wolfgang E. Nagel, and Stefan Pflüger

Center for Information Services and
High Performance Computing
Technische Universität Dresden
01062 Dresden, Germany
E-mail: {*robert.schoene, wolfgang.nagel, stefan.pflueger*}*@tu-dresden.de*

Intel Core 2 processors are used in servers, desktops, and notebooks. They combine the Intel64 Instruction Set Architecture with a new microarchitecture based on Intel Core and are proclaimed by their vendor as the "world's best processors". In this paper, measured bandwidths between the computing cores and the different caches are presented.
The STREAM benchmark[1] is one of the most used kernels by scientists to determine the memory bandwidth. For deeper insight the STREAM benchmark was redesigned to get exact values for small problem sizes as well. This analysis gives hints to faster data access and compares performance results for standard and tuned routines on the Intel Core 2 Architecture.

1 Introduction

For analyzing the details of a computer architecture and its implementation as well as software influences, a convenient performance measuring tool is necessary. For this kind of tasks BenchIT[2,3] has been developed at the Center for Information Services and High Performance Computing at the Technische Universität Dresden. BenchIT implements some features this paper benefits from. A variable problem size for measuring algorithms, remote measurement support, and easy comparison possibilities are some of them.
The memory performance is latency and bandwidth bound. Since the memory bandwidth in modern computer systems does not grow as fast as the arithmetical performance, caches are essential for the performance in most applications. This work will show that the transfer rate is not only bound to the hardware limitations but also depends on software, compiler, and compiler flags.

2 The Measured Systems

In this paper an Intel Core 2 Duo so called "Woodcrest" is the reference object of analysis. A short overview is shown in Table 1, more information can be obtained at the Intel Homepage[4]. Performance results for the other processors listed are presented in Section 4.

3 STREAM Benchmark Related Analysis

STREAM was first presented in 1991 and is a synthetic benchmark, based on different routines which use one-dimensional fields of double precision floating point data. Thus the total performance is bound to several factors: First of all, the total bandwidth in all

parts of the system between the FPU and the highest memory-level, in which the data can be stored. This can be limited by transfer rates between the different memory-levels but also by the width of the result bus for data transfers within cores. Secondly, there is the maximal floating point performance which is high enough in most cases[a].

3.1 STREAM Benchmark Overview

The benchmark consists of four different parts which are measured separately. Every part implements a vector operation on double precision floating point data. These parts are copy, scale, add, and triad. All operations have one resulting vector and up to two source vectors.

3.2 Implementation of Measuring Routines

The original STREAM benchmark is available as a source code in C and FORTRAN, but as binary for several systems as well. A fragment of the C code is listed below.

```
...
#define N 100000
#define OFFSET 0
...
double a [N+OFFSET];
double b [N+OFFSET];
double c [N+OFFSET];
...
  #pragma omp parallel for
  for (j=0;j<N;j++)
    c[j]=a[j];
  ...
  #pragma omp parallel for
  for (j=0;j<N;j++)
    b[j]=scalar*c[j];
  ...
  #pragma omp parallel for
  for (j=0; j<N; j++)
    c[j] = a[j]+b[j];
  ...
  #pragma omp parallel for
  for (j=0; j<N; j++)
    a[j] = b[j]+scalar*c[j];
...
```

Listing 42.1. Fragment of the STREAM Benchmark

3.3 First Performance Measurements

First measurements derived from the STREAM benchmark led to unsatisfying results. The timer granularity was not high enough to determine bandwidths for problem sizes which fit into the L1 Cache. However, results for the L2 Cache can be imprecise as well.
In order to reduce these effects, a better timer has been used (read time stamp counter `rdtsc`). Furthermore, the benchmark now has been adapted to fit the BenchIT-Interface.

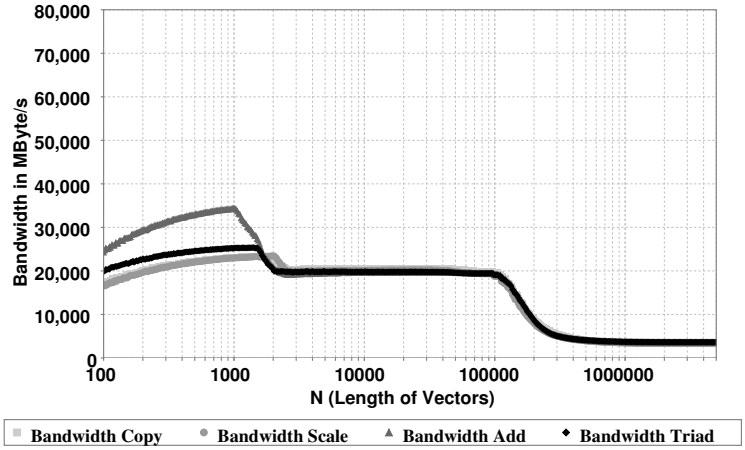


Figure 1. Measurement on Intel Xeon 5160 (woodcrest), non-optimized, compiler flag `-O3`, 1 core

Using the compiler flag `-O3` led to the performance results as shown in Fig. 1.
To benefit from special features of these processors, the compiler flag `-xP` can be used[b]. It adds support for SSE3 operations as well as all previous vectorization possibilities (e.g. MMX) which were added after the IA-32 definition. The usage of vector operations leads to a performance benefit of at least fifty percent which can be seen in Fig. 2.

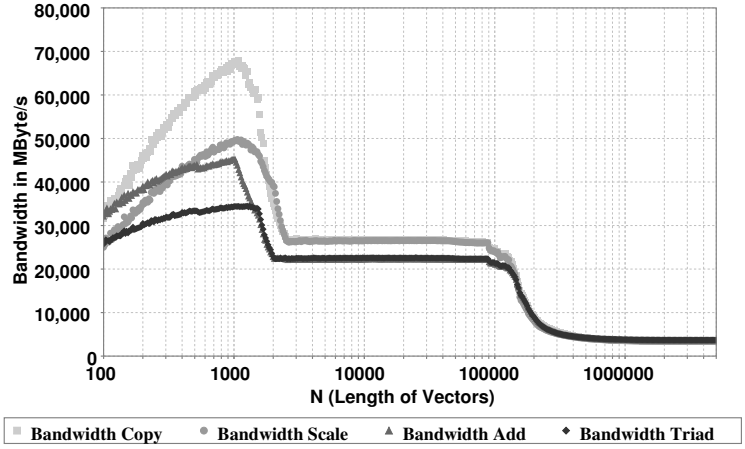


Figure 2. Measurement on Intel Xeon 5160 (woodcrest), optimizing compiler flags, non-optimized, compiler flags `-O3 -xP -openmp`

[a] An exception, for example, is SUNs UltraSPARC T1 which implements only one FPU for up to 8 cores.
[b] With compiler version 10.0 additional flags were introduced especially for the use with Core processors. These are `-xO` and `-xT`

To parallelize the vector operations, STREAM uses OpenMP which is supported by many compilers. When the flag `-openmp` is used additional to those mentioned before, a performance benefit appears in the L2 cache. For small problem sizes, the influence of the parallelizing overhead is too large to gain exact performance results. The complete results can be seen in Fig. 3.

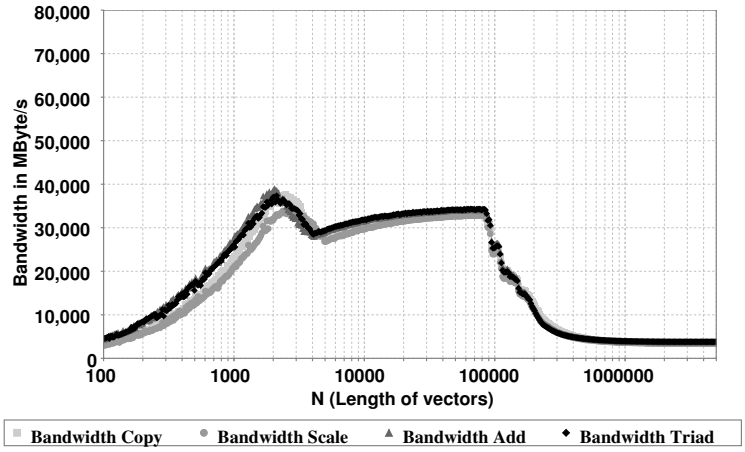


Figure 3. Measurement on Intel Xeon 5160 (woodcrest), 2 cores active (OpenMP), non-optimized, compiler flags `-O3 -openmp`, 2 cores

A performance comparison for different compiler flag combinations shows that when a loop is parallelized with OpenMP, the vectorizations are disabled. The compiler output also indicates that the `LOOP WAS PARALLELIZED` but not `VECTORIZED`.

3.4 Optimizations

As previous results have shown there are challenges which arise from overheads as well as the lack of vectorizing and OpenMP-parallelizing code simultaneously. The overhead causes inaccurate measurements for small problem sizes and can be "reduced" easily. When repeating the functions, the runtime is extended by the number of repeats as well. It may be possible that the compiler removes or alters these repetitions for at least copy and scale. This has been checked in all following measurements and did not occur. A repetition leads to other cache borders in the resulting figure. They are indicated by a shift related to the problem sizes of thirty percent later for copy and scale operations[c].
To combine SIMD- and OpenMP-parallelization, the loop is divided in two parallel parts. The first thread calculates the first half of the vectors, the second thread calculates the other fifty percent. When changing the loop, the timer is also moved into the OpenMP parallel region and surrounds the single vector operations with barriers which also reduces the overhead.

[c]This calculation is based on storing only two vectors with size N in cache instead of three.

However, the resulting performance is not yet as high as possible. Previous measurements have shown that an alignment of 16 bytes helps SSE memory operations to complete faster. The compiler directive `#pragma vector aligned` can be written on top of loops to give a hint that all vectors within this loop are 16 byte aligned[d]. A normal memory allocation does not guarantee this alignment, therefore specific routines should be used. For these cases Intels C-compiler allows the usage of the routine `_mm_malloc(...)` when including headers for SIMD support.
The implementation and usage of these hints and routines achieve a better performance but other negative effects for the L1 cache performance are visible. Looking at the results closely, it appears that the algorithm performs better on problem sizes which are multiples of 16. This fact can be declared easily. If the length of the vector is a multiple of 16, both cores compute on a part which is 64 byte aligned, which complies to the cache line size. When these cases are selected solely, the resulting performance is stable on all memory levels.

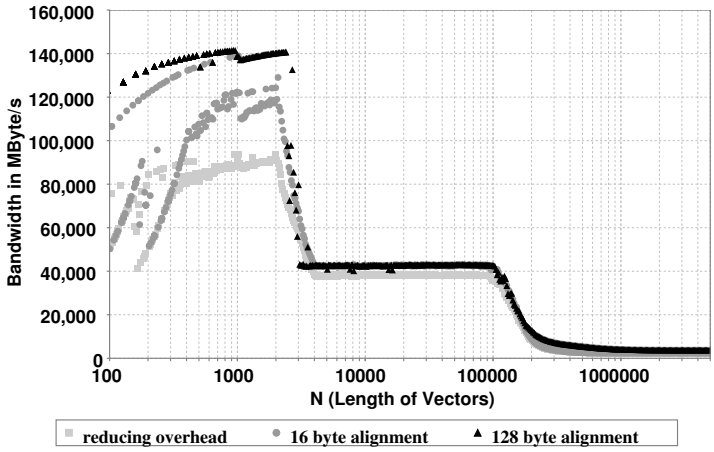


Figure 4. Measurement on Intel Xeon 5160 (woodcrest), Triad, optimized, compiler flags `-O3 -xP -openmp`, 2 cores

As an example the bandwidth for triad is shown for all optimization steps in Fig. 4. The speedup compared to a sequential execution is about 2 within the caches - no matter whether two cores are on the same die (as in the results shown before) or on different dies.speedup results can be seen in Fig. 4.

4 Comparison to other Dual Core Processors

After the performance has been optimized on the Intel Xeon 5160, those results are compared to previous x86 processors by the same vendor. These are an Intel Core Duo T2600

[d] This directive is also available under FORTRAN as `!DEC VECTOR ALIGNED`

and Intel Xeon 5060. A short overview of some key properties are summarized in Table 4. The predecessors in desktop and mobile computing are based on different microarchitectures: Whilst the Xeon 5060 is a representative of the Netburst era, the T2600 represents the Pentium M architecture used for mobile computing. Additionally, an AMD Opteron 285 processor has been tested.

	Intel Xeon 5160	**Intel Core Duo T2600**	**Intel Xeon 5060**	**AMD Opteron 285**
Codename	Woodcrest	Yonah	Dempsey	Italy
Compiler	icc 9.1-em64t	icc 9.1-32	icc 9.1-em64t	icc 9.1-em64t
Clockrate	3.0 GHz	2.167 GHz	3.2 GHz	2.6 GHz
L1 I-Cache per Core	32 kB	32 kB	16 kB	64 kB
L1 D-Cache per Core	32 kB	32 kB	12 kμops	64 kB
L2 Cache	4 MB shared	2 MB shared	2 * 2 MB	2 * 512 kB

Table 1. Overview about measured systems

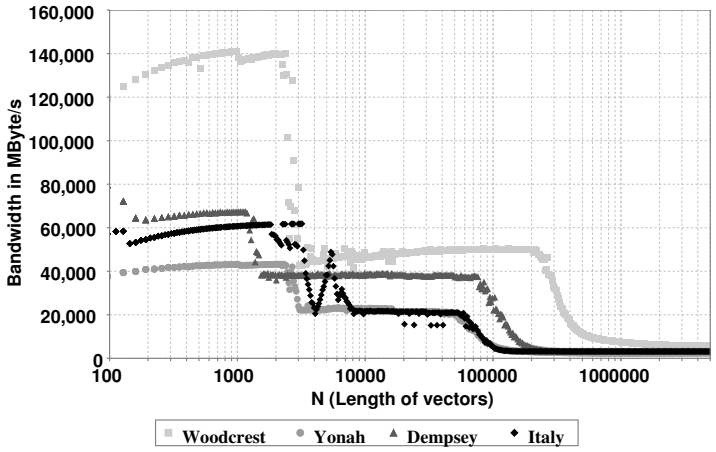


Figure 5. Measurement on different processors, Triad, compiler flags `-O3 -xP -openmp`, 2 cores

The results in Figs. 5 and 6 show that the Core 2 architecture outperforms other processors by at least factor two. The main reason has its origin within the processor core. The result bus was widened to 128 bit and the number of floating point operations that can be performed in one cycle were increased. Also the transfer rate between L1 Cache and core was widened so 32 byte can be read and 32 byte can be written per cycle.

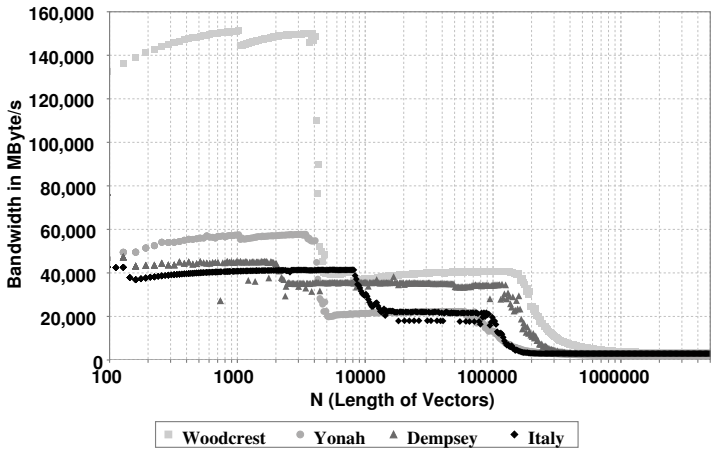


Figure 6. Measurement on different processors, Copy, compiler flags `-O3 -xP -openmp`, 2 cores

5 Conclusion

The Intel Core 2 Duo processors have a very high bandwidth within the cores when memory is accessed linearly. This can be achieved by using high optimizing compilers and architecture specific flags. Compiler optimizations are quite restricted and the user has to optimize manually to achieve reasonable results. When parallelizing loops with OpenMP, benefits from compiler flags may be lost as has been shown. In addition to the optimizing flags, a memory alignment of 128 byte and specific hints for the compiler like `#pragma vector aligned` provide the best performance in this case, significantly outperforming previous x86 processors.

Acknowledgement

This work could not have been done without help and granted access to several computer systems of the Regionales Rechenzentrum Erlangen HPC-Group.

References

1. J. D. McCalpin, *Memory bandwidth and machine balance in current high performance computers*, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, (1995).
2. G. Juckeland, S. Börner, M. Kluge, S. Kölling, W. E. Nagel, S. Pflüger, H. Röding, S. Seidl, T. William, and Robert Wloch, *BenchIT - Performance Measurement and Comparison for Scientific Applications*, Proc. ParCo2003, pp. 501–508, (2004). http://www.benchit.org/DOWNLOAD/DOC/parco2003$_-$paper.pdf
3. R. Schöne, G. Juckeland, W. E. Nagel, S. Pflüger, and R. Wloch, *Performance comparison and optimization: case studies using BenchIT*, Proc. ParCo2005, G. Joubert et al., eds., pp. 877–884, (2006).
4. Intel Corporation, *Intel Xeon Processor Website*, http://www.intel.com/design/Xeon/documentation.htm.

Analyzing Mutual Influences of High Performance Computing Programs on SGI Altix 3700 and 4700 Systems with PARbench

Rick Janda, Matthias S. Müller, Wolfgang E. Nagel, and Bernd Trenkler

Center of Information Services and High Performance Computing
Dresden University of Technology, 01162 Dresden, Germany
E-mail: rick.janda@zuehlke.com
E-mail: {matthias.mueller, wolfgang.nagel, bernd.trenkler}@tu-dresden.de

Nowadays, most high performance computing systems run in multiprogramming mode with several user programs simultaneously utilizing the available CPUs. Even though most current SMP systems are implemented as ccNUMA to reduce the bottleneck of main memory access, the user programs still compete in different ways for resources and influence the scheduler decisions with their generated load.

This paper presents the investigation results of the SGI Altix 3700Bx2 of the TU-Dresden and its successor system the Altix 4700 with the PARbench system. The PARbench system is a multiprogramming and multithreading benchmark system, which enables the user to assess the system behaviour under typical production work load and identify bottlenecks and scheduling problems.

The Altix 3700 and 4700 with their directory based ccNUMA architecture are the largest SMP systems on the market and promise a high scalability combined with a good isolation of the several system nodes. Several tests validates these promised features and analyzes the connection network and the utilized Intel Itanium 2 Madison (Altix 3700Bx2) and dual core Itanium 2 Montecito (Altix 4700) CPUs.

The paper will also show practical problems of the shared system bus by two CPUs each in the 3700 system and compare these situation with the internally shared system bus of the dual core CPUs in the 4700 system.

Further tests examine the load balancing behavior and it's consequences to OpenMP parallelized programs under overload.

1 Introduction

Nowadays, most high performance computing systems run in multiprogramming mode with several user programs simultaneously utilizing the available CPUs. Even though most current SMP systems are implemented as ccNUMA to reduce the bottleneck of main memory access, the user programs still compete in different ways for resources and influence the scheduler decisions with their generated load.

Section 2 gives a very brief overview of the architecture of the Altix systems and shows potential levels of influence. The PARbench system outlined in Section 3 is a powerful tool to analyze resource limitations and the mutual influence of programs. Section 4 analyzes the measured kernel data in order to give a first picture of the performance of the Itanium 2 CPUs and show the OpenMP scalability. The last section presents the load scenarios that analyzes the shared CPU bus, the general scalability and the load balance behaviour.

2 Memory Subsystem of the Altix 3700Bx2 and Altix 4700

The Altix 3700 and 4700 with the directory based ccNUMA architecture are the largest SMP systems on the market. Both systems mainly differ in the utilized Itanium 2 CPUs. An Altix 3700Bx2 system bundles each two Itanium 2 Madison CPUs together with one memory node together on a CPU module. Figure 1 illustrates the overall structure of such a CPU module. The two CPUs share a single system bus to the SHub. The architecture of the Altix 4700 follows the same schema but integrates one or two dual core Itanium 2 Montecito CPUs on the CPU module. The provided 4700 system only possesses one Montecito per module which makes the both systems especially comparable.

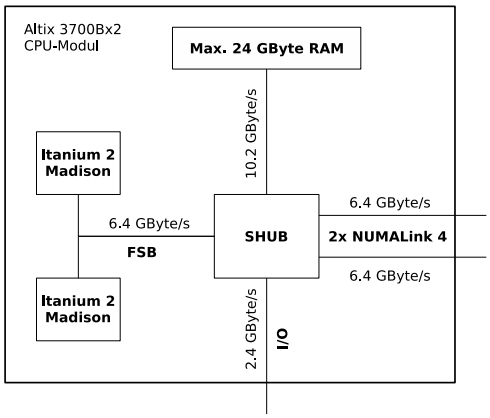


Figure 1. CPU board of an Altix 3700Bx2 system

Up to 256 CPU modules are connected by a NUMALink4 based dual plane fat tree and form a single ccNUMA system with a global shared main memory. The NUMALink connections ensure cache coherency over the whole system. The directory protocol integrated in each SHub reduces the amount of snooping signals for cache coherency by forwarding snooping signals to affected system nodes only. Together with an optimized memory placement that tries to allocate memory as near as possible to the CPU, the allocating process running on, the system architecture promises high scalability together with good isolation of the system nodes. Part of the evaluation was to validate these promised features.

3 Main Features of the PARbench System

OpenMP is a widely used approach to parallelize calculations on SMP systems. In the past, a lot of work has been presented that assesses the performance of OpenMP programs on dedicated SMP systems or measure the runtime performance under quite favourable circumstances and say nothing about the interaction of several user jobs in production environment.

PARbench was designed to address exactly this issue. The PARbench system enables the user to generate almost arbitrary load scenarios in benchmark capable runtime based

on synthetic benchmark kernels.

For this reason, 28 mathematical cores are combined with customizable data volumes, different strides in memory access and the OpenMP parallelization on customizable amounts of threads to a large amount of benchmark kernels with very different characteristics and very short runtime in the range of milliseconds. In addition the user can define different metrics based on hardware performance counters and measured times. Typical defined metrics could be Main Memory Transfer, FLOPS, Cache Transfers, Write Fraction, Bus-Snoops/s, I/O-Transfer, Load Balance or Parallelization Efficiency.

The initial Base Data Mode measures all defined metrics for every kernel on a dedicated system or subsystem and store them for later program generation. According to the described load scenarios, the PARbench system synthesizes sequences of the synthetic kernels, that create the described program behaviour and have the specified runtime on a dedicated system. The generation algorithm ensures uniform behaviour distribution across the whole runtime of each program and a good variation of the different metrics around the specified average behaviour. The subsequent execution of the whole load scenario measures the runtime and the CPU times of each program in the scenario. A GUI enables the user to analyze all measured results and compare them to each other.[7] contains a detailed description of the PARbench system.

4 Discussion of the Kernel Base Data

The evaluation of the measured kernel base data permits first conclusions about the CPU performance and the OpenMP scalability. For all following tests, the kernels were configured to the following parameters:

- Data volumes of 240 KByte, 2500 KByte and 25 MByte to place variations of each core in different levels of the memory hierarchy
- Predefined memory access strides (1, 2, 4, 8, 16, 32)
- OpenMP parallelization up to 32 threads (1, 2, 4, 6, 8, 12, 16, 24, 32)

The combination of these parameters with the mathematical cores generates an amount of about 3800 benchmark kernels.

4.1 Basic CPU Performance Data

Analyzing the measured metrics of the benchmark kernels outlines the general performance data of the utilized Itanium 2 CPUs. Figure 2 shows the achieved main memory transfer rate and the FLOPS of all sequential kernels on the Altix 4700 with the Montecito CPUs at 1.6 GHz. The older Madison in the Altix 3700Bx2 shows almost the same distribution but with less peak performance. For the Montecito, several kernels allmost reach the theoretical peak performance of 6.4 GFlops. The Madison could only reach about 5.2 GFlops at 1.5 GHz, which extrapolates to 5.5 GFlops at 1.6 GHz. The Montecito can leverage its improved cache subsystem here. A new second level instruction cache of one MByte was introduced in Montecito. Madison, in comparison, only possesses a 256 kByte shared second level cache for instructions and data. The conducted tests also showed, that

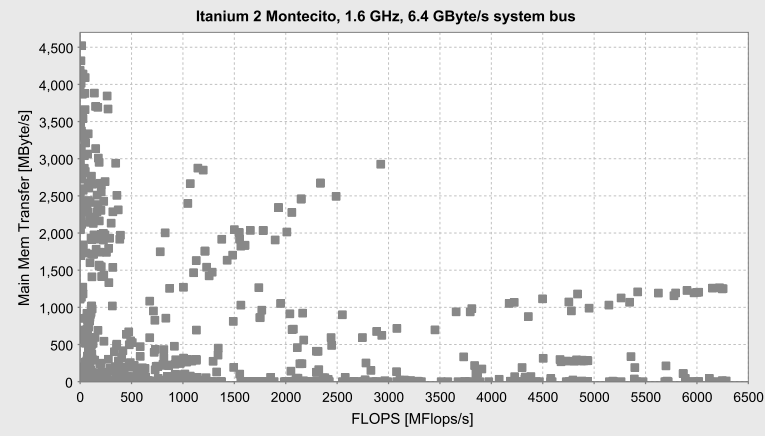


Figure 2. Floating point performance and main memory transfer of the utilized Itanium 2 Montecito (Intel Fortran Compiler 10.0.13 with highest optimization)

only the newest Intel compiler in version 10 was able to leverage the huge second level instruction cache, that was introduced in the Montecito. Compilations based on the version 9.1 reached only about 5.6 GFlops which corresponds to the extrapolated result from the Madison.

Considering the measured main memory transfer, it becomes clear that it is not possible to utilize the system bus with a single CPU respective a single CPU core. Madison and Montecito both only reaches about 4.5 GByte/s, which is far from the theoretical maximum of 6.4 GByte/s. This result is also supported by the work in[6].

4.2 OpenMP Scalability

In the given setup, the cores are parallelized with up to 32 OpenMP threads. Figure 3 shows the parallelization efficiency of all kernels with 32 threads. Many kernels perform very well with an efficiency of almost 100 percent. Some kernels show super linear speedup which is due to the increased overall amount of cache available with more CPUs. The kernels with very low efficiency are included by purpose and designed to stress the cache coherency protocol in multithreaded environments by means of accessing the data matrices in unfavourable direction. As of both systems use the same NUMAlink 4 based connection network, they both show equal results in this area.

5 Discussion of the Load Tests

5.1 Shared CPU bus as bottle neck

Each CPU board of the evaluated Altix 3700Bx2 system connects two Itanium 2 Madison CPUs over a shared system bus with the local SHub and therewith with the local memory node and the overall system. Other investigations have shown, that both CPUs together can utilize the available 6.4 GByte/s of the system bus in benchmark situations. The question

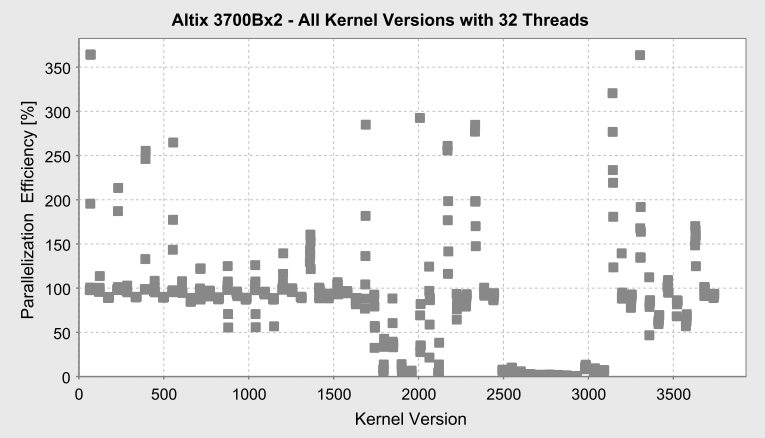


Figure 3. OpenMP scalability to 32 threads

is, how large is the impact in a practical environment, wherein both CPUs do not always use the bus in the same direction but create opposite requests. For this purpose, a load of eight sequential programs was executed on a subdivided CPU subset with four CPU boards. The programs were generated to an average main memory transfer of 2.14 GByte/s with mixed read/write access and 100 seconds runtime in a load free system. The memory transfer on the system bus of each of the four CPUboards should be about 4.4 GByte/s, which is far from the theoretical maximum of 6.4 GByte/s. Theoretically, the two programs should not interfere each other. The test result in Fig. 4 illustrates a different behaviour. The runtime of each program increases on average by 24.5 percent. The programs already influence each other considerably. The possible reasons could be bus read/write turn around cycles, that are necessary each time the transfer direction changes on the bus.

Figure 4. Runtime extension because of influences on the system bus (Altix 3700Bx2)

In order to determine the practical bus saturation, the test was repeated with different main memory transfer rates for all eight programs. The same tests were also conducted on the newer Altix4700 system, with one dual core Montecito CPU on each CPU board. Figure 5 shows the combined results of all these tests. Each bar represents the average runtime of all eight programs in a single load test. For the Altix 3700Bx2 system the measured performance impact impact is less than 10% only at a generated main memory transfer below 1 GByte/s per program. The two CPU cores in the Montecito show considerablely lesser mutual influence on their shared system bus which leads to only half the runtime extension in comparison to the two Madisons.

5.2 Overall System Scalability

To investigate the general system scalability, the first presented load test was repeated with larger amount of CPUs and appropriate amount of programs in the load scenario. The statistical examination does not show any changes in the runtime behaviour. The average runtime does not increase and also the min/max value remains the same.

5.3 Scheduling Behaviour Under Overload

Experiments with sequential overload clearly shows that the scheduling focuses on data locality and not on fair resource distribution. Figure 6 shows 17 sequential programs on 16 CPUs. 15 programs will be executed without interruption. The remaining two programs share one CPU until another CPU becomes available. According to the first touch policy, memory is allocated on the local memory node of a cpu board whereon a process or thread is executed at the time of allocation. Even if the process/thread is migrated to another CPU module, the data keeps on the same memory node and must be accessed by slower remote memory accesses. By binding processes and threads as long as possible to the same CPU, slower remote memory accesses are minimized. With this strategy, migrations are only

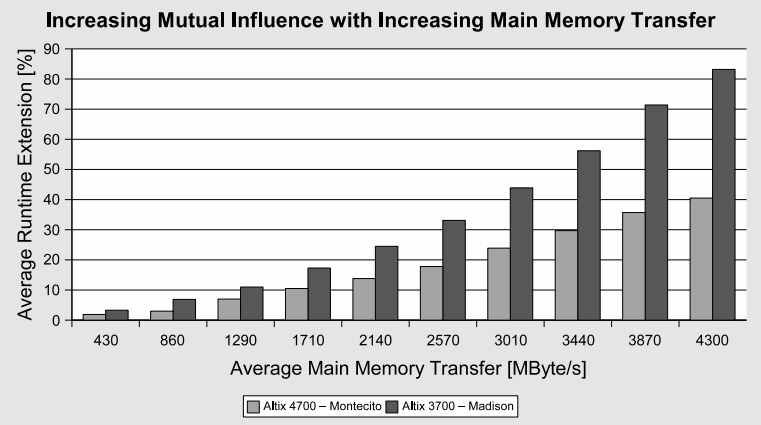


Figure 5. Runtime extension because of influences on the system bus

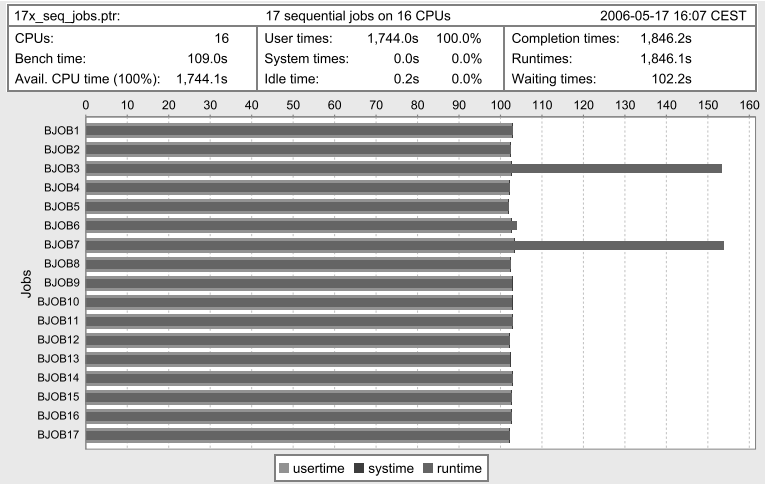


Figure 6. 17 sequential programs on 16 CPUs - focus on data locality

performed in order to utilize idle CPUs. Overload is not equally shared over all available CPUs.

This scheduling strategy has a strong influence on the behaviour of OpenMP programs and multithreaded programs in general. Figure 7 shows how multithreaded programs behave under overload. For this purpose, a scenario of 12 sequential jobs and four parallel jobs was executed on a 32 CPU subsystem.

Whereas the sequential programs in this high overload can almost finish their work undisturbed, the parallel programs substantially increase their runtime in comparison to the generated runtime in a dedicated environment. This behaviour does not significantly change, if the sequential programs are executed with lower scheduling priority. The reason for this stems from the scheduling behaviour and the way in which additional threads are spawned. If an additional thread is spawned by a process, the new thread is placed on the same CPU as the spawning process. The experiments with sequential overload already showed that the scheduler migrates only in order to utilize idle CPUs. Because all CPUs are already occupied by at least another process or thread, the additional threads of the OpenMP programs remain on the same CPU as the master thread. Under this circumstances, the further threads can not be leveraged and each OpenMP program is practically serialized on a single CPU.

6 Conclusion

The conducted tests showed the possible calculation power of the Itanium 2 CPUs. The improved cache subsystem of the Montecito leads to considerable more performance but only in conjunction with the newest Intel Fortran Compilers in version 10. The directory bases ccNUMA protocol together with the localized memory placement clearly confines the different system nodes and reduces the interaction effectively. High scalability and very good system node isolation could be clearly seen. The system bus of the CPUs showed

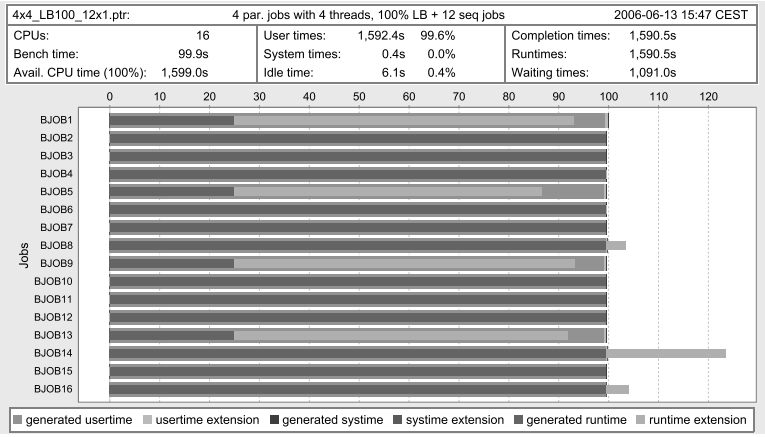


Figure 7. OpenMP programs under overload

some practical problems. One CPU/CPU core can not fully use it and two CPU/CPU cores influence each other to a large extent. With the integration of the two CPU cores on a single die in the Montecito the mutual influence over the system bus significantly decreased. The tests also showed that the scheduler clearly focuses on data locality and does not spread overload equally over the CPUs. Under these conditions, OpenMP programs practically becomes serialized under overload.

References

1. D. Lenoski et al., *The Standford Dash Multiprocessor* IEEE Computer Bd. **25**, 1992, 63–79, (1992).
2. M. Woodacre, D. Robb, D. Roe and K. Feind, *The SGI Altix 3000 Global Shared-Memory Architecture*, White Paper, (2003).
3. J. Aas, *Understanding the Linux 2.6.8.1 CPU Scheduler*, White Paper (2005). http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf
4. Intel, Inc., *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*. http://www.intel.com/design/itanium2/manuals/251110.htm
5. SGI, *SGI NUMAlink - Industry Leading Interconnect Technology*, White Paper (2005). http://www.sgi.com/pdfs/3771.pdf
6. G. Juckeland, *Analyse der IA-64 Architektur: Leistungsbewertung und Möglichkeiten der Programmoptimierung*, Center for Information Services and High Performance Computing, Dresden University of Technology, Diploma thesis, (2005).
7. R. Janda, *SGI Altix: Auswertung des Laufzeitverhaltens mit Hilfe von neuen PARbench-Komponenten*, Center for Information Services and High Performance Computing, Dresden University of Technology, Diploma thesis, (2006).

Low-level Benchmarking of a New Cluster Architecture

Norbert Eicker[1] **and Thomas Lippert**[1,2]

[1] Jülich Supercomputing Centre, Forschungszentrum Jülich
52425 Jülich, Germany
E-mail: {n.eicker, th.lippert}@fz-juelich.de

[2] Department C, Bergische Universität Wuppertal
42097 Wuppertal, Germany

JULI is a first of its kind project at Forschungszentrum Jülich accomplished during 2006. It was carried out to develop and evaluate a new cluster architecture. Together with the companies IBM, QLogic and ParTec a state-of-the-art cluster system was designed based on PPC 970MP CPUs and a this time novel PCIe version of the InfiniPath interconnect. Together with the ParaStation cluster management system these components guaranteed an effortless handling of the system resulting in start of full production by end of 2006.

We present results of synthetic, machine-oriented benchmarks shedding light on the capabilities of JULI's architecture. Problems revealed include the latency of the inter-node communication which suffers from many stages between CPU and host channel adapter (HCA). Furthermore, general issues like the memory-bandwidth bottlenecks of the dual-core processors—which will become more severe on upcoming multi-core CPUs—are discussed.

1 Introduction

The JULI[a] project is the first in a series of investigations carried out at the Zentralinstitut für Angewandte Mathematik (ZAM) of the Forschungszentrum Jülich (FZJ). The mission of these projects is to develop and evaluate new state-of-the-art cluster platforms to prepare the upgrade of FZJ's general purpose supercomputer system in 2008.

Following this philosophy, the JULI project aims at integrating very compact compute hardware with lowest power consumptions, smallest latency interprocessor communication, and best possible process management systems. As project partners from industry we have chosen IBM Development Lab Böblingen (Germany)[1] for node hardware and various software components, QLogic[2] as contributor of the InfiniPath interconnect and corresponding MPI library and ParTec[3] with their ParaStation cluster middleware.

A first meeting for preparation of JULI was held in November 2005 only shortly before the official start begin of 2006. A first prototype of the project's hardware solution was presented at IBM's booth during ISC 2006 in Dresden. Already mid of August a first version of the system has been installed in Jülich. Accompanying an extensive benchmarking period, parts of the hardware have been upgraded for performance optimization. In December 2006 production started.

Within this paper we will present and discuss results of synthetic low-level benchmarks on JULI. This will shed light on the most important hardware features of the system, like effective latency and effective bandwidth of memory and communication hardware or performance of the FPU. These parameters are the basis for any performance estimate in high-performance computing[4].

[a] **JU**elich **LI**nux Cluster

The paper is organized as follows: In the next section a brief overview of the JULI cluster is given. In Section 3 we will introduce the two synthetic low-level benchmarks used to analyze the cluster and present the attained results. We end with a summary and draw conclusions from the presented results.

2 Hardware and Software

2.1 Compute Nodes

JULI consists of 56 IBM JS21 BladeServers. Each BladeServer is equipped with 2 Dual-Core PowerPC 970MP CPUs running at 2.5 GHz. The PowerPC CPU has a pipelined, super-scalar architecture with out-of-order operations. Each core features two 21-stage floating-point units (FPU). Every FPU allows one double-precision multiply-add operation per cycles. All this leads to a theoretical peak-performance of 10 GFlop/s per core.

Both cores have their own hierarchy of two caches. While the L1 cache is segmented into 32 kB for data and 64 kB for instructions, the 1 MB L2 cache is used for both data and instructions. The main difference besides its size is the latency of the caches access. While it takes 2 cycles to fetch data from L1 cache, the processor has to wait 14 cycles until data from the L2 cache are available.

For the JS21 blades two varieties of DDR2 memory are available as main memory; slower SDRAM modules, running at 400 MHz, and faster ones, clocked with 533 MHz. In fact, we were able to test both types of memory. This enabled us to study the memory sub-system in detail and to analyze the effects of the different memory-speeds on both synthetic low-level benchmarks and real-world applications.

2.2 Networks

The nodes are interconnected by means of three networks. Two networks are implemented by Gigabit-Ethernet technology. They are responsible for I/O and management activities. The third network is for the actual applications, i.e. MPI. Here a 10-Gigabit technology is used: QLogic's implementation of the InfiniBand[7] standard called InfiniPath[5,6]. Its most remarkable feature is an extremely low latency compared to other implementations of this standard. This is due to the fact that the protocol is not handled by a full-fledged CPU on the HCA but is mapped to the logic of a state-machine implemented within an ASIC.

Nevertheless the wire-protocol is perfectly conforming with the InfiniBand standard. As a result, standard InfiniBand switches can be employed. In the case of JULI, we choose a Voltaire ISR 9096 switch with three line-cards summing up to a total of 72 ports.

2.3 Software

The software configuration of JULI mostly follows the main-stream in cluster computing. Linux is used as the operating system. We chose the PPC-version of SLES 10. For compatibility reasons, gcc is available. Nevertheless IBM's XL compiler is the default since it provides significantly better optimization for the PowerPC-architecture. We chose the IBM XL C-compiler for the tests we present within this article. Nevertheless for comparison reasons we also ran the tests using the gcc compiler and found no significant difference.

This is no surprise since these low-level benchmarks directly test hardware capability and should be insensitive towards optimizing compilers by design.

Of course some deviations from the main-stream were necessary: As MPI-library QLogic's InfiniPath-MPI was used rather than MPICH. It supports the InfiniPath interconnect in a most efficient way. Furthermore the process management does not employ the usual ssh/rsh constructs but relies on ParaStation, jointly developed by ZAM and ParTec Cluster Competence Center, Munich.

3 Benchmarks

In order to determine some of the crucial performance parameters of the system at low level, we employed two synthetic, machine-oriented benchmark-codes. The results enable us to set expectations on the performance of real world application.

While the effects of system characteristics on the performance of actual real world applications in general are hard to predict, the knowledge of these characteristics is important in order to be able to set limits on performance expectations. As an example, if it is possible to get an estimate which memory-bandwidth is actually achievable in practice, this will set an upper bound on the performance expectations of any algorithm which stresses the memory subsystem of a computer.

The tests of real world applications were done in parallel with the analysis presented here; the results are presented in a separate publication[11].

3.1 FPU Performance

In order to determine the on-node capabilities of JULI we ran the `lmbench`-suite[8,9] containing low-level performance benchmarks on one of JULI's compute-blades.

The latencies and throughput values of the PPC 970MP processor from the data-sheet are redisplayed in columns 2 and 3 of Table 1. We confirmed the specifications by carrying out the corresponding test within the `lmbench` suite.

op.	latency	throughput	single	double
add	6	2 / cycle	2.38	2.38
mult	6	2 / cycle	2.38	2.38
div	33	2 / 28 cycles	13.1	13.1

Table 1. FPU execution times in nsec for PPC 970MP from `lmbench`

The benchmark results are presented in columns 4 and 5. Based on a cycle-time of 0.4 nsec—in correspondence with a 2.5 GHz processor clock—the results agree with the values in the data-sheet. Here one has to consider that `lmbench`'s way to explore the floating-point performance does by intention not make use of the pipelining features of modern CPUs. Thus the time measured for `add` or `mult` in fact is the actual latency of the corresponding operation. In a sense the reported numbers are a kind of worst case performance for the given architecture besides further limitations that might have been introduced by restrictions of the memory sub-system, etc.

3.2 Memory Characteristics

Another set of results from the `lmbench` suite is discussed with the aim to understand the memory sub-system of the JS21 blades. The corresponding numbers are presented in Table 2.

As mentioned above, during the JULI project the compute-nodes have been equipped with two different types of memory-modules: JULI started with 400 MHz DDR2 SDRAM modules and upgraded to faster memory running at 533 MHz. This enables us to make interesting comparisons concerning the effects of the memory-bandwidth.

MHz	procs	cores	BW [MB/sec]		latency [nsec]		
			read	write	L1	L2	mem
400	1	any	2750	1740	1.19	5.2	40.7
	2	0 1	4000	2280	1.19	5.24	60.5
		0 2	4480	2295	1.19	5.24	52.2
	4	0 1 2 3	5090	2280	1.19	5.24	99.5
533	1	any	2830	1810	1.19	5.2	39.1
	2	0 1	4020	2590	1.19	5.24	60.0
		0 2	4870	2730	1.19	5.24	45.3
	4	0 1 2 3	6141	2635	1.19	5.24	81.6

Table 2. Results for bandwidth and latency from `lmbench`'s memory test suite.

The upper part of the table shows results obtained with the older and slower memory (400 MHz), the lower part results are obtained with the faster modules (533 MHz). The four lines of each block show the outcome of the test running with 1, 2 or 4 instances simultaneously. Column 2 denotes the number of instances. The difference between the two lines referring to two instances is due to the cores that are used within a single test as indicated in column 3. For this purpose the two processes conducting the operations were bound to core 0 and 1 or 0 and 2, respectively, by using the `sched_setaffinity` functionality of the Linux kernel.

Columns 4 and 5 of Table 2 show the results for consecutive reads and writes to the main memory testing the memory-bandwidth of the system. The results for read operations are significantly larger than for write operations. In HPC practice this should be no major problem since for most algorithms the reading access to memory dominates the write operations.

It is clearly visible that for all applications with a performance-characteristic that is sensitive to memory-bandwidth we cannot expect a linear scaling within a node. Even for the faster memory the total read-bandwidth obtained for four processes is only twice as large as the one we see for one process.

The last 3 columns of Table 2 show latencies as determined by `lmbench`. As expected we see no dependence on the type of memory used when testing L1 or L2 cache as displayed in column 6 and 7, respectively. In the last column we present the latencies to access the main memory. Here effects concerning the type of memory only show up if the instances of the test make use of both sockets.

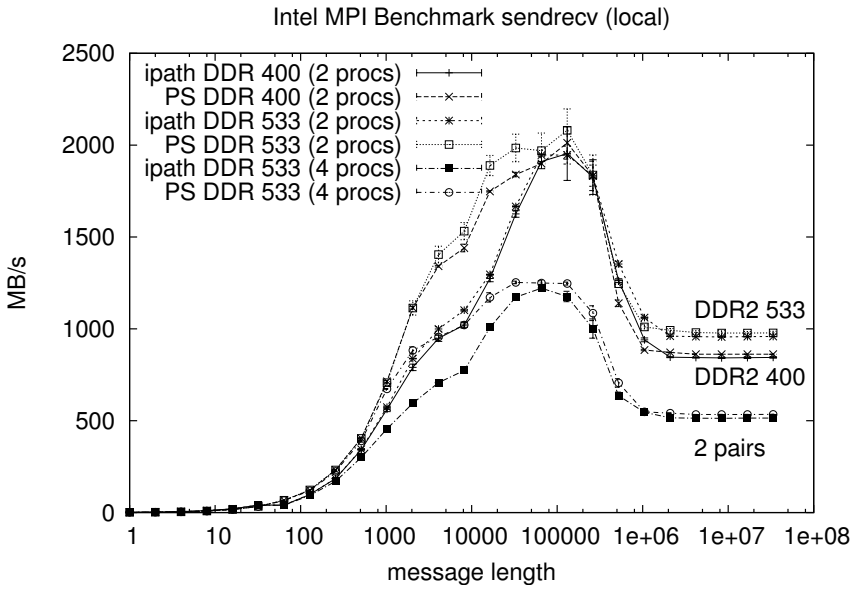


Figure 1. Intra-node MPI-communication bandwidth for ParaStation-MPI (PS) and InfiniPath-MPI (ipath). Tests utilizing two processes where executed using both types of memory – DDR2400 MHz and DDR2533 MHz. Tests with four processes only use the faster memory.

Comparing these numbers to the ones from the data-sheet given in Section 2.1, we see that in practice accessing the L1 cache takes 3 cycles instead just 2 cycles. On the other hand we see that data from L2 cache is already available after 13 cycles in most cases – in contrast to the 14 cycles guaranteed by the documentation.

3.3 The Intel MPI Benchmark

Depending on the programming model, often MPI intra-node communication is required within a parallel application. Therefore we investigate both intra-node and inter-node communication performance. We employ the Intel MPI Benchmark (IMB)[10].

3.3.1 Intra-node communication

Communication between processes allocated to the same node is expected to be highly dependent on the available memory bandwidth. In general, this type of operation will not require the communication-hardware but relies on a segment of shared memory used to copy the data from the address-space of one process to another.

Therefore IMB is another tool well suited to get a feeling of the capabilities of the memory sub-system of the JS21 blades. Within our tests of the intra-node communication we made use of two different implementations of MPI: on the one hand, we employed QLogic's InfiniPath-MPI library that also supports communication between the nodes. On the other hand, we used an implementation of MPI which is part of ParTec's ParaStation

suite. ParaStation will serve as a reference-implementation of local shared-memory communication.

Figure 1 shows results of IMB's `sendrecv`-test for various combinations of MPI-implementation, type of memory-modules and number of processes. We carried out tests between two processes (i.e. one pair passing messages) with both types of memory and both MPI implementations. Furthermore, results obtained from runs with 2 pairs of processes on the fast memory are presented.

It can be observed that the ParaStation MPI shows consistently better performances than InfiniPath-MPI. This is true for both types of memory and over the whole range of message lengths. In all cases one has to discriminate two regions of results: all message-sizes smaller than about 512 kB will be handled within the caches of the involved CPU[b]. Only messages larger than this threshold will actually be sent via the main memory. Accordingly, only results for larger messages are sensitive to the different memory speeds.

For both regions the ParaStation implementation is superior, even if the difference for large messages is just in the 5 % range. Only in the region of 1 MB messages is QLogic's implementation on par with the ParaStation MPI.

3.3.2 Inter-Node Communication

Inter-node communication makes use of the high-speed InfiniPath network. Since the ParaStation-MPI has no optimized support for this type of interconnect, only results using the InfiniPath-MPI are presented. We ran all tests for both types of memory. However, we saw almost no differences. We conclude that the memory bandwidth is not a bottleneck for the inter-node communication.

By means of IMB's `pingpong` benchmark we determined the network latency on MPI-level. We notice a half round-trip time of less than 2.75 μsec, which is good compared to other InfiniBand implementations. However, the difference to results with InfiniPath on other PCIe platforms—less than 2 μsec—is still substantial. As a comparison, we also ran the `pingpong` test on another Cluster in Jülich. This one is equipped with InfiniPath and two dual-core Opteron CPUs per node. The latencies observed here are as low as 1.7 μsec or 1.9 μsec depending on the CPU used[c].

We conclude that the complex architecture of the JS21 blades and the long path from the PPC CPUs to the InfiniPath HCAs presumably is responsible for this result. In fact, besides a comparable south-bridge in both, the JS21 and the Opteron nodes, the JS21 involves a north-bridge which introduce the additional latency. However, it is beyond the scope of this paper to analyze in detail the actual cause of this problem.

Further tests were carried out in order to analyze the bandwidth of the interconnect. Again we apply the `sendrecv` benchmark of IMB. In Figure 2 one has to distinguish three sets of tests: the two topmost graphs show bandwidth-results for communication between two processes, i.e. one pair of processes passing messages to each other. For the two graphs in the middle two pairs of processes make use of the same HCAs and physical

[b] Since the `sendrecv`-test concurrently sends and receives data to separate buffers, only half of the cache can be used for a single message. Additionally the shared-memory segment occupies further cache-lines.

[c] Within the Opteron nodes the south-bridge hosting the PCIe-subsystem is directly connected by a HyperTransport (HT) link to one of the CPU-sockets (the one with core 0). All other CPU-sockets have to go via the first socket in order to reach the south-bridge.

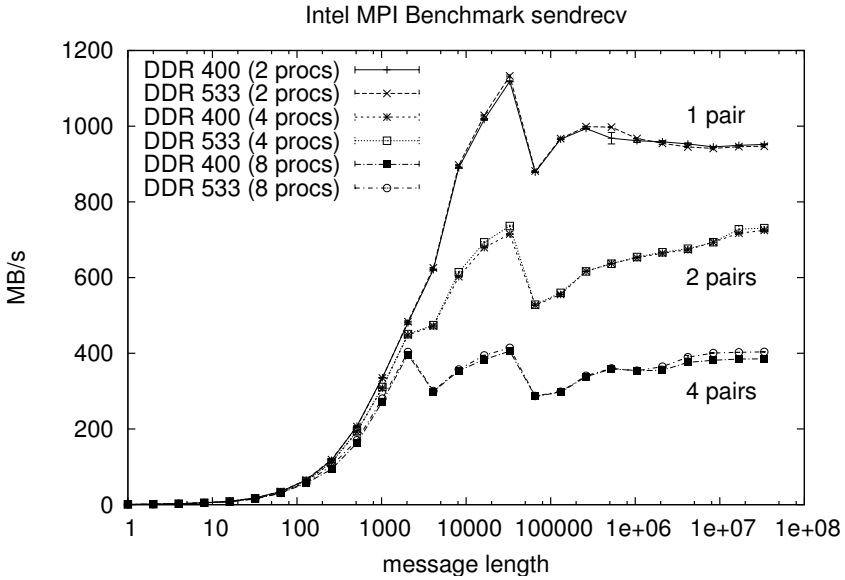


Figure 2. Inter-node MPI-communication bandwidth on JULI. Results for 1, 2 and 4 pairs of processes passing messages to each other concurrently are shown. Bandwidth shown is per pair of processes.

wire at a time. The two graphs at the bottom are for four pairs of processes occupying a single physical wire during the measurement.

Again, two regions have to be distinguished: For messages up to a length of about 2000 bytes the bandwidth-results do not depend significantly on the number of process-pairs passing data. This changes above this threshold where the observed bandwidth per pair of processes is strongly correlated to the number of pairs. In fact not the message-length is the crucial parameter, but the total amount of bandwidth occupied – which in fact also depends on the message-length. For 2k messages we see a bandwidth of about 400 MB/s per pair. Employing 4 pairs this leads to a total (bidirectional) bandwidth of 1.6 GB/s – which is close to the physical limit of 2 GB/s.

Nevertheless one puzzle remains for large messages: While we see a total bidirectional bandwidth of 1.6 GB/s using 4 pairs, this drops to 1.4 GB/s for two pairs and even below 1 GB/s if only one pair of processes generates traffic.

The reason for this behaviour might be a sub-optimal implementation of the actual transport layer within the InfiniPath-MPI. A good indication that this is the case might be found in the fact that we see a break-down of bandwidth for messages of size 64k. At this message-size other MPI-implementation like MPICH switch to a rendezvous-protocol. A more fine-tuned version of the InfiniPath-MPI library might fix this problem.

4 Summary and Conclusion

The conclusions we can draw from JULI extend the project and the type of hardware involved. The finding that dual-core or multi-core architectures will complicate life in HPC

significantly is not surprising: until now the growth-rate of processor performances was slightly larger than the increase of memory-bandwidth. With the appearance of architecture with multiple cores the amount of memory bandwidth required to balance the system for HPC increases by factors significantly larger than 1 for each generation of processors.

Furthermore, the JULI project shows that in HPC we have to minimize the distance between processor and HCA, i.e. we need to keep the complexity of the infrastructure in between as small as possible. In the case of the JULI cluster the latency penalty for inter-node communication already is significant: the time a CPU has to wait for data from an interconnect will become more and more expensive with increasing number of cores: JULI's penalty of 1.0 μsec compared to other PCIe architectures corresponds to $\mathcal{O}(2500)$ cycles of each CPU, a number that already amounts to $\mathcal{O}(40000)$ floating point operations per node.

5 Acknowledgements

We thank Ulrich Detert, who is responsible for the JULI system at FZJ for his continuous support.

We are grateful to Otto Büchner for his kind support of our test on the JUGGLE Opteron Cluster in Jülich.

Furthermore we thank the teams of our project partners, ParTec, QLogic and IBM-Böblingen, who together with the ZAM-team have developed JULI with remarkable enthusiasm and devotion.

References

1. http://www-5.ibm.com/de/ibm/produkte/entwicklung.html
2. http://www.qlogic.com
3. http://www.par-tec.com
4. An overview on available models can be found in: E. Sundarajan and A. Harwood, *Towards parallel computing on the internet: applications, architectures, models and programming Tools*. arXiv:cs.DC/0612105.
5. QLogic InfiniPath Install Guide Version 2.0 (IB0056101-00 C).
6. QLogic InfiniPath User Guide Version 2.0 (IB6054601-00 C).
7. http://www.infinibandta.org
8. L. McVoy and C. Staelin. "lmbench: Portable tools for performance analysis". In Proc. Winter 1996 USENIX, San Diego, CA, pp. 279-284.
   C. Staelin. "lmbench – an extensible micro-benchmark suite." HPL-2004-213.
9. http://www.bitmover.com/lmbench/
10. http://www.intel.com/cd/software/products/asmo-na/
    ... eng/307696.htm#mpibenchmarks
11. U. Detert, A. Thomasch, N. Eicker, J. Broughton (Eds.) JULI Project - Final Report; Technical Report IB-2007-05

Comparative Study of Concurrency Control on Bulk-Synchronous Parallel Search Engines

Carolina Bonacic[1] **and Mauricio Marin**[2]

[1] ArTeCS, Complutense University of Madrid, Spain
CEQUA, University of Magallanes, Chile
E-mail: cbonacic@fis.ucm.es

[2] Yahoo! Research, Santiago, Chile
E-mail: mmarin@yahoo-inc.com

In this paper we propose and evaluate the performance of concurrency control strategies for a parallel search engine that is able to cope efficiently with concurrent read/write operations. Read operations come in the usual form of queries submitted to the search engine and write operations come in the form of new documents added to the text collection in an on-line manner, namely the insertions are embedded into the main stream of user queries in an unpredictable arrival order but with query results respecting causality.

1 Introduction

Web Search Engines use the inverted file data structure to index the text collection and speed up query processing. A number of papers have been published reporting experiments and proposals for efficient parallel query processing upon inverted files which are distributed on a set of P processor-memory pairs[1,2,3,4,5]. It is clear that efficiency on clusters of computers is only achieved by using strategies devised to reduce communication among processors and maintain a reasonable balance of the amount of computation and communication performed by the processors to solve the search queries.

An inverted file is composed of a vocabulary table and a set of posting lists. The vocabulary table contains the set of relevant terms found in the collection. Each of these terms is associated with a posting list which contains the document identifiers where the term appears in the collection along with additional data used for ranking purposes. To solve a query, it is necessary to get the set of documents ids associated with the query terms and then perform a ranking of these documents so as to select the top K documents as the query answer.

Query operations over parallel search engines are usually read-only requests upon the distributed inverted file. This means that one is not concerned with multiple users attempting to write information on the same text collection. All of them are serviced with no regards for consistency problems since no concurrent updates are performed over the data structure. Insertion of new documents is effected off-line in Web search engines. However, it is becoming relevant to consider mixes of read and write operations. For example, for a large news service we want users to get very fresh texts as the answers to their queries. Certainly we cannot stop the server every time we add and index a few news into the text collection. It is more convenient to let write and read operations take place concurrently. This becomes critical when one think of a world-wide trade system in which users put business documents and others submit queries using words like in Web search engines.

The concurrency control algorithms we propose in this paper are designed upon a particular way of performing query processing. In Section 2 we describe our method for read-only queries and in Section 3 we extend it to support concurrency control under read-/write operations. Section 4 presents an evaluation of the algorithms and Section 5 presents concluding remarks.

2 Query Processing

The parallel processing of queries is basically composed of a phase in which it is necessary to fetch parts of all of the posting lists associated with each term present in the query, and perform a ranking of documents in order to produce the results. After this, additional processing is required to produce the answer to the user. At the parallel server side, queries arrive from a receptionist machine that we call the *broker*. The broker machine is in charge of routing the queries to the cluster processors and receiving the respective answers. For each query the method produces the top-K documents that form the answer.

The processor in which a given query arrives is called the *ranker* for that query since it is in this processor where the associated document ranking is performed. Every query is processed using two major steps: the first one consists on fetching a K-sized piece of every posting list involved in the query and sending them to the ranker processor. In the second step, the ranker performs the actual ranking of documents and, if necessary, it asks for additional K-sized pieces of the posting lists in order to produce the K best ranked documents that are passed to the broker as the query results. We call this *iterations*. Thus the ranking process can take one or more iterations to finish. In every iteration a new piece of K pairs (doc_id, frequency) from posting lists are sent to the ranker for every term involved in the query.

Under this scheme, at a given interval of time, the ranking of two or more queries can take place in parallel at different processors along with the fetching of K-sized pieces of posting lists associated with other queries. We assume a situation in which the query arrival rate in the broker is large enough to let the broker distribute Q P queries onto the P processors.

The search engine is implemented on top of the BSP model of parallel computing[6] as follows. In BSP the computation is organized as a sequence of *supersteps*. During a superstep, the processors may perform computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors. The underlying communication library ensures that all messages are available at their destinations before starting the next superstep.

Thus at the beginning of each superstep the processors get into their input message queues both new queries placed there by the broker and messages with pieces of posting lists related to the processing of queries which arrived at previous supersteps. The processing of a given query can take two or more supersteps to be completed. All messages are sent at the end of every superstep and thereby they are sent to their destinations packed into one message per destination to reduce communication overheads. In addition, in the input message queues are requests to index new documents and merge them into the distributed inverted file the resulting pairs (id_doc,frequency).

The total running time cost of a BSP program is the cumulative sum of the costs of

its supersteps, and the cost of each superstep is the sum of three quantities: w, $h\,G$ and L, where w is the maximum of the computations performed by each processor, h is the maximum of the messages sent/received by each processor with each word costing G units of running time, and L is the cost of barrier synchronizing the processors. The effect of the computer architecture is included by the parameters G and L, which are increasing functions of P.

The two dominant approaches to distributing the inverted file onto P processors are (**a**) the document partitioned strategy in which the documents are evenly distributed onto the processors and an inverted file is constructed in each processor using the respective subset of documents, and (**b**) the term partitioned strategy in which a single inverted file is constructed from the whole text collection to then distribute evenly the terms with their respective posting lists onto the processors.

In the document partitioned approach the broker performs a broadcast of every query to all processors. First and exactly as in the term partitioned approach, the broker sends one copy of the query to their respective ranker processors. Secondly, the ranker sends a copy of every query to all other processors. Next, all processors send K/P pairs (doc_id, frequency) of their posting lists to the ranker which performs the documents ranking. In the case of one or more query terms requiring another iteration, the ranker sends messages to all processors asking for additional K/P pairs of the respective posting lists.

In the term partitioned approach we distribute the terms and their posting lists in an uniformly at random manner onto the processors (we use the rule id_term mod P to determine in which processor is located a given term). In this scheme, for a given query, the broker sends the query to its ranker processor which upon reception sends messages to the other processors holding query terms, to get from them the first K pairs (doc_id, frequency) of every term present in the query. After the ranking of these pieces of posting lists, new iterations (if any) are processed in the same way. Similar to the document partitioned index, the ranker is selected in a circular manner and without paying attention in which processors are located the query terms. Notice that the effect of imbalance due to frequent terms in queries is minimized because the ranking computations (which is most costly component in the running time) are well balanced and disk accesses for frequent terms are reduced by LRU disk caching of the posting list segments being required by iterations.

3 Concurrency Control Strategies

Timestamp protocol. A first point to note is that the semantics of supersteps tell us that all messages are in their target processors at the start of each superstep. That is, no messages are in transit at that instant and all the processors are barrier synchronized. If the broker assigns a correlative timestamp to every query and document that it sends to the processors, then it suffices to process all messages in timestamp order in each processor to avoid R/W conflicts. To this end, every processor maintains its input message queue organized as a priority queue with keys given by id_query integer values (timestamps). The broker selects the processors to send documents and queries in a circular manner. Upon reception of a document, the respective processor parses it to extract all the relevant terms.

In the document partitioned inverted file the posting lists of all the parsed terms are updated locally in the same processor. However, this is not effected in the current superstep but in the next one in order to wait for the arrival of broadcast terms belonging to queries

placed in the same superstep, which can have timestamps smaller than the one associated with the current document insertion operation. To this end, the processor sends to itself a message in order to wait one superstep to proceed with the updating of the posting lists.

In the term partitioned inverted file the arrival of a new document to a processor is much more demanding in communication. This because once the document is parsed a pair (id_doc,frequency) for each term has to be sent to the respective processor holding the posting list. However, insertion of a new document is expected to be comparatively less frequent than queries in real-life settings.

A complication arises from the fact that the processing of a given query can take several iterations (supersteps). A given pair (id_doc,frequency) cannot be inserted in its posting list by a write operation with timestamp larger than the query being solved throughout several iterations. We solve this by keeping aside this pair for those queries and logically including the pair in the posting list for queries with timestamps larger than the one associated with the pair. In practice the pair is physically inserted in the posting list in frequency descending order but it is not considered by queries with smaller timestamps. Garbage collection can be made periodically every certain number of supersteps by performing a parallel prefix operation in order to determine the largest timestamp among the fully completed query timestamps at the given superstep.

Exclusive write supersteps. Another strategy is to let the broker control the order in which the queries and documents are sent to the cluster processors. Write operations associated to each new document are granted exclusive use of the cluster. That is, upon reception of a new document, the broker waits for the current queries in the cluster to be completed and queues up all new arriving queries and documents. Then it sends the new document to the cluster where it is indexed and the respective posting lists are updated. Once these write operations have finished the broker let the next set of queued read operations to be processed in the cluster until the next document and so on.

Two-phases locks protocol. Concurrent access to terms and their respective posting lists can be protected by using read and write locks[3]. We employ a distributed version of the locks protocol in which every processor is in charge of administering a part of the lock and unlock requests associated with the processing of queries and documents. To this end, we assume terms evenly distributed onto the processor using the rule id_term % P. To insert a new document it is necessary to request a write lock for every relevant term. This is effected by sending to the respective processors messages with the lock requests. Once that these locks are granted and modifications to the inverted file are finished the locks are released by sending unlock messages. Similar procedure is applied for read operations required by queries but in this case one or more read locks can be granted for each term. Write locks are exclusive.

Optimal protocol. For comparison purposes we use this hope-for-the-best protocol. This is a case in which we assume that queries and documents are submitted to the search engine in a way in which read/write conflicts never takes place. To this end, we process the queries and documents without including extra-code and messages to do any concurrency control. This represents the best a protocol can do in terms of total running time.

3.1 Tradeoff Between the Term and Document Partitioned Indexes

To motivate the comparative study presented in the next section we describe the BSP cost of the timestamp protocol. This to expose the tradeoff of using either of these distributed

inverted files.

Let us assume that at the beginning of a superstep i there is a certain average number Q of new queries per processor per superstep and t_q terms per query on the average. The average length of posting lists is ℓ. Also an average of D new documents arrive per processor per superstep containing on average t_d relevant terms. The steps followed by the term and document partitioned indexes are the following.

[*Term partitioned index*]

Superstep i Each processor obtains from its input message queue the arriving messages in timestamp order where messages can be queries or documents.

- For each document send all its terms to the processors holding the respective posting lists. Cost $t_d\, D\, (1 + G) + L$.
- For each query send its terms to the processors holding the respective posting lists. Cost $t_q\, Q\, (1 + G) + L$.

Superstep $i + 1$ Each processor obtains the new messages in timestamp order from its input queue.

- For each message containing the triplet (id_term, id_doc, freq) update the posting list associated with the term. Cost $t_d\, D$ updateListCost(ℓ).
- For each message containing a query term retrieve from disk the respective posting list to obtain the best K pairs (id_doc, freq) and send them to the ranker processor. Cost $t_q\, Q\, [\,\text{diskCost}(\ell) + K\, G\,] + L$.

Superstep $i + 2$ In each processor and for each query started in the superstep i get the messages containing the K pairs (id_doc, freq), effect the final ranking (we assume queries with one iteration) and report to the broker the top-K documents. Cost Q rankingCost(t_q, K) $+ L$.

On the other hand, in the document partitioned index the timestamp strategy works as follows.

[*Document partitioned index*]

Superstep i Each processor obtains from its input message queue the arriving messages in timestamp order where messages can be queries or documents.

- Store each document and wait to the next superstep.
- Send each query to all the processors (broadcast). Cost $t_q\, Q\, (1 + P\, G) + L$.

Superstep $i + 1$ Each processor obtains the new messages in timestamp order from its input queue.

- For each message containing a document extract all its relevant terms and update the respective posting lists. Cost $t_d\, D$ updateListCost(ℓ/P).
- For each message containing a query obtain the posting lists associated with its terms and get the best K/P documents and send them to the ranker processor. Cost $t_q\, Q\, P\, [\,\text{diskCost}(\ell/P) + (K/P)\, G\,] + L$.

Superstep $i+2$ Each processor and for each query started in the superstep i, obtain the P messages per query with K/P pairs (id_doc, freq) and perform the final ranking and report to the broker the top-K documents. Cost Q rankingCost(t_q, K) + L.

Basically the difference between the two strategies is given by the tradeoff in communication arising in the first superstep. That is, the tradeoff is given by the factors $t_d\, D\, G$ and $t_q\, Q\, P\, G$ in the cost of the term and document partitioned indexes respectively.

4 Evaluation

For the performance evaluation we used a text database which is a 12 GB sample of the Chilean Web taken from the `www.todocl.cl` search engine. Using this collection we generated an index structure with 1,408,447 terms. Queries were selected at random from a set of 127,000 queries taken from the todocl log. The query log contains 33,000 terms. The experiments were performed on a cluster with dual processors (2.8 GHz) that use NFS mounted directories. This system has 2 racks of 6 shelves each with 10 blades to achieve 120 processors. All the results were obtained using 4, 8, 16 and 32 processors as we found this to have a practical relation between the size of the inverted file and the number of processors. With 32 processors we observed significant imbalance in some cases. In addition, the number of queries available for experimentation makes 32 processors large enough. On average, the processing of every query finished with $0.6K$ results after 1.5 iterations. Before measuring running times and to avoid any interference with the file system, we load into main memory all the files associated with queries and the inverted file. To evaluate the effect of write operations on the inverted files we inserted documents taken at random from the same text sample. On average documents contain 200 relevant terms which are also in the query log. We observed a large number of read/write conflicts during execution of query and document operations.

In the experiments the first fact to become clear is the poor performance achieved by the locks protocol. We observed that its throughput was too small causing an increasingly large queue of pending operations. The assignment of a write lock to each relevant term of the document being inserted/updated restrains significantly the rate of user queries processed per unit time. Figure 1 shows the percentage of scheduled write/read operations that are completed in each superstep considering that in each superstep new queries and documents are injected using P= 4 processors. In this experiment the total number of document insertion operations is a 22% of the total number of operations (queries plus documents). The figure shows that on the average a 16% of the queries and a 8% of the document insertions are completed which lead to an average of 10% of unlock operations completed per superstep.

In Fig. 2 we show results for the timestamp (TS) and exclusive supersteps (ES) protocols for both the term and document partitioned indexes. We also show results for the optimal protocol. Also the results are for different query traffic Q and different number K of documents presented to the user as the answer for each query, and number of processors P= 4, 8, 16 and 32. The curves are labeled “R” for the case in which 80,000 queries are distributed uniformly at random onto the P processors and no documents are inserted in the inverted file, and “W” for the case in which the 80,000 queries are randomly mixed with other 80,000 operations of insertion of new documents.

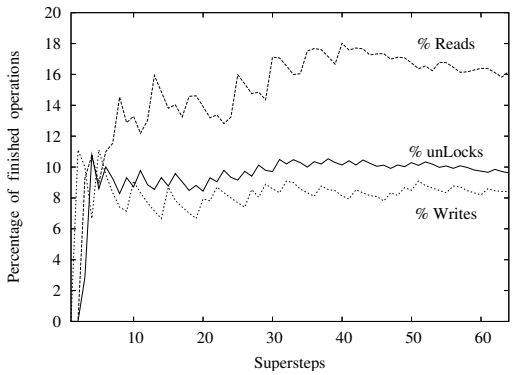


Figure 1. Locks protocol. The curves show the percentage of read, write and unlock operations finished.

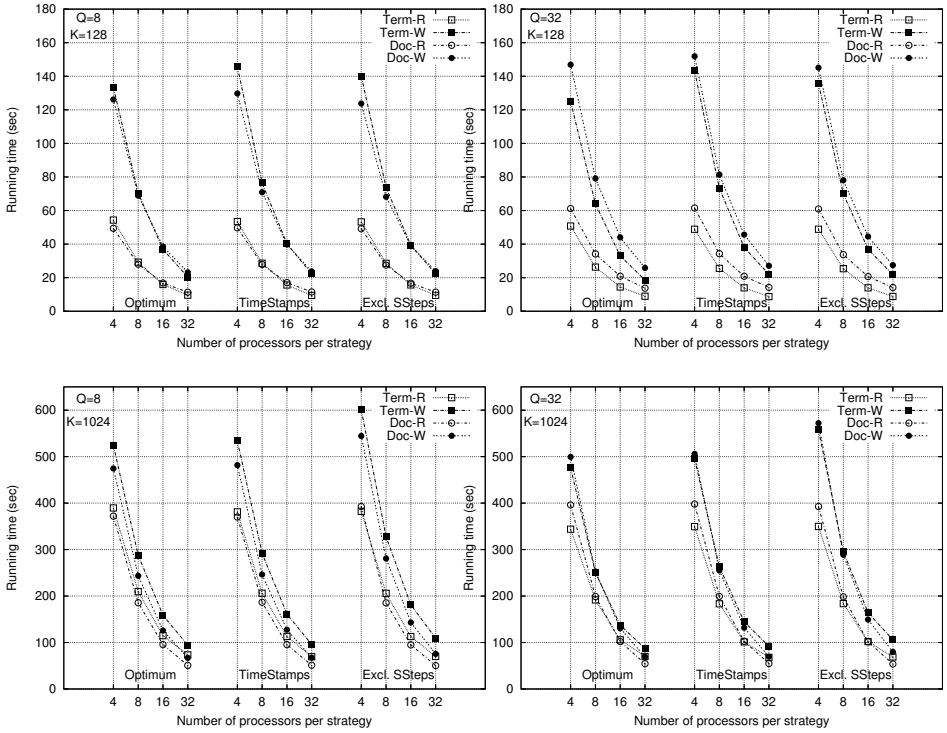


Figure 2. Running times for the timestamp and exclusive supersteps protocols.

The results in Fig. 2 show that the performance of both protocols is fairly similar to the performance of the optimal protocol. This indicates that overheads are small. The term

partitioned index performs slightly better than the document partitioned index. Even for the case with large number of write operations the term partitioned index performs efficiently. The ratio $t_d\,D\,G\,/\,t_q\,Q\,P\,G$ is not detrimental to the term partitioned index. This is because the cost of ranking queries can be more significant than the cost of indexing and updating the posting lists. This is seen in the Fig. 2 when one compares the difference between the results for K= 128 and K= 1024. Low traffic $Q = 8$ tends to produce higher running times as this causes imbalance across processors. For the case K= 1024 the ES protocol tends to be noticeably less efficient than the TS protocol. In this case imbalance is more significant for two reasons: each document is indexed in one processor during the write-only supersteps and posting lists are updated in parallel, and ranking of queries is split in several supersteps which reduces the average number of ranking operations per processor per superstep.

5 Conclusions

We have presented a comparison of different protocols to perform concurrency control in distributed inverted files. The particular manner in which we organize query processing and insertion of new documents allows a suitable bulk-synchronous organization of parallel computation. This provides a simple but very efficient timestamp protocol for controlling concurrency. The empirical results show a performance similar to the optimal protocol.

We have also found that the write exclusive supersteps protocol achieves competitive performance. In this case it is not necessary to barrier synchronize the processors as it only requires to detect when all current queries have been finished to send the document insertion operation and then detect when all associated write operations have finished to start a new set of queries and so on. This scheme can be implemented in message-passing asynchronous implementations of inverted files in which case operation termination must be handled using extra messages. Even in this case it is clear that this scheme is more efficient than the locks protocol since this protocol requires similar number of extra messages and our results show that it is not efficient because it achieves a very low throughput.

Acknowledgement: Partially funded by CWR Grant P04-067-F, Mideplan, Chile.

References

1. A. Arusu, J. Cho, H. Garcia-Molina, A. Paepcke and S. Raghavan, *Searching the web*, ACM Trans., **1**, 2–43, (2001).
2. A. Barroso, J. Dean and U. H. Olzle, *Web search for a planet: The google cluster architecture*. IEEE Micro, **23**, 22–28, (2002).
3. A. A. MacFarlane, S. E. Robertson and J. A. McCann, *On concurrency control for inverted files*, in: 18th BCS IRSG Annual Colloquium on Information Retrieval Research, pp. 67–79, (1996).
4. W. Moffat, J. Webber, Zobel and R. Baeza-Yates, *A pipelined architecture for distributed text query evaluation*, Information Retrieval, (2006).
5. S. Orlando, R. Perego and F. Silvestri, *Design of a parallel and distributed web search engine*, in: Proc. 2001 Parallel Computing Conf., pp. 197–204, (2001).
6. L. G. Valiant, *A bridging model for parallel computation*, Comm. ACM, **33**, 103–111, (1990).

Gb Ethernet Protocols for Clusters: An OpenMPI, TIPC, GAMMA Case Study

Stylianos Bounanos and Martin Fleury

University of Essex, Electronic Systems Engineering Department
Colchester, CO4 3SQ, United Kingdom
E-mail: {sbouna, fleum}@essex.ac.uk

Gigabit Ethernet is a standard feature of cluster machines. Provision of fast network interconnects is negated if communication software cannot match the available throughput and latency. Transparent Inter Process Communication (TIPC) has been proposed as an alternative to TCP in terms of reduced message latency and system time. This study compares through low-level tests and application benchmarks the relative performance of TIPC, Open MPI, and also what improvement the GAMMA User-Level Network interface can bring compared to both of these. TIPC's system time usage is reduced compared to TCP, leading to computational gains, especially on loaded cluster nodes. GAMMA is shown to bring significant improvement in computational performance on an unloaded cluster but a more flexible alternative is Open MPI running over TIPC.

1 Introduction

Gigabit (Gb) Ethernet is a standard feature of Linux clusters and, indeed, BlueGene/L machines with this interface have headed up the Top500 supercomputer list.[a] The question arises: what is the most appropriate protocol software in terms of throughput, latency, compatibility, flexibility and so on? The traditional TCP/IP protocol suite was developed in the early 1970s for Internet applications not clusters and for relatively fast CPUs rather than network links that keep pace with the CPU. On the other hand, the Transparent Inter Process Communication (TIPC) protocol[1] promises improved short-message latency compared to the normal TCP/IP protocol suite and at data-link layer two, User-Level Network (ULN)interfaces , such as GAMMA[2], are a more direct way to improve latency and throughput. As 'Ethernot' network interfaces, Myrinet, Infiniband, QsNet, exist with custom hardware off-loaded lower layer protocol stacks, portability remains a concern, but fortunately Open MPI[3] has made it easier to integrate non-standard protocols into its component infrastructure.

This paper's contribution is a port of TIPC to Open MPI v. 1.0.2 to thoroughly benchmark the benefits of a cluster-optimized protocol compared to TCP/IP. The tests are conducted with and without background load. In turn, the paper also examines: the value of hardware off-loading of some compute-intensive TCP features onto the Network Interface Card (NIC); and whether the GAMMA Linux kernel module is a way of improving standard MPI with MPICH implementation. The communication software is evaluated by low-level metrics of performance and by a standardized set of NAS application benchmarks[4]. The cluster under test, with Gb switching and high-performance AMD processors, scales up to thirty processors; it is of a moderate but accessible size.

[a] The list is to be found at `http://www.top500.org`, checked June 2007.

2 Computing Environment

The cluster employed consists of thirty-seven processing nodes connected with two Ethernet switches. Each node is a small form factor Shuttle box (model XPC SN41G2) with an AMD Athlon XP 2800+ Barton core (CPU frequency 2.1 GHz), with 512 KB level 2 cache and dual channel 1 GB DDR333 RAM. The cluster nodes each have an 82540EM Gb Ethernet Controller on an Intel PRO/1000 NIC. This Ethernet controller allows the performance of the system to be improved by interrupt mitigation/moderation[5]. It is also possible to hardware offload routine IP packet processing to the NIC, especially TCP header Cyclic Redundancy Check (CRC) calculation and TCP segmentation.

In TCP Segmentation Offload (TSO), the driver passes 64 kB packets to the NIC together with a descriptor of the Maximum Transmission Unit (MTU), 1500 B in the case of standard Ethernet. The NIC then breaks the 64 kB packet into MTU-sized payloads. A NIC communicates via Direct Memory Access (DMA) to the CPU, over a single-width 32-bit Peripheral Component Interface (PCI) running at 33 MHz in the case of the cluster nodes. Though both the single-width and double-width (64-bit at 66 MHz) standard PCI busses should cope with 1 Gb throughput, the GAMMA optimized ULN the short-width PCI bus throughput has been found[2] to saturate at about 40 MB/s for message sizes above 2E15 B in length.

The nodes are connected via two 24 port Gigabit (Gb) Ethernet switches manufactured by D-Link (model DGS-1024T). These switches are non-blocking and allow full-duplex Gb bandwidth between any pair of ports simultaneously. Typical Ethernet switch latency is identified[6] as between 3 μs and 7 μs, whereas generally more costly Myrinet switch latency is 1 μs. The switches are unmanaged and, therefore, unable to carry "jumbo" 9000 B Ethernet frames, which would lead to an increase in communication efficiency of about 1.5% and, more significantly, a considerable decrease in frame processing overhead[7].

3 Cluster Communication Software

Transparent Inter Process Communication (TIPC) is a feature of some Linux version 2.4 kernels and is now incorporated into Linux kernel v. 2.6.16. A number of studies have identified the importance to cluster applications of latency especially[8] for short messages. Reduced latency for small-sized messages and logical addressing are two of the attractions claimed by TIPC for clusters.

Logical addressing allows realtime calculation of routes based on the zone (group of clusters), cluster, or subnet within a cluster. TIPC fuses protocol layers, which reduces the number of memory-to-memory copies within the stack, a prime cause of increased latency in protocol stacks. Connectionless and single message connection handshake remove the impediment of TCP's three-way handshake when dynamic real-time messaging is required. TIPC employs a static sliding window for flow control rather than TCP's adaptive window, which avoids complex window size calculation algorithms. In case of link congestion, message bundling up to MTU is practised. TIPC also delegates checksum error detection to the data-link layer, which without hardware-assist is a serious burden.

To compare TIPC and TCP/IP, TIPC was ported by us to Open MPI[3] and its component-based architecture. Within open MPI, the TIPC stack is accessed through the socket API and then directly to an Ethernet frame or native data-link layer protocol. Open

MPI implements the full functionality of MPI-1.2 and MPI-2. It also merges the code bases of LAM/MPI, LA-MPI, and FT-MPI, capitalizing on experience gained in their development. While open MPI is probably aimed at terascale processing at national laboratories, its flexibility allows alternative protocols to be introduced. The component architecture makes it possible to provide drivers for diverse physical layers and diverse protocols. For example, Myrinet, Quadric's qsnet, Infiniband, and Gb Ethernet are alternative physical layers, while TCP/IP and now TIPC are alternative protocols. However, Open MPI currently employs TCP for out-of-band (OOB) spawning of processes through the Open Runtime Environment (ORTE). For ease of protocol implementation, this feature was retained by us (though implementations without TCP OOB are also possible[9]).

Open MPI is organized as a series of layers of which the Byte Transfer Layer (BTL) is concerned with protocols. Open MPI supports the normal MPI point-to-point semantics, namely standard, buffered, ready, and synchronous. However, Open MPI's low-level library calls employ different types of internal protocols for point-to-point communications between Open MPI modules. The type of internal protocol depends on message size. For some message sizes, Open MPI supports software Remote DMA (RDMA).

In software RDMA, an initial message portion contains a message descriptor. If the receiver has already registered an interest in the message, then the remainder of the message is requested and transferred directly to user memory. Otherwise, the transfer only takes place when the receiver registers an interest and supplies the target user address. Function callbacks at sender and receiver in this rendezvous operation are used in the manner of Active Messages[10]. Compared to MPICH, Open MPI therefore has an extra overhead from message matching but avoids the overhead from unexpected large messages. An identifying message tag in the current implementation of Open MPI[9] is of size 16 B.

Open MPI supports three internal protocols:

1. An eager protocol for small messages, up to 64 KB in the TCP BTL. If the receiver has not registered for a message, then there is an overhead in copying the message from the receive buffers to user memory.

2. A send (rendezvous) protocol with support for RDMA for messages up to the BTL's maximum send size; parameterized as 128 KB for TCP.

3. A pipeline protocol which schedules multiple RDMA operations up to a BTL-specific pipeline depth, for larger contiguous messages. In the pipeline, memory registration is overlapped with RDMA operations.

The Genoa Active Message MAchine (GAMMA) communication layer[2] employs NIC-dependent operations through a kernel-level agent. These operations include enabling/disabling interrupts, reading the NIC status, and polling the NIC. The code is highly optimized: with inline macros rather than C functions, and receiving messages in a single and immediate operation, without invoking the scheduler. GAMMA has a 'go-back N' internal flow control protocol. GAMMA has been ported to MPI, albeit with a customized version of MPICH protocols. GAMMA also utilizes `ch_p4` for remote spawning of MPI processes.

In Linux, the New API (NAPI) can be set to reduce the rate of interrupts from the Ethernet controller by substituting clocked interrupts or polling, in a scheme originating from[11]. A by-product is that arriving packets remain in a DMA send ring[6] awaiting transfer over

the PCI bus. In the tests in Section 4, NAPI `poll rx` was activated in the Linux e1000 driver, allowing a protocol processing thread to directly poll the DMA ring for packets. In fact, by judging link congestion, NAPI is able to maintain a flat packet service rate once the maximum loss-free receive rate has been reached, reactivating NIC-generated interrupts if need be. Performance tests show a 25-30% improvement in per packet latency on a Fast Ethernet card when polling is turned on under NAPI. The 82540EM provides absolute timers which delay notification for a set period of time. The 82540EM also provides packet send/receive timers which interrupt when a link has been idle for a suitably long time. Interrupt throttling is available, allowing interrupts to be suppressed if the interrupt rate goes above a maximum threshold.

4 Results

The first set of tests considered the latency of TIPC compared against several different ways of running TCP. For the majority of the tests in this Section, the version of the Linux kernel was 2.6.16.11, and the compiler was gcc/g++ 4.0.4. The TCP-BIC version of TCP[12] is the default setting of TCP in Linux kernel 2.6.16 and this was the TCP version employed. Comparative tests between TCP-BIC and TCP New Reno previously established that no appreciable difference occurred in the outcomes of the tests, and, hence, TCP-BIC was retained. The socket send and receive buffers were 103 KB in size, this being the default and immutable size for TIPC. The default value for the Gb Ethernet NIC transmit queue of 1000 (`txqueuelen`) was also retained.

In Fig. 1, the plot with the legend 'with hw offloading' refers to off-loading of the TCP/IP checksum calculations and TCP segmentation to the NIC, TSO (Section 2). The plot with legends '. . . and with NODELAY' refer to turning off Nagle's algorithm, which causes small messages to be aggregated by TCP. The comparison was made by normalizing the TCP latencies to that of TIPC. The lower latency of the two TCP varieties without hardware offloading, may be attributable to applying TSO to small message sizes, as the PCI overhead of the segmentation template is relatively large. Another factor in hardware offloading is that, because the CPU processor runs faster than that on the Intel NIC, as the processor is repeatedly called for CRC calculation for small messages, their latency will be reduced. Of course, there is a trade-off against available processing for the application. After 64 KB, those varieties of TCP with hardware offloading have lower latency compared to TIPC, because PCI overhead is relatively reduced. In the TIPC implementation under test, the maximum message payload size was set in the TIPC header and at various points in the source code of the kernel module to 66 kB. Therefore, for message sizes above this limit, it was necessary to make two system calls, whereas the TCP socket only involves one system call. It has been stated[1] that TIPC spends 35% less CPU time per message than TCP, up to sizes of 1 KB, when the host machine is under 100% load. This is an important practical advantage, as clearly if TIPC spends relatively less time processing that time is available for computation. We were able to confirm this in Fig. 2, which shows that TIPC's throughput relative to system time (captured with the Unix utility `getrusage`) is dramatically better.

Comparing TIPC and TCP running under Open MPI, Fig. 3 shows that for message-sizes below about 16 KB, TIPC has a clear advantage in latency. There is also a sharp increase in TIPC latency at around 128 KB. TCP was able to employ an appropriate Open

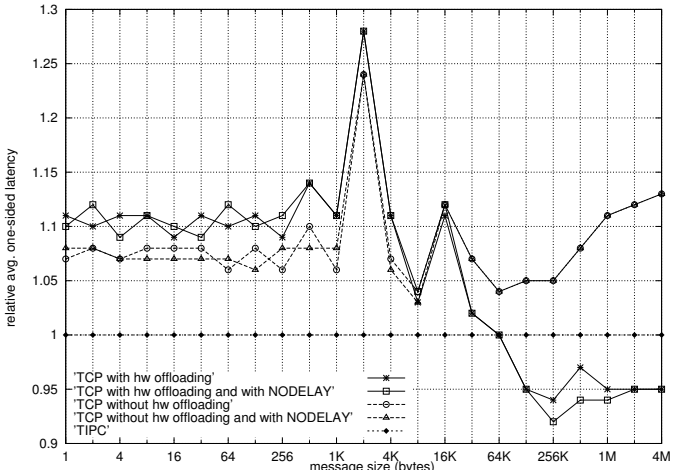


Figure 1. Comparative latency of TCP and TIPC across a range of message sizes, normalized to TIPC. (Note the log. scale on the horizontal axis.)

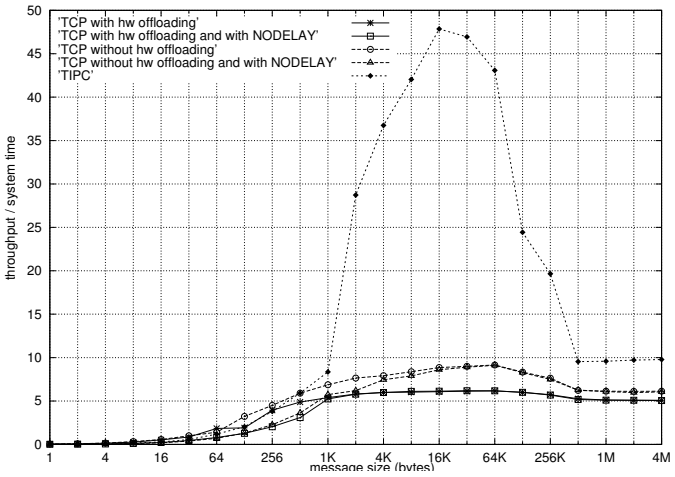


Figure 2. Comparative resource usage of TCP and TIPC across a range of message sizes. (Note the log. scale on the horizontal axis.)

MPI internal protocol, depending on message size. The same message parameterizations were retained in the TIPC BTL as in the TCP BTL. In the TIPC implementation under test, the maximum message payload size was again internally set to 66 kB. To avoid problems with open MPI in the pipelined version of RDMA, the third internal protocol for larger messages was disabled in the TIPC BTL, and the send protocol maximum was set to 66 kB. For message sizes of 128 KB and above, TIPC performs up to three rendezvous operations (for the first eager fragment, a maximum send-sized portion, and any remainder via the

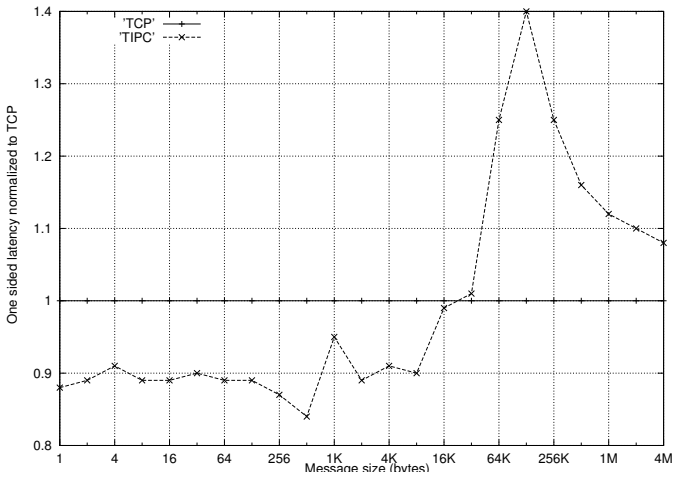


Figure 3. Comparative latency of OMPI/TCP variants and OMPI/TIPC across a range of message sizes, normalized to TCP. (Note the log. scale on the horizontal axis.)

send internal protocol), whereas the TCP BTL negotiates a single RDMA operation for the entire message. Within the latency range that TIPC has the advantage, there is a 'sweet-spot' at about 512 KB, possibly due to internal buffering arrangements.

Apart from low-level message passing tests, we also conducted application level tests. The NAS Parallel Benchmarks (version 3.2)[4] are a set of eight application kernels/programs, *viz.* CG, IS, LU, FT, MG, BT, and SP, and EP, with MPI source code. Open MPI was compiled with Fortran 77 bindings to match the NAS source code. We used NAS W class problems, as the dimension of these problems is appropriate to the cluster's processors. Accurate recording of CPU load is available through Andrew Morton's 2003 utility `cyclesoak`, which can also act as a source of load. Following from the findings on relative system time usage, two sets of tests were conducted, with and without load. In fact, the EP benchmark could also be adapted as a timewaster, as it continually creates pseudo-random numbers. Notice that as EP is CPU-bound, it only communicates at the end of its operations. In the tests, the best of five runs was taken rather than the median of five runs. This was because one of the application kernels, the IS integer sort, was found to be particularly sensitive to any system activity. For the others, the median does not differ noticeably from the best. LU and SP are computational fluid dynamics, while the others are diverse 'kernels'. For example, MG is a multigrid kernel for solving a 3D Poisson Partial Differential Equation set, being a hierarchical version with rapid convergence of the application.

Table 1 records the results. The Table also includes results for the MPICH version of MPI, with `ch_p4` being the standard MPICH method of spawning remote processes via the Unix utility rsh. The first two rows of Table 1 are taken from Table 2 of[13], also for 16 processors and W class problems. It will be seen that the message sizes are large for some of the tests, though for LU and MG they are about 1 KB and below the MTU. From the results, observe that MPICH almost always under-performs Open MPI. Noticeable gains in

	Application benchmark					
Test	CG	IS	LU	MG	SP	EP
Comms. freq. (MB/s)	36.24	22.10	5.58	20.21	19.50	0.00
Comms. freq. (# msg./s)	6103	2794	5672	13653	1657	16
Avg. msg. size (B)	5938	7909	983	1480	11768	
mpich/ch_p4	436.65	88.18	4134.11	2140.89	1698.00	140.86
OMPI/tcp	493.16	91.49	4688.89	2220.64	1792.43	141.84
OMPI/tipc	503.36	90.77	4921.43	2280.77	1812.18	140.88
tipc rel. tcp (%)	+2.07	-0.79	+4.96	+2.71	+1.10	-0.68
ch_p4 rel. tcp (%)	-11.46	-3.62	-11.83	-2.71	-3.59	-0.69
mpich/ch_p4 (loaded)	474.26	90.44	4137.57	2152	1686.01	141.55
OMPI/tcp (loaded)	494.70	91.56	4021.40	2237.23	1522.23	141.62
OMPI/tipc (loaded)	506.31	91.54	4356.74	2308.87	1574.66	141.71
ch_p4 rel. tcp (%)	-4.13	-1.22	+2.89	-3.77	+10.76	-0.05
tipc rel. tcp (%)	+2.35	-0.02	+8.34	+3.20	+3.44	+0.06

Table 1. NAS W class benchmark results (MOP/s) for sixteen processors, including relative (rel.) performance.

performance occur for the TIPC variant of Open MPI in the LU and MG application, both with shorter messages. Lastly, TIPC's relative performance compared to TCP increases when there is background load, suggesting an efficient stack.

We were also interested in the performance of GAMMA. A further set of tests were conducted in which later versions of the software were availed of. TIPC version 1.6.2, which we applied to kernel 2.6.18, removes the 66 kB send limit in the sense that the programmer no longer need make two system calls for messages over this limit, though the underlying MTU size of course remains. Table 2 shows very good relative performance of GAMMA but much weaker performance under load. At this point in time, the cause of GAMMA's weak performance under load has not been established by us despite investigations. We have not included MPICH results in Table 2 as its relative performance was weak, particularly when there was no background load. This decision was taken as we could not be sure that new software settings had resulted in disadvantaging native MPICH.

5 Conclusion

The tests comparing TCP and TIPC reveal that TIPC has very real advantages over TCP, both within and outside an OMPI environment. This is the paper's strongest conclusion, as it is shown for low-level benchmarks and for NAS application kernels. Short message latency was reduced. Moreover, the efficiency of the TIPC stack is demonstrated for nodes with high background load. The TIPC protocol has only recently been transferred to the Linux kernel and will gain as its software matures. A speculation is what would be the gain if TIPC benefited from hardware offloading to the NIC for larger messages, as TCP is able to do. The GAMMA user-level network interface (with MPI) is also capable of much improved performance over a combination of TCP and MPI. However, on a heavily-loaded machine, its computational performance may be matched by OMPI/TIPC.

	Application benchmark					
Test	CG	IS	LU	MG	SP	EP
mpich/GAMMA	876.07	140.37	7608.99	3222.95	2408.11	146.47
OMPI/tcp	521.73	94.62	4807.69	2289.76	1850.37	145.68
OMPI/tipc	524.16	94.46	5003.33	2310.14	1811.60	145.67
GAMMA rel. tcp (%)	+67.92	+48.75	+58.27	+40.75	+30.14	+0.54
tipc rel. tcp (%)	+0.47	-0.17	+4.07	+0.89	-2.10	-0.01
mpich/GAMMA (L)	56.69	18.46	1261.28	607.35	1299.29	134.67
OMPI/tcp (L)	478.62	92.90	3844.91	2276.31	1516.27	139.77
OMPI/tipc (L)	508.21	92.75	4022.34	2293.77	1503.29	140.06
GAMMA rel. tcp (L,%)	-88.16	-80.13	-67.20	-73.32	-59.87	-3.65
tipc rel. tcp (L,%)	+6.18	+0.16	+4.61	+0.77	-0.86	+0.21

Table 2. NAS W class benchmark results (MOP/s) for sixteen processors with GAMMA, including relative (rel.) performance; (L) indicates loaded benchmark.

References

1. J. P. Maloy, *TIPC: providing communication for Linux clusters*, in: Linux Symposium, vol. **2**, pp. 347–356, (2004).
2. G. Ciaccio, M. Ehlert and B. Schnor, *Exploiting gigabit ethernet for cluster applications*, in: 27th IEEE Conf. on Local Computer Networks, pp. 669–678, (2002).
3. E. Gabriel *et al.*, *Open MPI: goals, concept, and design of a next generation MPI implementation*, in: 11th Eur. PVM/MPI Users' Group Meeting, pp. 97–104, (2004).
4. D. Bailey *et al.*, *The NAS parallel benchmarks 2.0*, Tech. Rep. NAS-95-020, NASA Ames Research Center, Moffet Field, CA, (1995).
5. *Interrupt moderation using Intel gigabit ethernet controllers*, Tech. Rep., Intel Corporation, Application note AP-450, (2003).
6. A. Gallatin, J. Chase, and K. Yochum, *Trapeze/IP: TCP/IP at near gigabit speeds*, in: USENIX'99 Technical Conference, (1999).
7. R. Breyer and S. Riley, *Switched, Fast, and Gigabit Ethernet*, (Macmillan, San Francisco, 1999).
8. P. Buonadonna, A. Geweke, and D. Culler, *An implementation and analysis of the Virtual Interface Architecture*, in: Supercomputing Conference, pp. 1–15, (1998).
9. B. W. Barrett, J. M. Squyres, and A. Lumsdaine, *Implementation of Open MPI on Red Storm*, Tech. Rep. LA-UR-05-8307, Los Alamos National Lab., (2005).
10. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, *Active Messages: a mechanism for integrated communication and computation*, in: 19th Annual International Symposium on Computer Architure, pp. 256–266, (1992).
11. J. C. Mogul and K. K. Ramakrishnan, *Eliminating receive livelock in an interrupt-driven kernel*, ACM Transactions on Computer Systems, **15**, 217–252, (1997).
12. L. Xu, K. Harfoush, and I. Rhee, *Binary increase congestion control for fast, long distance networks*, in: IEEE INFOCOM, pp. 2514–2524, (2004).
13. K. Morimoto *et al.*, *Performance evaluation of MPI/MBCF with the NAS parallel benchmarks*, in: 6th Eur. PVM/MPI Users' Group Meeting, pp. 19–26, (1999).

Performance Measurements and Analysis of the BlueGene/L MPI Implementation

Michael Hofmann[a] **and Gudula Rünger**

Department of Computer Science
Chemnitz University of Technology
E-mail: {mhofma, ruenger}@informatik.tu-chemnitz.de

The massively parallel architecture of the BlueGene/L supercomputer poses a challenge to the efficiency of parallel applications in scientific computing. The specifics of the BlueGene/L communication networks have a strong effect on the performance of the MPI implementation. Various optimizations and specially adapted algorithms cause varying performance results for communication operations. In this paper, we present performance results of various MPI operations for a BlueGene/L system using up to 4,096 nodes. We discuss their efficiency in terms of the communication hardware and investigate influences on the performance of commonly used point-to-point and collective operations. The results give an overview of the efficient usage of the MPI operations and can be used to optimize the performance of parallel applications.

1 Introduction

During the last few years, one important progress in high performance computing was the development and deployment of the IBM BlueGene/L (BG/L) supercomputer. This was an important milestone of the ongoing effort to build a petaflop supercomputer that provides sufficient computing power for advanced scientific computations. Beside the superior theoretical peak performance of about 360 teraflops, the most interesting and also challenging properties arise from the unique architecture. The BG/L is a massively parallel distributed memory system with several special purpose networks. Message passing programming is supported by an MPI implementation especially adapted to the properties of the communication networks. Achieving high performance in parallel scientific applications requires an adaption to the target environment. This involves the communication infrastructure given through the MPI implementation. Knowledge about their efficient usage is essential for programmers to prepare their applications to scale well up to thousands of nodes. The properties of the BG/L system have a direct influence on the performance of MPI communication operations. Due to specific optimizations, the performance of MPI operations varies depending on their particular usage. In this paper, we present performance results of MPI operations for a BG/L system[b] using up to 4,096 nodes. We discuss their origins and derive implications for an efficient usage.

The rest of this paper is organized as follows. In the following, we introduce the BG/L system and list related work. Section 4 presents performance results of MPI point-to-point communication operations and Section 5 shows results of collective MPI operations. Sections 6 presents results for communication schemes like overlapping communication and nearest neighbour communication. We conclude in Section 7.

[a] Supported by Deutsche Forschungsgemeinschaft (DFG)
[a] Supported by Deutsche Forschungsgemeinschaft (DFG)
[b] Measurements are performed on the BlueGene/L system at the John von Neumann Institute for Computing, Jülich, Germany. http://www.fz-juelich.de/zam/ibm-bgl

2 BlueGene/L System

The BlueGene/L supercomputer consists of up to 65,536 (64Ki[c]) dual-processor compute nodes with 700 MHz PowerPC based processors and either 512 MiB or 1 GiB main memory. Two operation modes are available for utilizing the dual-processors nodes: In *co-processor mode* (CO) each node runs only one process and allows the second processor to be used as a communication or computation coprocessor. In *virtual node mode* (VN) two separate processes run on every node with evenly divided resources. (The following investigations use CO mode unless otherwise stated.) The system features several communication networks, three of those available for message passing communication. The *global interrupt* network can be used to perform a barrier synchronization on a full system in 1.5 μs. The *collective* network is a tree with fixed point arithmetic support in every node and can be used to perform broadcast and reduction operations. For reduction operations with floating-point data, a two-phase algorithm consisting of multiple fixed point reduction operations was introduced[1]. The collective network has a payload bandwidth of about 337 MB/s[c] and a latency of 2.5 μs on a 64Ki nodes system. For point-to-point communication, a three dimensional *torus* network connects every node to six neighbors through bi-directional links. Data is sent packet-wise with packet sizes up to 256 bytes and the maximum payload bandwidth per link and direction is about 154 MB/s (we refer to this as the single link bandwidth). The *deposit bit* feature of the torus network supports fast broadcasting of packets to a line of nodes. A 64Ki nodes system is connected by a 64x32x32 torus network and can be split into multiple independent partitions. The BG/L MPI implementation[2] is based on MPICH2 and especially adapted to make use of the different communication networks.

3 Related Work and Motivation

The BlueGene/L has been introduced in detail[3] and a lot of information about application development and tuning is available[4]. Details about the BG/L MPI implementation[2] include information about different communication layers as well as performance results of single MPI operations and parallel application benchmarks. The collective MPI operations have been extensively optimized[1] to make use of the specialties of the communication networks. Separate algorithms for short and long message communication are introduced and performance results for varying message sizes are shown. The implementation of a one-sided communication interface[5] enables the usage of global address space programming models and one-sided operations of MPI-2 and presents additional efforts for bandwidth and latency optimizations. The topology functions of MPI have also been optimized[6] to improve the mapping of virtual (application specific) topologies to the physical topology of the torus network. MPI performance analysis tools were ported to the BG/L system and have been tested using a number of scientific applications[7].

Most of the work about the BG/L MPI focuses on implementation details for certain MPI operations and optimized algorithms for these implementations. In contrast, this article concentrates on the performance of MPI operations when used within a parallel appli-

[c] Prefixes Ki, Mi, and Gi are used to denote the base-1024 versions of Kilo, Mega, and Giga. Prefixes K, M, and G represent the base-1000 versions.

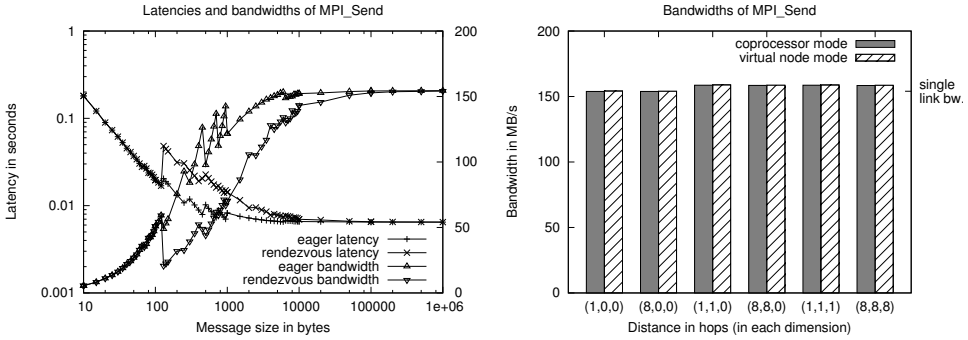


Figure 1. Latencies and bandwidths of point-to-point communication depending on the message size (left) and on the distance to the target node (right).

cation. The detailed measurements show how specific implementations of an MPI operation have an effect on the application program. Various influences on the performance of MPI operations are investigated and it is shown how certain communication schemes of an application program can be implemented best using the underlying MPI library.

4 MPI Point-to-Point Communication

All MPI point-to-point communication uses the torus network with either *deterministic* routing (packets use fixed paths) or *adaptive* routing (path depends on the current load). Depending on the size of the message to be sent, one of three protocols is chosen. The *eager* protocol is optimized for latency and used for medium size messages. The sender immediately starts sending the packets assuming that the receiver is able to handle them. The *short* (or *one packet*) protocol is used for messages that can be sent with one packet and causes a very low overhead. Both protocols use deterministic routing. The *rendezvous* protocol is a bandwidth optimized protocol for large messages. It uses a handshake mechanism between sender and receiver to establish a message context. Adaptive routing is used to maintain a balanced network load. The usage of the eager or the rendezvous protocol can be controlled with the `BGLMPI_EAGER` variable.

Figure 1 (left) shows the latencies and the bandwidths for sending 1 MB data with `MPI_Send` and varying message sizes to a neighboring node. The usage of the eager and the rendezvous protocol is forced using the `BGLMPI_EAGER` variable. Up to a message size of about 120 bytes the short protocol is used. In this case, with an increasing message size the total number of packets decreases resulting in constantly decreasing latencies. For message sizes of about 120-130 bytes the switch to the eager or the rendezvous protocol occurs. There is a clear difference between the two protocols with the eager protocol being about two times faster. For both protocols, there is a zigzag scheme visible for message sizes up to 1000 bytes. The peaks occur when an additional packet is required to send messages of the particular size. With increasing message sizes, the differences between the two protocols vanish and their latencies become fairly equal and constant for messages sizes above 10000 bytes. The short protocol achieves about 38 % of the single link bandwidth, while the eager and the rendezvous protocol reach about 99 %.

Figure 1 (right) shows the bandwidths for sending a message of size 1 MB to nodes at different distances in a 16x16x16 partition. It can be observed that the distance has no effect on the achievable bandwidth, because distances (1,0,0) and (8,0,0) show the same results. A slight increase occurs if sender and receiver nodes are not located on the same axis. However, the results indicate that only one link is used for a point-to-point operation at a time, since the bandwidth does not clearly exceed the single link bandwidth. Communication between processes on the same node in VN mode is done with memory copies and achieves a bandwidth of about 1670 MB/s. Apart from that, the operation mode has no effect on the results presented above.

The results confirm that the performance of the point-to-point operations strongly depends on the message size and the communication protocol used. The eager protocol shows generally the best performance and has significant performance peaks for certain message sizes. Comparable performance with the rendezvous protocol requires message sizes of at least 10000 bytes. However, with a higher network load, the rendezvous protocol can become more important. The distance between sender and receiver has no significant effect on the bandwidth of single communication operations and efforts for improving the locality are unnecessary in this case. The bandwidth of a single point-to-point operation is limited to the bandwidth of a single torus link and multiple link performance as demonstrated for one-sided communication[5] is currently not achievable.

5 MPI Collective Communication

For efficient implementations of MPI collective communication the BG/L system has two special purpose networks, the global interrupt network and the collective network. However, the usage of these networks is limited to operations on `MPI_COMM_WORLD`. Otherwise, the torus network is used with a number of optimized algorithms[1]. Especially for rectangular subsets of nodes optimizations based on the deposit bit feature are used.

MPI_Bcast / MPI_Allreduce

The following results of broadcast and reduction operations are obtained using a fixed message size of 1 MB. Figure 2 (top) shows latencies of `MPI_Bcast` and `MPI_Allreduce` (with `MPI_SUM`) for varying numbers of nodes using the CO mode. For `MPI_COMM_WORLD` the broadcast and the reduction operation with integer data have almost identical results independent from the number of nodes. The latency of about 2.98 ms for 1 MB data corresponds to a bandwidth of about 336 MB/s which is close to the payload bandwidth of the collective network. The latencies for floating-point data reductions are about 3-4 times higher, because of the missing arithmetic support in the collective network. With subsets of nodes, the torus network is used and optimized algorithms based on the deposit bit feature of the torus network are applied for rectangular subsets. With a 16x16x16 partition the broadcast operation achieves full single link bandwidth for broadcasting in an incomplete line (2-15 nodes), full double link bandwidth for broadcasting in a full line (16x1x1) or incomplete plane (16x2x1, ..., 16x15x1), and exceeds the maximum bandwidth of the collective network for broadcasting in a full plane (16x16x1) or cubic subset (16x16x2, ..., 16x16x16). The results of `MPI_Allreduce` show similar optimizations for rectangular subsets, but without achieving maximum bandwidths and

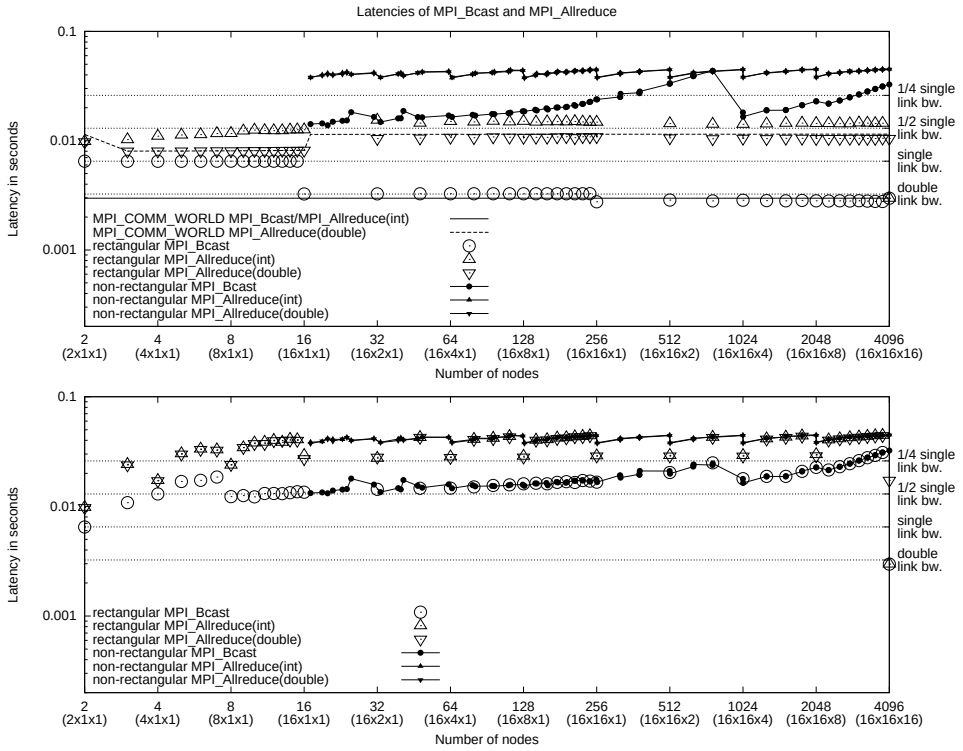


Figure 2. Latencies of MPI broadcast and reduction operations depending on the number of participating nodes and the shape (rectangular or non-rectangular) of the subset of nodes in CO (top) and VN (bottom) mode.

independent from the number of utilized links. With non-rectangular subsets of nodes the latencies are at least 3-4 times higher. Figure 2 (bottom) shows results using the VN mode. The improvements due to the optimizations for rectangular subsets of nodes have almost disappeared. Only for certain rectangular subsets of nodes there is a small decrease in latency for the reduction operation. For the broadcast operation there is no difference between rectangular and non-rectangular subsets.

Figure 3 (left) shows latencies of `MPI_Allreduce` depending on the reduction operation and the data type in three different situations (`MPI_COMM_WORLD`, rectangular and non-rectangular subsets). A user-defined reduction operation (summation) disables the optimizations and achieves almost equal results in all three situations. The results for the non-rectangular subset are independent from the pre-defined reduction operation and the data type. For `MPI_COMM_WORLD` the collective network can be used, but the maximum performance is only achieved with integer data and `MPI_SUM`. With `MPI_PROD` the latency increases by an order of magnitude. The results with floating-point data are the same for the pre-defined reduction operations and fairly independent from the number of nodes.

The results show that the performance of the collective operations strongly depend on the optimizations due to the BG/L communication hardware. The actual number of participating nodes has only a small influence, thus indicating good scalability of the collective

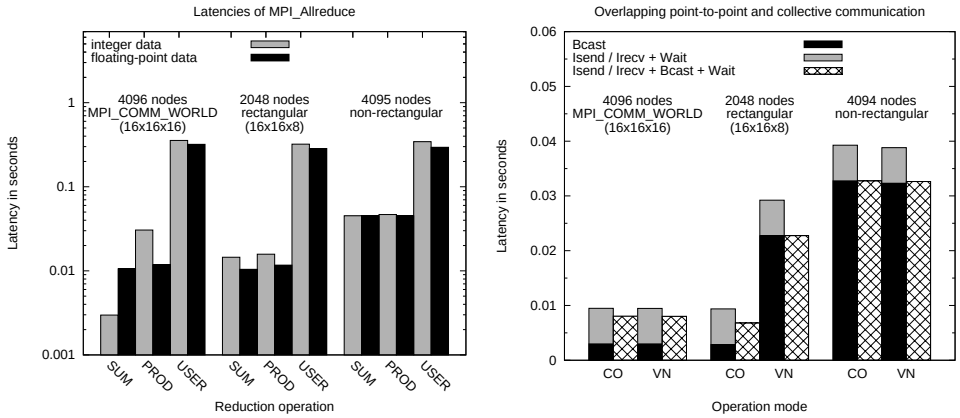


Figure 3. Latencies of `MPI_Allreduce` with different reduction operations and data types. (left) Latencies for overlapping point-to-point and collective MPI operations (`MPI_Bcast`). (right)

operations. Non-rectangular subsets of nodes prevent the usage of optimized algorithms, resulting in a significant performance loss. Therefore, it is almost better to increase the number of nodes participating in a collective operation to gain a rectangular shape. Additionally, it is recommend to spread subsets of nodes at least across two dimensions of the torus network to benefit from the usage of multiple torus links.

6 Communication Schemes

Overlapping Communication

The non-blocking communication operations of MPI can be used to overlap communication and computation. This is a common technique for hiding latencies of communication operations. However, with the current BG/L MPI version this kind of processing has no benefit. Overlapping point-to-point communication with a single matrix multiplication task produces exactly the same latency results as if they were performed one after another. This behaviour was the same using CO and VN mode. Figure 3 (right) shows measurements for overlapping point-to-point communication with a collective operation. Pairs of nodes exchange 1 MB messages with non-blocking operations while they are participating in a broadcast of a 1 MB message. Results are shown for three situations (`MPI_COMM_WORLD`, rectangular and non-rectangular subsets) with different broadcast implementations in use. Overlapping the two communication operations leads to a decrease of the latencies of about 15 % to 25 %. Significant differences between CO and VN mode occur only with the rectangular subset of nodes. In this case, the VN mode leads to an increased latency of the broadcast operation (as already show in Section 5). However, the improvements due to the overlapping are the same using CO and VN mode.

Nearest Neighbour Communication

Figure 4 (left) shows the bandwidths of concurrent communication of one node with its six direct neighboring nodes. Results are shown for receiving, sending, and exchang-

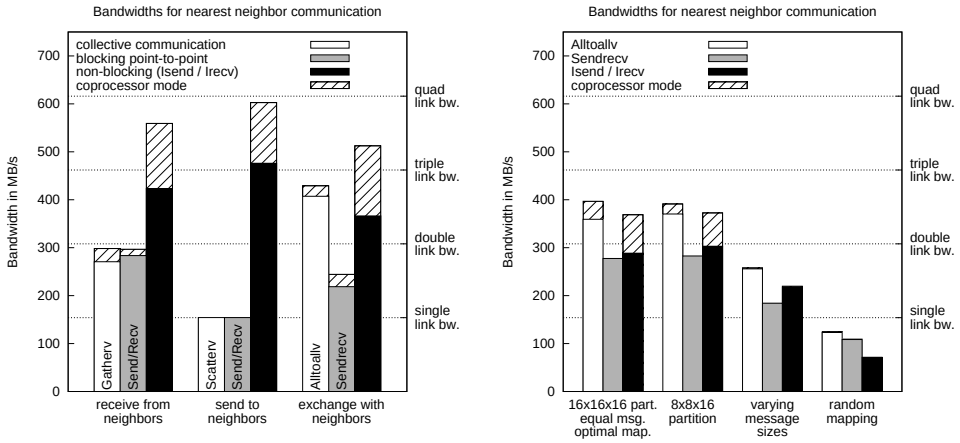


Figure 4. Bandwidths of one node communicating with its six neighboring nodes in the torus network using different MPI operations. (left) Per node bandwidths if all nodes communicate with their neighbors in a 16x16x16 torus network using different MPI operations. (right)

ing (concurrent send and receive) messages of size 1 MB with each neighbour using different communication operations (collective, blocking, and non-blocking point-to-point). When only sending or receiving, the differences between collective (`MPI_Gatherv` or `MPI_Scatterv`) and blocking point-to-point communication are rather small. Sending data to the neighbors achieves single link bandwidth, while data from the neighbors is received with about double link bandwidth. Higher rates are achieved when performing all communication at once with non-blocking operations. For exchanging data, the bandwidth of the collective operation (`MPI_Alltoallv`) is much better than with consecutive blocking point-to-point operations. The usage of `MPI_Sendrecv` achieves only about 80 % of the maximum bi-directional performance. With the CO mode, the usage of non-blocking operations achieves the best results, while with the VN mode the collective operation performs best. In general, the bandwidths of the non-blocking operations with the CO mode are about 30 % higher in comparison to the VN mode. The benefits for the collective and the blocking operations are rather small.

Figure 4 (right) shows the bandwidths per node if all nodes communicate with their neighbors. With a 16x16x16 partition, equal message sizes (1 MB) and an optimal mapping (communication with direct neighbors in the torus network only) the usage of `MPI_Alltoallv` achieves the best result of about 400 MB/s. The bandwidth with non-blocking point-to-point operations is only slightly lower and `MPI_Sendrecv` achieves again only about 80 % of the maximum bi-directional bandwidth. With a smaller partiton (8x8x16) the bandwidths for all communication operations remain fairly unchanged. Using messages with varying sizes (0.5-1.5 MB) leads to a decrease of the bandwidths of about 30-40 %. A random mapping of the nodes in the torus network decreases the locality of the communication significantly. Therefore, the bandwidths for all communication operations fall below the single link bandwidth and the non-blocking operations become the worst. Using the CO mode improves the performance of collective and non-blocking operations, but only under optimal conditions (equal message sizes, optimal mapping).

Even though the non-blocking operations of the BG/L MPI cannot be used to overlap communication and computation, they still can be used to perform multiple communication operations at once. Performance benefits can be achieved from the concurrent usage of the collective and the torus network as well as from the utilization of multiple torus links. The usage of all-to-all operations for data exchange with neighbors in a grid achieves similar good performance like a direct implementation with point-to-point operations. With non-optimal conditions like varying message sizes or decreased locality of the communication, the usage of all-to-all communication performs best. Using the CO mode increases the performance, especially for performing multiple non-blocking operations at once.

7 Conclusion

The measurements have shown that the specifics of the BG/L communication hardware have significant effects on the performance of single MPI operations. Optimized algorithms make of use the different communication networks and improve the performance of collective MPI operations. However, their usage is subject to a number of restrictions. Maximum performance in terms of hardware and software limitations is achieved for several communication operations and enables the prediction of latency values. Non-blocking operations have shown to be useful to communicate with various nodes at once and to utilize multiple links of the torus network. The all-to-all communication operation provided with the BG/L MPI turns out to be a good choice for exchanging data, especially under non-optimal conditions. In general, the experiments have shown that good scalability of communication operations can be achieved even for up to thousands of nodes.

References

1. G. Almási et al., *Optimization of MPI collective communication on BlueGene/L systems*, in: ICS '05: Proc. 19th annual international conference on Supercomputing, pp. 253–262, (2005).
2. G. Almási et al., *Design and implementation of message-passing services for the BlueGene/L supercomputer* , IBM J. of Research and Development, **49**, no. 2/3, (IBM, 2005).
3. G. L.-T. Chiu, M. Gupta, and A. K. Royyuru, (Eds.), *IBM J. of Research and Development: Blue Gene* , **49**, no. 2/3, (IBM, 2005).
4. O. Lascu et al., *Unfolding the IBM eServer Blue Gene solution* , IBM Redbooks, (2005).
5. M. Blocksome et al., *Design and implementation of a one-sided communication interface for the IBM eServer Blue Gene supercomputer*, in: SC '06: Proc. 2006 ACM/IEEE Conference on Supercomputing, p. 120, (2006).
6. H. Yu, I-H. Chung and J. Moreira, *Topology mapping for Blue Gene/L supercomputer*, in: SC '06: Proc. 2006 ACM/IEEE conference on Supercomputing, p. 116, (2006).
7. I.-H. Chung, R. E. Walkup, H.-F. Wen and H. Yu, *MPI performance analysis tools on Blue Gene/L*, in: SC '06: Proc. 2006 ACM/IEEE conference on Supercomputing, p. 123, (2006).

Potential Performance Improvement of Collective Operations in UPC

Rafik A. Salama and Ahmed Sameh

Department of Computer Science
American University in Cairo
E-mail: {*raamir, sameh*}*@aucegypt.edu*

Advances in high-performance architectures and networking have made it possible to build complex systems with several parallel and distributed interacting components. Unfortunately, the software needed to support such complex interactions has lagged behind. The parallel language's API should provide both algorithmic and run-time system support to optimize the performance of its operations. Some developers, however, choose to play clever and start from the language's primitive operations and write their own versions of the parallel operations. In this paper we have used a number of benchmarks to test performance improvement over current Unified Parallel C (UPC) collective implementations and prove that in some circumstances, it is wiser for developers to optimize starting from UPC's primitive operations. We also pin point specific optimizations at both the algorithmic and the runtime support levels that developers could use to uncover missed optimization opportunities.

1 Introduction

Collective communication operations in many parallel programming languages are executed by more than one thread/process in the same sequence taking the same input stream(s) to achieve common collective work[1]. The collective operations can either be composed by developers using the primitive operation's API that the language provides, or by parallel programming language writers who provide API for effective implementations of these collective operations. The extra effort of the language writers is meant to provide ease of use for developer to just call the collective operation rather than rewriting them using several primitive operations, and to supply highly optimized collective operations at two separate levels of optimization:

- *System runtime optimization*: The runtime library provides optimization opportunities at both hardware and system software levels. These optimizations may result in, for example, native use of the underlying network hardware and effective calls to Operating systems' services.
- *Algorithmic Optimization*: The algorithm is the core for optimizing collective operations. The collective operations can be highly optimized with the best proven algorithms.

UPC, or Unified Parallel C, is a parallel extension of ANSI C which follows the Partitioned Global Address Space (PGAS) distributed shared memory programming model that aims at leveraging the ease of programming of the shared memory paradigm, while enabling the exploitation of data locality. UPC is implemented by many universities (Berkley, Michigan, George Washington, Florida), companies (HP, IBM, Cray) and open source community (GNU GCC Compiler, ANL)[2,3]. According to a latest research comparing the

performance of various UPC implementations, it has been established that the Berkley implementation is currently the best implementation of UPC.

Assuming that the Berkley UPC collective operations are highly optimized (at both runtime support and algorithmic levels), we used them as a reference for comparison with the less optimized collective operations provided by Michigan University[3]. Then starting from Michigan implementation of UPC primitive operations that provides two techniques as options to handle input streams, Push (Slave Threads pushing data to the master thread or the master thread pushes data to the slave threads) and the Pull (Master thread pulling data from the slave threads, or slave threads pulls data from the master thread), we build collective UPC operations by applying both algorithmic and runtime support. Most of these optimizations are borrowed from similar MPI collective operations. We have investigated in many implementations of the collective operations applied to the distributed and shared memory and then selected as a start the LAM implementation of MPI[5]. We have implemented two versions of each Michigan UPC collection operation, one based on Michigan Push technique and another based on Michigan Pull technique and compared them to the current Berkley UPC collective optimizations.

Finally, new non-LAM algorithmic optimizations were tried for the intensive collective operation AllReduce. We have also identified some bit falls in the UPC runtime support that couldn't implement specific performance optimization techniques for AllExchange.

The rest of the paper is organized as follows; the next section describes the test bed of the performance comparison benchmarks, the Third section presents the experimental results of the benchmarks showing the potential performance enhancement over Berkeley UPC collectives, the Fourth section describes algorithmic non-LAM optimization of the UPC collectives, the final section is a conclusion of our work.

2 Comparison Test Bed

2.1 Cluster Configuration

The performance comparison of benchmarks was done on a cluster developed by Quant-X which is a 63 GFLOPs (TPP: Theoretical Peak Performance) supercomputing facility with 14 nodes dual Intel Pentium IV Xeon 2.2 GHz, with 512 MB memory, Intel 860 chipset, 36 GB SCA hard disk (for a total of 15*36 GB), CD-ROM, Floppy, Ikle graphics cards, and M3F Myrinet 2000 Fiber/PCI 200 MHz interface cards[6].

2.2 Software Configuration

The Berkley UPC with the GASNET is installed over the 14 nodes of the cluster described above and the LAM MPI is also installed over the 14 nodes. The Berkley UPC GASNET is configured to use both the SMP (2 processors in each node) and LAM MPI for communication between the nodes. Also the Michigan UPC that confirms to UPC V1.1 is installed on the cluster[7].

2.3 Benchmarks

2.3.1 NPB framework Tailored for Collective

The NAS parallel benchmarks (NPB) are developed by the Numerical Aerodynamic simulation (NAS) program at NASA Ames Research Center for the performance evaluation of parallel supercomputers[8]. The NPB performance measurement framework were used and tailored the "NPB IS" benchmark (since it involves integer operations) to focus only on the collective operation and measure their timing. This collective focus implementation simply took the major workload classes (i.e S, A, B), the general function of data preparations, the data validation functions and the time measurement methods then started putting instead of the normal IS computation another function that only executes a collective operation with the various workloads and processor numbers given. For example, to measure the UPC AllReduce; the function simply works on the array already prepared by the NPB2.4 framework with a simple collective reduction operation.

2.3.2 Collective Optimization measurement

The collective optimization measurement is a comparison between the execution time taken by the **native** Berkeley UPC collective operation provided by the language that contains the runtime performance optimization and the **primitive** reference implementation of the Michigan UPC provided in both the Push and the Pull versions with added LAM-MPI optimizations.

3 Experimental Result

The experimental results have shown surprises for both the Push and Pull techniques. Although we will be exploring all results, but in general the choice of the PUSH or PULL according to the collective operation shows a notable performance potential. This is the case since UPC has the ability to recognize local-shared memory accesses, and perform them with the same low overhead associated with private accesses. The local-shared memory accesses can be divided into two categories: thread local-shared accesses, and SMP local-shared accesses, when a thread accesses data that is not local to the thread, but local to the SMP. The latter requires that implementation details are exploited using run-time systems, while the former can be exploited by compilers. Another PUSH/PULL optimization is the aggregation of remote accesses to amortize the overhead and associated latencies. This is done using UPC block transfer functions. Due to UPC thread-memory affinity characteristic, UPC compilers can recognize the need for remote data access at compile time, thus provide a good opportunity for pre-fetching. In all the results below, the y-axis of the graphs is calculated by the following equation:

$$OptimizationPercentage = (P/O - 1)\% \tag{3.1}$$

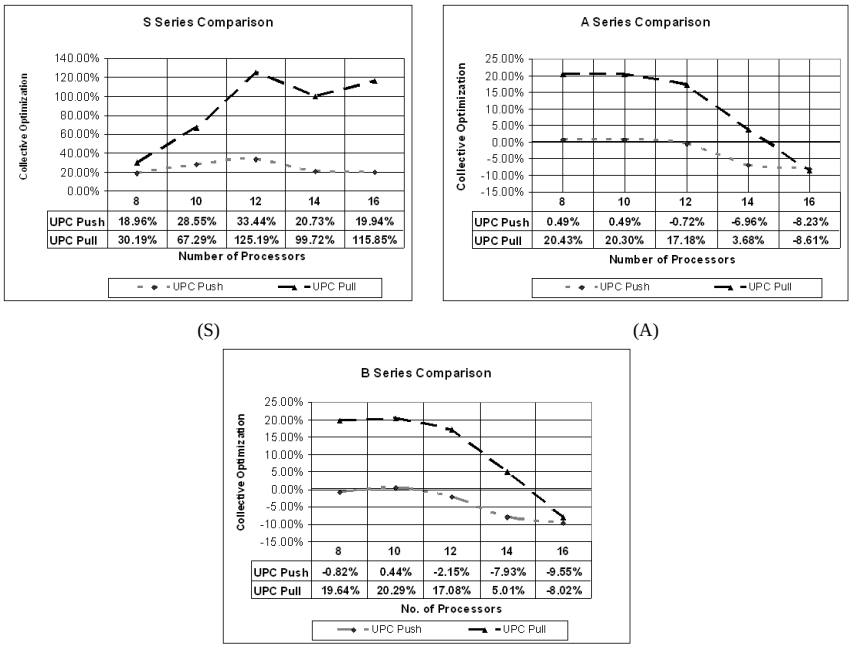
Where:

P	The primitive Michigan collective operations running Time
O	The native Berkely optimized collective operations running time.

Here the assumption is that collective operation should perform better than primitive operations which means higher than 1, so:

- The higher the value of the percentage above shows that the native Berkeley collective running time is lower than the Michigan primitive
- Zero means that native Berkeley collective is performing equal to the Michigan primitive operations
- Negative values indicate that the native Berkeley collective operations running time is higher than the Michigan primitive operations

Each point in the graphs below is an average of 600 actual result points for both the Michigan primitive collective as 300 point and the native Berkeley collective operations as 300 point.



(S) (A)

(B)

Figure 1. AllGather Collective Optimization Comparison (Push & Pull vs. Native)

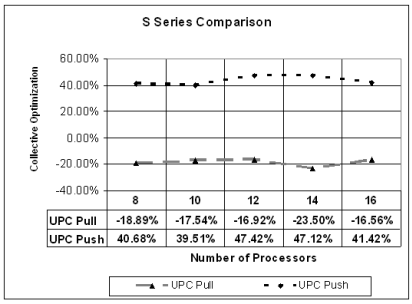
3.1 All Gather

ALL Gather was tested using the test bed described above showing as a general trend that the native Berkeley collective operations is better than the Michigan primitive collective operations with smaller data (S) as in Fig.1, but the performance kept getting worth with

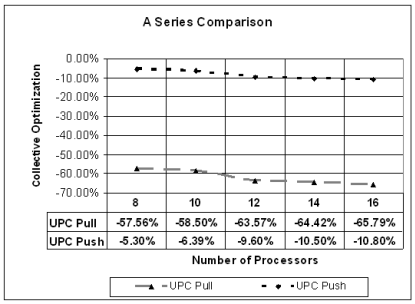
larger data sizes (A,B) as in Fig.1. Also the comparison of the Push and Pull technique favoured the Push technique as expected, since the effort of sending the data to the parent process is distributed among all the slaves, rather than the Pull where the parent thread gets to do everything.

3.2 All Scatter

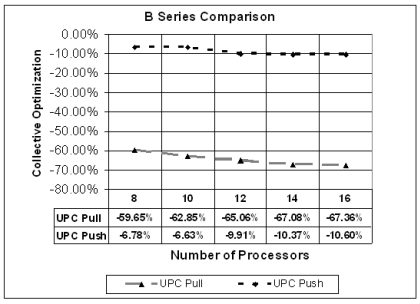
ALL Scatter testing results have shown an improvement by an average of 16 % for the pull technique in the small sizes S as shown in Fig.2. Over and above, the improvement has even become better with the larger sizes using the Pull technique as shown in Fig.2. This concludes that the Michigan primitive implementation using the Pull technique is better than the current Berkeley collective implementation. The Push technique on the other hand didn't show any enhancement over the current collective implementation and over the Pull technique which is more logical. A simple explanation is that the Pull technique divides the effort needed for data distribution among all the threads rather the Push technique which would have mandated for the parent thread to copy the data for the slave threads, rendering the parent thread a bottle neck in the collective operation.



(S)



(A)

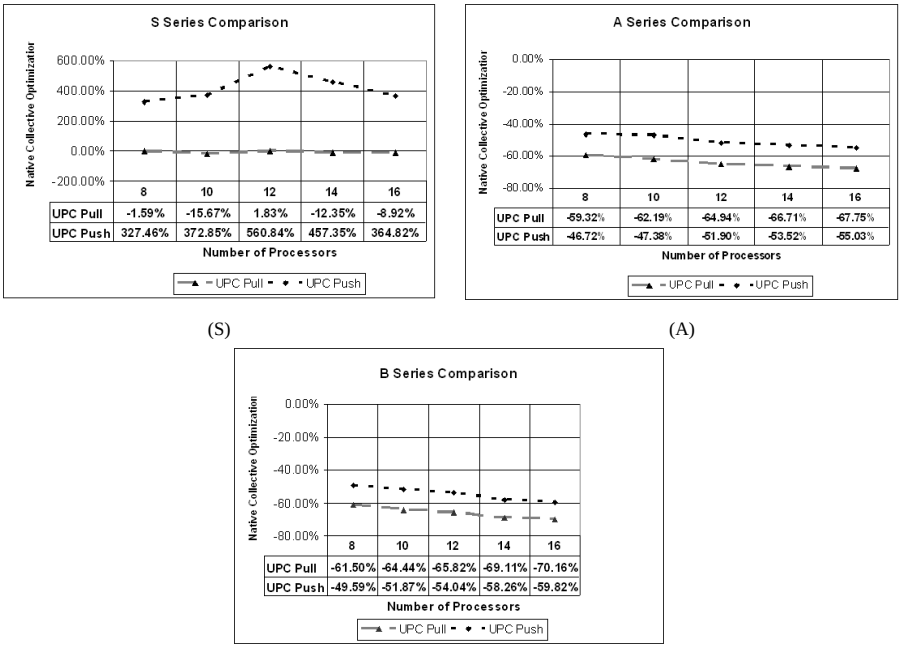


(B)

Figure 2. AllScatter Collective Optimization Comparison (Push & Pull vs. Native)

3.3 All Broadcast

ALL Broadcast Michigan primitive native have generally shown better performance than the native Berkeley collective. The Push and Pull technique have shown that the Pull technique is much better than the Push technique in smaller sizes as in Fig.3, while the Push technique is almost the same as the Pull technique at higher sizes as in Fig.3.



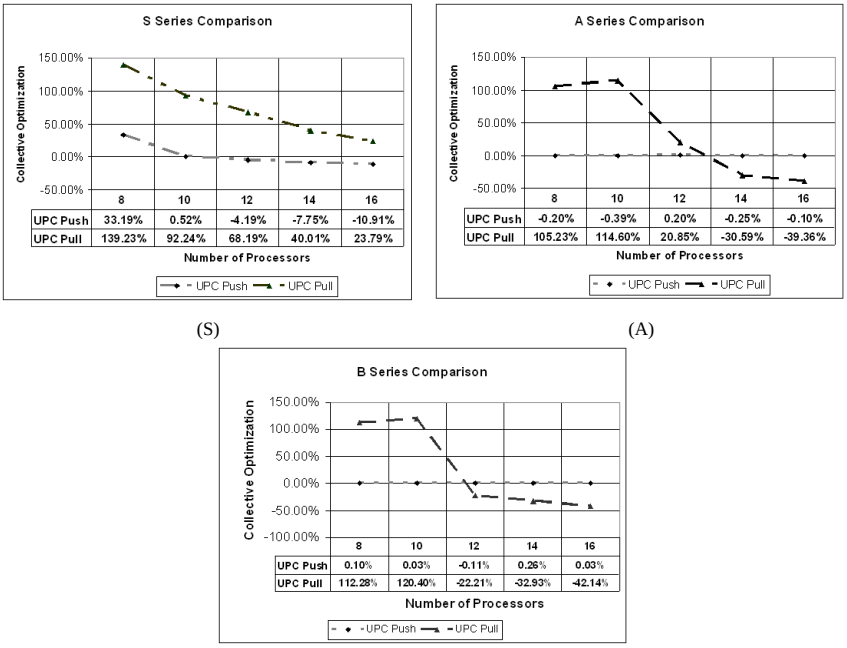
(S) (A)

(B)

Figure 3. AllBroadcast Collective Optimization Comparison (Push & Pull vs. Native)

3.4 All Exchange

ALL Exchange have shown different behaviour at different sizes and different processors numbers. Initially the Push technique has shown better performance than the Pull technique at smaller data sizes (S) as shown Fig.3 and larger processor numbers (12 - 16). The Push technique on the other hand has shown better performance at larger data sizes (A, B) and larger processor numbers (12 - 16) as shown in Fig.4. This would conclude that generally the native Berkeley collective is performing better at small processor numbers, and there is a notable enhancement performance for the larger processors in the alternative use of the Push - Pull techniques according to data size.



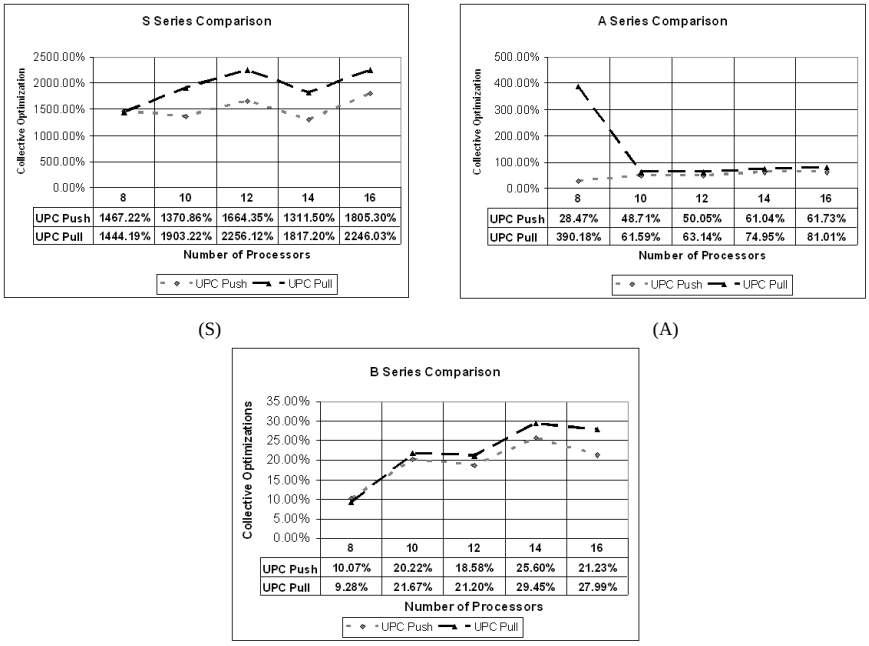
(S) (A)

(B)

Figure 4. AllExchange Collective Optimization Comparison (Push & Pull vs. Native)

3.5 All Reduce

ALL Reduce Berkeley native collective operations have generally shown better performance than the Michigan primitive collective operations. The primitive collective operations have the worst performance in the small sizes (S), see Fig.5, while it gets better in the larger sizes (A, B) as shown in Fig.5.



(S) (A)

(B)

Figure 5. AllReduce Collective Optimization Comparison (Push & Pull vs. Native)

4 UPC Collective Operations Further Optimization

The native Berkeley UPC collective operations testing against the reference Michigan implementation have shown low performance in some operations while it has shown better performance in others. In our research we have explored the various optimizations done in the area of the collective operations enhancements and borrowed some of them to prove the room for enhancements. In this area, there have been many newly proposed algorithms as in[9,10,11,12] which offers a new set of algorithm for the MPI collectives as for example, the pipelining style which breaks the large messages into segments. These algorithms can be borrowed to the UPC collective implementations to enhance its current state. In the area of collective operation enhancements for the shared memory which is our focus, there have been many proposals as for the MPI Collective Algorithms over SMP[13] where the authors present an enhanced algorithms for the collective operations to provide a concurrent memory access feature, which would server a better performance as they have proved the efficiency of these algorithms to enhance the running time of the collective operations. The same algorithms could be borrowed for the UPC collective operations to enhance reflecting this in the compilation of the program or even reflecting it in the GASNET libraries in the SHM scheme. Also One of the modern techniques proposed in this area is the self tuned adaptive routines for the collective operations which delays the decision of the collective operation algorithm until the platform are known, this technique is called ”delayed finalization of MPI collective communication

routines (DF)"[14], it actually does allow for the MPI collective operations to detect the SMP architecture and automatically apply the needed algorithm for this architecture using the Automatic Empirical Optimization of Software (AEOS) technique[15].
In our comparison we borrowed as a start the optimizations done in LAM-MPI reference implementation since the MPI have shown better performance than the UPC in the collective operations[1]. Although MPI is a library while the UPC is a language (with its own compiler) but we have borrowed the optimizations of the MPI library to apply it for the UPC collective operations library.

In this section further optimization is done by applying further LAM MPI optimizations:

1. *Allgather & AllExchange*:

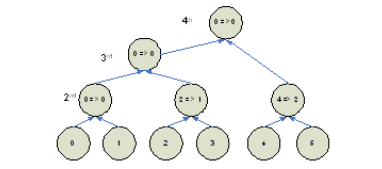
   Allgather & AllExchange are further investigated to borrow the same MPI Allgather & ALLtoALL optimization techniques respectively using the pairwise-exchange which have shown better performance[5], but we couldn't apply this optimization as this technique required the use of **(**asynchronous communication) which couldn't be simulated in the UPC due to lack in runtime support. As a matter of fact all the copying techniques in UPC are synchronous so it would have been so much helpful if the UPC supports asynchronous communication, which would offer apart from the collective operations a wealth of algorithms that uses the asynchronous communication.

2. *AllReduce*:

   ALLReduce on the other hand was highly optimized using a binary tree algorithm resulting in enhanced performance than the current collective operations.

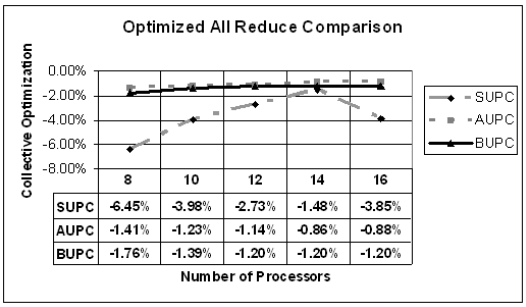
   (a) *Algorithm*: the binary tree algorithm used is almost similar to the parallel binary tree algorithm[3], except for one fact, that the tree ranks is reconstructed again from the available nodes as shown in Fig.6

   (b) *Experimental Results*: the experimental results have shown enhancements over the current collective especially in the smaller sizes of data -S- as shown in Fig.7, and almost similar results with the higher sizes of data



While (Zero thread is not the last thread)
Compute current rank of the threads according to the step of the algorithm, if step zero, then nothing to be done.
Odd threads send their array to even threads.
Even threads receive odd array and compute the reduction with their own arrays.
Increment step by one
Continue

Figure 6. AllReduce Binary Tree Algorithm



	8	10	12	14	16
SUPC	-6.45%	-3.98%	-2.73%	-1.48%	-3.85%
AUPC	-1.41%	-1.23%	-1.14%	-0.86%	-0.88%
BUPC	-1.76%	-1.39%	-1.20%	-1.20%	-1.20%

Figure 7. Optimized All Reduce Results

5 Conclusion

Group communications are commonly used in parallel and distributed environments. However, MPI collective communication operations have gone through several optimization enhancements. In this project, we borrow some of these optimization techniques into the UPC world. Experimental results show the borrowed techniques are solid and effective in improving UPC performance. The allreduce primitive collective was further optimized using a binary tree algorithm which showed better performance. So generally, there is a potential performance improvement and the current UPC Michigan implementation have shown an important need for the asynchronous memory communication where major MPI algorithms could be efficiently borrowed.

References

1. George Washington University and IDA Center for Computing Sciences, *UPC Consortium, UPC Collective Operations Specifications V1.0*, (2003).
2. Z. Zhang and S. Seidel, *Benchmark Measurements of Current UPC Platforms*, in: 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS'05, (2005).
3. Michigan University UPC, *UPC Collective Reference Implementation V 1.0*, (2004), http://www.upc.mtu.edu/collectives/col1.html
4. George Washington University and IDA Center for Computing Sciences, *UPC Consortium, UPC Language Specifications V 1.2*, (2005).
5. J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel and J. J. Dongarra, *Performance analysis of MPI collective operations*, Cluster Computing, **10**, 127–143, (2007).
6. http://www.cs.aucegypt.edu/~cluster
7. www.upc.mtu.edu/
8. www.nas.nasa.gov/Software/NPB/
9. A. Faraj and X. Yuan, *Bandwidth efficient All-to-All broadcast on switched clusters*, in: 19th IEEE International Parallel and Distributed Processing Symposium, Boston, MA, (2005).
10. T. Kielmann, et. al., *MPI's collective communication operations for clustered wide area systems*, ACM SIGPLAN PPoPP, **1**, 131–140, (1999).
11. R. Rabenseifner, *A new optimized MPI reduce and allreduce algorithms*, (1997).
12. R. Thakur, R. Rabenseifner and W. Gropp., *Optimizing of collective communication operations in MPICH*, ANL/MCS-P1140-0304, Mathematics and Computer Science Division, Argonne National Laboratory, (2004).
13. S. Sistare, R. vandeVaart and E. Loh., *Optimization of MPI collectives on clusters of large scape SMPs*, in: Proc. High Performance Networking and Computing, SC'99, (1999).
14. S. S. Vadhiyar, G. E. Fagg and J. Dongarra, *Automatically tuned collective communications*, in: Proceedings of High Performance Networking and Computing, SC'00, (2000).
15. R. C. Whaley and J. Dongarra, *Automatically tuned linear algebra software*, in: High Performance Networking and Computing, SC'98, (1998).

Parallel Data Distribution and I/O

Optimization Strategies for Data Distribution Schemes in a Parallel File System

Jan Seidel[1], **Rudolf Berrendorf**[1], **Ace Crngarov**[1], **and Marc-André Hermanns**[2]

[1] Department of Computer Science
University of Applied Sciences Bonn-Rhein-Sieg, 53754 St. Augustin, Germany
E-mail: mail@janseidel.net, {rudolf.berrendorf, ace.crngarov}@fh-bonn-rhein-sieg.de

[2] Central Institute for Applied Mathematics
Research Centre Jülich, 52425 Jülich, Germany
E-mail: m.a.hermanns@fz-juelich.de

Parallel systems leverage parallel file systems to efficiently perform I/O to shared files. These parallel file systems utilize different client-server communication and file data distribution strategies to optimize the access to data stored in the file system. In many parallel file systems, clients access data that is striped across multiple I/O devices or servers. Striping, however, results in poor access performance if the application generates a different stride pattern. This work analyzes optimization approaches of different parallel file systems and proposes new strategies for the mapping of clients to servers and the distribution of file data with special respect to strided data access. We evaluate the results of a specific approach in a parallel file system for main memory.

Keywords: Parallel I/O, memory file system, I/O optimizations

1 Introduction

In parallel file systems file data is stored on multiple storage devices and multiple paths to data are utilized to provide a high degree of parallelism. The majority of todays parallel file systems, including GPFS[1], Lustre[2] and PVFS2[3] implement a simple striping data distribution function. This distribution function distributes data with a fixed block size in a round-robin manner among all participating servers or storage devices. In many file systems the block size is the only variable factor of the data distribution, it can be adjusted according to application requirements.

However, different access pattern analyses[4,5] demonstrate that many parallel applications utilize complex logical data distribution patterns even for regular data structures. The input and output data sets of parallel applications are often matrices or other multidimensional data structures that are distributed among the MPI processes (clients). The logical distribution of multidimensional arrays results in complex nested strided I/O access patterns of the clients, as illustrated in Fig. 1. According to[6], the standard approach of striping file data yields multiple drawbacks for these complex access patterns, including:

- Complex access patterns of clients result in the exchange of many small messages between clients and servers, because the accessed data of clients is fragmented in small parts on multiple servers, resulting in poor spatial locality. Network and storage devices, however, are optimized for the transfer of large blocks.
- Contention of accesses at I/O servers can decrease the achievable parallelism between servers.

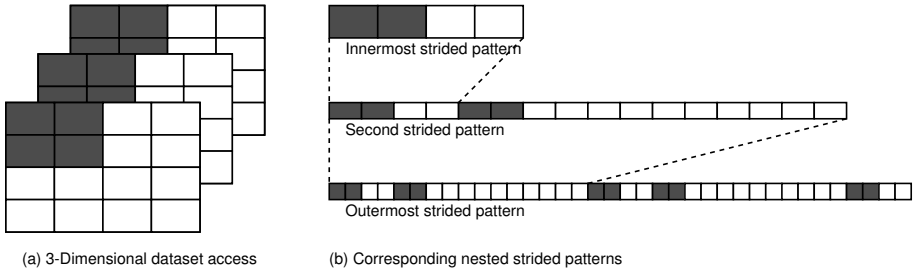


Figure 1. Nested-Strided Example (Example based [7])

- The mismatch between the logical and the physical distribution of file data requires complex mappings between client and server data distributions for each data transfer.

We present a new approach to automatically relate the physical partitioning of file data to the logical distribution of data between application clients. This mechanism is based on MPI file views that define the accessible file regions for clients. File views are commonly used in MPI programs to partition complex data structures between clients and constrict the access pattern for each client to the regions that are visible in the particular file view.

2 Related Work

The MPI-IO implementation ROMIO utilizes two techniques to optimize strided access to file data. Data sieving is used to reduce the effect of high I/O latency by performing less requests to the file system[8]. Data sieving increases the amount of data that is transferred between the client and the server. It also increases the network load and requires a temporary buffer as large as the number of bytes from the first requested byte to the last requested byte. This technique is useful if the amount of overhead data is relatively small.

The second optimization of ROMIO is two-phase-I/O which is used in collective operations. It is useful when multiple clients partition contiguous data so that each single client accesses noncontiguous parts, e.g. when partitioning a matrix. Two-phase-I/O also requires an additional temporary buffer and has to exchange additional data between clients in the redistribution phase.

Some parallel file systems support more complex data distribution functions than striping that are oriented on the logical distribution of file data. PVFS2 introduces a mechanism where hints can be passed to the file system to give information about the data access pattern of an application[3]. Using this information, the data distribution function can be adjusted to the application requirements. PVFS2 therefore natively supports structured data access using the same constructs as MPI (e.g. MPI datatypes).

Another approach that targets this direction is Clusterfile[6]. It supports flexible physical partitioning of file data. The data distribution model is optimized for multidimensional array partitioning. Clusterfile uses its own data type constructors, the nested pitfalls. It differentiates between the logical partitioning (the file views) and the physical partitioning (the storage on different servers) of a file, supporting overlapping file views. Both the logical and the physical data distribution are described by nested pitfalls and there exist

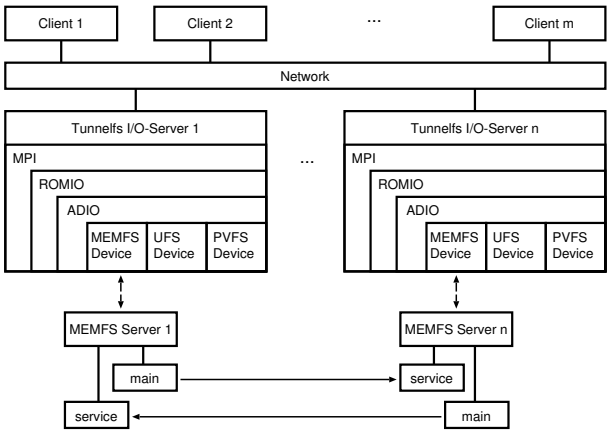


Figure 2. MEMFS setup with clients and servers

efficient mapping and redistribution algorithms between nested pitfalls. Mapping functions between physical and logical distributions are used to transfer data between files and buffers at I/O operations. Clusterfile supports arbitrary physical partitions and it is optimized for nested-strided partitions. The physical partition can be passed from the application to Clusterfile through an interface.

3 Design of a Specialized Parallel File System

We designed and implemented a parallel file system targeted primarily to flexible Grid environments. This file system stores file data in the main memory of remote clusters utilizing an optimized communication device for large distance high bandwidth networks [9]. In our approach, the file data is automatically distributed according to the file views of application clients. In contrast to the approaches of PVFS2 and Clusterfile, where the application needs to define a data distribution function, our development does not require to define a distribution function within the application.

The development of our parallel file system is split into two parts: TUNNELFS provides transparent access to remote servers in a grid environment, and MEMFS builds a parallel file system in the main memory of multiple nodes. This design is illustrated in Fig. 2 where clients communicate with the TUNNELFS I/O servers through the TUNNELFS ADIO device.

Together with TUNNELFS, MEMFS is utilized for high-bandwidth parallel I/O to the main memory of remote clusters in VIOLA[10]. MEMFS operates without any disk I/O and therefore completely removes the bandwidth limitations of hard disks. Instead, it leverages the available memory resources of different cluster sites and is therefore able to satisfy very high bandwidth demands.

The I/O performance of an application using TUNNELFS and MEMFS is defined by the transfer of data from the clients to the servers at write operations and vice-versa at read operations. Two parts of our parallel memory file system together define the transfer of

file data: The distribution of file data among the servers and the communication pattern of clients and servers. These two parts are now introduced.

4 Data Distribution on I/O Server

Given a set of I/O clients $C = \{c_1, \ldots, c_k\}$ (i.e. MPI processes) and a set of I/O servers $S = \{s_1, \ldots, s_l\}$ a distribution of file data is the (disjoint) partitioning of a file F on a subset of the available I/O servers S. A distribution function defines a partition element p_i for each server $s_i \in S$. Two constraints define the potential partitioning patterns: First, the union of all partition elements has to define the complete file F. Second, the partition elements have to describe non-overlapping file regions. Taken together, these constraints guarantee that each file byte is mapped onto exactly one partition element (one server).

To optimize the data access of scientific applications MEMFS distributes file data according to file views of application clients. The data region of a file that a client accesses is described by its file view. The accessible region of each client is passed to the file system as a MPI datatype. The file system can analyze this datatype to optimize for a specific client-data access scheme. In MEMFS, the accessible file region of each client is stored physically sequential in a so-called subfile, eliminating the access gaps introduced by non-sequential file view descriptions.

The different subfiles are stored contiguously on one or by striping on multiple I/O servers, eliminating the gaps of the file view description. If the number of clients $|C|$ is not a multiple of the number of servers $|S|$, the subfiles of $\lfloor \frac{|C|}{|S|} \rfloor$ clients are assigned to one specific server, while the subfiles of the remaining $|S| \bmod |C|$ clients are striped across disjunct sets of servers. This is a basic mechanism to balance the load and the storage amount between the servers. This works well if the amount of I/O data is balanced between the clients. In the future we plan to investigate into more sophisticated load balance mechanisms that take the size of defined file views into account.

Our approach is primarily targeted for high-bandwidth I/O from a single application rather than for a general purpose parallel file system with concurrent accesses from many applications. With our data distribution mechanism, changes to file views of clients require complex movement of file data between I/O servers. It requires two mappings, one from the subfiles to the logical sequential byte stream of the file and one from this stream to the new file views. Using these mapping functions, the data is redistributed to the newly set logical distribution. As the definition of a file view is separated from subsequent accesses this time consuming redistribution can be done by the servers in the background. Without a redistribution the accesses would be still work but would be slower.

The view-based data distribution mechanism is especially expedient for complex nested-strided logical file partitions that occur with multidimensional data partitioning. Logically sequential accesses of a client translate into physically sequential accesses, providing higher I/O performance than with data striping where these partitions result in numerous small noncontiguous accesses at multiple servers (compare to Section 1). Data can be transferred from single servers in large blocks, the optimal access pattern of I/O and network devices.

In the current implementation of the file view based data distribution, overlapping file views are not supported. File views are directly mapped to subfiles (partition elements) which have to define non-overlapping file regions. This includes overlapping file access

of different communicators to the file. To support overlapping file views, an additional definition of subfiles is required. I/O accesses can then be performed with a mapping function between the file views and the subfiles. This strategy can be similar to the approach of Clusterfile, with independent logical and physical data distributions. However, to our knowledge there currently exist no efficient mapping algorithms between arbitrary MPI datatypes. As a solution to this, either these mapping algorithms need to be developed or the file system must convert to another data distribution representation, like nested pitfalls in Clusterfile.

MEMFS supports the activation and deactivation of the file-view-based data distribution by a hint. Applications that use overlapping file views can deactivate this distribution and fallback to the standard striping approach. The hint can be set for each file separately.

5 Client-Server Mapping

As a restriction due to the separation of TUNNELFS and MEMFS, the TUNNELFS clients do not have knowledge about the MEMFS-specific file data distribution used to partition a file among multiple I/O servers. Instead, TUNNELFS provides an interface to MEMFS that is used to dynamically map clients to servers.

A client accesses its server for all I/O operations until the mapping is dynamically changed. A client-server mapping shall optimize the *data matching* of client requests. It is defined as the fraction of requested data located at the contacted server: Given a request R described by an offset off, a file view $view$ and a request size rs ($R = (off, view, rs)$), the function $g : R \times S \rightarrow \mathbb{R},\ \ g(r, s_i) = x, 0 \leq x \leq 1,\ \ r \in R, s_i \in S, x \in \mathbb{R}$ computes the data matching for a request r and a server s_i, where x is the fraction of r that is stored on s_i. The maximum reachable data matching for a request is the maximum fraction of r stored at a server $s \in S$. A high data matching means that a high fraction of data can be transferred directly between the client and the contacted server. The remaining parts of the request, $1 - x$, need to be transferred from other servers (this is done transparently to the client by the client-contacted server), introducing additional hops and potentially requiring transfer of small pieces separated by gaps.

Using the data distribution approach introduced in the previous section, each client gets mapped to one of the servers that stores its subfile. For the clients whose subfiles are stored on a single server – in most cases the majority of clients as only $|S| \bmod |C|$ subfiles are striped – this mechanism reaches a data matching of 1 for all client requests. The complete accessible file region of the client is then stored at a single server, enabling data transfer in one communication operation.

The clients whose subfiles are striped among multiple servers are assigned to the server that holds the subfile data that starts where the last request of that client ended. This algorithm assumes a sequential access pattern of the client, which is the most common but not the only access pattern of common scientific applications (see [4,5]).

6 Results

In[9] we demonstrated the capability of our parallel memory file system MEMFS to provide high-performance I/O to applications by storing file data in the memory of remote nodes.

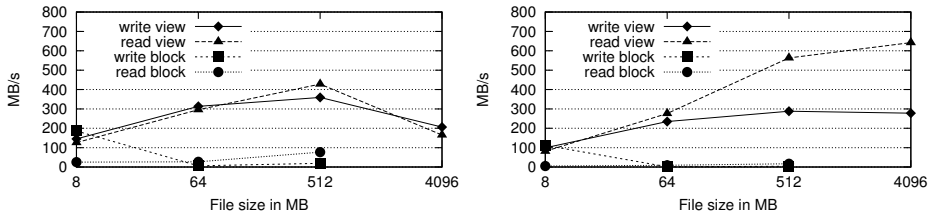


Figure 3. Strided benchmark for MEMFS with two different data distribution schemes. Left: 6 clients and servers. Right: 12 clients and servers.

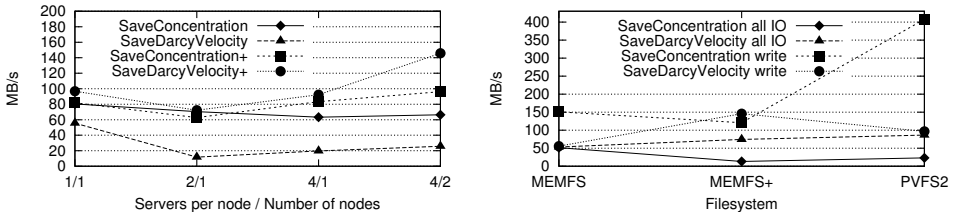


Figure 4. Left: MetaTrace Results for unoptimized MEMFS and optimized MEMFS (+) with varying amount of clients and servers. Right: Maximum MetaTrace transfer rates for unoptimized MEMFS, optimized MEMFS (+) and PVFS2

In this former version, high bandwidth I/O was limited to accesses to large, contiguous regions. Noncontiguous accesses showed drastic performance decreases mainly because of under-utilization of the underlying network connections.

Using the data distribution based on the file views of clients and a related client server mapping we are now able to reach high-performance even for complex, nested-strided accesses of clients. To measure the strided I/O performance of MEMFS, we benchmarked two applications. The first is a synthetic benchmark of the Indiana University, which uses MPI-IO operations to distribute a matrix between multiple processes[11]. The application defines a 3-dimensional matrix which is distributed blockwise in each dimension among the participating processes. This is a common distribution pattern of applications with regular array decomposition. The blockwise matrix distribution is defined as an MPI datatype which is used as the fileview for the MPI-IO data transfer. Each application process writes its data amount, reads it back and verifies it for correctness.

The benchmark was performed on two clusters physically separated with a distance of around 20 kilometers and connected through a 10 Gbit/s optical network. One of the clusters is a 6-node 4-way Opteron based system with 8 GB main memory and one Gigabit Ethernet network adapter per node. The maximum achievable transfer rate is therefore 6 Gb/s or 750 MB/s. The second cluster (PCC cluster) is a 32-node cluster with 2-way Intel Xeon processors and 1 GB main memory.

The 3-dimensional block distribution of the 3-dimensional matrix in this application results in complex, nested strided access patterns. We compared the results of the new data distribution with the former MEMFS approach of striping as described in Section 1 and that is used in most parallel filesystems. Figure 3 shows the results for 6 (12) servers

placed at one cluster and 6 (12) clients placed at the other cluster. In this figure ”block” denotes the results of the striping distribution (block-based) and ”view” denotes the view-distribution. Figure 3 shows a very large gap between the results collected with striping and the view-distributed results for all file sizes larger than 8 MB. These results show the problems of a mismatch between the logical and the physical file layout. With striping each client needs to access data parts from all servers and transfers many small messages over the network. The view distribution reaches high transfer rates because large data chunks are transferred between clients and servers.

With the view distribution the read operations reach more than 600 MB/s at a file size of 4096 MB, while the write operations still reach approximately 290 MB/s. The higher read performance partially results from the increased costs of dynamically allocating memory at write operations. The corresponding maximum number for the traditional strided distribution approach for this access pattern are 80 MB/s for read operation and 20 MB/s for write operations respectively at a file size of 512 MB. We were not able to run the application with a file size of 4096 MB and the strided distribution. We assume that this is a problem resulting from the numerous strided data accesses with this distribution type.

The performance of the optimization was also tested with the scientific application Metatrace. Metatrace simulates the diffusion of harmful substances in groundwater.The application is composed of two subprograms both writing simulation data in parallel. We benchmarked two functions, one writing contiguous blocks of data to a shared file (SaveConcentration) and one writing a block-distributed 3-dimensional data array (SaveDarcyVelocity). 8 processes were started for each subprogram of Metatrace on one of the clusters and up to 8 I/O server processes on the second cluster. In our testcase the different functions wrote between 55 MB and 85 MB of data per timestep with 20 simulation steps. In the function that distributes a 3-dimensional data array among processes, the view distribution version of MEMFS reaches a maximum of 145 MB/s with 8 servers, while MEMFS striping reaches a maximum of 60 MB/s (see Fig.4) and PVFS2 a maximum of 95 MB/s. PVFS2 clearly shows the best performance in the function that writes data sequentially to a file. Here, the MEMFS data distribution optimizations can not improve the I/O performance of MEMFS, because data is accessed sequentially by each client. The large gap between the sequential and strided data accesses in PVFS2 show the demand of efficient data distribution strategies for the very common nested-strided data accesses. The results of the optimized MEMFS version show that directly using the information of MPI file views can help to accelerate strided data transfers.

7 Conclusion and Future Work

Many parallel file systems store data distributed over several I/O servers / devices with a unit stride. This may result in poor I/O performance if I/O accesses in the application do not match this stride because clients then request data stored on many different servers.

We described an optimization technique that stores data on multiple I/O servers based on an user defined MPI application file view rather than in a simple strided way. I/O operations on MPI clients therefore touch data on only one or few I/O servers. This can significantly reduce overhead that gets introduced by traditional stripe-based server distribution strategies. A client server mapping related to the distribution function additionally reduces communication overhead. These optimizations are implemented in our parallel file

system MEMFS and can be introduced in any parallel file system.

High I/O bandwidth can be reached if physical application access patterns match the strided storage pattern. We have shown that utilizing MPI file views for distributing data can result in high-bandwidth for nested-strided access patterns.

Acknowledgements

This work was supported within the VIOLA project by the German Ministry of Education and Research under contract number FKZ 01AK605L, and by a research grant of the University of Applied Sciences Bonn-Rhein-Sieg.

References

1. IBM Corporation, *IBM General Parallel Filesystem*, (2006). http://www.ibm.com\discretionary{-}{}{}/systems/clusters/software/gpfs.html.
2. Inc. Cluster File Systems, *Lustre whitepaper: A scalable, high-performance file system*, (2002). http://www.clusterfs.com/
3. N. Drakos, R. Moore and R. Latham, *Parallel Virtual File System, Version 2: PVFS2 Guide*, (2006), http://www.pvfs.org/pvfs2-guide.html.
4. N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Ellis and M. Best, *File-Access Characteristics of Parallel Scientific Workloads*, IEEE Transactions on Parallel and Distributed Systems, **7**, 1075–1089, (1996).
5. F. Wang, Q. Xin, B. Hong, S. Brandt, E. Miller, D. Long and T. McLarty, *File system workload analysis for large scale scientific computing applications*, in: Proc. 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies, (2004).
6. F. Isaila and W. F. Tichy, *Clusterfile: a flexible physical layout parallel file system.*, Concurrency and Computation: Practice and Experience, **15**, 653–679, (2003).
7. W. Gropp, E. Lusk and T. Sterling, *Beowulf Cluster Computing with Linux*, vol. **2**, (MIT Press, 2003).
8. R. Thakur, W. Gropp and E. Lusk, *Optimizing noncontiguous accesses in MPI-IO*, Tech. Rep., Mathematics and Computer Science Division - Argonne National Laboratory, (2002).
9. J. Seidel, R. Berrendorf, M. Birkner and M.-A. Hermanns, *High-bandwith remote parallel I/O with the distributed memory filesystem MEMFS*, in: Proc. PVM/MPI 2006, Lecture Notes in Computer Science, pp. 222–229, (Springer-Verlag, 2006).
10. The VIOLA Project Group, *Vertically Integrated Optical testbed for Large scale Applications*, 2005, http://www.viola-testbed.de/
11. Z. Meglicki, *High Performance Data Management and Processing, darray.c*, Tech. Rep., Indiana University, (2004). http://beige.ucs.indiana.edu\/I590/node102.html

Parallel Redistribution of Multidimensional Data

Tore Birkeland and Tor Sørevik

Dept. of Mathematics, University of Bergen, Norway
E-mail: {*tore.birkeland, tor.sorevik*}*@math.uib.no*

On a parallel computer with distributed memory, multidimensional arrays are usually mapped onto the nodes such that only one or more of the indexes becomes distributed. Global computation on data associated with the reminding indexes may then be done without communication. However, when global communication is needed on all indexes a complete redistribution of the data is needed. In higher dimension ($d > 2$) different mappings and subsequent redistribution techniques are possible. In this paper we present a general redistribution algorithm for data of dimension d mapped on to a processor array of dimension $r < d$.

We show by a complexity analysis and numerical experiments that while using a $1D$ processor grid is the most efficient for modest number of processors, using $2D$ processor grid has better scalability and hence work best for higher number of processors.

1 Introduction

A common situation in parallel computation on multidimensional data is to perform calculations which involve all data along a specific dimension, while the calculations are completely decoupled along the other dimensions. For such problems it is possible to simplify the parallel algorithms by keeping at least one dimension local, while distributing the other dimensions among the processors. In most cases, however, it is necessary to perform calculations along all dimensions sequentially, which introduces the problem of redistributing the data among the processors. A prime example of this problem is the multidimensional fast Fourier transforms (FFT), where the full FFT can be calculated by performing 1D FFTs along all dimensions sequentially[1,2,3,4].

The standard way of dealing with problems of this type, is to distribute only one dimension of the data set at a given time. Redistribution can then be performed with the calls to the MPI function MPI_Alltoall. As other authors have pointed out[5,6], this approach has limited scalability as the number of processors, P, is limited by the smallest dimensional grid size $P < \min N_i$. Furthermore, many massively multiprocessor computers utilizes special network topologies (i.e. toroidal), which this technique is not able to exploit.

In this paper we describe a generalized algorithm for data redistribution and its implementation using the MPI. We also analyse the computational complexity of the algorithm, and discuss in which cases it is favourable over the standard approach.

2 Problem Definition and Notation

Consider a d-dimensional dataset of size $N_0 \times N_1 \times \cdots \times N_{d-1}$ which is mapped onto an r-dimensional processor array of size $P_0 \times P_1 \times \cdots \times P_{r-1}$. $1 \leq r < d$. The mapping is done by splitting the data set along r dimensions in equal pieces. We get different splittings depending on which dimensions we choose to split. There is of course $\binom{d}{r}$

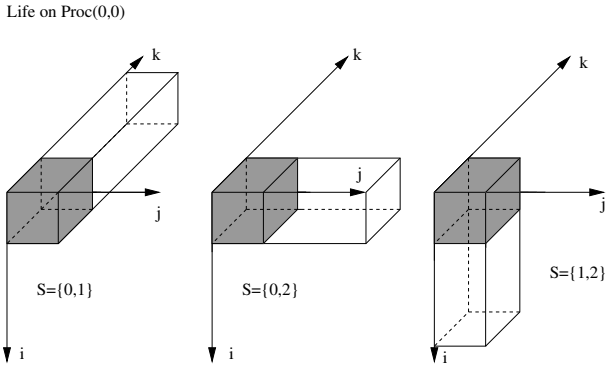


Figure 1. *This figure shows the 3 different slices of 3D data onto 2D processor array. The slice of the data shown here is the local data to $P_{0,0}$. The shaded part is the portion of the local data that is invariant for all 3 different slices. Thus it does not have to be moved whenever a redistribution is needed.*

possible mappings. In a computation, as exemplified by the d-dim FFT, the mapping will have to change during the computation.

For convenience we assume that $N_i \bmod P_j = 0$ for all $i = 0, \cdots, d-1$ and $j = 0, \cdots, r-1$. This requirement is easy to overcome in practise, but it simplifies the notation significantly in this paper. In practise we have some control over r and the P_j's, while d and N_i are defined by the problem. A straight forward way to deal with the $N_i \bmod P_j = 0$ requirement is to set $P_1 = P_2 = \cdots = P_r = P$ and pad the data array with zeroes to satisfy $N_i \bmod P = 0$. Another way (which we have used in our implementation) is to modify the algorithm slightly so that it can work with different amounts of data residing on each processor.

Let $S = \{i_0, i_1, ..., i_r\}$ be an index set where $0 \leq i_j < d$ for $j = 0, \cdots, r-1$ Then S_{now} denotes the dimensions which are distributed among the r-d processor array. A dimension can only be distributed over one set of processors, which gives $i_j \neq i_k$, if $j \neq k$. If we want to do computation on dimension k, where $k \in S_{now}$, a redistribution is required.

Let S_{next} be a distribution where $k \notin S_{next}$. The dimensions $S_{now} \backslash S_{next}$ will be distributed, while the dimensions $S_{next} \backslash S_{now}$ will be gathered.

3 Algorithm

3.1 Redistribution of One Dimension

We will now assume that the difference between S_{now} and S_{next} is exactly one index, i.e. the operation to be performed is an all-to-all along one dimension. For such an operation, the processors can be organised in groups, where a processor only communicates with other processors in the same group. For redistribution along different dimensions in the processor grid, different groups will have to be formed. In general, one set of groups will be formed for each dimension in the processor array. A processor P_α, where $\alpha = (\alpha_0, \alpha_1, ..., \alpha_{r-1})$, will be a part of the groups $G^j_{\alpha_j}$, for $j = 0, 1, ..., r-1$ (Figure 3.1).

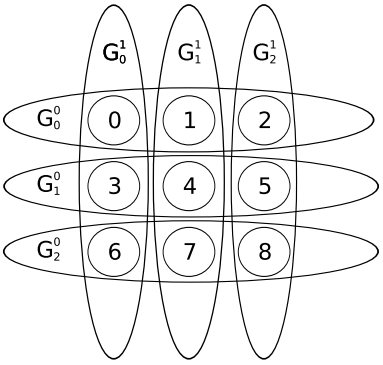


Figure 2. *A $3x3$ processor array. The processors are organised into one group for each dimension in the processor array. For redistributing the pth dimension in the processor array, processors in the G^p groups will communicate internally.*

For communication within one group, an algorithm similar to the standard implementation of all-to-all is used. Below is an implementation of this algorithm in simplified Python-like syntax. `inData` and `outData` are the input and output data arrays local to the current processor. `fullShape()` returns the shape of the global array, and `shape(x)` returns the local size the array `x`. `inDistrib` and `outDistrib` are the dimensions of the data set which is distributed at the beginning and end of the algorithm respectively. `groupSize` is the number of processors in the communication group.

```
sendSize = fullShape(inDistr)/groupSize
recvSize = fullShape(outDistr)/groupSize

for i in range(groupSize):
  sendProc = (groupRank + i) % groupSize
  recvProc = (groupRank + groupSize - i) % groupSize

  sendSlice = shape(inData)
  sendStart = sendProc*sendSize
  sendEnd = (sendProc+1)*sendSize
  sendSlice[inDistr] = sendStart:sendEnd
  sendBlock = inData[sendSlice]

  recvSlice = shape(outData)
  recvStart = recvProc*recvSize
  recvEnd = (recvProc+1)*recvSize
  recvSlice[outDistr] = recvStart:recvEnd
  recvBlock = outData[recvSlice]

  irecv(fromProc, recvBlock)
  isend(toProc, sendBlock)
  wait()
```

A test implementation of the above algorithm has been made in C++. To set up the processor groups we have used the Cartesian topology routines in MPI, and set up a communicator for each processor group. This is an easy way to set up a r-dimensional processor grid, and allows for optimised MPI implementations to exploit locality in the underlying network topology without user interaction.

For handling multidimensional data in C++ we have used the excellent blitz++ library[7]. Using blitz++ and MPI datatypes we have been able to hide the details of sending and receiving a strided hyperslab, which has simplified the implementation of the redistribution considerably.

3.2 Redistribution of Several Dimensions

If $r = 1$ or $r = d-1$ the above algorithm cover all possibilities. However, for $1 < r < d-1$ one might pose the problem of how to redistribute more than one dimension at the time. One alternative is to apply the above algorithm several times. Assume that $S_{now} \rightarrow S_{next}$ is a simple redistribution, that is if $s_i^{now} \neq s_i^{next} \rightarrow s_i^{now} \notin S_{next}$ $s_i^{next} \notin S_{now}$. For such a redistribution, the following simple algorithm will change distribution from `sNow` to `sNext`.

```
for i in range(r):
  if sNow[i] != sNext[i]:
    redistribute(data, groupIndex=i,          \
      inDistrib=sNow[i] outDistrib=sNext[i])
```

4 Lower Dimensional Projections

An alternative method for redistributing several dimensions simultaneously, is to map the data set to a lower dimensional data set, and distribute that array on a low dimensional processor array. In the extreme, one could map the data to a 2 dimensional data set and use a 1 dimensional processor array. However, if $N_i \bmod P_j \neq 0$, one risks splitting up a dimension in the data set in an unpredicted way, and one must therefore take care to use the correct indexes in such situations.

In our previous work this method has been deployed with great success[8],[9]. It is simple and in many cases efficient. It does however has restricted scalability. The reason for this is twofold. First there is a theoretical limit on the number of processors used which require us to have $\min(N_1, N_2) \geq P$. The prime example of this is for $d = 3$ and the data array is of the size $N \times N \times N$. In that case we will have N^3 data and only being able to use $P \leq N$ processors.

Secondly when all processors are involved in the same all-to-all communication this is a more severe stress for the bisectional bandwidth than when they are divided into smaller groups, each group doing an internal all-to-all simultaneously. This effect is increasing with P and may also dependent on the network topology.

5 Complexity Analysis

This analysis is not meant as a detailed analysis, suitable for accurate prediction of the communication time. It only indicates the pro- and con's of the different strategies, and their dependence on the actual parameters.

For simplicity we assume that $N_0 = N_1 = \cdots = N_{d-1} = N$ and $P_0 = P_1 = \cdots = P_{r-1} = P$ and $N \bmod P = 0$. Then each processor will store N^d/P^r data items. The

algorithm of Section 3.1 will send pieces of N^d/P^{r+1} data in each of the $P-1$ steps. With latency t_s and reciprocal bandwidth t_w the complexity of this algorithm becomes:

$$(P-1)(t_s + t_w \frac{N^d}{P^{r+1}}) \tag{5.1}$$

Initially $d-r$ directions are not splitted, and local work in these dimensions can be carried out. Then, in each step, the algorithm provides one new direction for local computation. Thus to get data associated with the remaining r indexes to appear locally, the redistribution algorithm needs to be repeated r times. Thus for a complete sweep of local computation in all d direction the total work becomes:

$$W_1 = r(P-1)(t_s + t_w \frac{N^d}{P^{r+1}}) \approx r(Pt_s + t_w \frac{N^d}{P^r}) \tag{5.2}$$

Alternatively we might map the data to a 2D data set and the processor array to a 1D array of P^r processors. Each processors will still have the same total amount of data. However when applying our algorithm the piece of data to be sent is now only N^d/P^{2r} and the number of loop iterations becomes $P^r - 1$. However the upside is that only one redistribution is needed for a full sweep. The complexity becomes

$$W_2 = (P^r - 1)(t_s + t_w \frac{N^d}{P^{2r}}) \approx P^r t_s + t_w \frac{N^d}{P^r} \tag{5.3}$$

The amount of data transferred is less in the second case, but the number of start-ups might be much higher. We notice that the problem parameters (N, d), system parameters related to processors configuration (P, r) as well as those related to communication network (t_s, t_w) all play a role in determining the fastest communication mode.

We would like to caution about using the above formula for predicting the fastest way of organising your computation. Additional factors such as network topology as well as system and communication software will also be of importance. Note also that when arranging the processors as a 1D-array one more easily get significant load imbalance when the $N \bmod P = 0$ condition is violated.

6 Numerical Experiments

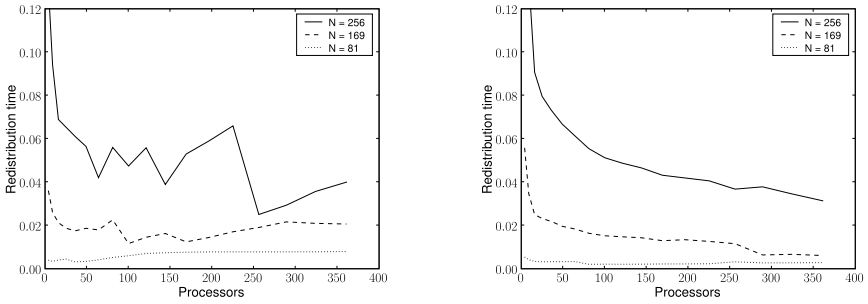
We have performed several numerical experiments to determine the properties of the redistribution algorithms. The implementation has been tested on different platforms with different network topologies to determine if we are able to effectively exploit special topologies more efficiently than with the previously used algorithm. For measuring efficiency, we have used the wall clock time it takes to complete two complete sweeps of redistribution. That is $2r+1$ redistributions with the algorithm presented in 3.1.

In order to minimise any effect of function calls to the timer, the process has been repeated 10 times and the averaged time is recorded. Each of these tests was run 10 times and to minimise the effect background processes in the operating system, we report the minimal time. Furthermore the complete test was run twice with several days separation in order to detect if any of the tests were influenced by other jobs running on the computer.

The main performance test is to redistribute a $3D$ data set of size $N \times N \times N = N^3$, on processor grid of $P \times P = P^2$ processors. The test is run for several values of P and N, comparing the time it takes to redistribute the data set on a $2D$ vs. a $1D$ processor grid.

	Platform 1	Platform 2
Provider	NTNU	Argonne National Lab
System	IBM p575+	IBM Blue Gene/L
CPU	IBM Power5+ 1.9 GHz	PowerPC 440 700 MHz
Cores pr. node	16	2
Total nodes	56	1024
Interconnect	HPS	Blue Gene/L Torus
Topology	Fat tree	3D Torus

Table 1. The platforms used for testing the redistribution algorithms



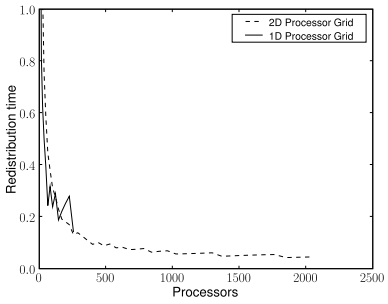
(a) $1D$ processor grid (b) $2D$ processor grid

Figure 3. Platform 1 redistribution time as a function of number of processors P^2 plotted for different grid sizes. The left panel shows wall time using a $1D$ processor grid, and the right panel shows the results for a $2D$ processor grid.

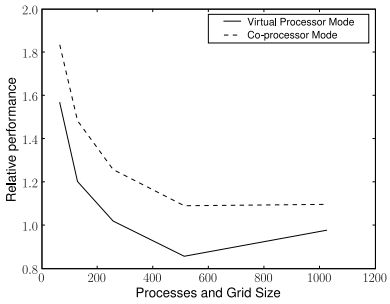
To our disposal we have had 2 different platforms. The key hardware features of these are described in Table 1.

Platform 1 uses a high speed HPS interconnect. We have not detected any significant dependence on how the processor grid is mapped onto the physical processors, suggesting that the network topology is not an important factor for this platform. In Figure 3 the results for the main performance test is shown both for a $1D$ and a $2D$ processor grid. As expected, for few processors, the $1D$ processor grid is superior. However, the $2D$ configuration gives better scaling and eventually becomes faster then the $1D$ configuration. The crossover-point appears to be $P^r \approx N/4$. The jagged form of the curves in left figure is a consequence of the fact that when the data can not be distributed evenly among the processors, the performance will be dictated by the processor with most data rather than any of the latency effects described in Section 5. The case $N = 256$ and $r = 1$ illustrate this explanation. Here we observe local minima for $P = 128$ and $P = 256$.

Platform 2 is an IBM Blue Gene/L system, and uses a special 3D toroidal network for bulk data transfer. The network topology is interesting as it should fit well to a logical $2D$ configuration of the processors, and by carefully mapping the logical $2D$ processor grid onto the physical $2D$ toroidal node grid, one might expect some performance improvement compared to the $1D$ processor grid. In order to test this hypothesis, we have run four test cases. One reference test using a $1D$ processor grid, and three tests with a $2D$ processor grid. Using the $2D$ processor grid, we have varied the mapping of the processor grid on the



(a) Platform 2 redistribution time as a function of number of processors P^r plotted for grid size $N = 256$ for a $1D$ and $2D$ processor grid.



(b) Relative performance of a $2D$ processor grid compared to $1D$. $N = P^r$ for different values of N. Values larger than 1 means the $2D$ processor grid is faster

Figure 4. Numerical results from Platform 2

physical nodes in order to detect if the performance is dependent on the topology. However, through all our tests, we have not been able to observe any difference in performance between the three $2D$ processor grids used.

Figure 4 shows results from Platform 2. In essence, the results are similar results to that of Platform 1. For a given N, a $1D$ processor grid performs best for few processors, but a $2D$ processor grid scales better. Interrestingly, we observe that for $P^r = N$ (Figure 4(b) (Which is the highest possible processor count for the $1D$ processor grid) the $2D$ processor grid performs better, even though twice the amount of data is being transferred. This means that not only does the $2D$ processor grid allow one to use more processors, it also enables more efficient utilization of the the network, most likely due to larger blocks of data being sent at each step in the redistribution. The relative decrease in efficiency of the $2D$ processor grid seen for increasing values of N and P^r, can be explained from the fact that block blocksize increases cubically with N, and decreases linearly with P^r. As the blocksize increase, we expect the startup effects for the $1D$ processor grid to decrease. In virtual processor mode, the performance of the $2D$ processor grid suffers most, because the bandwidth of each process is effectively halved compared to co processor mode.

7 Concluding Remarks

We have designed and implemented and algorithm for redistribution of multidimensional arrays. Complexity analysis and experiments show that our algorithm has better scalability than the standard algorithm which view the processors organised as a $1D$-grid. As the current trend in design of HPC-system is that number of processors increases much faster than their individual speed, sustained petaflop computing can only be achieved through highly scalable algorithms.

Our implementation is a generalization of the standard algorithm, making it easy to change the dimension of the processor grid at runtime depending on the size of the problem and the number of processors available at runtime.

Acknowledgement

We gratefully acknowledge the support of NOTUR, The Norwegian HPC-project, for access to their IBM p575+ at NTNU, and to Argonne National Laboratory for access to their Blue Gene/L System.

References

1. H. Q. Ding, R. D. Ferraro and D. B. Gennery, *A Portable 3D FFT Package for Distributed-Memory Parallel Archite ctures*, in: PPSC, pp. 70–71, (1995).
2. M. Frigo and S. G. Johnson, *FFTW: An adaptive Software Architecture for the FFT*, in: Proc. IEEE International Conference on Acoustics, Speech and signal Processing (ICASSP), pp. 1381–1394, (1998).
3. C. E. Cramer and J. A. Board, *The development and integration of a distributed 3D FFT for a cluster of workstations*, in: Proc. 4th Annual Linux Showcase and Conference, pp. 121–128, (2000).
4. P. D. Haynes and M. Cote, *Parallel fast fourier transforms for electronic structure calculations*, Comp. Phys. Commun., **130**, 132–136, (2000).
5. M. Eleftheriou, B. G. Fitch, A. Rayshubskiy, T. J. C. Ward and R. S. Germain, *Scalable framework for the 3D FFTs on the Blue Gene/L supercomputer: Implementation and early performance measurements*, IBM J. Res. & Dev., **49**, 457–464, (2005).
6. A. Dubey and D. Tessera, *Redistribution strategies for portable parallel FFT: a case study*, Concurrency and Computation: Practice and Experience, **13**, 209–220, (2001).
7. T. L. Veldhuizen, *Arrays in Blitz++*, in: Proc. 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98), (Springer-Verlag, 1998).
8. T. Sørevik, J. P. Hansen and L. B. Madsen, *A spectral method for integration of the time-dependent Schrödinger equation in hyperspherical coordinates*, Phys. A: Math. Gen., **38**, 6977–6985, (2005).
9. T. Matthey and T. Sørevik, *Performance of a parallel split operator method for the time dependent Schrödinger equation*, in: Computing: Software Technology, Algorithms, Architectures and Applications, pp. 861–868, (2004).

Parallel I/O Aspects in $\mathrm{PIMA(GE)^2}$ Lib

Andrea Clematis, Daniele D'Agostino, and Antonella Galizia

Institute for Applied Mathematics and Information Technologies
National Research Council, Genoa, Italy
E-mail: {*clematis, dago, antonella*}*@ge.imati.cnr.it*

Input/Output operations represent a critical point in parallel computations, and often results in a bottleneck with a consequent reduction of application performance. This paper describes the parallel I/O strategy applied in $\mathrm{PIMA(GE)^2}$ Lib, the Parallel IMAGE processing GEnoa Library. The adoption of a parallel I/O results in an improvement of the performance of image processing applications developed using the library. To achieve this goal we performed an experimental study, comparing an MPI 1 implementation of a classical master-slave approach, and an MPI 2 implementation of parallel I/O. In both cases we considered the use of two different file systems, namely NFS and PVFS. We show that MPI 2 parallel I/O, combined with PVFS 2, outperforms the other possibilities, providing a reduction of the I/O cost for parallel image processing applications, if a suitable programming paradigm for I/O organization is adopted.

1 Introduction

The scientific evolution allows the analysis of different phenomena with great accuracy and this results in a growing production of data to process. Parallel computing is a feasible solution, but a critical point becomes an efficient I/O management. This issue may derive from an hardware level and/or from a poor application-level I/O support; therefore I/O performance should be improved using both parallel file systems and effective application programming interface (API) for I/O appropriately[1].

In this paper we focus on these aspects for parallel image processing applications. We developed $\mathrm{PIMA(GE)^2}$ Lib, the Parallel IMAGE processing GEnoa Library; it provides a robust implementation of the most common low level image processing operations. During the design of the library, we look for a proper organization of I/O operations, because of their impact on application efficiency. In fact, a parallel application interacts with the underlying I/O hardware infrastructure through a *software stack*, depicted in Fig. 1; the key point is to enable a proper interaction between the different levels[2].

In particular a parallel image processing application developed with $\mathrm{PIMA(GE)^2}$ Lib exploits a software level, or *I/O Middleware*, aimed to perform I/O operations using MPI. The I/O Middleware imposes the logical organization of the parallel processes and the I/O pattern to access data. It also interacts with the file system, e.g. PVSF, NFS, that in turn effectively exploits the I/O hardware, managing the data layout on disk. Thus a proper use of the I/O API provided by MPI leads to a more efficient management of the I/O primitives of the file systems.

The adoption of a parallel I/O in scientific applications is becoming a common practice. Many papers provide a clear state of the art of the problem; an in-depth analysis of I/O subsystems, and general purpose techniques to achieve high performance are proposed[1,5]. They mainly suggest the use of specific software to enable parallel access to the data. It is actually obtained considering parallel file systems combined with scientific data library, in

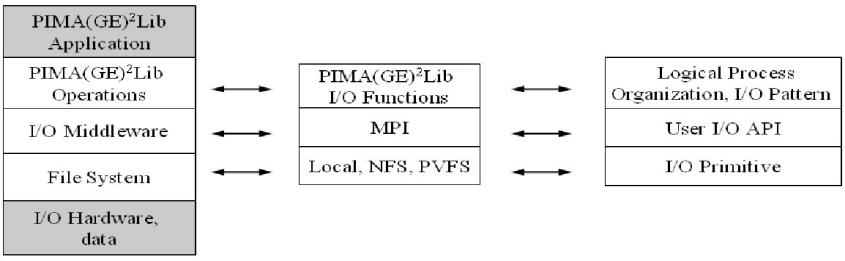


Figure 1. *Software stack for parallel image processing applications. Starting from the left, it is explained how an application interacts with data and I/O infrastructure, the tools used in* $\mathrm{PIMA(GE)^2}$ *Lib, and the roles of each software level.*

order to exploit their optimization policies. The user has to consider the strategy that better fits with the application requirements and the available I/O subsystem.

The proposed solutions have been adopted in different works, for example in visualization problems managing great amount of data[6], in simulations using particle-mesh methods[7], in biological sequence searching[8], in cosmology applications based on the adaptive mesh refinement[9]. Surprisingly, such strategies have not been sufficiently considered in the image processing community, although an increasing attention is paid to parallel computations. In fact it is possible to find different and actual examples of parallel libraries, ParHorus[10], PIPT[11], EASY-PIPE[13] and Oliveira et al.[12]. However they do not consider a parallel I/O, and apply a master-slave approach during the data distribution.

Starting from these remarks, we performed a case study about different approaches in the imaging community, where these aspects received only a little attention. We made several tests of the logical organization in I/O operations to determine the most efficient strategy to apply in $\mathrm{PIMA(GE)^2}$ Lib. To achieve this goal we compared a master-slave approach implemented with MPI 1, and a parallel I/O using the functionalities of the MPI-IO provided by MPI 2. In both cases we tested the interaction with the most common file systems for parallel and distributed applications, that are PVFS, Parallel Virtual File System[3], and NFS, Network File System[4], both open source. We show that MPI 2 parallel I/O, combined with PVFS 2, outperforms the other possibilities, providing a reduction of the I/O cost for parallel image processing applications. More in general a parallel I/O approach is effective in the image processing domain.

The paper is organized as follows: in the next Section a brief presentation of the $\mathrm{PIMA(GE)^2}$ Lib is given, including an overview of both I/O approaches we tested. In Section 3 we analyse the experimental results. The conclusions are outlined in Section 4.

2 A Brief Overview of the $\mathrm{PIMA(GE)^2}$ Lib with Different Parallel Organizations

The Parallel IMAGE processing GEnoa Library, shortly $\mathrm{PIMA(GE)^2}$ Lib, is designed with the purpose of providing robust and high performance implementations of the most common low level image processing operations, according to the classification provided

in Image Algebra[14]. The library has been implemented using C and MPICH, and allows the development of compute intensive applications, achieving good speed up values. The operations are performed both in a sequential and in data parallel fashion, according to the user requirements, on 2D and 3D data sets. The parallelism in $\mathrm{PIMA(GE)}^2$ Lib is hidden from the users through the definition of an effective and flexible interface, that appears completely sequential[15]. Its aim is to shield the users from the intrinsic complexities of a parallel application. An optimization policy is applied in the library in order to achieve good performance; the optimization aspects are transparent as well.

Let us focus on the I/O aspect of the library, with a description of the possible strategies to perform the I/O operations. Typically image processing applications acquire and produce data stored on files.

In order to avoid many small noncontinuous accesses to a possibly remote disk made from multiple processors, the classic logical organization for data distribution in parallel applications is the master-slave one. It means that a process, the master, entirely acquires

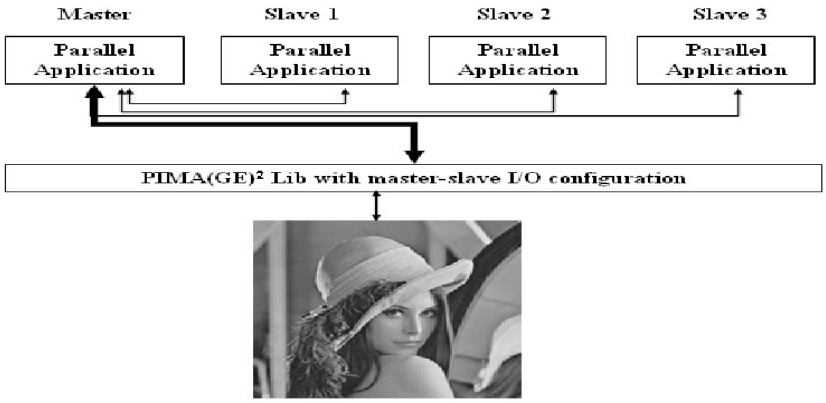


Figure 2. *Master-slave approach in accessing data. Only the master accesses the file, and sends portions of the image to the other processes.*

data and distributes them among the other MPI processes, the slaves, according to the I/O pattern. A specular phase of data collection is necessary for the output operations. Therefore the master is in charge of collecting/distributing data and performing I/O to a single file through a sequential API. This behaviour is depicted in Fig. 2.

However the data collection on a single process results in a serialization of the I/O requests and consequently in a bottleneck, because of the time the master spends in loading the entire data set and in sending the partial data to each process. A specular situation occurs for the Output operations. The waiting time for the distribution or the collection of data increases with the data set size. In case of huge data sets, the I/O execution may result very inefficient or even impossible. A further problem is the possibility to overwhelm the memory capacity, with the consequent necessity of exploiting the virtual memory.

An alternative approach to perform I/O operations and avoid unnecessary data movements is provided by a parallel access to the data file. It means that all processes perform

I/O operations, but each of them acquires or produces its specific portion of data. The situation is represented in Fig. 3.

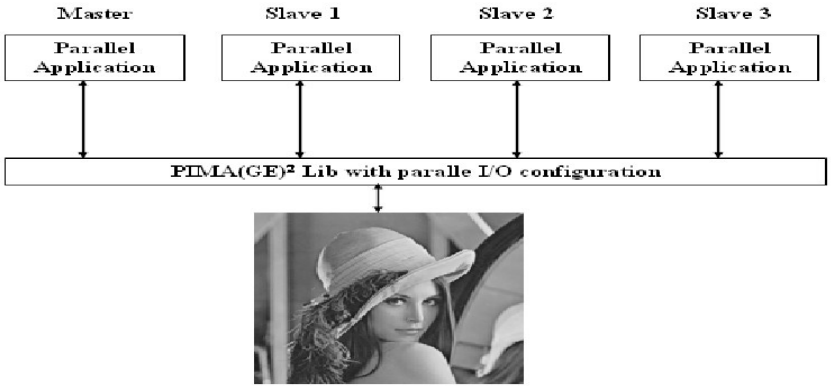


Figure 3. *The parallel I/O approach in accessing data. All the processed access data, and acquire their portion.*

However the partial data of each process may correspond to non contiguous small chunks of global data; it implies that each process accesses the I/O file to load small pieces of information non contiguously located. This situation worsens the performance even if sophisticated parallel file systems are used. In fact it is not possible to exploit file system optimization policy, since it is mainly designed to efficiently support the parallel access to large chunks of a file or of different files.

Another important point is given by the mismatch between the data access patterns and the actual data storage order on disk. This topic could be stressed if data are distributed on different machines. In this case, chunks of data requested by one processor may be spanned on multiple I/O nodes; or multiple processors may try to access the data on a single I/O node, suffering from a long I/O request queue. However combining the use of proper file systems, and API for I/O in a suitable way, avoids such situations and enables to effectively exploit a parallel I/O. In fact in this way it is possible to span data cleverly and remove the long queue in data acquisition.

3 I/O Experimental Results

We performed an experimental study about I/O organization in image processing, comparing the approaches already described, and fixing as I/O pattern a block partition. The master-slave approach was implemented through MPI 1, the parallel I/O using the functionalities of the MPI-IO provided by MPI 2. We are interested in the analysis of the I/O scalability and the impact of file systems on it; therefore, we measured how the growth of the number of processes affects the execution time in each case, and how each I/O approach interacts with different file systems, considering the use of PVFS 2, and NFS. In this paper we are not interested in the evaluation of the $\mathrm{PIMA(GE)}^2$ Lib performance, thus we do not consider other operations of the library.

3.1 Experimental Conditions

Tests have been performed on a Linux Beowulf Cluster of 16 PCs, each one equipped with 2.66 GHz Pentium IV processor, 1 GB of Ram and two EIDE 80 GB disks interfaced in RAID 0; the nodes are linked using a dedicated switched Gigabit network.

The experimental results were collected using the Computed Tomography (CT) scan of a Christmas Tree (XT)[a], and the CT scan of a Female Cadaver (FC)[b]. We considered such data sets because of their sizes; the XT data set can be considered a medium size data set, the partial image size varies from 499.5 MB to 31.2 MB considering respectively 1 and 16 processes, while the FC data set is a quite large size data set, and the partial image size varies from 867 MB to 54.2 MB in the same conditions.

With respect to the utilized software tools, let us provide few concepts about MPI-IO, and two widely used file systems for file sharing using clusters NFS and PVFS2.

NFS was designed to provide a transparent access to non local disk partitions. It means that, if we consider the machines of a local area network that mount the NFS partition on a specific directory, they are able to share and access all the file of that directory. Therefore a file sitting on a specific machine, looks to the users on all the machines of the network, as if the file resides locally on each machine.

PVFS stripes file data across multiple disks in different nodes in a cluster. It allows multiple processes to access the single part of the global file that have been spanned on different disks concurrently. We considered the second version of PVFS, (PVFS2), that allows the exploitation of fast interconnection solutions and the minimization of bottleneck due to the retrieving of metadata regarding the file to acquire or produce.

MPI-IO permits to achieve high performance for I/O operations in an easy way. Indeed it enables the definition of the most common I/O patterns as MPI derived data types, and in this way permits parallel data accesses. We considered the use of ROMIO[16,2], a high-performance, portable implementation of MPI-IO distributed with MPICH. ROMIO contains further optimization features such as Collective I/O and Data Sieving. These aspects have been implemented for several file systems, including PVFS2, NFS.

3.2 The Master-Slave Approach

We implemented the data partition through a sequential read operation performed by the master that immediately after scatters partial data to the slaves. The execution time (in seconds) are presented in Figure 4(a) for XT and in Figure 4(b) for FC data set.

In the tests involving NFS, we consider two different nodes of the cluster to store data and to run the master. In such situation, the data are not located on the same machine of the master, therefore we actually exploit the use of NFS in the data access. However we tested also a slightly different situation, i.e. the master has the data locally. In fact managing data through a remote file system, we have to take into account the overheads due to the latencies deriving from the file system and from the transmission time. We verified that considering XT data set the I/O using the local disk requires 0.9 seconds, and through NFS 1.2; while considering FC we have 1.4 and 4.6 respectively.

[a] The XT data set was generated from a real world Christmas Tree by the Department of Radiology, University of Vienna and the Institute of Computer Graphics and Algorithms, Vienna University of Technology,

[b] The FC data set is a courtesy of the Visible Human Project of the National Library of Medicine (NLM), Maryland.

In the master-slave approach, the striping of the data among multiple disks obtained with PVFS represents a drawback. Actually, since there is only one sequential access to the data file, the master process has to acquire data by accessing small parts on multiple disks. This represents an useless time consuming step. Indeed in Figure 4 we can see that on both data sets, the I/O performance do not scale and the execution time is almost constant. It is due to the overhead related with the use of the file system to access remote part of the data. It results higher than the time spent to send/receive data, in fact we do not verify a significant variation in the execution time even with the growth of the number of processes, i.e. the number of send/receive operations to perform.

On contrary, the use of a single disk through NFS represents on average the best solution, despite the data set size may really affect the performance. As it is possible to see in Figure 4, the use of NFS performs better than PVFS when we consider a medium size data set; but when we manage a large data set the use of PVSF leads to better performance when the number of process increases.

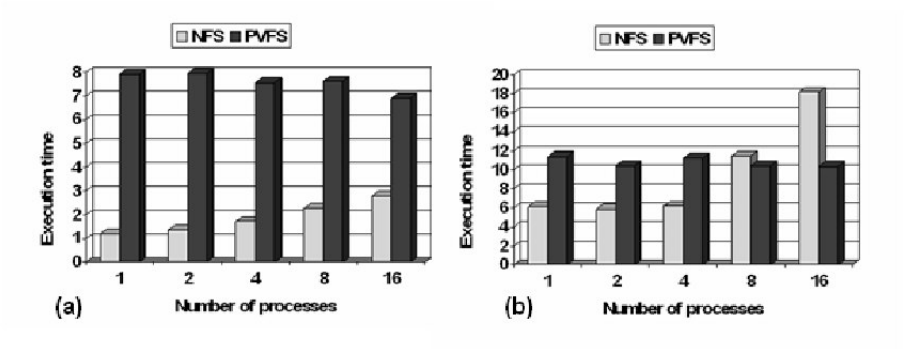


Figure 4. The execution time (in seconds) of the I/O operations using the master-slave approach on the XT data set (499.5 MB) (a) and the FC data set (867 MB) (b)

In fact in this case, even if data file is accessed through NFS, the data set is placed on a single disk of the cluster; that implies a lower overhead due to the file system. On the other hand, when the number of processes grows, the execution times suffer from the communication overheads. Considering up to 4 processes, the time could be considered similar to the sequential case; however the performance are really deteriorated if an higher number of processes is considered.

3.3 Parallel I/O

We implemented the parallel I/O using the collective I/O features and the derived datatypes provides by MPI 2. The execution time (in seconds) are presented in Figure 5(a) considering the XT data set, and Figure 5(b) for the FC data set. We can see that the parallel I/O combined with the use of PVFS outperforms the use of NFS and both master-slave solutions.

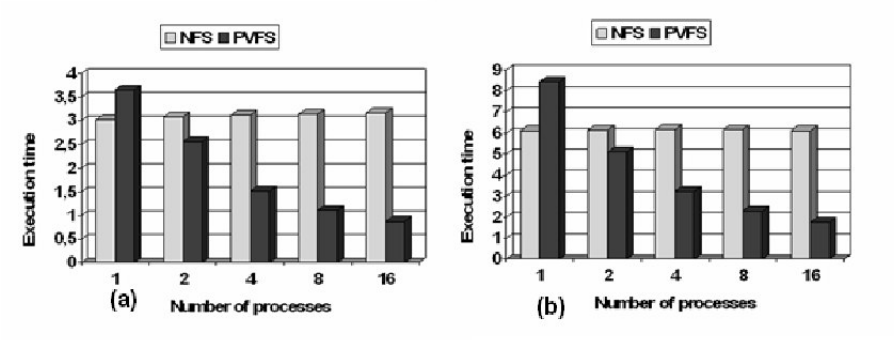


Figure 5. The execution time (in seconds) of the I/O operations using the MPI2 functionalities on the XT data set (499.5 MB) (a) and the FC data set (867 MB) (b)

It is mainly due to a combination of the two-phase I/O strategy adopted by the collective I/O operations, and the striping of the data among multiple disks performed by PVFS. In fact through the collective I/O operations, we significantly reduce the number of I/O requests that would otherwise result in many small non-contiguous I/O operations. By using the MPI derived data-types, the file data is seen as a 3D array, and the part requested by each process as a sub-array. When all processes perform a collective I/O operation, a large chunk of contiguous data file is accessed. The ROMIO implementation on PVFS2 is optimized to efficiently fit the I/O pattern of the application with the disk file striping. Thus the possible mismatches between the I/O pattern of the application and the physical storage patterns in file are minimized.

In Figure 5, we can see that MPI 2 and PVFS scales well with the number of processes, since in this case we effectively exploit data access to multiple disks. Instead it does not happen using NFS, since NFS was not designed for parallel applications that require concurrent file access to large chunk of files. Therefore as the number of processors and the file size increase, the use of NFS leads to an important bottleneck due to the serialization of the I/O operations. Actually the execution time is almost constant, although the number of processes increases.

4 Conclusions

The efficient acquisition and production of data is a major issue for parallel applications; this is of particular importance in the imaging community where these aspects received only a little attention.

In this paper we present how we tackled the problem in the design of $\mathrm{PIMA(GE)}^2$ Lib. Our solution is based on the use of the more sophisticated I/O routines provided by MPI2. This work represents an experimental study about the adoption of a parallel I/O, obtained comparing its performance with that achieved by the classical master-slave approach. The results demonstrated the effectiveness of parallel I/O strategy. In this manner we improve the overall performance of an application developed using the library. Furthermore, at the best of our knowledge, $\mathrm{PIMA(GE)}^2$ Lib is one of the few examples of image processing

library where a parallel I/O is strategy is adopted.

Acknowledgements

This work has been supported by the regional program of innovative actions PRAI-FESR Liguria.

References

1. J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon and A. White, *The Sourcebook of Parallel Computing*, Morgan Kaufmann, (2002).
2. R. Thakur, W. Gropp and E. Lusk, *Optimizing Noncontiguous Accesses in MPI-IO*, Parallel Computing, **28**, 83–105, (2002).
3. Parallel Virtual File System Version 2, see http://www.pvfs.org
4. Sun Microsystems Inc., *NFS: Network File System Version 3 Protocol Specification*, Sun Microsystems Inc., Mountain View, CA, (1993).
5. A. Ching, K. Coloma, J. Li and A. Choudhary, *High-performance techniques for parallel I/O*. In Handbook of Parallel Computing: Models, Algorithms, and Applications, CRC Press, (2007).
6. H. Yu and K.L. Ma, *A study of I/O methods for parallel visualization of large-scale data*, Parallel Computing, **31**, 167–183, (2005).
7. I.F. Sbalzarini, J.H.Walther, B. Polasek, P. Chatelain, M. Bergdorf, S.E. Hieber, E.M. Kotsalis, and P. Koumoutsakos, *A Software Framework for the Portable Parallelization of Particle-Mesh Simulations*, in Proceedings of Euro-Par 2006, LNCS **4128**, 730–739, (Springer, 2006).
8. Y. Zhu, H. Jiang, X. Qin and D. Swanson, *A Case Study of Parallel I/O for Biological sequence Search on Linux Clusters*, in: IEEE Proceeding of Cluster 2003, 308–315, (2003).
9. J. Li, W. Liao, A. Choudhary and V. Taylor, *I/O analysis and optimization for an AMR cosmology application*, in IEEE Proc. Cluster 2002, 119–126, (2002).
10. F. J. Seinstra and D. Koelma, *User transparency: a fully sequential programming model for efficient data parallel image processing*, Concurrency and Computation: Practice & Experience, **16**, 611–644, (2004).
11. J. M. Squyres, A. Lumsdaine and R. L. Stevenson, *A toolkit for parallel image processing*, in Parallel and Distributed Methods for Image Processing II, Proc. SPIE, vol **3452**, (1998).
12. P. Oliveira and H. du Buf, *SPMD image processing on Beowulf clusters: directives and libraries*, in: IEEE Proc. 7th IPDPS, (2003).
13. C. Nicolescu, P. Jonker, *EASY-PIPE an easy to use parallel image processing environment based on algorithmic skeletons*, in IEEE Proc. 15th IPDPS, (2001).
14. G. Ritter and J.Wilson, *Handbook of Computer Vision Algorithms in Image Algebra*, 2nd edition. (CRC Press, 2001).
15. A. Clematis, D. D'Agostino and A. Galizia, *An Object Interface for Interoperability of Image Processing Parallel Library in a Distributed Environment*, in Proc. ICIAP 2005, LNCS 3617, 584-591, (Springer, 2005).
16. ROMIO home page, http://www.mcs.anl.gov/romio

Fluid and Magnetohydrodynamics Simulation

Parallelisation of a Geothermal Simulation Package: A Case Study on Four Multicore Architectures

Andreas Wolf[1], **Volker Rath**[2], **and H. Martin Bücker**[1]

[1] Institute for Scientific Computing
RWTH Aachen University, D–52056 Aachen, Germany
E-mail: {wolf, buecker}@sc.rwth-aachen.de

[2] Institute for Applied Geophysics
RWTH Aachen University, D–52056 Aachen, Germany
E-mail: v.rath@geophysik.rwth-aachen.de

An OpenMP-based approach to parallelise a geothermal simulation package is investigated. Like its predecessor SHEMAT, the new simulation software is capable of solving the coupled transient equations for groundwater flow and heat transport. The overall parallelisation strategy consists of an incremental approach to parallelise the most time-consuming tasks: the assembly and the solution of two coupled systems of linear equations. We compare the parallel performance on recent dual- and quad-core architectures based on various Xeon and Opteron processors. The case study demonstrates that it is possible to obtain good speedups for a moderate number of threads with only modest human effort.

1 Introduction

Small and medium-sized enterprises consider the transition from serial to parallel computing as extremely difficult because of the large human effort required for this move. Since hardware cost of future multicore processors[1] are expected to be modest, it is particularly important to reduce human effort. By enabling an incremental approach to shared-memory programming, the OpenMP[2,3] programming paradigm promises to offer a smooth transition from serial to multicore architectures. In this study, we assess the performance of such an OpenMP approach applied to a specific geothermal simulation package that is used by several academic institutions and different small enterprises in geotechnical engineering.

The organisation of this note is as follows. The geothermal simulation software is described in Sect. 2. The approach to its OpenMP parallelisation is presented in Sect. 3. In Sect. 4, we compare the parallel performance on four recent multicore platforms, using up to eight threads.

2 The Geothermal Simulation Package

A new geothermal simulation package is currently being developed at the Institute for Applied Geophysics, RWTH Aachen University. It is based on its predecessor SHEMAT[4] (Simulator for HEat and MAss Transport). SHEMAT as well as the new software are capable of solving the coupled transient equations for groundwater flow, heat transport, and the transport of reactive solutes at high temperatures in three space dimensions. Throughout this study, we consider a simplified test case involving steady-state flow and heat transport. Also, the chemically reactive transport is not taken into account. In particular, the

groundwater flow equation is reformulated as

$$\nabla \cdot \left(\frac{\rho_f \, g}{\mu_f} \mathbf{k} \cdot \nabla h \right) + Q = 0, \tag{2.1}$$

using the hydraulic head h as the primary variable and denoting the source term by Q. In this equation, the symbol $\mathbf{k}$ denotes hydraulic permeability, ρ_f and μ_f are density and dynamic viscosity of the pore fluid, respectively, and g denotes the gravity constant. Both ρ_f and μ_f depend significantly on temperature T and pore water pressure P. The latter has to be calculated from the distribution of head and the depth z according to the definition given by de Marsily[5]

$$P(z,h) = P(0) + \int_0^z \rho_f(\tilde{z})\, g(h-\tilde{z}) d\tilde{z}\,, \tag{2.2}$$

where $P(0) \approx 10^5$ Pa is the pressure at the surface.

The temperature is controlled by the steady state heat transport equation

$$\nabla \cdot (\lambda_e \nabla T) - \rho_f \, c_f \, \mathbf{v} \cdot \nabla T + A = 0, \tag{2.3}$$

which comprises a conductive term, a source A similar to the head equation, and an additional advective transport term containing the filtration velocity $\mathbf{v}$, where c_f is the fluid specific heat capacity and λ_e is the effective thermal conductivity of the rock. The filtration velocity can be calculated from hydraulic head by Darcy's law,[5]

$$\mathbf{v} = \frac{\rho_f g}{\mu_f} \, \mathbf{k} \cdot \nabla h\,. \tag{2.4}$$

Thermal conductivities of the rock matrix and fluid as well as the other fluid properties, ρ_f, μ_f and c_f, and consequently the filtration velocity in (2.4), depend on temperature and pressure. Together with (2.2) these functions constitute a nonlinear coupling between (2.1) and (2.3). To ease the adaption to the highly variable conditions in the subsurface, the nonlinearities resulting from fluid and rock properties are formulated in a modular way.

Numerically, (2.1) and (2.3) are discretized by standard finite-difference techniques, as described in the SHEMAT documentation.[4] With the appropriate boundary conditions incorporated, this leads to the discrete versions of the coupled basic equations

$$\begin{aligned} \mathbf{M}^h(T,h)\,\mathbf{h} &= \mathbf{b}^h(T,h), \\ \mathbf{M}^T(T,h)\,\mathbf{T} &= \mathbf{b}^T(T,h), \end{aligned} \tag{2.5}$$

where the coefficient matrices $\mathbf{M}^h$ and $\mathbf{M}^T$ of these linear systems as well as the right-hand sides $\mathbf{b}^h$ and $\mathbf{b}^T$ depend on the solutions, head and temperature, respectively. This nonlinear system of equations is solved by a simple alternating fixed-point iteration.[6] In each step of the iterative solution of (2.5), two large sparse systems of linear equations have to be solved for $\mathbf{h}$ and $\mathbf{T}$ in subsequent phases. The current solution $\mathbf{h}$ is needed in setting up the coefficient matrix $\mathbf{M}^T$. Similarly, the current solution $\mathbf{T}$ is required to build up $\mathbf{M}^h$ in the next iteration.

3 Parallelisation Strategy

In contrast to current computational models simulated on today's serial processors, future multicore architectures promise the potential to consider even more realistic geological

models typically require larger problem sizes, due to finer discretisation and more sophisticated presentation of physical processes. Hence, parallelisation of existing and new programs becomes an issue of increasing importance. In the following, we describe the parallelisation of the two most time- and memory-intensive tasks involved in the solution of (2.5). More precisely, the aim is to parallelise (a) the assembly of the coefficient matrices and (b) the solution of the resulting large sparse systems of linear equations. Both tasks dominate the overall runtime of the simulation with over 90%.

For parallelising a matrix assembly, the corresponding routines are modified such that all the rows of the matrices $\mathbf{M}^T$ and $\mathbf{M}^h$ can be computed independently. This parallelisation is possible because the values of the matrix elements solely depend on values computed in the previous nonlinear iteration or geometrical and physical properties that do not change during the solution of the nonlinear system. The assembly of a coefficient matrix is dominated by independent local computations, allowing to neglect cache affinity effects.

The two linear systems involved in (2.5) are solved by a parallel version of the biconjugate gradient method (BiCGStab)[7] for nonsymmetric systems. The key feature of this parallel version is the rearrangement of statements so as to minimise the number of synchronisation points (where all threads have to wait before continuing the execution of operations). This is accomplished by combining reduction operations for computing inner product-like operations during an iteration step of the BiCGStab algorithm. In contrast to methods like GMRES[8] that are based on long recurrences, BiCGStab falls into the class of Krylov methods using short recurrences[9]. That is, time and storage requirements of a single BiCGStab iteration are constant and do not vary with index of the iteration, enabling the solution of large-scale systems without the need to explicitly control memory consumption.

With the exception of matrix-vector products and preconditioning, all operations occurring within BiCGStab are vector-vector operations carried out by the BLAS.[10] The regular memory access of these operations allows the compiler to generate highly-efficient code. However, limits of the overall memory bandwidth can be approached, potentially leading to a dramatic reduction of the overall performance. The computational work on these vectors and their memory placement are statically assigned to the threads, taking into account non-uniform memory architectures where multiple processors can simultaneously access fast and directly-connected memory.

Currently, three preconditioners are implemented: diagonal scaling, SSOR, and ILU(0). The ILU(0) preconditioner has proven to be sufficient for the specific regularly-structured sparse linear systems under consideration arising from our types of model problems. The resulting linear systems usually reach convergence after a few iterations. A blocked version of this preconditioner based on the mesh geometry is implemented to set up as many computationally independent tasks as possible. That is, when solving the resulting triangular systems in ILU(0), blocking is used to not only exploit the availability of efficient data caches but also to increase the degree of parallelism.

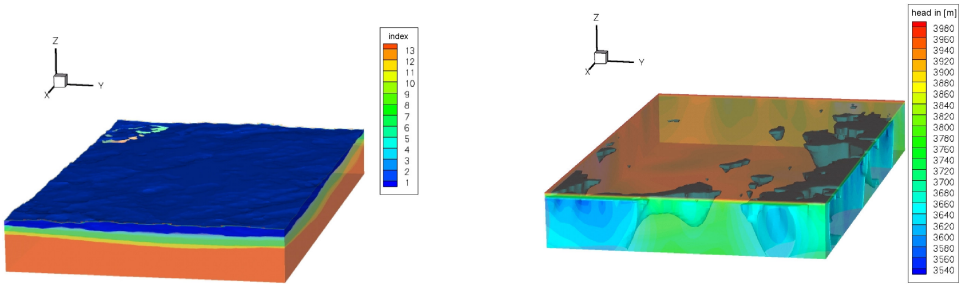
In the computation of a matrix-vector product, all threads reuse this blocked data layout, thus exploiting the carefully-designed data locality. For ILU(0) preconditioning, in order to avoid side effects caused by false sharing on caches and also in order to take advantage of the non-uniform memory access (NUMA), each thread works on its own private copy of the data blocks for which it is responsible. This approach thus mimics an MPI-style programming that would be used for distributed-memory architectures.

4 Parallel Performance on Multicore Systems

In this case study, we are interested in a conductive model of a synthetic sedimentary basin consisting of 13 zones with different rock properties depicted in Fig. 1(a). The coupled equations (2.1) and (2.3) are solved for temperature and hydraulic head. The head prescribed at the Earth's surface follows the topography. It is combined with a Dirichlet boundary condition of a constant temperature of $10°\mathrm{C}$. Moreover, a heat flow of $0.05\ \mathrm{W/m^2}$ is specified at the bottom. The resulting distribution of hydraulic head for this configuration is sketched in Fig. 1(b).

Throughout the numerical experiments, we examine two different scenarios on an $N \times N \times N$ grid. We consider a large test case with $N = 200$ equidistant grid points in each of the three dimensions for solving the model described above. Furthermore, a small test case with $N = 30$ solving an inverse problem related to the above forward model is used. Both test cases are configured to use the BiCGStab linear solver with ILU(0) preconditioning. Since the focus is on the parallel performance and not on the convergence of the numerical scheme, we agree upon the following procedure in all our performance experiments: For the case $N = 200$, a single nonlinear iteration is carried out avoiding long runtimes needed for convergence. The smallest execution time for that iteration out of four identical runs is taken. For the case $N = 30$, the nonlinear iteration is carried out until convergence and the best runtime out of ten runs is reported. Except for certain outliers, most of the measurements for a given test case, platform and number of threads differ in only 10% of the runtime.

We compare the performance of the OpenMP-parallelised code on four recent multicore platforms whose characteristics are given in Table 1. Each system has a certain number, p, of processors with a specific number, c, of cores on each processor. The Clovertown is a quad-core system, while the Woodcrest is based on dual-core processors. Both systems offer uniform memory access (UMA) and consist of Xeon processors using shared caches, meaning that two cores can share the same 4 MByte L2-cache offering the potential to reduce synchronisation time. Like the Woodcrest, the two Opteron platforms also use dual-core processors. However, their access to memory is non-uniform. The Opteron 280 consists of two processors whereas the Opteron 885 is a bigger variant having eight processors. The performance experiments on the Opteron 885 are carried out with a high back-



(a) (b)

Figure 1. The computational domain consists of 13 different zones shown in (a). The hydraulic head for this domain using $N = 200$ is given in (b).

System	Processor	$p \times c$	Memory	Clock Rate
Power Edge 1950	Clovertown, Xeon 5355	2×4	UMA	2.66 GHz
Power Edge 1950	Woodcrest, Xeon 5160	2×2	UMA	3.00 GHz
D20z-M4	Opteron 280	2×2	NUMA	2.40 GHz
Sun Fire X4600	Opteron 885	8×2	NUMA	2.60 GHz

Table 1. Characteristics of the four multicore architectures with p processors each having c cores.

ground load. Thus, the Opteron 885 is considered to be a non-competitive configuration that is included to give an impression of an application in a practical multi-user environment. The experiments on all other platforms are carried out without any background load on a dedicated system.

In an attempt to compare the Intel-based systems Clovertown and Woodcrest with the two AMD-based Opteron systems in an unbiased way, we considered the Linux Intel compiler 10.0.023 with special support for Xeon processors as well as the PGI-6.1 and GCC-4.1 compilers with optimisation flags for Opteron processors activated. Since all resulting binaries performed similarly with respect to serial runtime, all experiments are carried out with an executable built with the Intel compiler using the flags "-O3 -axT -openmp -fpp2." The special flag "-axT" generates an executable capable of running different program paths, depending on which processor architecture is found. Each program path is optimised for a specific set of processor features. For instance, it includes special commands to use the vectorisation unit of the processor. To ensure comparability, we avoid using precompiled external performance libraries for the BLAS. Rather, explicit Fortran sources of BLAS operations are compiled with the same flags as given above and linked with the main program.

For performance measurements with n threads, it is useful to specify a subset of n cores on which the program should run in parallel. This thread binding is supported by the Linux operating system in several ways. We used the taskset tool to specify a set of cores on which threads are allowed to run. In this case, the operating system often did not fully utilise the core set, i.e., more than one thread was executed on one core at the same time, while other cores of the set were idle. An alternative method would have been direct thread binding where each thread binds itself on the first free core. We did not use direct thread binding because of the lack of a standardised support within OpenMP. However, without explicit thread binding, the operating system succeeded to manage n threads on n different processor cores, though changes in the affiliation of threads and cores occurred occasionally. We decided to neglect these effects, and use the default unbound setting for all experiments.

The speedup for the case $N = 30$ is shown in Fig. 2 using the four different multicore architectures. Here, the execution time of the parallel program with one thread is taken as the reference point on each platform. As shown in this figure, an almost linear speedup on up to five threads is achieved for all architectures. Note that the time-intensive parts of the program can load most of the data into cache, resulting in good overall cache re-use. For six and more threads that are available only on Clovertown and Opteron 885, no further

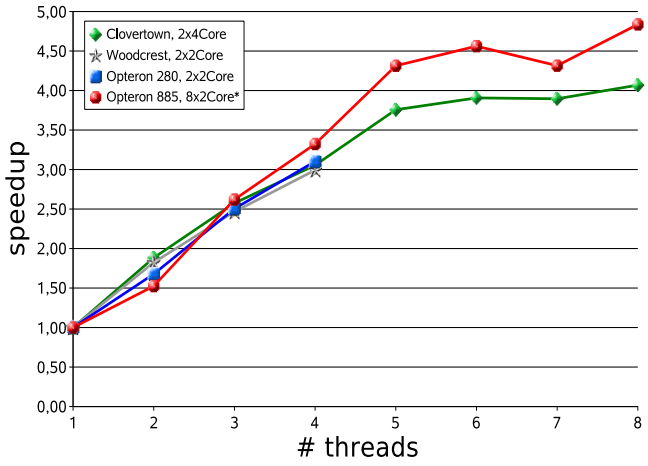


Figure 2. Speedup versus number of threads for the case $N = 30$. *There is background load on the Opteron 885.

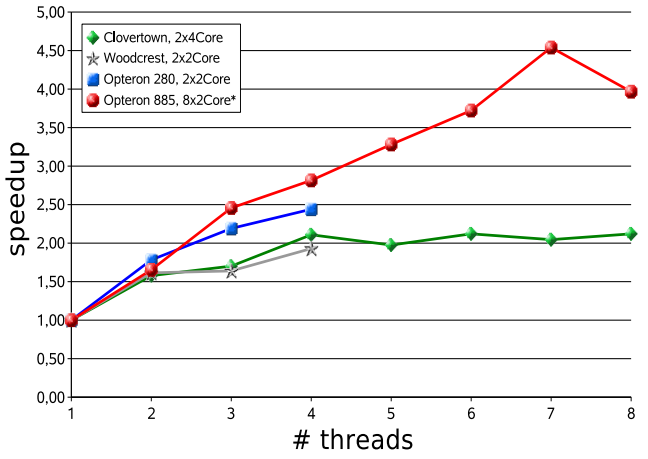


Figure 3. Speedup versus number of threads for the case $N = 200$ based on the timings given in Table 2. *There is background load on the Opteron 885.

increase of the speedup is observed. The reason for this behaviour is the relatively small number of blocks that can be handled independently in the ILU(0) preconditioner, leading to a load imbalance for small problem sizes. Overall, the results using up to five threads demonstrate a significant improvement over a serial execution for the case $N = 30$.

For the larger case with $N = 200$ whose speedup is depicted in Fig. 3, we can expect a saturation of the speedup already for the two UMA systems, Clovertown and Woodcrest. The increase of the speedup when increasing the number of threads from one to three

shows that the memory bandwidth between memory and processor is still high enough to fetch and process data. However, the limit of the memory bandwidth is approached when using four or more threads. This leads to a full saturation with a speedup of 2.1 on the Clovertown. The corresponding execution times for the case $N = 200$ are shown in Table 2. Having an estimate of 2.3 TByte for the data traffic from memory to processors at hand, we conclude from the runtime of 409.2 sec with four threads on the Clovertown a corresponding data throughput of 5.7 GByte/sec. This memory bandwidth is close to the practical hardware limit of the main memory that was also observed in other applications at the Center for Computing and Communication, RWTH Aachen University.

In contrast, the two NUMA Opteron systems provide a direct memory connection for each processor. That is, when adding a thread running on an additional processor, the memory bandwidth is also increased. This can result in a better scaling and the improved speedups are shown in Fig. 3. Here, only the Opteron 885 system has the ability to scale well with a speedup up to 4.5 on seven threads similar to 4.8 for $N = 30$ on eight threads given in Fig. 2.

Although, as given in Table 1, the clock rate of the Opteron 885 is higher than the one of the Opteron 280, the execution time using a single thread is larger on the Opteron 885. More precisely, its execution time of 1413.6 sec is approximately 40% larger than 1027.8 sec on the Opteron 280. The background load on the Opteron 885 contributes to this phenomenon. Therefore, it is also not reasonable to consider more than eight threads on this platform even though it has 16 cores.

# Threads	Clovertown	Woodcrest	Opteron 280	Opteron 885*
1	862.8	758.4	1027.8	1413.6
2	546.6	469.8	576.6	855.0
3	507.0	462.6	469.2	575.4
4	409.2	393.0	421.2	502.2
5	436.8	—	—	430.8
6	406.8	—	—	379.8
7	421.8	—	—	311.4
8	406.8	—	—	356.4

Table 2. Execution times in seconds for the case $N = 200$. *There is background load on the Opteron 885.

5 Concluding Remark

The OpenMP programming paradigm is used to parallelise a geothermal simulation package in an incremental fashion. More precisely, the assembly and the solution of the resulting systems of linear equations are parallelised, while the remaining parts of the simulation remain serial. The performance results on four recent multicore architectures (Clovertown, Woodcrest, and two Opteron-based platforms) demonstrate that it is possible to obtain moderate to good speedups using up to four threads with only modest human effort. We

believe these results to be encouraging. However, algorithms getting performance out of these multicore architectures have to be designed for efficient cache use, while synchronisation is of less importance. Memory throughput limitations are also still an issue. Finally, with the experience limited to today's multicore processors with a small number of cores, it will be crucial to address the problem of scalability to larger number of cores for future multicore architectures.

Acknowledgements

We would like to thank Dieter an Mey, Christian Terboven and Samuel Sarholz of the Center for Computing and Communication at RWTH Aachen University for stimulating discussions and access to the four multicore architectures.

References

1. J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*, (Morgan Kaufmann, San Francisco, 4th edition, 2007).
2. OpenMP Architecture Review Board, *OpenMP Application Program Interface, Version 2.5*, (2005).
3. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*, (Morgan Kaufmann Publishers, San Francisco, 2001).
4. C. Clauser, ed., *Numerical Simulation of Reactive Flow in Hot Aquifers. SHEMAT and PROCESSING SHEMAT*, (Springer, New York, 2003).
5. G. de Marsily, *Quantitative Hydrogeology*, (Academic Press, 1986).
6. P. S. Huyakorn and G. F. Pinder, *Computational Methods in Subsurface Flow*, (Academic Press, 1983).
7. H. A. van der Vorst, *BI-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, **13(2)**, 631–644, (1992).
8. Y. Saad and M. H. Schulz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM Journal on Scientific and Statistical Computing, **7(3)**, 856–869, (1986).
9. Y. Saad, *Iterative Methods for Sparse Linear Systems*, (SIAM, Philadelphia, 2nd edition, 2003).
10. C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, *Basic linear algebra subprograms for Fortran usage*, ACM Trans. Math. Soft., **5**, 308–323, (1979).

A Lattice Gas Cellular Automata Simulator on the Cell Broadband Engine^{TM}

Yusuke Arai, Ryo Sawai, Yoshiki Yamaguchi
Tsutomu Maruyama, and Moritoshi Yasunaga

Graduate School of Systems and Information Engineering
University of Tsukuba Ten-ou-dai 1-1-1 Tsukuba, Ibaraki, 305-8573, Japan
E-mail: {arai, yoshiki}@islab.cs.tsukuba.ac.jp

The Cell Broadband Engine (CBE) was developed as a high-performance multimedia processing environment. A CBE is a heterogeneous multicore processor that incorporates the PowerPC Processor Element (PPE) and Synergistic Processor Elements (SPEs). Each SPE is a special purpose RISC processor with 128-bit SIMD capability. In this paper, we describe a new computational method for simulating fluid dynamics on a CBE using the FHP-Ⅲ model. The approach is suitable for simulating very complicated shapes. In our implementation, the speedup of the FHP-Ⅲ model is about 3.2 times compared with an Intel Core2 Duo E6600 running at 2.4 GHz when there are about 11 millions lattice sites.

1 Introduction

A considerable number of studies have been conducted on the Cellular Automata (CA) that John Von Neumann and Stan Ulam proposed[1]. The CAs have been widely used for modeling different physical systems, such as systems with an emphasis on spin systems and pattern formations in reaction-diffusion systems. Here, the Lattice Gas Automata (LGA) that are a class of the CAs designed for simulating fluid dynamics have been one of the central models.

Within the LGA, a transition function is broken down into two parts: collision stage and propagation stage. In the collision stage, particle collisions are handled by a collision rule, which broadly can be classified into two groups: the Hardy, Pazzis and Pomeau (HPP) model[2], and, the Frisch, Hasslacher, and Pomeau (FHP) model[3,4]. The FHP model is used in this paper. In the propagation stage, the particles move from one lattice site to another adjacent site. Each of the lattice sites can be computed from the values of its own lattice site and adjacent lattice sites. The LGA has computational locality and therefore many approaches with parallel and distributed systems have been proposed[5,6].

The Cell Broadband Engine (CBE) is the multicore processor developed by Sony Computer Entertainment Inc./Sony, TOSHIBA, and IBM[7,8]. As shown in Fig.1, the CBE has nine processor cores, one PowerPC Processor Element (PPE), and eight Synergistic Processor Elements (SPEs) connected by the Element Interconnect Bus (EIB).

The CBE's peek performance is 204.8 GFlops (single precision) or 14.6 GFlops (double precision) running at 3.2 GHz. This is about 11 times better than the performance of an Intel Core2 Duo E6600 processor which achieves 19.2 GFlops (single precision) running at 2.4 GHz.

In this paper, we implement a particle simulation using the FHP-Ⅲ model on a Cell Reference Set (CRS)[10,11]. The Cell Reference Set produced by Toshiba Co. Ltd. includes one CBE, Toshiba's own boards and a cooling system as shown in Figs. 2, 3, and 4.

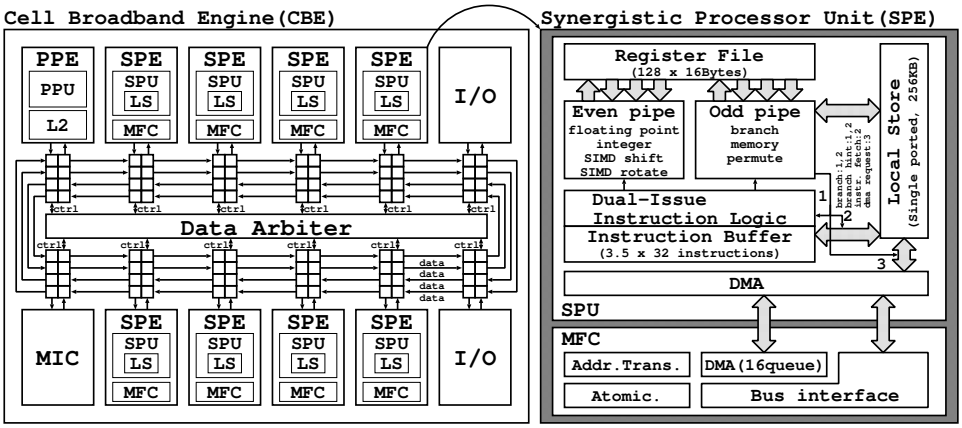


Figure 1. Hardware Block Diagram of the Cell Broadband Engine (CBE)

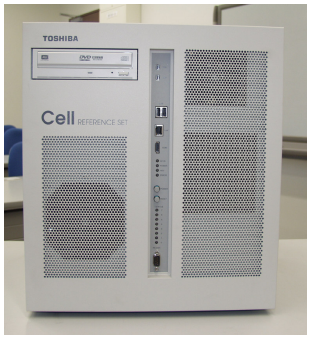


Figure 2. Cell Reference Set (CRS)

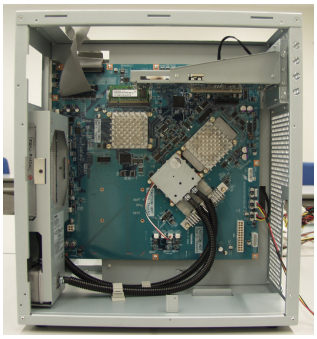


Figure 3. The inside of CRS

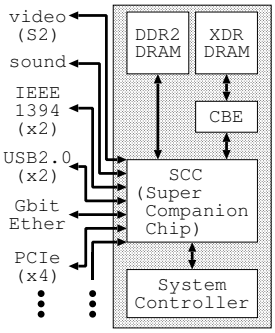


Figure 4. CRS Block Diagram

This paper organized into four sections. In Section 2, we describe the architecture of the CBE. Section 3 reports on the implementation of the FHP-Ⅲ LGA model on the Cell Reference Set with one CBE. Section 4 summarizes our findings.

2 Cell Broadband Engine (CBE)

The overview of the CBE is shown in Fig.1. The CBE is composed of one PowerPC Processor Element (PPE), eight Synergistic Processor Elements (SPEs), one Memory Interface Controller (MIC), I/O, and an Element Interconnect Bus (EIB) as shown in Fig.1. The PPE is a 64 bit Power PC architecture which can run a 64 and 32 bit operating system and applications, and manages the SPEs. The one PPE and eight SPEs are connected with the EIB.

Each SPE has a Synergistic Processor Unit (SPU) and a memory flow controller (MFC). It does not have branch prediction hardware but it allows for branch hint instructions and has a high-performance floating point unit[12, 13]. Therefore, the computational per-

formance of one SPU (3.2 GHz) is 25.6 GFlops (single precision) and 1.83 GFlops (double precision), and high performance has been reported in scientific applications[9]. Each SPU has a 128 bit SIMD processing unit, 128 128 bit registers, and a 256 KB local storage. SPE programs are allocated in the local storage. Direct Memory Access (DMA) is used for the data transfer among the local storage, other SPEs, a PPE, and external memories through the EIB.

3 Implementation of Lattice Gas Automata

3.1 FHP Model

The FHP model is designed for fluid analysis[3,4]. The lattice structure is shown in Fig. 5. One lattice site is connected to six neighbour sites and includes up to seven particles. The particles are divided into two groups according to particle velocity. Six particles and one particle have unit velocity and no velocity, respectively (Fig. 5). The unit velocity, $\mathbf{c}^i$, is obtained by the following equation, where i (=1$\sim$6) is a moving direction.

$$\mathbf{c}^i = \left(cos\frac{(5-i)\pi}{3}, sin\frac{(5-i)\pi}{3} \right) \tag{3.1}$$

Suppose that $\mathbf{x}_k$=(x_k,y_k) is the k-th lattice coordinate, a state on the k-th lattice site at time t ($\mathbf{n}_k(n_k^0, n_k^1, .., n_k^6)$) is described by

$$n_k^i(\mathbf{x}_k + \mathbf{c}_k^i, t+1) = n_k^i(\mathbf{x}_k, t) + \Delta_k^i\left[\mathbf{n}_k(\mathbf{x}_k, t)\right] \tag{3.2}$$

where the function, Δ_k^i, is a variation of $n_k^i(\mathbf{x}_k, t)$ by the collision.

In this paper, we consider an implementation called the FHP-Ⅲ with all 76 collision rules shown in Fig. 6. Each group lists a collection of states that can be in existence before and after a collision with successor states chosen randomly among the alternatives.

3.2 Region Splitting Method on CBE

In this paper, we simulated the FHP-Ⅲ LGA model on a 3392×3392 two-dimensional space with 11 million lattice sites. The size of the LGA is too large for the local storage of SPEs in the CBE because the total of a local storage is only 1,792 KB, sufficient only for up to 1354×1354 lattice sites.

We consider the following issues:

(A) computational procedure for a single SPE,

(B) data organization for the FHP-Ⅲ, and

(C) parallel processing arrangement for a whole simulation space.

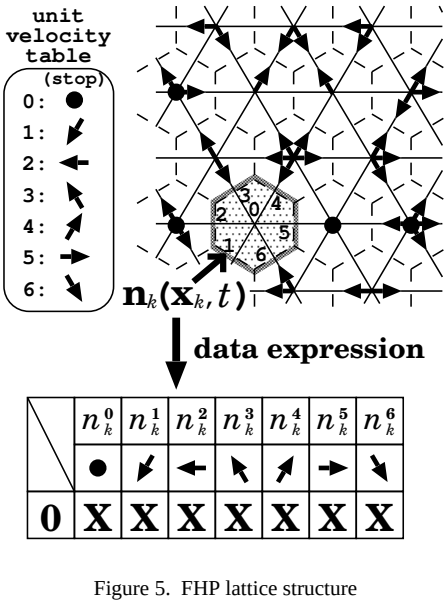


Figure 5. FHP lattice structure

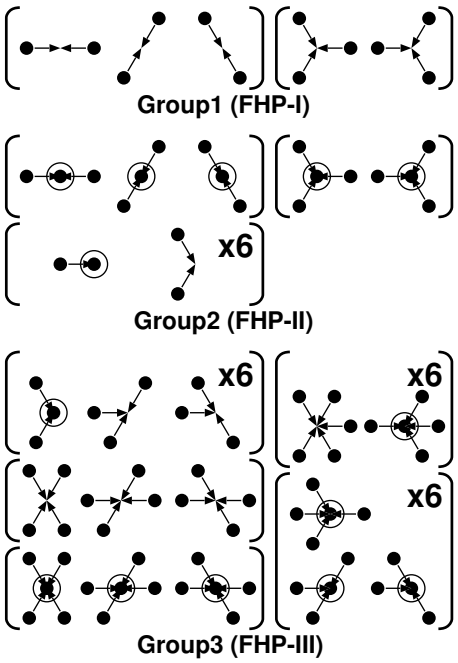


Figure 6. FHP collision rules

Concerning (A), the LGA model is very simple and its computation requires only particle migrations and collisions. It alternates between migration stages and collision stages until the whole simulation finishes. In this procedure, pseudo random numbers are frequently generated and the computational effort is not negligible. We have adopted the combined Tausworthe method[14] and therefore can also reduce the program size on each local storage.

Concerning (B), at each lattice site, a collision among particles happens based on Fig. 6. We can assign a single bit to indicate whether there is a particle or not on each site which is depicted as a shaded hexagon in Fig. 5, and one site can be stored in one byte as shown by the bottom of Fig. 5. Consequently, one byte is a computing unit in our target application. In our implementation, as shown by Fig. 7, we put together 4 procedures of 76 collusion rules in Fig. 6. This enables us to use no branch instruction in collision computation and, using SIMD instructions[15], we achieve high performance with the CBE.

The size of our simulation space can be estimated at about 11 MB ($= 1\ Byte \times 3392 \times 3392$) which is too large for the local storage (256 KB) in a SPE. Hence, concerning (C), we must divide the simulation space to suit a CBE, and face the bottleneck between each SPE and external XDR DRAM (Fig. 4)[16]. DMA data transfers are required every some dozens of microseconds and they cause serious performance loss. For this reason, we consider the region splitting method shown in Fig. 8.

In Fig. 8, suppose that the whole simulation space is 14×14 lattice sites, and 6×6 lattice sites can be stored in a local storage in each SPE. At step 0, the top-left region **A** is loaded to a local storage. After loading **A**, the SPE starts its iteration for the region. This

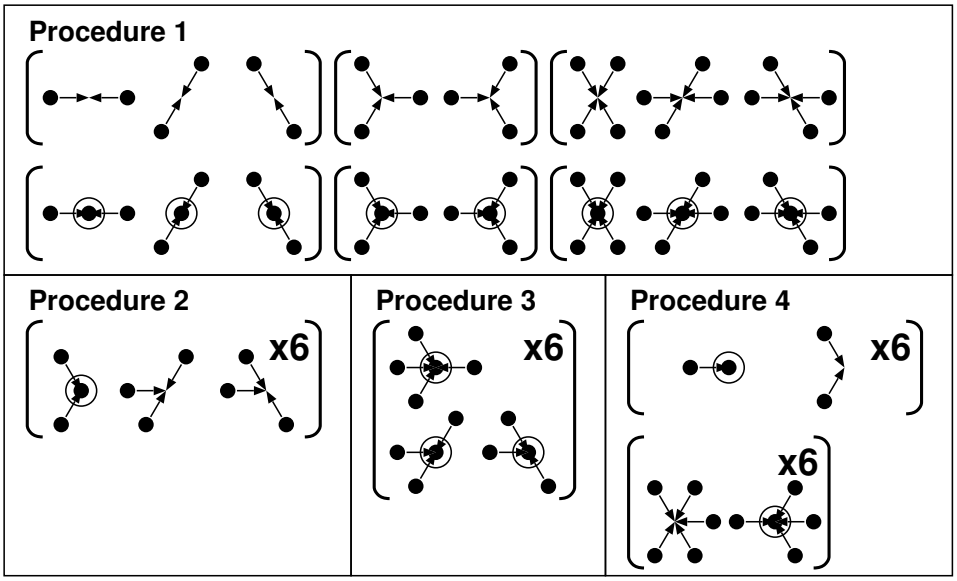


Figure 7. Collision rule integration for the redaction of computational procedure

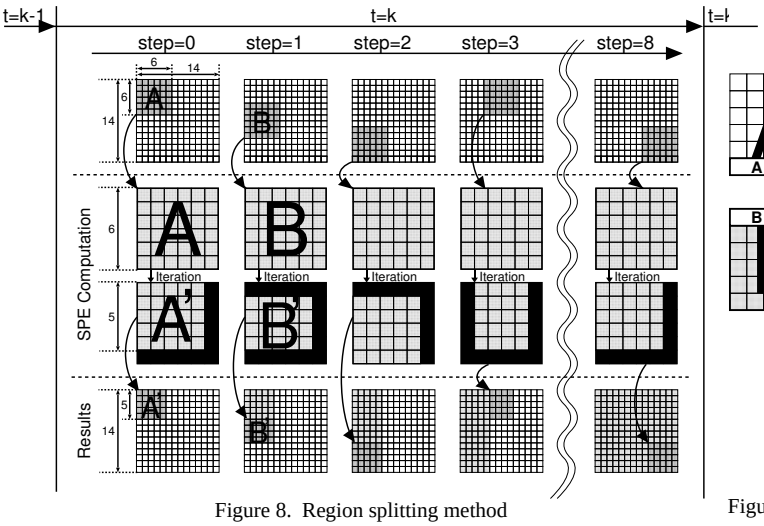


Figure 8. Region splitting method

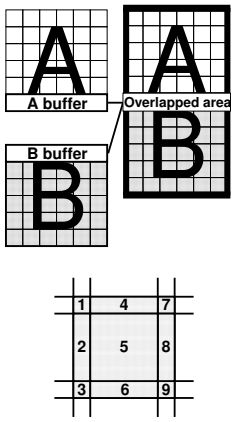


Figure 9. Overlapped area

block approach decreases the number of data transfers.

The LGA is one of stochastic fluid models and we must ensure the consistency of overlapping areas. Focusing on **A** and **B** in Fig. 8, **A buffer ($\mathbf{A}_{buf}$)** and **B buffer ($\mathbf{B}_{buf}$)** in Fig. 9 are the same region. Hence, we must use the same pseudo random number generators (RNGs) and their seeds for the computation because the result of $\mathbf{A}_{buf}$ is consistent with the result of $\mathbf{B}_{buf}$. Here, we prepare 3 RNGs, $\mathbf{A}_{RNG}$, $\mathbf{B}_{RNG}$ and $\mathbf{C}_{RNG}$ for these

Figure 10. Speedup gain

	Core2 Duo	CBE
clock (GHz)	2.4	3.2
# of core	2	7
L2 Cache	4 MB	7×256 KB
time(msec)	75.7	23.4
speedup	1	3.24

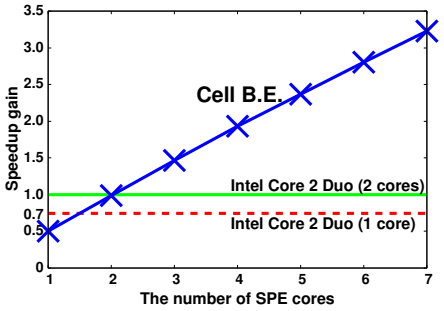


Figure 11. The relationship of speedup gains and no. of SPE cores

computation, which are used for the independent region of **A** and **B**, and the overlapped region, respectively. Therefore, $\mathbf{A}_{RNG}$ and $\mathbf{C}_{RNG}$ are used for SPE1, and $\mathbf{B}_{RNG}$ and $\mathbf{C}_{RNG}$ are used for SPE2 when **A** and **B** are computed by SPE1 and SPE2, respectively. As shown in the bottom of Fig. 9, in our current implementation, one region has normally 9 RNGs to maintain the consistency of surrounding overlapped area.

3.3 Evaluation of Results and Discussion

The computing time and speedup gains are shown in Table 10 using kernel 2.6.17-1 and Intel compiler 9.1 (option: -O3 -xT -parallel -static -openmp). According to Table 10, we achieve about 3.2 times speedup with the CBE compared to an Intel Core2 Duo E6600 running at 2.4 GHz. When we compare one SPE in the CBE to a single core in an Intel Core2 Duo E6600, an Intel Core2 Duo E6600 is about 1.5 faster than one SPE. But the performance is the reflection of cache size; an Intel Core2 Duo E6600 has 4 MB L2 Cache but the SPE has only 256 KB.

In our experimental result, a speedup ratio was calculated by dividing a computational time required by an Intel Core2 Duo E6600 using both cores by a computational time for a simulation on the CBE. A solid line in Fig. 11 shows the base line (=1.0) and a dashed line illustrates the speedup gain (= 0.74) by dividing dual cores' computational time by a single core's computational time. Figure 11 also illustrates the relationship between the speedup ratio and the number of SPE cores used in a simulation. The speedup ratio vs. the number of SPE cores is almost linear and is approximated by the following equation.

$$(Speedup\,ratio) \;=\; 0.45 \times \;(number\;of\;SPE\;cores) \;+ 0.08 \qquad (3.3)$$

As can be expected from Eq. 3.3, the CBE can keep high scalability. One of reasons is that the SPEs are connected by the EIB bus and the result can be read/written from/to the other SPEs. The other is that memory bandwidth of the CBE is larger than an Intel Core2 Duo E6600.

4 Conclusion

In this paper, we reported on the implementation of the LGA on the Cell Reference Set. The speedup ratio of a CBE running at 3.2 GHz, compared with an Intel Core2 Duo E6600

running at 2.4 GHz, was about 3.2 times. Thus, the CBE can achieve sufficient speedup even though the implementation of the LGA does not require floating point computation which is SPE's strong point. In addition, experimental results lead us to the conclusion that the CBE has high scalability in our target application.

Future tasks are to decrease the use of a local storage and the penalty of the instruction for branch on condition, and extract the part that can be parallelized dynamically and executed with SIMD operations. So, implementing the programs with SPEs divided into two groups, and utilizing the PPE not used at the time, we plan to categorize SPEs into the data control elements which extract the computable part and data operating elements which compute the data generated by the data control elements, examine the performance of CBE as one chip.

Acknowledgements

We would like to thank here TOSHIBA CORPORATION and TOSHIBA Semiconductor Company for their cooperation and support of this work. We are also indebted to Dr. Taiji, Deputy Project Director, Computational and Experimental Systems Biology Group in RIKEN GSC.

References

1. J. von Neumann, *The Theory of Self-Reproducing Automata*, A. W. Burks (ed), Univ. of Illinois Press, Urbana and London, (1966).
2. J. Hardy, et al., *Molecular dynamics of a classical lattice gas: transport properties and time correlation functions*, Phys. Rev. A, **13**, 1949–1961, (1976).
3. U. Frisch, B. Hasslacher and Y. Pomeau, *Lattice gas automata for the Navier-Stokes equation*, Phys. Rev. Lett., **56**, 1505–1508, (1986).
4. U. Frish, et al., *Lattice gas hydrodynamics in two and three dimensions*, Complex Systems, **1**, 649–707, (1987).
5. T. Toffoli and N. Margolus, *Cellular Automaton Machines – New Environment for Modeling*, (MIT Press, 1987).
6. G. S. Alamsi and A. Gottlieb, *Highly Parallel Computing*, (Benjamin-Cummings, 1994).
7. D. Pham, et al, *The design and implementation of a first-generation CELL processor*, in: IEEE International Solid-State Circuits Symposium, pp. 184–186, (2005).
8. J. Kahle, et al., *Introduction to the Cell multiprocessor*, IBM Journal of Research and Development, vol. **49**, No.4/5, pp. 589–604, (2005).
9. S. Williams, et.al., *The potential of the Cell processor for scientific computing*, in: ACM International Conference on Computing Frontiers, May (2006).
10. J. Amemiya, et. al., *Cell configuration of Cell reference set software*, Toshiba Review, **61**, 37–41, (2006).
11. S. Osawa, et. al., *Cell software development environment*, Toshiba Review, **61**, 47–51, (2006).
12. Y. Kurosawa, et. al., *Cell Broadband Engine Next-Generation Processor*, Toshiba Review, **61**, 9–15, (2006).

13. Sony, *Synergistic Processor Unit (SPU) Instruction Set Architecture*, Ver. 1.1, Jan. (2006).
14. M. Barel, *Fast hardware random number generator for the Tausworthe sequence*, in: 16th Annual Simulation Symposium, pp. 121–135, (1983).
15. Y. Arai, et al., *An approach for large-scale fluid simulation with a multi-core processor*, in: Symposium on Advanced Computing Systems and Infrastructures, Vol. **2007**, pp. 159–160, (2007).
16. R. Sawai, et al., *Relationship between SPEs and EIB on Cell Broadband Engine*, IEICE Technical Report, Vol. **106**, pp. 87–92, (2006).

Massively Parallel Simulations of Solar Flares and Plasma Turbulence

**Lukas Arnold, Christoph Beetz, Jürgen Dreher,
Holger Homann, Christian Schwarz, and Rainer Grauer**

Institute for Theoretical Physics I
Ruhr-University Bochum, Germany
E-mail: grauer@tp1.rub.de

Some of the outstanding problems in space- and astrophysical plasmasystems include solar flares and hydro- or magnetohydrodynamic turbulence (e.g. in the interstellar medium). Both fields demand for high resolution and thus numerical simulations need an efficient parallel implementation. We will describe the physics behind these problems and present the numerical frameworks for solving these problems on massive parallel computers.

1 Introduction

In this paper, we will describe numerical simulations of fundamental plasma phenomena like the evolution of solar flares and Lagrangian statistics of compressible and incompressible turbulent flows. In the following, we will first describe the framework *racoon* which is an adaptive mesh refinement framework for hyperbolic conservation laws. Simulations of solar flares and compressible turbulence are performed utilizing this framework. The incompressible turbulence simulations are based on the spectral solver LATU. After describing this solver and discussing performing issues for both *racoon* and LATU, we will present physical results obtained from the simulations.

2 The Framework *racoon*

All simulations using finite volume and finite differences are performed using the framework *racoon*[1]. *racoon* is a computational framework for the parallel, mesh-adaptive solution[2] of systems of hyperbolic conservation laws like the time-dependent Euler equations in compressible gas dynamics or Magneto-Hydrodynamics (MHD) and similar models in plasma physics. Local mesh refinement is realized by the recursive bisection of grid blocks along each spatial dimension, implemented numerical schemes include standard finite-differences as well as shock-capturing central schemes[3], both in connection with Runge-Kutta type integrators. Parallel execution is achieved through a configurable hybrid of POSIX-multithreading and MPI distribution with dynamic load balancing based on space-filling Hilbert curves[4] (see Fig. 1).

racoon also has the ability to advect tracer particles with the flow using the same parallelisation strategy as for the blocks. The main numerical work is spent in the interpolation routines from cell values to the actual particle positions.

Benchmarks on IBM p690 machines with 32 CPUs show that the hybrid concept in fact results in performance gain over a pure MPI parallelization, which, however requires a careful optimization of the multi-threaded implementation.

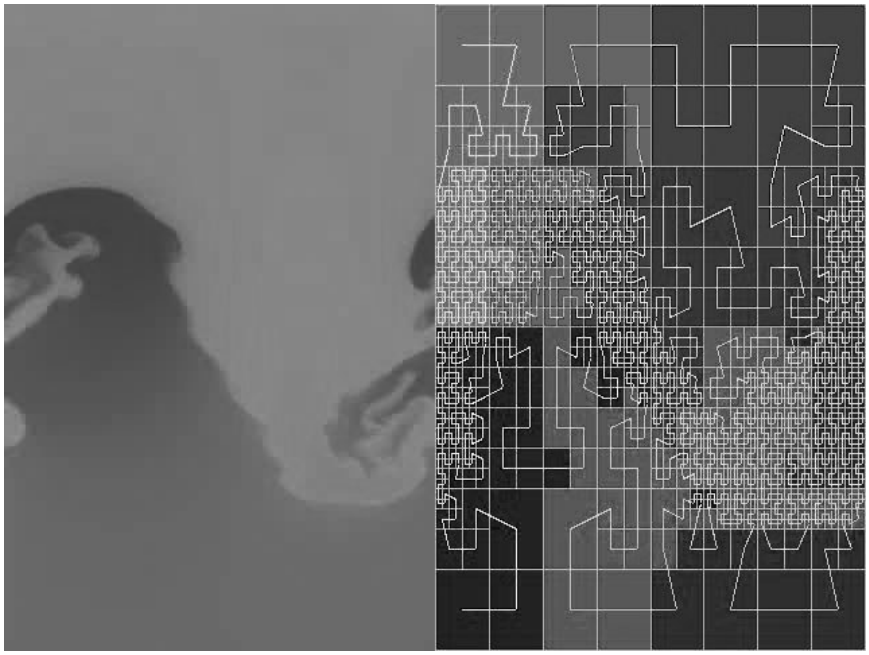


Figure 1. Simulation of a Rayleigh-Taylor instability. Load balancing is based on a Hilbert curve distribution (right side)

A key issue here is the efficient use of the CPU cache, which in the first place can be naturally obtained in AMR by the use of small grid block sizes that fit well into the cache. In addition, each thread in the current implementation creates its own effective data subspace consisting of a fixed subset of grid blocks and the block connectivity information, all allocated in the thread itself in order to achieve small CPU-memory distances in NUMA architectures. Block assignment to threads is based on the same space-filling curve algorithm that determines the distribution among processes, and which thereby tends to minimize not only inter-process communication but also inter-thread memory accesses with potential cache conflicts. To finally achieve the desired gain from multi-threading, the affinity between tasks and CPUs must be enforced manually by binding the working threads to individual CPUs.

For the MPI part of the communication, it turned out that the creation of fewer messages of moderate size (1 MB and below) by collecting the small inter-block messages which are addressed to the same target processor is favourable compared to mapping the typically small messages (one to few kB) between blocks directly to MPI messages, despite the fact that all MPI traffic is channelled through one thread in the message collection method. Here, the (de-) serialization of compound MPI messages can occur concurrently by many threads. These results indicate that the concurrent access to the MPI layer for the completion of many small-sized messages, even with multi-thread abilities, should be used carefully with respect to the overall performance. In the end, the hybrid concept proved to work satisfactory and resulted in floating point performances in the range of 7-10% of

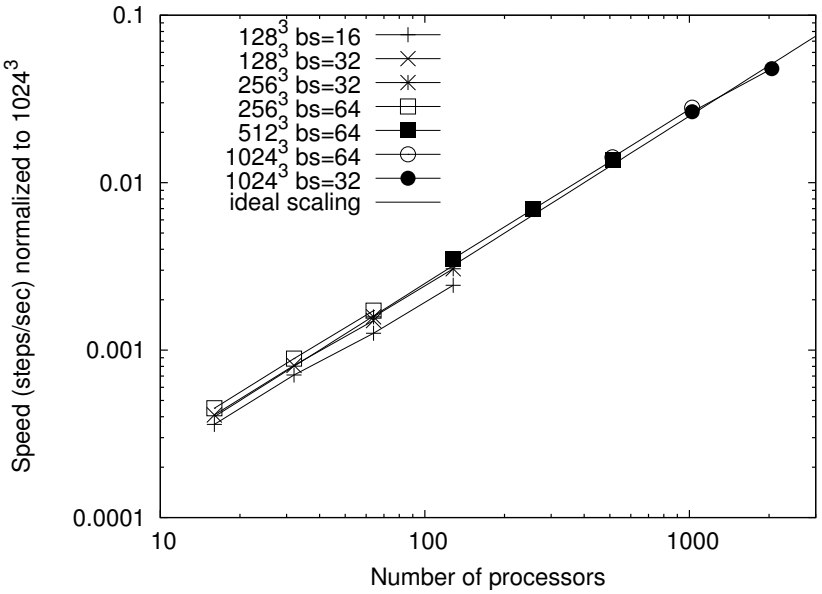


Figure 2. Strong scaling for different blocksizes (bs) varying from 16^3 to 64^3.

the theoretical peak performance on 64 processors for the described application. Naturally, there is still some room for further improvement, for example in connection with automatic estimates for the size of MPI compound messages. For practical use, further development of high-level interfaces for the control of task and memory affinity on high performance computers would be helpful, as the method of explicit CPU binding that was chosen here has the potential to conflict with the job dispatcher and load distribution algorithms in larger settings. One interesting initiative for IBM's platform is the VSRAC interface project (www.redbooks.ibm.com/redpapers/pdfs/redp3932.pdf), that might be extended in the near future to allow a thread-level control in addition to its current process-level control (see also R. Rabenseifer[5] for hybrid parallel programming).

On very massive parallel machines like the IBM BlueGene, a MPI only version is used. Scaling tests on the BlueGene JUBL at the FZ Jülich reveal linear scaling up to 2048 processors (see Fig. 2).

3 LATU: An Incompressible MHD Spectral Solver with Passive Tracer Particles

The numerical simulations of incompressible turbulence are performed using a pseudo-spectral code. The underlying equations are treated in Fourier-space, while convolutions arising from non-linear terms are computed in real space. A Fast-Fourier-Transformation (FFT) is used to switch between this two spaces. This scheme is very accurate and produces negligible numerical dissipation. The code is written in C++ and parallelizes efficiently. The time scheme is a Runge-Kutta third order. The inter-process communication uses the

Message Passing Interface (MPI) and the FFT are performed using the portable FFTW library or the San Diego P3DFFT routines using the IBM ESSL library. Simulations with 1024^3 collocation points were performed using up to 512 CPUs on the IBM p690 machine of the John von Neumann-Institut in Jülich. Results of preliminary scaling tests on BlueGene using up to 16384 processors and the P3DFFT routines are depicted in Fig. 3.

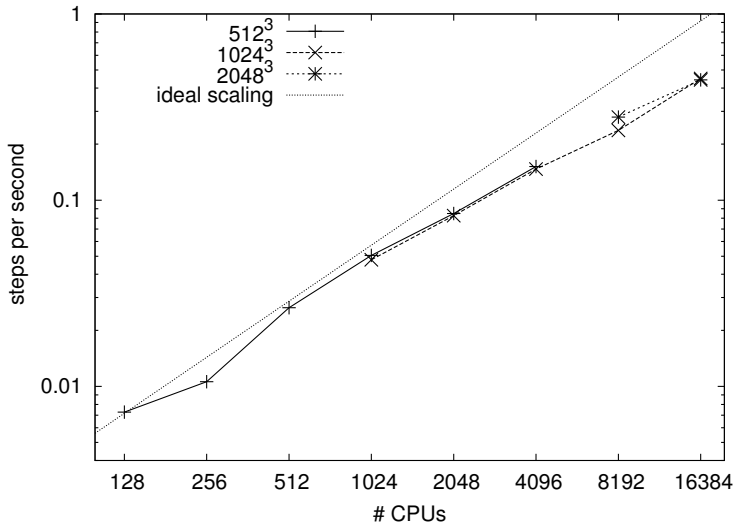


Figure 3. Mixture of strong and weak scaling of the LaTu code on BlueGene.

The implementation of the passive tracer particles is also done in a parallel way. This is necessary, because a large number of particles has to be integrated in order to sample the considered volume homogeneously and obtain reliable statistical results. We performed simulations with up to 10^7 particles on the IBM p690 machine.

The crucial point is the interpolation scheme needed in order to obtain the velocity field at the particle positions from the numerical grid. The code uses a tri-cubic interpolation scheme which on the one hand provides a high degree of accuracy and on the other hand parallelizes efficiently. Comparisons of the numerical results to simulations with a trilinear scheme are reported in[6].

4 FlareLab and Solar Flares

Observations of various phenomena at the solar surface and atmosphere – the so called solar transition region and solar corona – exist since a long time. Although successful work has been done in this time, mostly to explain and understand a special phenomenon, as yet there is no fundamental comprehension. These are for example the solar flares, the coronal mass ejections or the long living filaments and prominences. The connecting part in all these structures is the magnetic field. In these regions the magnetic forces and

energies dominate the gravitational and thermal ones. This puts the investigation focus on magnetic structures and its evolution.

The *FlareLab* project is intended to simulate solar flares. It splits in two parts, an experimental and a numerical. The experiment is basically a plasma arc created by a gas discharge in an arcade like magnetic field, whereas the setup is orientated on the previous work by[7,8]. The evolution of this current arc and the structure formation are the key aspects of the experiment. Accompanying numerical simulations on the one hand are performed to analyze the experiment and to create a link to solar conditions. Numerical experiments on the other hand are able to approximate the influence of plasma parameter and magnetic topology - as well as its geometry - before the experiment is modified.

The plasma in these regions is well described by the magneto-hydrodynamical equations. This set of partial differential equations gives the coupled temporal evolution of the plasma fluid and the magnetic field. They are solved by using finite differences for the spatial derivations and a third order Runge-Kutta method for the temporal integration.

The fundamental property of a magnetic field is solenoidality. This property is in general not conserved in numerical simulations. One way to ensure a physical magnetic field is to extend the MHD equations. The corresponding extension was proposed by[9]. In the framework *racoon* the divergence of the magnetic field is transported to outer regions and damped on its way. This works only for localized problems, here it is an arc shaped plasma in a huge empty domain.

The localization in this problem offers the possibility for the use of an adaptive mesh refinement. Here the grid is adapted to the electric current density, so that the current arc is well resolved during its evolution without resolving the other, plasma and current free, regions.

A first comparison of the experiment FlareLab and the adaptive mesh simulations is shown in Fig. 4.

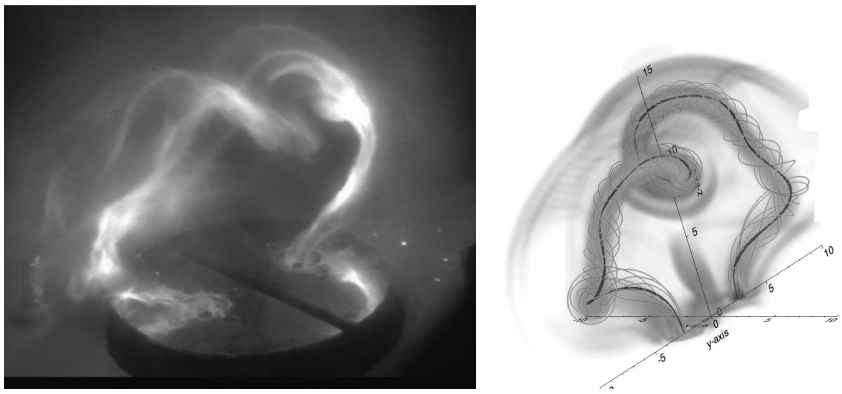


Figure 4. Comparison of the experiment FlareLab with simulations. Shown are high temperature regions (left) and current density and magnetic field lines (right).

5 Turbulence

Turbulence is an important and wide spread matter in today's research. From gas in molecular clouds to blood streaming through a heart valve one has to deal with turbulent flows and its properties. Although their generation is based on different forces and although they are enclosed by specific boundaries, there are features which all turbulent flows have in common.

The forces and boundaries naturally act on the large scales of the motion. From these scales turbulence generates a whole range of structures of different sizes down to the smallest scales where the dissipation transforms the kinetic energy into heat. The universality sets in at scales smaller than the boundary or forcing scale down to scales larger than the dissipation scales. Here the information of the geometry and the specific dissipation mechanism of the flow is lost and the motion is completely determined by the non-linear inertial interaction of the eddies. This range is called inertial range. Physical theories often deal with fundamental features such as scaling behaviour and intermittency of this range of scales. In order to analyze its properties numerically it is necessary to provide a large amount of scales. Numerical simulations of Euler- and Navier-Stokes-turbulence revealed that a resolution of at least 512^3 collocation points is needed to obtain an inertial range of scales within a turbulent flow.

Numerical investigations of different flows have to focus on different concerns. Incompressible flows are conveniently solved using pseudo-spectral codes. The formation of

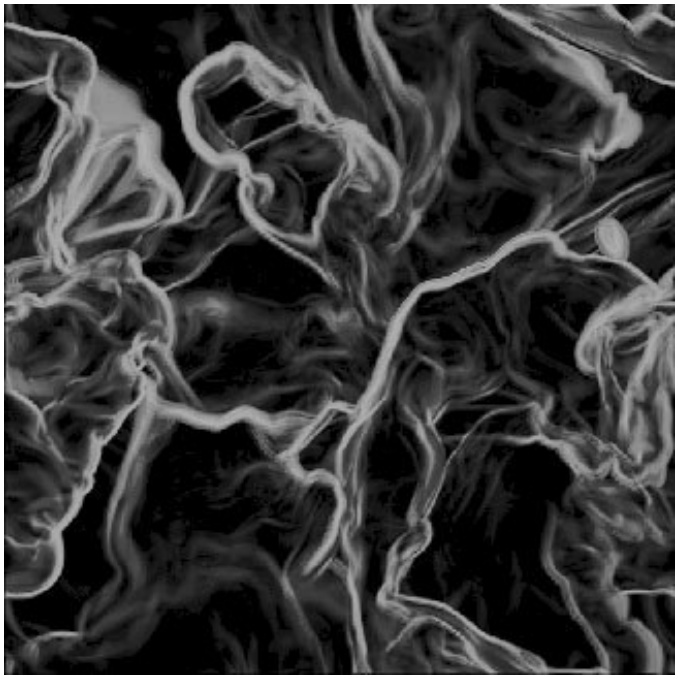


Figure 5. Volume rendering of vorticity of compressible turbulence

strong shocks in compressible gas dynamics needs a shock capturing central scheme and allows the application of adaptive mesh refining techniques (AMR)[10], so this problem is predestined to the *racoon* framework (see below). With this we examined isothermal Euler turbulence up to an effective resolution of 1024^3 cells. Fig. 5 shows the vorticity of a high Mach number compressible simulation.

While it is natural to perform incompressible MHD simulations with a pseudo-spectral code, compressible MHD simulations in real space appear to be more difficult. The proper way seems to be the constraint transport method[11] on a staggered grid in combination with divergence-free reconstruction[12] for the AMR.

Lagrangian statistics of turbulent fluid and magnetohydrodynamic flows has undergone a rapid development in the last 6 years to enormous progress in experimental techniques measuring particle trajectories[13]. Lagrangian statistics is not only interesting for obtaining a deeper understanding of the influence of typical coherent or nearly-singular structures in the flow but also of fundamental importance for understanding mixing, clustering and diffusion properties of turbulent astrophysical fluid and plasma flows. Tracer particles are employed in the incompressible as well as in the compressible code.

Concerning the incompressible case we computed the Lagrangian statistics of MHD- and neutral turbulence. The comparison revealed the intriguing and differing influence of the flow structures on the Eulerian and Lagrangian statistics[14] (see Fig. 6). The issue of intermittency was addressed by the computation of probability density functions (PDFs) and structure functions and comparison to a multifractal model[15].

Figure 6. Particle trajectories near singular events (left: Navier-Stokes, right: MHD)

In compressible turbulence we found an explanation for the PDF of the spatial particle distribution as a counterpart of the mass density field[16].

References

1. J. Dreher and R. Grauer, *Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws*, Parallel Comp. **31**, 913–932, (2005).
2. M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, J. Comp. Phys. **82**, 64–84, (1989).
3. A. Kurganov and E. Tadmor, *New high-resolution central schemes for nonlinear conservation laws and convection-diffusion equations*, J. Comp. Phys. **160**, 241–282, (2000).
4. G.W. Zumbusch, *On the quality of space-filling curve induced partitions*, Z. Angew. Math. Mech. **81**, 25–28 (2001).
5. R. Rabenseifer, *Hybrid Parallel Programming on HPC Platforms*, Fifth European Workshop on OpenMP, Aachen, (2003).
6. H. Homann and R. Grauer, *Impact of the floating-point precision and interpolation scheme on the results of DNS of turbulence by pseudo-spectral codes*, Comp. Phys. Comm. **177**, 560–565, (2007).
7. J. Hansen and P. Bellan, *Laboratory simulations of solar prominences*, APS Meeting Abstracts, 1089, (2001).
8. J. Hansen, S. Tripathi and P. Bellan, *Co- and counter-helicity interaction between two adjacent laboratory prominences*, Physics of Plasmas **11**, 3177–3185, (2004).
9. A. Dedner, F. Kemm, D. Kröner, C.-D. Munz, T. Schnitzer and M. Wesenberg, *Hyperbolic Divergence Cleaning for the MHD Equations*, J. Comp. Phys. **175**, 645–673, (2002).
10. A. Kritsuk, M. L. Norman and P. Padoan, *Adaptive mesh refinement for supersonic molecular cloud turbulence*, The Astrophysical Journal **638**, L25–L28, (2006).
11. U. Ziegler, *A central-constrained transport scheme for ideal magnetohydrodynamics*, J. Computat. Phys. **196**, 393–416, (2004).
12. D. S. Balsara, *Second-Order-accurate Schemes for Magnetohydrodynamics with Divergence-free Reconstruction*, The Astrophysical Journal Supplement Series **151**, 149–184, (2004).
13. A. La Porta, G. A. Voth, A. M. Crawford, J. Alexander and E. Bodenschatz, *Fluid particle accelerations in fully developed turbulence*, Nature **409**, 1017–1019, (2001).
14. H. Homann, R. Grauer, A. Busse and W. C. Müller, *Lagrangian Statistics of Navier-Stokes- and MHD-Turbulenz*, to appear in J. Plasma Phys, doi:10.1017/S0022377807006575, (Published online 31 May 2007).
15. L. Biferale, G. Boffetta, A. Celani, B. J. Devenish, A. Lanotte and F. Toschi, *Multifractal Statistics of Lagrangian Velocity and Acceleration in Turbulence*, Phys. Rev. Lett. **93**, 064502, (2004).
16. Ch. Beetz, Ch. Schwartz, J. Dreher and R. Grauer, *Density-PDFs and Lagrangian Statistics of highly compressible Turbulence*, arXiv:0707.1798v1 [physics.flu-dyn]

Object-Oriented Programming and Parallel Computing in Radiative Magnetohydrodynamics Simulations

Vladimir Gasilov, Sergei D'yachenko, Olga Olkhovskaya, Alexei Boldarev, Elena Kartasheva, and Sergei Boldyrev

Institute for Mathematical Modelling, Russian Academy of Sciences
Miusskaya Sq. 4-A, 125047 Moscow, Russia
E-mail: {gasilov, boldar, bsn}@imamod.ru

The subject of this paper is the computer simulation of transient processes in strongly radiative plasma, that refers to solving the problems of radiative magnetohydrodynamics (MHD). The program system MARPLE developed in IMM RAS is described, where the application of OOP and parallel computing is emphasized. The strategy and scheme of code parallelizing are explained, with the outcome being presented in the results of practical computations.

1 Introduction

Modern problems in pulsed-power energetics issue a real challenge to the computer simulation theory and practice. High-performance computing is a promising technology for modelling complex multiscale nonlinear processes such as transient flows of strongly radiative multicharged plasmas. An essential part of such numerical investigations is devoted to computer simulation of pinches resulted from electric explosion of cold matter, e.g. gas-puff jets, foam strings, or metallic wire arrays. These investigations were significantly stimulated by impressive results in the soft X-ray yield produced by wire cascades or double arrays obtained in Sandia National Laboratory, US[1]. The goal of numerical research in pulsed-power is to study the evolution of very intensive transient electric discharges and to perform a multiparametric optimization of future experimental schemes.

2 Mathematical Model and Numerical Methods

Numerical modelling of strongly radiative plasma flows goes along with development and sophistication of research codes. This paper concerns the code MARPLE (Magnetically Accelerated Radiative PLasma Explorer) created in the Institute for Mathematical Modelling, RAS[2]. MARPLE performs calculations in terms of cylindrical (r, z) frame of reference, while (r, φ) or Cartesian (x, y) geometry is also available. A plasma flow is considered in the single-fluid magnetohydrodynamics model given by the well-known theory of Braginsky. The MHD equations are written in a so-called 2.5-dimensional fashion, where the flowfield vectors are presented by all three components: velocity $\vec{w} = (w_r, w_\varphi, w_z)$, magnetic inductance $\vec{B} = (B_r, B_\varphi, B_z)$, and electric intensity $\vec{E} = (E_r, E_\varphi, E_z)$. The anisotropy of dissipative processes in presence of magnetic field is taken into account. The energy balance is described in terms of the electron-ion-radiation relaxation. The governing system of equations includes the radiative transport equation for spectral intensity, and is completed by data bases and/or analytical description of plasma state, coefficients of transport processes, spectral opacities and emissivities.

Depending on the studied problem the governing system can also be supplemented by an equation describing electric current evolution for the whole electric circuit. For a typical pulsed-power problem the circuit includes an electric generator, current supply facilities, and a discharge chamber.

2.1 Governing System of Equations for 2-Temperature MHD

Non-dissipative MHD:

$$\frac{\partial \rho}{\partial t} + \nabla(\rho \vec{w}) = 0, \qquad \frac{\partial \rho w_i}{\partial t} + \sum_k \frac{\partial}{\partial x_k} \Pi_{ik} = 0,$$

$$\Pi_{ik} = \rho w_i w_k + P\delta_{ik} - \frac{1}{4\pi}\left(B_i B_k - \frac{1}{2}B^2\delta_{ik}\right),$$

$$\frac{\partial \vec{B}}{\partial t} - \nabla \times (\vec{w} \times \vec{B}) = 0,$$

$$\frac{\partial}{\partial t}\left(\rho\varepsilon + \frac{1}{2}\rho w^2 + \frac{B^2}{8\pi}\right) + \nabla \vec{q} = 0,$$

$$\vec{q} = \left(\rho\varepsilon + \frac{1}{2}\rho w^2 + P\right)\vec{w} + \frac{1}{4\pi}\vec{B} \times (\vec{w} \times \vec{B}), \qquad P_{e,i} = P_{e,i}(\rho, \varepsilon_{e,i}).$$

Dissipative processes:

$$\vec{E} = \frac{\vec{j}_{\parallel}}{\sigma_{\parallel}} + \frac{\vec{j}_{\perp}}{\sigma_{\perp}},$$

$$\operatorname{rot} \vec{B} = \frac{4\pi}{c}\vec{j} = \frac{4\pi}{c}\hat{\sigma}\vec{E}, \qquad \frac{1}{c}\frac{\partial \vec{B}}{\partial t} = -\operatorname{rot} \vec{E},$$

$$\frac{\partial \rho\varepsilon_e}{\partial t} = -\operatorname{div}(\hat{\kappa}_e \operatorname{grad} T_e) + Q_{ei} + G_J + G_R,$$

$$\frac{\partial \rho\varepsilon_i}{\partial t} = -\operatorname{div}(\hat{\kappa}_i \operatorname{grad} T_i) - Q_{ei}, \qquad \varepsilon = \varepsilon_i + \varepsilon_e, \quad P = P_i + P_e.$$

2.2 The Physical Splitting Scheme and Numerical Algorithms

Non-dissipative MHD	Dissipative processes	
	Magnetic field diffusion, heat transfer, electron-ion exchange, Joule heat	Radiative energy transfer
Local processes	Quasi-local processes	Non-local processes
Explicit scheme	Implicit scheme	Explicit fractional-step scheme
High-res. TVD scheme with flux correction	Integro-interpolation finite volume schemes	Characteristic scheme
Time advance: 2nd order predictor-corrector scheme		

The code provides for the problem solution in a complex geometry domain with use of unstructured triangular and quadrilateral meshes where all physical variables are stored at the mesh vertices (nonstaggered storage). The conservation laws for mass, momentum, and energy are approximated by means of the finite volumes technique. In the current version of code the finite volumes are formed by modified Voronoi diagrams. An example of computational mesh and finite volumes is presented below in Fig. 1.

The MHD system is solved by the generalized TVD Lax-Friedrichs scheme which was developed for the unstructured mesh applications[3]. For the case of regular triangulation this scheme ensures the second order approximation of spatial derivatives (the third order is possible with a special choice of the antidiffusion limiters). For solving parabolic equations, describing magnetic diffusion and conductive heat transfer, we developed new finite-volume schemes constructed by analogy with the mixed finite-element method. The time-advance integration is explicit, the second approximation order being achieved due to the predictor-corrector procedure. The time step is restricted by the Courant criterion.

2.3 Radiation Transport

Radiative energy transfer is described by the equation for spectral radiation intensity. In practice, calculations are performed via multi-range spectral approximation. We solve the equation by means of a semi-analytical characteristic interpolation algorithm constructed on the base of Schwarzschild-Schuster approximation. The analytical solution along a characteristic direction is constructed by means of the forward-backward angular approximation to the photon distribution function[4,5]. The two-group angular splitting gives us the analytical expression for radiation intensity dependent on opacity and emissivity coefficients. The energy exchange between the radiation field and the gas is taken into account via radiative flux divergence, which is incorporated into the energy balance as a source function (G_R in the above system).

The transport equation for quasistationary radiation field in cylindrical geometry:

$$\sin\theta\left(\cos\varphi\frac{\partial I_\omega}{\partial r}+\frac{\sin\varphi}{r}\frac{\partial I_\omega}{\partial\varphi}\right)+\cos\theta\frac{\partial I_\omega}{\partial z}=-\aleph_\omega I_\omega+j_\omega.$$

A set of equations for the forward/backward intensity functions $I^{f/b}$:

$$\frac{\cos\theta_n-\cos\theta_{n+1}}{\Delta\theta_n}\left(\frac{\partial I^f_{n+1/2}}{\partial r}+\frac{I^f_{n+1/2}}{r}\right)+\frac{\sin\theta_{n+1}-\sin\theta_n}{\Delta\theta_n}\frac{\partial I^f_{n+1/2}}{\partial z}=-\aleph I^f_{n+1/2}+j,$$

$$\frac{\cos\theta_{n+1}-\cos\theta_n}{\Delta\theta_n}\left(\frac{\partial I^b_{n+1/2}}{\partial r}+\frac{I^b_{n+1/2}}{r}\right)+\frac{\sin\theta_{n+1}-\sin\theta_n}{\Delta\theta_n}\frac{\partial I^b_{n+1/2}}{\partial z}=-\aleph I^b_{n+1/2}+j.$$

The radiative energy density (radiative contribution in energy balance):

$$G_R=\sum_{s=1}^{k}\frac{\pi}{c}\sum_{n=1}^{N}\left(I^f_{n+1/2}+I^b_{n+1/2}\right)(\cos\theta_n-\cos\theta_{n+1}).$$

The forward/backward intensities along the i-th ray in the direction $\theta_{n+1/2}$:

$$I^f_{j+1}=\left(I^f_j-J_j\right)\exp(-\aleph_j l_j)+J_j,\qquad I^b_j=\left(I^b_{j+1}-J_j\right)\exp(-\aleph_j l_j)+J_j.$$

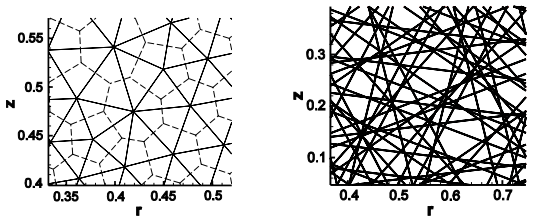


Figure 1. Fragments of the triangular mesh with finite volumes and of the grid of rays.

For the purpose of radiative energy transfer calculation a special grid of characteristics is constructed in the computational area. This grid is comprised by a number of sets (families) of parallel lines and is further referred to as the grid of rays. Each set of rays is characterized by the angle of inclination to coordinate axes and by spatial density of rays. The grid of rays introduces some discretization of computational domain in the plane (r, z) and with respect to the angle θ ($0 \leq \theta < \pi$), that is required for numerical integration of the radiation transport equation according to the above described model. The grid of rays is superimposed on the initial computational mesh designed for MHD and heat transfer computations. A fragment of the grid (12 angle sectors) is shown in Fig. 1.

3 Parallel Implementation

All in all, the described sort of problems is computationally very hard, due to the big number of involved physical processes and corresponding equations. It consumes substantial amounts of processing time even on meshes which might look rather small for some other commonly known problems. At this, we often need to carry out quite long computations, because processes under investigation are inherently time-dependent. And industrial applications usually require numerous series of similar computations. So it makes code parallelization for us not only challenging, but naturally highly desirable.

3.1 Parallelism & C++

Certain specificity of introducing parallelism into a program complex in our case relates to the object-oriented nature of MARPLE code, which essentially employs C++ language facilities, such as polymorphism, encapsulation, inheritance, and parametric programming.

Our data model takes its origin in the finite-element technique and provides a description of geometrical and topological properties for computational domains as well as for numerical meshes. Special data structures based on the concept of topological complex have been elaborated to provide problem statement in an arbitrary domain, and for handling unstructured meshes, including dynamic mesh changes. From some point, the programming language C++ was chosen for further elaboration of MARPLE codes, because it provides the most convenient way to implement data structures for handling dynamic unstructured meshes. Besides, the program implementation of various types of boundary conditions, equations of state, and some other physical elements of the model, can be realized in a natural way by use of inheritance and virtual functions. Also, specialized classes

have been developed for handling data tables, comprising parameters of ionization, electric conductivity, thermodynamic and optical properties. The C++ language and, on the whole, the object-oriented approach, simplifies the creation and further development of large complicated computer codes, especially in case of team development.

When the challenge to parallelize the algorithm arises, the necessity to modify the data structures normally appears too. Some of the above mentioned structures have to be adapted to allow for parallel computations and data exchanges, taking into account also the requirement of keeping interprocessor communication adequately small. Although, as a matter of fact, many of the classes and functions can be used in each branch of parallel code as well as in a usual sequential code.

For the problems like those described above and solved on triangular meshes, the geometrical parallelism is commonly used. It implies that each branch of the code processes a subset of computational mesh, while the data exchanges provide the possibility of computations in the vicinity of "inter-branch" bounds. In case of explicit schemes on unstructured meshes it means that each branch of the code stores two types of the computational mesh elements. The "real" elements require regular computations specified by the used scheme, while the "fictive" elements are the images of some "real" elements processed by another branch, and the data in these "fictive" elements are not computed but received from that branch. The "fictive" elements form "margins" along the inter-branch bounds.

It should be noted that the "fictive" elements and the "margins" concepts may be used as well in a single-branch (sequential) code for problems with some spatial symmetry (problems with translational or rotational periodicity). The "fictive" elements in this case are the images of some "real" elements in the geometrical transformation, with respect to which the problem is symmetrical. But the principal idea remains the same: instead of regular computations in a "fictive" element, the data associated with this element are transferred from the element-preimage.

So, in case of using geometrical parallelism, the data structures describing computational domains and meshes have to be modified, in order to handle the "fictive" mesh elements and encapsulate the methods for proper data exchanges, when needed.

3.2 Radiation Transport

Highly accurate simulation of radiative energy transfer, including detailed reproducing of the radiation spectrum, is among the most important requirements to the developed code. The reason is its great significance to the design of powerful X-ray pulse sources. Thereto, the entire spectrum is divided into a number of frequency ranges (from tens to hundreds). It is necessary to reconstruct the radiation field with respect to its angular distribution for each frequency range. So, the radiative transfer is calculated independently in each waveband, and then the corresponding results are summarized. Besides, as a rule, the grid of rays has the number of points much bigger than in the basic mesh. This is why the radiation transport computation is one of the most laborious steps in radiative MHD simulations. Profilings run at a monoprocessor computer showed that sometimes up to 80-90% of the total processing time may be spent on the radiative transfer computation, depending on a particular problem. For instance, typical figures corresponding to our benchmark problem (see Section 4 below) with 100 frequency ranges are shown in Table 1.

Therefore, the radiation transport was naturally chosen as our primary target for parallelizing. Now it's time to note, that in the above shown scheme of physical processes

Ideal MHD:	3.4%
Magnetic field diffusion:	4.7%
Heat conduction + e-i exchange:	3.0%
Joule heat:	4.0%
Radiative energy transfer:	84.9%

Table 1. Computational costs on different physical processes.

radiative transport stands apart in certain sense, because its most computations are performed on a special grid of rays, different from the basic grid, and have an essentially non-local character. Hence, domain decomposition and geometrical parallelism seem to be not applicable here. We chose another way of distributing computational load, which appears more natural and usable in this case. Indeed, the frequency ranges model requires repeating large volume of uniform computation with different sets of opacity and emissivity values for each range. It seems appropriate to carry out these computations concurrently on several processors, and then to collect the results by simple summation, using the fact that all frequency ranges produce uniform and independent contributions to the total radiative energy fluxes.

Thus, the whole volume of computation within each time step can be divided into two principal parts: the computations performed on the basic triangular mesh, and the computations performed on the grid of rays. Since currently the parallelization of the former part is not yet finished, the same calculations in this part are carried out by all processors simultaneously. In fact, this doesn't lead to any time loss, but helps to avoid any data exchange before the second part begins, as all processors will have exactly the same data by that moment. Then computations along the rays are performed in parallel, each processor calculating energy fluxes in its own set of wavebands. As these computations are well independent, no data related to the rays or ray points need to be exchanged. It makes this parallel computation quite efficient, provided that the number of processors is the divisor of the number of wavebands. However, to finalize this part, the radiative energy density must be summed up over all frequency ranges, so the global data collection is once required, actually in a scale of one variable defined on the triangular mesh (G_R).

3.3 C++ & MPI

Parallel computations on our cluster systems (with distributed memory architecture) were organized by means of the MPI framework, which is well efficient and highly portable as the prevalent today standard for message-passing communication model. However, the original library doesn't make use of OOP capabilities, that could be quite profitable in simplifying parallel programming[6]. On the other hand, the library contains plenty of efficient specialized functions with numerous options giving great flexibility to users, but practically many researchers tend to use some very limited subset of this richness for their particular computational problems. In order to simplify library usage and bring it closer to the C++ character of MARPLE codes, an intermediate interface has been elaborated, that hides the less exploitable details of the library but at the same time conveys some appropriate amount of flexibility provided by MPI to the core computational codes. As

a result, the interprocessor communications are organized in the overall manner of C++ paradigm, i.e. by using some set of datatype-secure polymorphic functions, convenient for our actual needs. E.g. asynchronous Send calls will look like the following:

```
aSend( int dest_proc, DataType *data_ptr, int data_count );
aSend( int dest_proc, DataType &variable );
```

However, this is yet our first approach realized for the moment. In fact, the sending/receiving functions are called now directly from modules that implement numerical methods and schemes. The next step will be encapsulating such calls into classes that are responsible for data description and storage, so that a command given by a computing module would be not to send/receive/share some particular data array, but to exchange or refresh, or acquire any specific data structures, like those described above in Section 3.1.

4 Practical Results

The program codes have been proved in practical computations on two cluster systems currently available in IMM RAS. Both systems are Linux-driven self-made clusters based on the Intel platform[a]. The results obtained in a set of test computations are presented in Table 2. Here, the triangular mesh contains 21221 node, and the grid of rays consists of 6160 rays with total of 1054553 points for computation. The number of frequency ranges for this series of computations was taken 100. Times given in seconds correspond to one time step calculations on our single-core Xeon-based cluster.

	Number of processors				
	1	2	4	10	20
Full time (100%)	48.82 sec	29.51 sec	18.76 sec	13.56 sec	11.16 sec
Grid of rays	41.44 (85%)	20.8 (70%)	10.1 (54%)	3.94 (29%)	1.89 (17%)
Data exchange	0 (0%)	0.29 (1.0%)	0.51 (2.7%)	0.55 (4.1%)	0.58 (5.2%)
Speed-up on rays	1.00	1.99	4.10	10.52	21.90
Overall speed-up	1.00	1.65	2.60	3.60	4.37
Overall efficiency	100%	83%	65%	36%	22%

Table 2. Timings of practical computations.

As it can be clearly seen from the table, the overall efficiency rapidly falls down with the growing number of processors, although the part of computation run on the grid of rays exhibits a perfect linear acceleration. This is an evident consequence of the fact that the latter is the only part of computation actually parallelized by now. So even if on a single processor it takes 85% of the total time cost, this proportion tends to change very quickly. Actually, for the small numbers of frequency ranges the parallelization gives here very limited effect, e.g. initially in the above problem with 20 wavebands the contribution

[a] The first cluster consists of 26 modules, each of them containing two Intel Xeon 3 GHz processors and 2 Gb of RAM, all modules being connected by Gigabit Ethernet network. The second cluster is similarly comprised by 14 dual-processor modules with 4 Gb RAM, where each processor is the dual-core Xeon 2.66 GHz.

of calculations along the rays into the whole computation time was only about 50%, so it would be hard to reach even the acceleration of factor 2. Obviously, doing only some distinct part of computations in parallel is not enough, though we must admit that even such limited acceleration is quite useful for us in practice.

Thus, from this point we are going to parallelize the rest of our code. First of all, we think to employ physical splitting scheme further, so that to parallelize computations on different physical processes independently, using different techniques (e.g. for explicit and implicit schemes), and dedicating different groups of processors for these tasks. The nearest stage of code development is to detach hyperbolic processes from parabolic ones, and to organize additional iterations within the latter group, together with radiative transfer. Hence, we suppose that parallelizing radiation transport computation has been only the first though important step towards making the whole MARPLE code parallel.

5 Conclusions

It proved possible to make good use of the proposed technique in numerical simulation of heterogeneous liners implosion with powerful X-ray pulse generation. The corresponding experiments are performed at "ANGARA 5-1" facility (TRINITI, Troitsk, Moscow reg., Russia)[7]. Parallel computing technology applied to the radiative energy transfer calculation helped us to reduce the total processing time by factor of 2 to 4. This is a significant practical achievement, since for the experiment scheme optimization a big series of similar numerical simulations is actually needed. Another concurrent advantage is that the number of ranges in a spectrum can be increased, that gives immediate effect on the accuracy and quality of numerical solutions. For example, in the above mentioned numerical experiments initially only 20 frequency ranges were differentiated, and for parallel computations this number was notably increased to one hundred.

Acknowledgements

The work is carried out with the help of Russian Academy of Sciences, Mathematical Sciences Department (program 10002-251/OMN-03/026-023/240603-806), and Russian Foundation for Basic Research (grant No. 05-01-00510).

References

1. J. P. Chittenden, et al., Plasma Phys. Control Fusion, **46**, B457–B476, (2004).
2. V. A. Gasilov, et al., Mathematical Modelling, **15**, No. 9, (2003).
3. V. A. Gasilov and S. V. D'yachenko, *Quasimonotonous 2D MHD scheme for unstructured meshes*, in: Mathematical Modelling: Modern Methods and Applications, pp. 108–125 (Janus-K, Moscow, 2004).
4. R. Siegel and J. R. Howell, *Thermal Radiation Heat Transfer*, (McGraw-Hill, 1972).
5. B. N. Chetverushkin, *Mathematical modelling in radiative gasdynamic problems*, (Nauka, Moscow, 1985).
6. C. Hughes and T. Hughes, *Parallel and Distributed Programming Using C++*, (Addison-Wesley, 2004).
7. V. V. Alexandrov, et al., Plasma Phys. Reports, **27**, No. 2, (2001).

Parallel Simulation of Turbulent Magneto-hydrodynamic Flows

Axelle Viré[2], Dmitry Krasnov[1], Bernard Knaepen[2], and Thomas Boeck[1]

[1] Fakultät für Maschinenbau, Technische Universität Ilmenau
P.O. Box 100565, 98684 Ilmenau, Germany
E-mail: {*thomas.boeck, dmitry.krasnov*}*@tu-ilmenau.de*

[2] Université Libre de Bruxelles, Service de Physique Théorique et Mathématique
Campus Plaine - CP231, Boulevard du Triomphe, 1050 Brussels, Belgium
E-mail: {*bknaepen, avire*}*@ulb.ac.be*

We compare the performances of a pseudospectral and a finite-volume method for the simulation of magnetohydrodynamic flows in a plane channel. Both Direct Numerical Simulations (DNS) and Large-Eddy Simulations (LES) are performed. The LES model implementation is validated for non-conductive flows. The application of the LES model in the case of flows subjected to low intensity magnetic fields is shown to be successful.

1 Introduction

Turbulent magnetohydrodynamic (MHD) flows at low magnetic Reynolds number, i.e. flows of electrically conducting liquids in the presence of an external magnetic field, occur in a variety of metallurgical processes. MHD flows are affected by the Lorentz force arising from the induced electric currents in the liquid. Important examples are the electromagnetic braking of molten steel in continuous casting or the electromagnetic stirring of melts[1]. By comparison with ordinary fluid flows, the experimental investigation of MHD flows is complicated by the opacity and corrosiveness of liquid metals. Experiments typically provide very limited information on the flow structures and statistics. For this reason, the accurate prediction of such flows by numerical simulations is of particular interest.

The purpose of the present study is to investigate the MHD channel flow by performing Large-Eddy Simulations (LES) and to compare two numerical approaches, the pseudospectral method and the finite-volume one, in terms of parallel computing performances. The attention is restricted to the case of a plane channel with a wall-normal magnetic field and electrically insulating boundaries. Such a geometry is known as the Hartmann flow. As illustrated in Fig. 1 (from Boeck *et al.*[2]), its remarkable features are a suppression of turbulent fluctuations (i.e. flat profile) in the middle of the channel and two boundary layers at the walls, where the current loops close. Moreover, the boundary layers become thinner as the imposed magnetic field intensity increases.

As a consequence, compared to the hydrodynamic case, the presence of the magnetic field imposes the use of a finer mesh, close to the walls, leading to higher computational costs. Hence, LES is likely to prove very useful to simulate MHD flows in that it reduces the global number of grid points. However, considerable computational ressources are still needed when more flexible codes are used. Direct Numerical Simulations (DNS) of MHD channel flows have been investigated intensively by Boeck *et al.*[2]. Their results, obtained with a pseudospectral code, are used here as a benchmark for the LES simulations of MHD

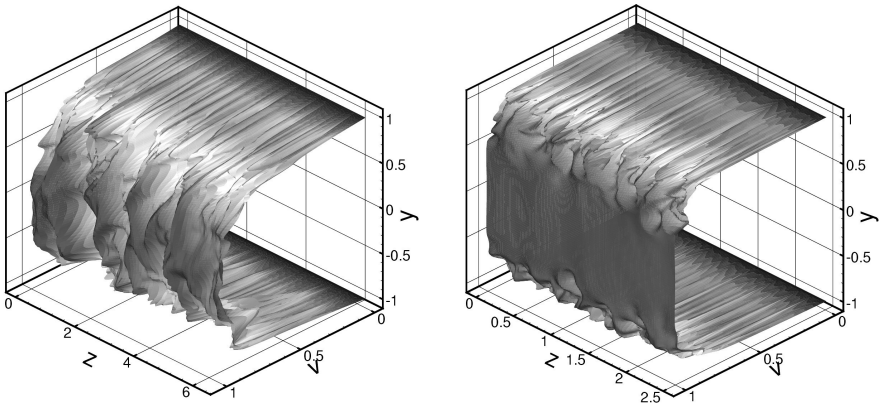


Figure 1. Effect of the wall-normal magnetic field on a turbulent channel flow. Snapshots of the streamwise velocity component in a plane $x = const$ for a non-magnetic flow at a Reynolds number $Re_c = U_c\delta/\nu = 3300$, based on the mean centreline velocity U_c and the channel half-width δ (left); and for a turbulent Hartmann flow at $R = 500$ and $Ha = 30$ (right) (for additional definitions see Section 2.1).

flows.

2 Mathematical Model

2.1 Basic MHD Equations

When an electrically conducting fluid is set in motion in the presence of an external magnetic field, the flow is affected by the Lorentz force. This has globally dissipative and anisotropic effects and, therefore, it modifies the turbulence structures present in the flow. The relative importance of the nonlinear and the diffusion terms in the magnetic induction equation is quantified by the magnetic Reynolds number Re_m. In the limit of low Re_m, the so-called *quasi-static approximation*, the effect of the magnetic field is taken into account through an extra damping term[3]. This leads to the incompressible Navier-Stokes Eq. (2.1), in which ρ, μ, σ are the fluid density, molecular viscosity and electrical conductivity, respectively; and u, P, B are the velocity, pressure and external magnetic field. The induced electric current j is given by Ohm's law, $\mathbf{j} = \sigma(-\nabla\Phi + \mathbf{u} \times \mathbf{B})$, assuming that the electric field is the gradient of the electric potential $\Phi = \nabla \cdot (\mathbf{u} \times \mathbf{B})$. In Eq. (2.1), ϵ_{ijk} represents the permutation symbol.

$$\rho\partial_t u_i + \rho\partial_j(u_i u_j) = -\partial_i P + \mu\partial_j\partial_j u_i + \sigma\epsilon_{ijk} j_j B_k \qquad i,j,k = 1,...,3 \tag{2.1}$$

Notice that we use the summation convention, and that the incompressibility requires $\partial_j u_j = 0$. By introducing the friction velocity u_τ and the characteristic length δ, Eq. (2.1) can be non-dimensionalized, leading to

$$\partial_t u_i^+ + \partial_j(u_i^+ u_j^+) = -\partial_i p^+ + \frac{1}{Re_\tau}\partial_j\partial_j u_i^+ + N\epsilon_{ijk} j_j^+ B_k^+ \qquad i,j,k = 1,...,3. \tag{2.2}$$

In Eq. (2.2), $p = P/\rho$ and $\nu = \mu/\rho$ are the kinematic pressure and viscosity, respectively. In addition, non-dimensional parameters are introduced: the friction Reynolds

number $Re_\tau = u_\tau \delta/\nu$ and the interaction parameter $N = \sigma B^2 \delta/\rho u_\tau$. Additional non-dimensional parameters are often used to describe MHD flows: the Hartmann number $Ha = B\delta\sqrt{\sigma/\rho\nu}$ which quantifies the intensity of the magnetic field, and the Reynolds number $R = U_0 d/\nu$ based on the laminar centreline velocity U_0 and on the Hartmann layer thickness $d = B^{-1}\sqrt{\rho\nu/\sigma}$. The superscript $+$, which denotes non-dimensional quantities, will be omitted in the rest of this paper.

2.2 Filtering and Subgrid-Scale Model

LES equations are obtained by spatially filtering Eq. (2.2). In Eq. (2.3), the overbar denotes the "grid-filter", whose kernel is G, and which eliminates the small scale part of the discrete velocity field. The filtered velocity $\overline{u}_i$ is defined by Eq. (2.4).

$$\partial_t \overline{u}_i + \partial_j(\overline{u_i u_j}) = -\partial_i \overline{p} + \frac{1}{Re_\tau}\partial_j\partial_j \overline{u}_i + N\overline{\epsilon_{ijk} j_j B_k} \tag{2.3}$$

$$\overline{u}_i(\mathbf{x}) = \int G(\mathbf{x},\mathbf{y}) u_i(\mathbf{y}) d\mathbf{y} \tag{2.4}$$

Filtering of the incompressibility condition leads to $\partial_j \overline{u}_j = 0$. Because we are using the quasi-static approximation, the only non-linear term in Eq. (2.3) is the convective term. Therefore, writing Eq. (2.3) as:

$$\partial_t \overline{u}_i + \partial_j(\overline{u}_i \overline{u}_j) = -\partial_i \overline{p} + \frac{1}{Re_\tau}\partial_j\partial_j \overline{u}_i + N\overline{\epsilon_{ijk} j_j B_k} - \partial_j \overline{\tau}_{ij}, \tag{2.5}$$

we see that only the subgrid-scale stress tensor $\overline{\tau}_{ij} = \overline{u_i u_j} - \overline{u}_i \overline{u}_j$ has to be modelled in order to close the equation in terms of the filtered velocity. The most commonly used model is the Smagorinsky model,

$$\overline{\tau}_{ij} = -2\nu_e \overline{S}_{ij} + \frac{1}{3}\delta_{ij}\overline{\tau}_{kk}, \tag{2.6}$$

in which δ_{ij} is the Kronecker symbol, $\nu_e = C_s \overline{\Delta}^2 |\overline{S}|$ is the eddy viscosity, C_s is the Smagorinsky constant, $\overline{\Delta}$ is the grid-filter width, and $|\overline{S}| = (2\overline{S}_{ij}\overline{S}_{ij})^{1/2}$ is the magnitude of the large-scale strain-rate tensor $\overline{S}_{ij} = 1/2(\partial_j \overline{u}_i + \partial_i \overline{u}_j)$. However, this model uses a single constant C_s and turns out to be inefficient in several cases: modelling of inhomogeneous turbulence, backscatter, etc. For this reason, a dynamic procedure has been developed[4], which aims to overcome the previously mentioned drawbacks by dynamically computing C_s. This is achieved by introducing a second, coarser spatial filter, the "test-filter", denoted by $\widehat{...}$, whose width is $\widehat{\overline{\Delta}}$. The LES equation is re-written as

$$\partial_t \widehat{\overline{u}}_i + \partial_j(\widehat{\overline{u}}_i \widehat{\overline{u}}_j) = -\partial_i \widehat{\overline{p}} + \frac{1}{Re_\tau}\partial_j\partial_j \widehat{\overline{u}}_i + N\widehat{\overline{\epsilon_{ijk} j_j B_k}} - \partial_j \widehat{T}_{ij}, \tag{2.7}$$

in which $\widehat{T}_{ij} = \widehat{\overline{\tau}}_{ij} - \widehat{\overline{u}}_i \widehat{\overline{u}}_j + \widehat{\overline{u_i u_j}}$. Assuming that both levels of filtering use the same C_s (self-similarity hypothesis), which means $\widehat{C}_s \approx C_s$, the Smagorinsky parameter is expressed by

$$C_s = \frac{1}{2}\frac{\langle L_{ij}\widehat{\overline{S}}_{ij}\rangle_{xz}}{\langle M_{ij}\widehat{\overline{S}}_{ij}\rangle_{xz}}, \tag{2.8}$$

where

$$L_{ij} = \widehat{T}_{ij} - \widehat{\overline{\tau}}_{ij}, \quad M_{ij} = \overline{\Delta}^2 |\,\widehat{\overline{S}}\,|\,\widehat{\overline{S}}_{ij} - \widehat{\overline{\Delta}}^2\,|\,\widehat{\overline{S}}\,|\,\widehat{\overline{S}}_{ij}. \tag{2.9}$$

Because of the flow homogeneity in the wall-parallel directions x, z of the channel, a single value of the Smagorinsky parameter can be assumed in each $y = const.$-plane[5]. Thus, in Eq. (2.8), $\langle ... \rangle_{xz}$ denotes an averaging procedure in these planes. All LES results presented in this paper are obtained with the dynamic Smagorinsky model (DSM) by applying local grid and test filters in planes parallel to the channel walls, whereas the averaging procedure in the computation of C_s is performed globally in these planes. The test to grid filter ratio $\widehat{\overline{\Delta}}/\overline{\Delta}$ is set to 2.

3 Numerical Codes for Wall-Bounded Flows

3.1 Pseudospectral Method (PSM)

The pseudospectral code used in this study is described in Boeck *et al.*[2]. The method applies a Fourier expansion in horizontal directions where periodic boundary conditions are imposed, and a Chebyshev polynomial expansion in the vertical direction between insulating walls (no-slip conditions). The time-stepping scheme uses three time levels for the approximation of the time derivative and is second-order accurate. The computation of the subgrid-scale (SGS) term can be disabled, so that the flow solver can be used for both DNS and LES calculations. The parallelization of the pseudospectral algorithm is accomplished by a domain decomposition with respect to the x or z direction. Only the Fourier transforms will then require inter-process communication. In our implementation, the transform proceeds as a successive application of one-dimensional transforms with respect to x, z and y. The transform for the divided direction is avoided by a transposition of the data array containing the expansion coefficients. The interprocess communication utilizes the Message Passing Interface (MPI) library.

3.2 Finite-Volume Method (FVM)

The numerical solution is based on the discretization in finite volumes of the integral form of the Navier-Stokes equations. Compared to spectral codes, the finite-volume method has the advantage that it can deal with very complex geometries often encountered in industrial applications. Depending on the grid arrangement, two formulations can be defined: the first is based on a staggered mesh, in which the pressure is computed at the cell centre, whereas the velocity components are calculated at the cell faces; the second uses a collocated mesh, in which both pressure and velocity components are computed at the cell centre. In both cases, interpolations are necessary to obtain variables at the faces from the grid ones. For simulations in complex geometries, the second method is favoured over the first one due to its simpler form in curvilinear coordinates[7]. However, non-staggered grids require special care to handle the well-known velocity/pressure decoupling. In the present code, this is done by introducing an implicit smoothing of the pressure by interpolation[8]. For this reason, a collocated-mesh scheme introduces a non-conservation error, unlike the staggered-mesh formulation[7]. The present simulations are performed with the CDP code developed at the Center for Turbulence Research (NASA Ames/Stanford University)[8,9].

A semi-implicit (Adams–Bashforth/Crank–Nicholson) time-splitting method is chosen for time advancement[10] while a collocated formulation is adopted for the spatial discretization. The parallelization is based on the MPI standard. The solver has been extended to deal with LES of MHD flows.

4 Results

4.1 Code Validation

Implementations are validated by comparing hydrodynamic channel flow simulations with those of Kim *et al.*[11]. A constant mass flow rate is imposed so that $Re_b = 2U_b\delta/\nu = 5600$, where Re_b is the bulk Reynolds number and $U_b = \int_{-\delta}^{\delta} U(y)dy/2\delta$ is the bulk velocity (δ is chosen as the channel half-width). This results in $Re_\tau \approx 180$, where u_τ is defined through the wall shear stress ρu_τ^2. Table 1 shows the three test cases considered. In both codes, periodic boundary conditions are applied in the homogeneous directions (i.e. streamwise and spanwise) and the mesh spacing is stretched in the wall-normal direction according to the distribution of Chebyshev collocation points. The domain size is $(4\pi\delta) \times (2\delta) \times (2\pi\delta)$ in the streamwise, wall-normal and spanwise directions, respectively. Results obtained

Case ID	Mesh resolution	Subgrid-scale model
udns64	$64 \times 64 \times 64$	No
les64	$64\times 64 \times 64$	DSM
udns128	$128 \times 128 \times 128$	No

Table 1. Hydrodynamic channel flow simulations. The acronym udns refers to an unresolved direct numerical simulation, i.e. without LES model and insufficient grid resolution.

from both codes are in good agreement with the literature ones. For the $udns128$ case, the friction Reynolds number is equal to 177.98 using the FVM and to 183.16 with the PSM. However, the PSM approach is more accurate than the FVM for coarser resolutions. In fact, the $les64$ case gives $Re_\tau \approx 167.12$ using the FVM and $Re_\tau \approx 178.32$ using the PSM. Hence, considering that a difference in Re_τ of approximately one percent compared to the DNS value is fairly good, the accuracy of the PSM is satisfactory at coarse resolutions if a LES model is used. The inaccuracy of the finite-volume simulation is due to a higher numerical dissipation and a more detailed analysis of the different contributions to the total dissipation has to be performed.

4.2 Parallel Performance Benchmarks

The aim of this part is to compare the computational cost of the PSM with that of the FVM, with and without LES model. All the simulations are run on our cluster made of four 3.0 GHz, 8-core Intel Xeon-based Mac Pro nodes. The nodes are connected through a standard gigabit Ethernet network. Internal communications utilize the local bus and shared memory.

Case	Number of CPU's	Run time per 10000 iterations [hours]	% CPU
udns64	1 (1 node)	1.8	100
udns64	2 (1 node)	1.00	100
udns64	4 (1 node)	0.71	92
udns64	8 (1 node)	0.65	46
les64	1 (1 node)	5.53	100
les64	2 (1 node)	3.32	93
les64	4 (1 node)	2.86	85
les64	8 (1 node)	2.57	43
udns128	1 (1 node)	16.57	100
udns128	2 (1 node)	9.74	95
udns128	4 (1 node)	6.76	90
udns128	8 (1 node)	6.25	46

Table 2. Performance study for the PSM. % CPU refers to the utilization of individual CPUs as shown by the UNIX *top* command.

Table 2 illustrates that, using our cluster, the cost of the LES implementation in the spectral code is about three times the total run time without model, for the same mesh resolution. An evaluation of the time spent on inter-process communication is also performed by comparing performances using 8 cores located in the same node or on two different nodes. The main results obtained with the finite-volume code are given in Table 3.

The advantage of the FVM is that the DSM only represents 10.7% of the total run time which is much less than for the PSM. However, this comes at the cost of accuracy which is higher in the PSM (see Section 4.1). In fact, the FVM becomes competitive at very high mesh resolutions or in the case of complex geometries, which cannot be studied with a spectral approach. Moreover, the performance of the gigabit Ethernet connection is evaluated by comparing the run time of the $udns64$ case on 8 CPU's in one node with that in two nodes. Using the FVM, 19.4% of run time is saved when the 8 CPU's are in the same node. Finally, as the run time of the $udns64$ simulation on 4 CPU's is identical

Case	Number of CPU's	Run time per 10000 iterations [hours]	% CPU
udns64	1 (1 node)	22.5	100
udns64	2 (1 node)	14.58	100
udns64	4 (1 node)	8.61	100
udns64	8 (1 node)	6.94	100
udns64	8 (2 nodes)	8.61	85
les64	8 (1 node)	7.77	100
udns128	16 (2 nodes)	52.77	82

Table 3. Performance study for the FVM.

Case	R	Ha	Mesh resolution in DNS[2]	Mesh resolution in LES
case1	700	20	$256 \times 256 \times 256$	$64 \times 64 \times 64$
case2	700	30	$512 \times 256 \times 512$	$128 \times 128 \times 128$

Table 4. Results for turbulent Hartmann flows from pseudospectral code.

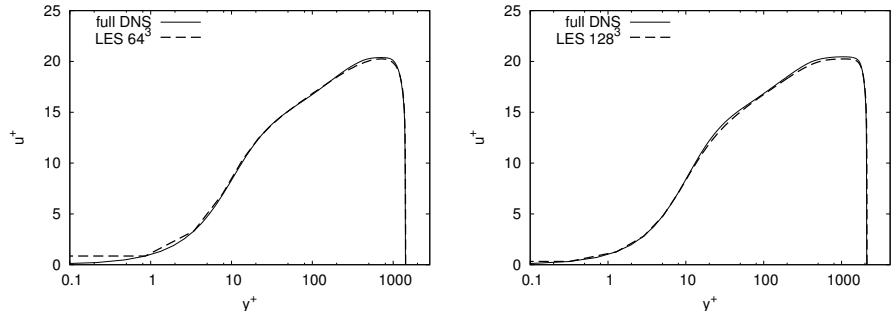


Figure 2. DNS vs. LES for turbulent Hartmann flows, mean velocity profiles in wall units for case1 (left) and case2 (right). DNS and LES resolutions are correspondingly 256^3 vs. 64^3 for the first case and $512^2 \times 256$ vs. 128^3 for the second one.

to that on 8 CPU's but on two nodes, the main bottleneck of the cluster is the time lost in network connections.

4.3 Large-Eddy Simulations of Hartmann Flow

In this section, the performance of the DSM is analysed in the case of the turbulent Hartmann flow. The two cases considered are presented in Table 4 and results are compared with the database made by Boeck *et al.*[2]. Using the DSM in LES runs, $case1$ and $case2$ yield to $Re_\tau = 707.4$ and $Re_\tau = 1064.3$, respectively, compared with $Re_\tau = 708.6$ and $Re_\tau = 1047.3$ for the corresponding DNS values. The performance of the SGS model is also illustrated by Fig. 2 which shows profiles of the mean velocity in wall units (i.e. non-dimensionalized by u_τ and δ). The comparison clearly indicates that LES mean velocities are in good agreement with the DNS ones. As a result, the standard DSM proves to be successful in modelling MHD flows using numerical grids about four times coarser than the DNS ones.

5 Conclusions

This work aims to evaluate the performance of the PSM and FVM, in terms of computational cost and accuracy. It reveals that, if the same mesh resolution is used in both approaches, the flexibility of the finite-volume code comes at the cost of accuracy. In other words, the FVM requires higher mesh resolutions than the spectral one to obtain the same

level of accuracy. For this reason, LES is likely to prove very useful to decrease the resolution requirements. In that perspective, this study shows that the advantage of the FVM is that the additional cost introduced by the LES model is not significant compared with the total run time. In particular, for a standard LES test case, Section 4.2 shows that the cost of the LES model is about 300% using the spectral code compared with 10% with the finite-volume method. As a consequence, the FVM becomes competitive at very high resolution simulations. However, because of the loss of accuracy in the finite-volume approach, attention has to be paid to avoid the numerical dissipation from becoming higher than the LES contribution. In this context, further work needs to be done to evaluate the balance between numerical and subgrid-scale dissipations.

Acknowledgements

TB and DK acknowledge financial support from the DFG in the framework of the Emmy-Noether program (grant Bo1668/2-2) and computer resources provided by the NIC. AV is supported by a F.R.I.A. fellowship. BK acknowledges financial support through the EU-RYI (European Young Investigator) Award “Modelling and simulation of turbulent conductive flows in the limit of low magnetic Reynolds number”.

References

1. P. A. Davidson, *Magnetohydrodynamics in materials processing*, Annu. Rev. Fluid Mech., **31**, 273–300, (1999).
2. T. Boeck, D. Krasnov and E. Zienicke, *Numerical study of turbulent magnetohydrodynamic channel flow*, J. Fluid Mech., **572**, 179–188, (2007).
3. B. Knaepen and P. Moin, *Large-eddy simulation of conductive flows at low magnetic Reynolds number*, Phys. Fluids, **16**, 1255–1261, (2004).
4. M. Germano, U. Piomelli, P. Moin and W. H. Cabot, *A dynamic subgrid-scale eddy viscosity model*, Phys. Fluids, **A 3**, 1760–1765, (1991).
5. D. K. Lilly, *A proposed modification of the Germano subgrid-scale closure method*, Phys. Fluids, **A 4**, 633–635, (1992).
6. T. Boeck, D. Krasnov, M. Rossi, O. Zikanov and B. Knaepen, *Transition to turbulence in MHD channel flow with spanwise magnetic field*, in: Proc. Summer Programm 2006, Center for Turbulence Research, Stanford University, (2006).
7. F. N. Felten and T. S. Lund, *Kinetic energy conservation issues associated with the collocated mesh scheme for incompressible flow*, J. Comp. Phys., **215**, 465–484, (2006).
8. F. Ham and G. Iaccarino, *Energy conservation in collocated discretization schemes on unstructured meshes*, Annual Research Briefs, Center for Turbulence Research, Stanford University, 3–14, (2004).
9. K. Mahesh, G. Constantinescu and P. Moin, *A numerical method for large-eddy simulation in complex geometries*, J. Comp. Phys., **197**, 215–240, (2004).
10. J. Kim and P. Moin, *Application of a fractional-step method to incompressible Navier–Stokes equations*, J. Comp. Phys., **59**, 308–323, (1985).
11. J. Kim, P. Moin and R. Moser, *Turbulence statistics in fully developed channel flow at low Reynolds number*, J. Fluid Mech., **177**, 133–166, (1987).

Pseudo-Spectral Modeling in Geodynamo

Maxim Reshetnyak and Bernhard Steffen

Jülich Supercomputing Centre
Research Centre Jülich, 52425 Jülich, Germany
E-mail: {*b.steffen, m.reshetnyak*}*@fz-juelich.de*

Many stars and planets have magnetic fields. The heat flux causes 3D convection of plasma or metal, which can generate a large-scale magnetic field like that observed. The small-scale behaviour, demonstrating self-similarity in a wide range of the spatial and temporal scales, is a field of active research using modelling, as it is usually not observed.
Rapid rotation gives a geostrophic system, where convection degenerates in the direction of the axis of rotation and all variation along this axis is weak. Such a system is somewhere in between the full 3D and 2D-systems. Its special properties show up in the physical and the spectral space simultaneously. Pseudo-spectral modelling solves the PDE in the spectral space for easy calculations of integrals and derivatives. The nonlinear terms are calculated physical space, requiring many direct and inverse FFTs per time step. We apply this technique to the thermal convection problem with heating from below in a Cartesian box. Above a threshold of the kinetic energy the system generates the magnetic field.
The most time consuming part of our MPI code is FFT transforms. For efficiency, we selected a FFT library which makes use of the symmetry of the fields. The optimal number of processors is $\sim$ half the number of grid planes, with superlinear speedup. The single node performance is poor, each processor delivering only $\sim$ 5% of its peak rate.
We see cyclonic convection with a cyclone density of the $\sim\ E^{-1/3}$ (E Ekman number $\sim 10^{-15}$ for the Earth). This causes a high anisotropy of the convection even for high Reynolds numbers. Our simulations demonstrates the generation of the large-scale hydrodynamic helicity. Helicity is an integral of the Navier-Stokes equation, and it has close relation to the α-effect which generates the large scale magnetic field via the small-scale turbulence. The magnetic field grows exponentially from a small seed and finally levels off, when it damps primarily the toroidal part of velocity field.
Convection and dynamo systems are dissipative, so no equilibrium is reached but a state of very slow decay. The kinetic energy is injected into the system at the medium scale of cyclons, one sink of energy is at the small viscous scale, another at the large (magnetic field) scale. For some (small) scales the cascade of the energy is direct (like it is in the Kolmogorov's like turbulence), for others (larger than cyclones) it is inverse, like it is observed in 2D turbulence. At the small scales there is a constant energy flux, as is plausible from theory as well as from semi-empirical models[1].

1 Introduction

Almost all stars, the earth, and the planets larger than earth have large scale magnetic fields that are believed to be generated by a common universal mechanism - the conversion of kinetic energy into magnetic energy in a turbulent rotating shell. The details, however, - and thus the nature of the resulting field - differ greatly. The only fields observable with good accuracy are that of the earth and of the sun. The challenge for the dynamo theory is to provide a model that can explain the visible features of the field with realistic assumptions on the model parameters. Calculations for the entire star or planets are done either with spectral models[2] or finite volume methods[3], and have demonstrated beyond reasonable doubt that the turbulent 3D convection of the conductive fluid -in the core for earth, in an upper shell for the sun - can generate a large scale magnetic field similar to the

one observed out of small random fluctuations. However, both these methods cannot cover the enormous span of scales required for a realistic parameter set. For the geodynamo, the time scale of the large scale convection is $\sim 10^3$ years, during which the planet itself makes $\sim 10^6$ revolutions. Further on, viscosity operates at a scale of centimetres, compared to the convective scale of $\sim 10^6$ meters. Thus the effects of the small scale processes have to be averaged and transported to the finest scale resolved, which will be orders of magnitude larger. For the sun and other stars, the situation is not better, the difference of scales being frequently even larger. To verify the averaging approaches and to understand the interactions of the different scales of turbulence, calculations of small parts of the earth' core are required, at least partially bridging the gap in scale. For this, we look at a the thermal convection problem (in Boussinesque approximation) with heating below in a Cartesian box at the equatorial plane. This complements the calculations for the entire earth done by the other methods towards smaller scales.

2 The Equations

The geodynamo equations for the incompressible fluid ($\nabla \cdot \mathbf{V} = 0$) in the volume of the scale L rotating with the angular velocity Ω in the Cartesian system of coordinates $(x,\, y,\, z)$ in its traditional dimensionless form in physical space can be written as follows:

$$\begin{aligned}
&\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{V} \times \mathbf{B}) + \mathrm{q}^{-1}\, \Delta \mathbf{B} \\
&\mathrm{E}\,\mathrm{Pr}^{-1} \left[\frac{\partial \mathbf{V}}{\partial t} + (\mathbf{V} \cdot \nabla)\, \mathbf{V} \right] = -\nabla P - \mathbf{1_z} \times \mathbf{V} + \mathrm{Ra}\, T\, z \mathbf{1_z} + (\nabla \times \mathbf{B}) \times \mathbf{B} + \mathrm{E}\, \Delta \mathbf{V} \\
&\frac{\partial T}{\partial t} + (\mathbf{V} \cdot \nabla)\, (T + T_0) = \Delta T + G(\mathbf{r}).
\end{aligned} \tag{2.1}$$

The velocity $\mathbf{V}$, magnetic field $\mathbf{B}$, pressure $P = p + \mathrm{E}\,\mathrm{Pr}^{-1}\, V^2/2$ and typical diffusion time t are measured in units of κ/L, $\sqrt{2\Omega\kappa\mu\rho}$;, $\rho\kappa^2/\mathrm{L}^2$ and L^2/κ respectively, where κ is thermal diffusivity, ρ is density, μ permeability, $\mathrm{Pr} = \dfrac{\kappa}{\nu}$ is the Prandtl number, $\mathrm{E} = \dfrac{\nu}{2\Omega L^2}$ is the Ekman number, ν is kinematic viscosity, η is the magnetic diffusivity, and $\mathrm{q} = \kappa/\eta$ is the Roberts number. $\mathrm{Ra} = \dfrac{\alpha g_0 \delta T L}{2\Omega\kappa}$ is the modified Rayleigh number, α is the coefficient of volume expansion, δT is the unit of temperature (see for more details[4]), g_0 is the gravitational acceleration, and G is the heat source.

For boundary conditions, we assume a temperature gradient from the lower to the upper boundary large enough to drive a convection, and periodicity otherwise, which leads to odd or even symmetry relative to the equator.

Some features of the field can be seen from a dimensional analysis already. The extreme smallness of the Ekman number means that the terms without E must almost cancel out, which without magnetic field results in a geostrophic balance where all fluxes transverse to the rotation are blocked by the Coriolis force and cyclonic motion is generated. With the magnetic field, a magnetostrophic balance or a mixture seems possible, depending on the scale considered.

The direct numerical integration of these equations is not efficient, much larger time steps and easier calculations can be achieved by transforming the equations into wave

space[5]. Also, some features of the solution show up much better in wave space than in physical space, and especially the separation of scales becomes clear there. Because the calculations of the nonlinear terms in wave space would be full matrix operations, they have to be done in physical space, requiring FFT transform of all quantities in every time step, which will dominate the computing time. This pseudospectral method could also be applied to a box in spherical coordinates, but has problems at the rotaton axis.

After elimination of the pressure using $\mathbf{k} \cdot \mathbf{V} = 0$, $\mathbf{k} \cdot \mathbf{B} = 0$ and transforming the equations into wave space we come to[6]

$$\begin{aligned} \left[\frac{\partial \mathbf{B}}{\partial t} + \mathrm{q}^{-1}\, k^2 \mathbf{B}\right]_{\mathbf{k}} &= \left[\nabla \times (\mathbf{V} \times \mathbf{B})\right]_{\mathbf{k}} \\ \mathrm{E}\left[\mathrm{Pr}^{-1}\, \frac{\partial \mathbf{V}}{\partial t} + k^2 \mathbf{V}\right]_{\mathbf{k}} &= \mathbf{k}\mathcal{P}_{\mathbf{k}} + \mathbf{F}_{\mathbf{k}} \\ \left[\frac{\partial T}{\partial t} + k^2 T\right]_{\mathbf{k}} &= -\left[(\mathbf{V} \cdot \nabla)\, T\right]_{\mathbf{k}} + G(\mathbf{k}) \end{aligned} \tag{2.2}$$

$$\begin{aligned} &\text{with} \qquad \mathcal{P}_{\mathbf{k}} = -\frac{\mathbf{k} \cdot \mathbf{F}_{\mathbf{k}}}{k^2}, \qquad k^2 = k_\beta k_\beta, \qquad \beta = 1 \ldots 3 \\ &\mathbf{F}_{\mathbf{k}} = \left[\mathrm{Pr}^{-1}\, \mathbf{V} \times (\nabla \times \mathbf{V}) + \mathrm{Ra}\, T \mathbf{1}_{\mathbf{z}} - \mathbf{1}_{\mathbf{z}} \times \mathbf{V} + (\mathbf{B} \cdot \nabla)\, \mathbf{B}\right]_{\mathbf{k}}. \end{aligned} \tag{2.3}$$

For integration in time we use explicit Adams-Bashforth (AB2) scheme for the non-linear terms. The linear terms are treated using the Crank-Nicolson (CN) scheme. To resolve the diffusion terms we used a well known trick which helps to increase the time step significantly. We rewrite $\frac{\partial A}{\partial t} + k^2 A = U$ in the form $\frac{\partial A e^{k^0 \gamma t}}{\partial t} = U\, e^{k^2 \gamma t}$ and then apply the CN scheme.

3 Parallel FFT for the Equations

The program shows slightly superlinear speedup up to 9 processors (with 128 physical planes) and more than 90% parallel efficiency on 33 processors, so parallelizing is not a problem. However, the single node performance is a problem. The Fourier transforms dominate the computing times, in the implementation used they make up about 85%, so the FFTs are subject careful inspection. Special tests were done on a physical grid of 128*129*128 real values, (second dimension is logically 256 and even), and using 4 processors of jump for the calculations, both direct and inverse Fourier transform take slightly less than 6 ms each, with the difference less than the variations in the timings.

The direct transforms is a transform from 3D real with symmetry in the second coordinate (due to the conventional placement of coordinates in geophysics), size nx*ny*nz, into a 3D complex array of slightly more than half the size, if redundancies are eliminated. There is no software available doing exactly this, not even sequentially. Therefore the FFT was put together from the sequential 1D FFT provided by[7] based on[8,9] , and some data movements in the following (natural) order: The data comes distributed on the processors along the last component. As a first step, nx*nz transforms along the second component - real even symmetric to real even symmetric or real odd symmetric to purely imaginary odd

symmetric - are done using routine RDCT. (cosine transform) or RDST (Sine transform). After this, the array is transformed, such that now it is split along the second component. During the transform using $mpi\_alltoallv$, the second and third indices are exchanged for better access. Then nx*ny complex transforms in direction z follow, and finally ny*nz transforms in direction x.
This procedure was chosen for ease of development and flexibility, it is reasonably fast but not optimal. An analysis using hardware counter monitoring (hpmcount,[10]) gave the following result:

	Cosine Trans	FFT compl	reshape for alltoall	MPI_alltoallv
load and store ops	18.6 M	36.5 M	4.2 M	0.065 M
Instr. per cycle	0.60	0.95	0.82	1.310
MFlips per sec	206	302	21.2	0.0
user time	0.0516	0.0782	0.0098	0.006
Loads per TLB miss	3976	257	21667	22464
L2 cache miss rate	0.024	0.116	0.053	0.048

Both the cosine transform and the FFT include collecting the data to a stride 1 vector for every transform. According to gprof[11] data, in a longer run the 3D FFT takes 34.5 s, of which only 14 s are spend in the 1D transforms, and 21.5 s in moving data (including the MPI call). It is now clear that the actual FFT is not more than half the time, assembling the data is of the same order, even though the processor efficiency of the FFT code is low, it runs at $\sim 5\%$ of the peak performance. The main problem seems to be the TLB miss rate, though L2 and L3 cache misses are quite high, also. The inverse FFT performs the inverse operations in inverse order, the timing is almost the same.
For optimization of the FFT, the first approach is using a better FFT routine. since the program development, a new release of FFTW[12] has appeared which includes real symmetric FFT that was previously missing. It also contains provisions to do the ny plane FFTs in one step. Tests show FFTW to be faster than the FFT used till now.
A second step is moving from complex-to-complex plane transforms to real-to-complex, as the result of the first transform is real or purely imaginary. This saves some data handling and half the communication volume. For the even symmetry the process is straightforward, for the odd symmetry, the final result has to be multiplied with the imaginary unit, which costs one sweep over the data.
The third step is arranging the data in a way that memory access comes with as small a stride as possible. This means arranging the coordinates in an order different from the customary one, which is not an option during development of a program. It also requires changes in almost any part of the program, not only in the two subroutines organizing the FFT. All these together may cut the time for the FFT in half, which still means that it is the dominating part, so optimizing the implementation of other parts is not reasonable.

While the tests were performed using a small number of processors, production runs use much more, such that each processor may contain only two or four planes of the grid for every quantity. This does effect our discussion above only with respect to the array transform, which needs more but smaller messages for a larger number of processors. However, the communication time stays small up to 64 processors. Obviously, having

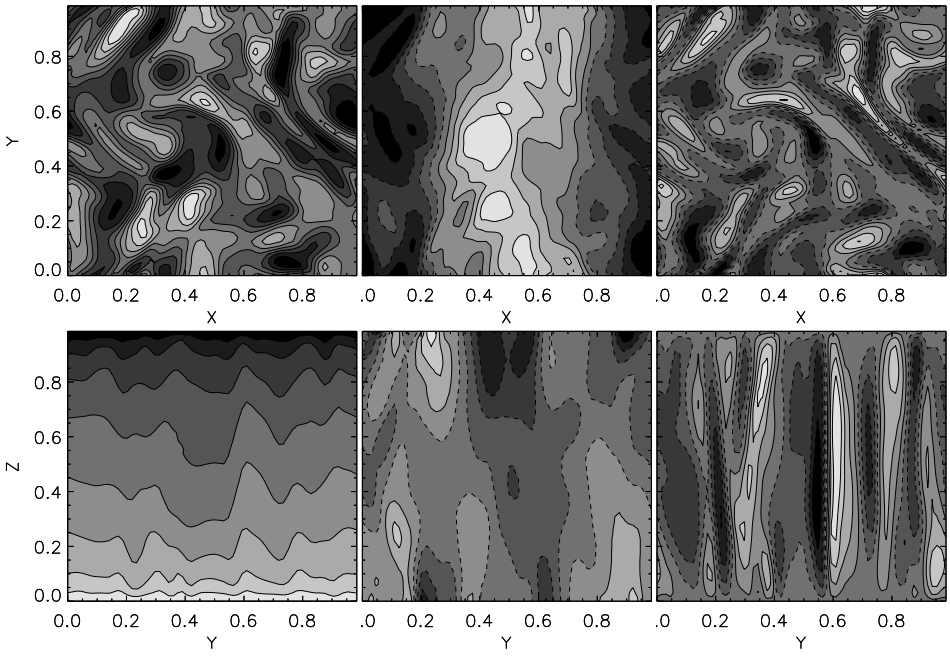


Figure 1. Thermal convection with rapid rotation ($N = nx = ny = nz = 64$) $\mathrm{Pr} = 1$, $\mathrm{Ra} = 1.3\,10^3$, $\mathrm{E} = 2\,10^{-5}$. for T (left column) (0, 1,0.43, 0.60), V_y (middle) (-745, 548, -904, 979) and V_z (right) (-510, 501,-651, 784). The upper line corresponds to the section $z = 0.5$, and the lower to $x = 0$. The numbers are (min,max) values of the fields.

more processors than planes in the grid would require a completely different organization.

4 Results

4.1 Pure Convection, no Magnetic Field

The thermal convection with a rapid rotation ($\mathrm{Ro} \ll 1$) is characterized by a large number of the vertical columns (cyclones and anticyclones). Their number depends on the Ekman number as $k_c \sim \mathrm{E}^{-1/3}$[13]. For the liquid core of the Earth $\mathrm{E} \sim 10^{-15}$, what is obviously prohibits simulations for the realistic range of parameters. Usually, one reach only with a $\mathrm{E} = 10^{-4} \div 10^{-6}$[4]. The second important parameter is a Rayleigh number describing intensity of the the heat sources. The critical Rayleigh number (when convection starts) depends on E as $\mathrm{Ra}^{\mathrm{cr}} \sim \mathrm{E}^{-1/3}$. Further we consider three regimes:

NR: Regime without rotation, $\mathrm{Ra} = 9 \cdot 10^5$, $\mathrm{Pr} = 1$, $\mathrm{E} = 1$, $\mathrm{Re} \sim 700$.

R1: Regime with rotation, $\mathrm{Ra} = 4 \cdot 10^2$, $\mathrm{Pr} = 1$, $\mathrm{E} = 2 \cdot 10^{-5}$, $\mathrm{Re} \sim 200$.

R2: Regime with rotation, $\mathrm{Ra} = 1 \cdot 10^3$, $\mathrm{Pr} = 1$, $\mathrm{E} = 2 \cdot 10^{-5}$, $\mathrm{Re} \sim 10^4$.

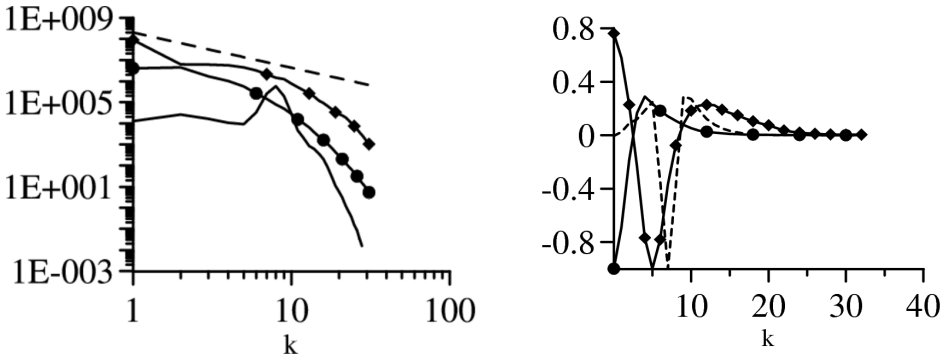


Figure 2. On the left is spectra of the kinetic energy for NR (circles), R1, R2 (diamonds). The dot line corresponds to the Kolmogorov's spectrum $\sim k^{-5/3}$, the solid line is $\sim k^{-3}$. On the right is a flow of the kinetic energy $\mathcal{T}$ in the wave space for NR (circle), R1 (dotted line), R2 (diamonds).

R1 corresponds to geostrophic state in vicinity of the threshold of generation with a typical regular columnar structure. Increase of Ra (regime R2) breaks regular structure, appearance of the small-scale flows, deviation from geostrophy (quasigeostrophic state). The non-linear term in the is comparable with the Coriolis force and pressure gradient. The temporal behaviour becomes chaotic.

Now we consider spectra of the kinetic energy Fig. 2. The spectrum without rotation NR is similar to the Kolmogorov's one. The marginal regime R1 has clear maximum at $k_c \sim 8$. Increase of Ra (R2) leads to the fill of the gap in the spectra at $k < k_c$ and spectra tends to the non-rotating spectra. However observable similarity of R2 and NR regimes does not mean similarity of the intrinsic physical processes. Thus, in the two-dimensional turbulence the spectra of the kinetic energy with a $\sim k^{-5/3}$ is observable also, however the energy transfer in contrast to the Kolmogorov's one (with a direct cascade) comes from the small scales to the large ones[14].

Consider what happens with a flow of the kinetic energy in the wave space. The non-rotating regime demonstrates well-known scenario of the Kolmogorov's direct cascade Fig. 2. For the large scales $\mathcal{T} < 0$: these scales are the sources of the energy. Coming to the infrared part of the spectra sign of the flow changes its sign to the positive: here is a sink of the energy. Note, that for the two-dimensional turbulence the mirror-symmetrical picture is observable[14] (the inverse cascade).

Rotation essentially changes behaviour of $\mathcal{T}$ in k-space. Now, the source of the energy corresponds to k_c. For $k > k_c$ we observe a direct cascade of the energy $\mathcal{T} > 0$. For $k < k_c$ picture is more complex: for the first wave numbers there is an inverse cascade $\mathcal{T} > 0$, while in the large part of the wave region $k < k_c$ the cascade is still direct $\mathcal{T} < 0$. Increasing Re leads to a narrowing of the inverse cascade region. Probably the appearance of the inverse cascade connected with a non-local interaction in k-space[15].

4.2 Full Dynamo Regime

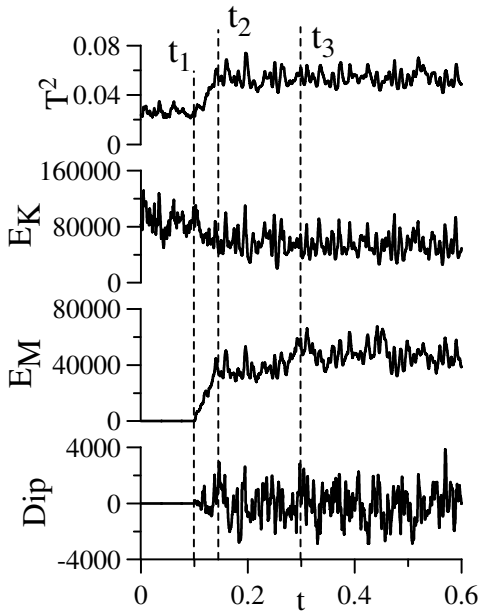


Figure 3. Evolution of the integral quantities (from the top): T^2, kinetic energy E_K, magnetic energy E_M, vertical dipole Dip for $\mathrm{E} = 10^{-4}$, $\mathrm{Pr} = 1$, $\mathrm{Ra} = 6 \cdot 10^2$, $\mathrm{q} = 10$. At t_1 we inject magnetic field. t_3 corresponds to an equilibrium regime, when growth of the magnetic field stops.

The magnetic field generation is critical phenomenon, its starts when the magnetic Reynolds number $\mathrm{R_m} = \mathrm{R_m}^{cr}$. Without feed back of the magnetic field on the flow (kinematic regime KC) magnetic energy grows $\mathrm{R_m} > \mathrm{R_m}^{cr}$ (or decays $\mathrm{R_m} < \mathrm{R_m}^{cr}$) exponentially in time. In Fig. 3 we present transition of the system from the pure convection regime ($0 < t < t_1$) to the KC regime ($t_1 < t < t_2$) and then to the full dynamo regime ($t > t_2$). It is clear, that influence of magnetic field onto the flow appears in decrease of $\mathrm{R_m}$. The more detailed analysis reveals that mainly the toroidal component of the kinetic energy is suppresses. In its turn, decrease of the toroidal velocity leads to decrease of the hydrodynamic helicity. So far the mean helicity is a source of the mean magnetic field[16] the suppression of the magnetic field generation starts.

5 Concluding Remarks

The magnetic fields of the planets are the main sources of information on the processes in the liquid cores at the times $10^2 - 10^3$yy and more. If the poloidal part of the magnetic field is observable at the planet surface, then the largest component of field (toroidal) as well as the kinetic energy distribution over the scales is absolutely invisible for the observer out of the core. Moreover, due to finite conductivity of the mantle even the poloidal part of the magnetic field is cut off at $k \ll k_c$. In other words the observable part of the spectra at the planets surface is only a small part (not even the largest) of the whole spectra of the field. That is why importance of the numerical simulation is difficult to overestimate. Here we showed that rapid rotation leads to the very interesting phenomenon: inverse cascade in

the kinetic energy transfer over the scales. The other interesting point is a hydrodynamic helicity suppression by a growing magnetic field which is key to understanding of how the dynamo system reaches the equilibrium state.

Acknowledgements

The computations were performed with a grant of computer time provided by the VSR of the Research Centre Jülich.

References

1. U. Frisch, *Turbulence: the Legacy of A. N. Kolmogorov*, (Cambridge University Press, 1995).
2. M. Kono and P. Roberts, *Recent geodynamo simulations and observations of the geomagnetic field*, Reviews of Geophysics, **40**, B1–B41, (2002)
3. M. Reshetnyak and B. Steffen, *Dynamo model in the spherical shell*, Numerical Methods and Programming, **6**, 27–32, (2005). http://www.srcc.msu.su/num-meth/english/index.html
4. C. A. Jones, *Convection-driven geodynamo models*, Phil. Trans. R. Soc. London, **A 358**, 873–897, (2000).
5. S. A. Orszag, *Numerical simulation of incompressible flows within simple boundaries. I. Galerkin (spectral) representations*, Stud. Appl. Math. **51**, 293–327, (1971).
6. B. Buffett, *A comparison of subgrid-scale models for large-eddy simulations of convection in the Earth's core*, Geophys. J. Int., **153**, 753–765, (2003)
7. S. Kifowit, http://faculty.prairiestate.edu/skifowit/fft
8. P. N. Swarztrauber, *Symmetric FFTs*, Mathematics of Computation, **47**, 323–346, (1986).
9. H. V. Sorensen, D. L. Jones, M. T. Heideman and C. S. Burrus, *Real-valued Fast Fourier Transform algorithms*, IEEE Trans. Acoust., Speech, Signal Processing, **35**, 849-863, (1987).
10. L. DeRose, *Hardware Performance Monitor (HPM) Toolkit* (C) IBM, (2002).
11. *Performance Tools Guide and Reference* (C) IBM, (2002)
12. M. Frigo and S. G. Johnson, *The design and implementation of FFTW3*, Proc. IEEE **93**, 216–231, (2005).
13. S. Chandrasekhar, *Hydrodynamics and hydromagnetic stability*, (Dover, New York, 1961).
14. R. H. Kraichnan and D. Montgomery, *Two-dimensional turbulence*, Rep. Prog. Phys. **43**, 547–619, (1980).
15. F. Waleffe, *The nature of triad interactions in homogeneous turbulence*, Phys. Fluids. A, **4**, 350–363, (1992).
16. F. Krause and K.-H. Rädler, *Mean field magnetohydrodynamics and dynamo theory*, (Akademie-Verlag, Berlin, 1980).

Parallel Tools and Middleware

Design and Implementation of a General-Purpose API of Progress and Performance Indicators

Ivan Rodero, Francesc Guim, Julita Corbalan, and Jesús Labarta

Barcelona Supercomputing Center
Technical University of Catalonia
Jordi Girona 29, 08034 Barcelona, Spain
E-mail: {*ivan.rodero, francesc.guim, julita.corbalan, jesus.labarta*}*@bsc.es*

In High Performance Computing centres, queueing systems are used by the users to access and manage the HPC resources through a set of interfaces. After job submission, users lose control of the job and they only have a very restricted set of interfaces for accessing data concerning the performance and progress of job events. In this paper we present a general-purpose API to implement progress and performance indicators of individual applications. The API is generic and it is designed to be used at different levels, from the operating system to a grid portal. We also present two additional components built on top of the API and their use in the HPC-Europa portal. Furthermore, we discuss how to use the proposed API and tools in the eNANOS project to implement scheduling policies based on dynamic load balancing techniques and self tuning in run time.

1 Introduction

In High Performance Computing (HPC) centres, queueing systems are used by the users to access the HPC resources. They provide interfaces that allow users to submit jobs, track the jobs during their execution and carry out actions on the jobs (i.e. cancel or resume). For example, in LoadLeveler the llsubmit command is used to submit an LL script to the system. Once the job is queued, and the scheduler decides to start it, it is mapped to the resources by the corresponding resource manager.

After job submission, users lose control of the job and they only dispose of a very restricted set of interfaces for accessing data concerning the performance, or progress or job events. In this situation, the queueing system only provides a list of the submitted jobs and some information about them, such as the job status or the running time. Although this is the scenario in almost all the HPC centres, there is information about the jobs that have been submitted that is missing but required by the users. For example, they want to know when their application will start running, once started, when it will finish, how much time remains, and if their application is performing well enough.

If experienced users could obtain such information during the job run time, they would be able to take decisions that would improve system behaviour. For example, if an application is not achieving the expected performance, the user can decide to adjust some parameters on run time, or even resubmit this application with new parameters. In the worst case, if an application achieves low performance during its execution, it is cancelled by the system because of a wall clock limit timeout and thus consumes resources unnecessarily.

In this paper we present a general-purpose API which can implement progress and provide performance indicators of individual applications. The API is generic and it is designed to be used at different levels, from the operating system to a grid portal. The

design can also be extended, and the development of new services on top the API are easy to develop. The implementation is done through a lightweight library to avoid important overheads and starvation with the running application. We also present two additional components built on top of the API and explain how they ares in the HPC-Europa portal, which is a production testbed composed of HPC centres. The API and the additional tools can be used in both sequential and parallel applications (MPI, OpenMP and mixed MPI+OpenMP). Furthermore, we discuss how to use the proposed API and tools in the eNANOS project[11] to implement scheduling policies based on dynamic load balancing techniques and self tuning in run time, to improve the behaviour of the applications and optimize the use of resources.

In the following sections we describe the proposed API, some tools developed on top of the API, its design and implementation, some uses of these tools, and we discuss the benefits of using them in production systems.

2 Related Work

In both literature and production systems we can find several mechanisms which allow users to monitor their applications. Firstly, we can find mechanisms offered by operating system, such as the `/proc` file system in Unix and Linux systems, or the typical `ps` command. These mechanisms allow the user with information about what is happening during the execution of the application, basically in terms of time and resources used. In addition, several monitoring systems have been developed in different areas. The Performance API (PAPI) project[9] specifies a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. Ganglia[3] is a scalable distributed monitoring system for high-performance computing systems, typically for clusters. Other monitoring tools, such as Mercury[1] or NWS[14] are designed to satisfy requirements of grid performance monitoring providing monitoring data with the metrics.

We have found some similarities between the presented work in this paper and other monitoring APIs such as SAGA[4]. However, these efforts do not exactly fit our objectives and requirements, since with these existing monitoring tools it is difficult to give users a real idea of the progress and behaviour of their applications, while for us this is essential. In our approach we try to provide a more generic tool, open, lightweight, not OS-dependent, and easy to use and to extend in different contexts (from a single node to a grid environment).

3 Architecture and Design

The API has 9 main methods which provide information about the progress of the application and application performance. Furthermore, its generic design allows the developer to publish any kind of information concerning the job/application, independent of its nature and content. The API specification if presented below.

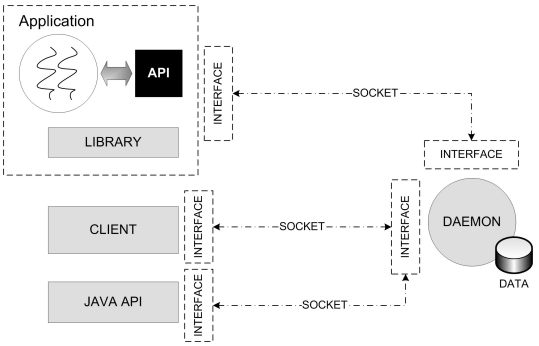


Figure 1. Overall architecture

```
int    CONNECT (char *appl_name)
/*Establishes a connection with the daemon for a particular application*/
int    DISCONNECT (void)
/*Closes a connection with the daemon*/
pi_id INIT_PROG_INDICATOR (char *name, pi_desc *description)
/*Initializes a progress indicator with the specified name & description
    */
int    INC_PROG_INDICATOR (pi_id id , pi_value *value)
/*Increments a progress indicator with the specified value*/
int    DEC_PROG_INDICATOR (pi_id id, pi_value *value)
/*Decrements a progress indicator with the specified value*/
int    SET_PROG_INDICATOR (pi_id id, pi_value *value)
/*Sets the value of a progress indicator with the specified value*/
int    RESET_PROG_INDICATOR (pi_id id)
/*Sets the default value of a progress indicator*/
int    GET_PROG_INDICATOR (pi_id id, pi_value *value)
/*Returns the value of the specified progress indicator by reference*/
int    REL_PROG_INDICADOR (pi_id id)
/*Releases a progress indicator. The identifier can be reused*/
```

We have also included the CONNECT_PID method, which takes into account the PID of the application. This method is specially useful when the real user of the progress and performance API is an application itself (a good example is the implementation of the tool constructed on top of the /proc file system). Its interface is shown below.

```
int    CONNECT_PID(char *appl_name, pid_t identifier)
/* Establishes a connection for a particular application & PID */
```

We present the architecture of the whole system in Fig. 1. The application uses the API and it is linked to a library which implements the API and provides an interface to connect the application to a daemon which manages the rest of the components. The daemon is the main component of the system and, basically, it is a multi-threaded socket server that gives service to the clients, to the library and to other external tools developed on top of the API. Apart from managing all the connections, it stores all the data and provides access to it.

We have implemented a client interface that can be used through a command line tool, and can also be used through an administration tool to start, stop and manage the main

functionality of the daemon. Moreover, the system provides a Java API that allow external Java applications to access the system functionality. This API is specially useful to implement services that use the progress and performance system in higher layers such as web and grid services (an example of this approach is presented later with the HPC-Europa portal). All the interface communications are implemented with sockets to ensure extensibility and flexibility; furthermore, it enhances efficiency.

The main data structures used in the whole system are shown below. For simplicity and efficiency, a progress or performance indicator is defined as a generic metric with some restrictions in terms of data types and the information that can include.

```
typedef int pi_id;
typedef enum {ABSOLUTE=1, RELATIVE} pi_type;
typedef enum {INTEGER=1, STRING, DOUBLE, LONG} TOD;
typedef union{     int        int_value;
                   char       str_value[BUF_SIZE]; /*e.g. BUF_SIZE=256*/
                   double     double_value;
                   long long  long_value;
             } pi_value;
typedef struct _pi_desc{     pi_value  init_value;
                             pi_type   type;
                             TOD       tod;
                         } pi_desc;
```

The “pi_id” is defined as an integer type and identifies the progress indicators. The “pi_type” define the kind of progress indicator and include both relative and absolute indicators. An example of an absolute indicator is the number of completed iterations of a loop and an example of a relative indicator is the percentage of completed iterations of this loop. “TOD”, that is the acronym of Type Of Data, defines the kind of the progress indicator data. The “pi_value” stores the value of the progress indicator depending on the data type specified in the “tod” structure. The full description of a progress indicator is composed of the type of indicator (pi_type), its type of data (TOD), and an initial value (for example 0 or an empty string).

The constructor of the metric class of the SAGA monitoring model[4] is shown below. The metric specification of SAGA seems even more generic than the API of the progress indicators because it includes some more attributes and uses the String data type for each of those attributes. However, as our daemon returns the information in XML format, we are able to provide more complex information. Thus, when the client requests information about a given indicator, the daemon provides information about the metric, plus some extra but useful information, for instance the application process ID or the hostname.

```
CONSTRUCTOR   ( in   string    name,
                in   string    desc,
                in   string    mode,
                in   string    unit,
                in   string    type,
                in   string    value,
                out  metric    metric);
```

The XML schema of the data obtained from the daemon with the client interfaces is shown in Fig. 2. It is composed of a set of connections (PI_connection). Each connection has a name to identify it, the name of the host in which the application is running (the

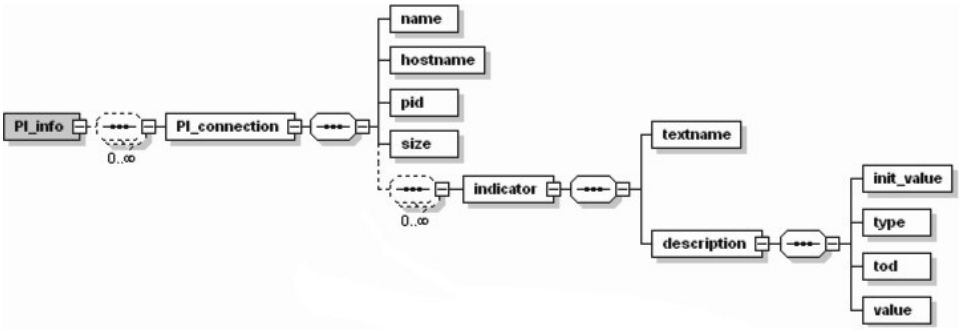


Figure 2. XML schema of the progress indicators data

master in the case of MPI), the ID process of the application, the number of indicators, and a set of indicators. An indicator has a name that identifies it, and a full description as presented previously. We have chosen XML to describe the information because it is a standard and can help external tools to manipulate the data.

4 Implementation

As well as implementing the API, we have developed a set of components on top of the main infrastructure presented above. Firstly, we have implemented a tool, called nanos-ind_proc, on top the API. It is designed for Linux systems, and provides a set of progress indicators with no need to modify the original application or to link it with any library. The nanos-ind_proc is an external process, and it requires the PID the application that will be monitored. It collects the information regarding the specified PID from the /proc file system, creates a set of progress indicators, and updates these indicators through a pooling mechanism. The pooling is configured by parameters when the nanos-ind_proc is executed. In this module it is essential to make the connection by using the method that has an input the PID. This is because the PID is used to map the information obtained from the /proc with the progress indicators. Using this mechanism we can define indicators of any of the information contained in the /proc, for example the user and system time or the memory usage. In case of MPI applications, the API is used for each process separately and the information is merged by the API daemon. The basic schema of this component is shown in Fig. 3.

Secondly, we have developed another external tool on top of the API to obtain information regarding the application hardware counters. With the current implementation, the original application only requires a very small modification. It only needs to add a call to a new method at the beginning and at the end of the application. In fact, the user only has to include a call to the nanos_ind_hwc_init and nanos_ind_hwc_close methods provided by a new library (nanos-ind_hwc). This library spawns a new thread (pthread) that creates a set of progress indicators (in this case performance indicators), initializes and gets the value of hardware counters, and updates the value of the initialized hardware counters through the progress indicators. We have developed two different implementations of this tool, one for Linux and another for AIX. For Linux we have used PAPI[9] to get

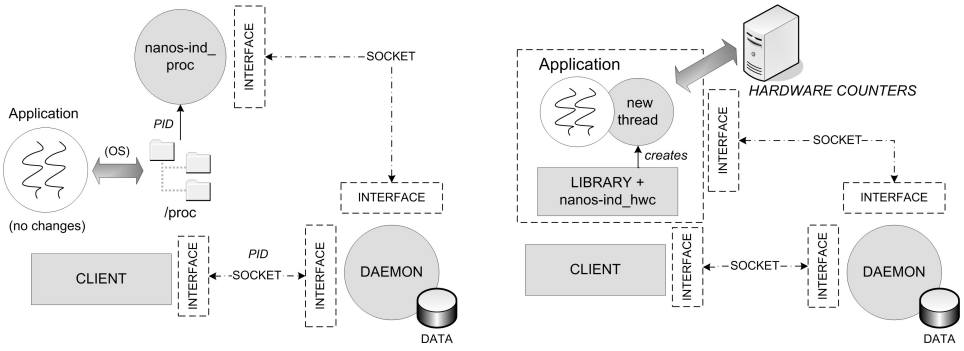


Figure 3. Schema of the "/proc" module (left), and schema of the hardware counters module (right)

and manage the hardware counters, and for AIX we have used PMAPI. Some examples of metrics that can be implemented with the performance indicators are the number of MIPS or MFLOPS that the application is achieving (absolute value) or the percentage of MFLOPS (respect the maximum possible in the resource) that the application is achieving. The hardware counters required in PAPI to calculate these metrics are: PAPI_FP_OPS (number of floating-point operations), PAPI_TOT_INS (total number of instructions) and PAPI_TOT_CYC (total number of cycles). In the case of MPI+OpenMP parallel applications, we monitor the MPI processes from the MPI master process, and we use collective metrics for the OpenMP threads. The basic schema of this component if shown in Fig. 3.

5 The HPC-Europa Testbed: a Case Study

In addition to developing some tools on top the API, we have implemented a Java API to allow grid services to access the API progress indicators – see below.

```
String getDaemonInfo(String daemonHostname)
/* Gets all the information from the daemon in the specified host */
String getDaemonInfoPid(String daemonHostname, int appl_pid)
/* Gets the connections and indicators of the specified application */
String getDaemonInfoConnection(String daemonHostname, String connectName)
/* Gets the information of the connection with the specified name */
String getDaemonInfoIndicator(String daemonHostname, String indicatorName
    )
/* Gets the information of the indicators with the specified name */
```

We have used the Java API in the HPC-Europa single point of access portal. One major activity of the HPC-Europa project is to build a portal that provides a uniform and intuitive user interface to access and use resources from different HPC centres in Europe[7]. As most of the HPC centres have deployed their own site-specific HPC and grid infrastructure, there are currently five different systems that provide a job submission and basic monitoring functionality in the HPC-Europa infrastructure: eNANOS[10], GRIA[5], GRMS[6], JOSH[8], and UNICORE[13].

The monitoring functionality is done with the "Generic Monitoring Portlet" as is shown in Fig. 4. In this figure we can see 3 submitted jobs, two of them running in the same

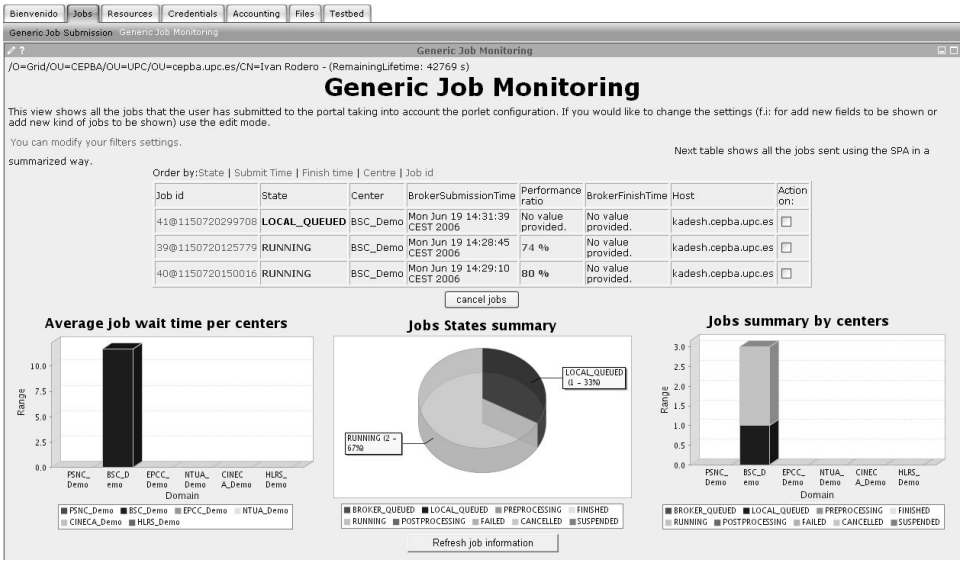


Figure 4. Monitoring functionality of the HPC-Europa SPA portal

machine. For each of these jobs, the monitoring portlet provides a set of data in which a "performance ratio" is included. This ratio indicates the percentage of MFLOPS the application is achieving in relation to the maximum MFLOPS the application can achieve in this machine. This value gives the user an idea of how its applications are behaving. In case of the example shown in Fig. 4, if the scheduler starts running another application in the same machine and the machine does not have any more free resources, the performance ratio of the running applications would decrease more and more, because the machine would be probably overloaded.

6 Discussion

In this paper we have discussed the lack of information on the progress and performance of the applications in a HPC system. We have also discussed the suitability of a progress and performance system which gives the user an idea of how its applications are progressing and helps to improve the performance and the use of resources of the HPC systems. In an attempt to improve the system, we have presented a generic API of progress and performance indicators and some tools which can be implemented on top of this API. We have also explained how we have used this API and tools in the HPC-Eruopa single point of access portal.

In addition to using the progress indicators API as a monitoring tool, we have introduced this API inside the eNANOS system[11] to improve its scheduling policies. The idea is using the progress indicators to perform the grid scheduling guided by the performance and progress indicators. Therefore, the job scheduling can be done based on dynamic load balancing techniques at the grid level. Furthermore, we have integrated the progress and

performance system into our grid brokering system[10] and also with other systems such as GRMS[6] in the framework of the CoreGRID project[2] as described in[12].

Future work will focus on improving the scheduling policies, using the progress indicators API more deeply with the current implementation. We believe that the progress indicators API can improve current scheduling policies by combining the information about the progress of the applications with the information obtained from our prediction system and the run-time information about load balance. Specifically, we believe that the dynamic adjustment of MPI+OpenMP applications on run-time can be improved with this additional information.

Finally, we are working to improve the current modules of the progress indicators API and to implement new functionalities. Our main objective is to make it possible for the user to avoid adding any line of code in the original application or recompiling the application to obtain the performance indicators obtained from the hardware counters. We are working on dynamic linkage for sequential applications and the usage of PMPI for MPI applications.

Acknowledgements

This work has been partially supported by the HPC-Europa European Union project under contract 506079, by the CoreGRID European Network of Excellence (FP6-004265) and by the Spanish Ministry of Science and Technology under contract TIN2004-07739-C02-01.

References

1. Z. Balaton and G. Gombis, *Resource and job monitoring in the grid*, in: Proc. Euro-Par, pp. 404–411, (2003).
2. CoreGRID Web Site. http://www.coregrid.net
3. Ganglia Web Site. http://ganglia.sourceforge.net
4. T. Goodale, S. Jha, T. Kielmann, A. Merzky, J. Shalf and C. Smith, *A Simple API for Grid Applications (SAGA)*, OGF GWD-R.72, SAGA-CORE WG, September 8, (2006).
5. GRIA project Web Site. http://www.gria.org
6. Grid Resource Management System (GRMS) Web Site. http://www.gridlab.org/grms
7. HPC-Europa Web Site. http://www.hpc-europa.org
8. JOSH Web Site. http://gridengine.sunsource.net/josh.html
9. PAPI Web Site. http://icl.cs.utk.edu/papi
10. I. Rodero, J. Corbalan, R. M. Badia and J. Labarta, *eNANOS Grid Resource Broker*, EGC 2005, Amsterdam, The Netherlands, LNCS **3470**, pp. 111–121, (2005).
11. I. Rodero, F. Guim, J. Corbalan and J. Labarta, *eNANOS: Coordinated Scheduling in Grid Environments*, in: ParCo 2005, Málaga, Spain, pp. 81–88, (2005).
12. I. Rodero, F. Guim, J. Corbalan, A. Oleksiak, K. Kurowski and J. Nabrzyski, *Integration of the eNANOS Execution Framework with GRMS for Grid Purposes*, CoreGRID Integration Workshop, Krakow, Poland, pp. 81-92, October 19-20, (2005).
13. UNICORE Web Site. http://www.unicore.org
14. R. Wolski, N. T. Spring and J. Hayes, *The network weather service : a distributed resource performance forecasting service for metacomputing*, Future Generation Computer Systems, **15**, 757–768, (1999).

Efficient Object Placement including Node Selection in a Distributed Virtual Machine

Jose M. Velasco, David Atienza, Katzalin Olcoz, and Francisco Tirado

Computer Architecture and Automation Department (DACYA)
Universidad Complutense de Madrid (UCM)
Avda. Complutense s/n, 28040 - Madrid, Spain
E-mail: mvelascc@fis.ucm.es, {datienza, katzalin, ptirado}@dacya.ucm.es

Currently, software engineering is becoming even more complex due to distributed computing. In this new context, portability while providing the programmer with the single system image of a classical JVM is one of the key issues. Hence a cluster-aware Java Virtual Machine (JVM), which can transparently execute Java applications in a distributed fashion on the nodes of a cluster, is really desirable. This way multi-threaded server applications can take advantage of cluster resources without increasing their programming complexity.

However, such kind of JVM is not easy to design and one of the most challenging tasks is the development of an efficient, scalable and automatic dynamic memory manager. Inside this manager, one important module is the automatic recycling mechanism, i.e. Garbage Collector (GC). It is a module with very intensive processing demands that must concurrently run with user's application. Hence, it consumes a very critical portion of the total execution time spent inside JVM in uniprocessor systems, and its overhead increases in distributed GC because of the update of changing references in different nodes.

In this work we propose an object placement strategy based on the connectivity graph and executed by the garbage collector. Our results show that the choice of an efficient technique produces significant differences in both performance and inter-nodes messaging overhead. Moreover, our presented strategy improves performance with respect to state-of-the-art distributed JVM proposals.

1 Introduction

A cluster-aware Java Virtual Machine (JVM) can transparently execute Java applications in a distributed fashion on the nodes of a cluster while providing the programmer with the single system image of a classical JVM. This way multi-threaded server applications can take advantage of cluster resources without increasing programming complexity.

When a JVM is ported into a distributed environment, one of the most challenging tasks is the development of an efficient, scalable and fault-tolerant automatic dynamic memory manager. The automatic recycling of the memory blocks no longer used is one of the most attractive characteristics of Java for software engineers, as they do not need to worry about designing a correct dynamic memory management. This automatic process, very well-known as Garbage Collection/Collector (GC), makes much easier the development of complex parallel applications that include different modules and algorithms that need to be taken care of from the software engineering point of view. However, since the GC is an additional module with intensive processing demands that runs concurrently with the application itself, it always accounts for a critical portion of the total execution time spent inside the virtual machine in uniprocessor systems. As Plainfosse[5] outlined, distributed GC is even harder because of the difficult job to keep updated the changing references between address spaces of the different nodes.

Furthermore, the node choice policy for object emplacement is an additional task that can facilitate or difficult an efficient memory management. The inter-node message production increases proportionally to the distribution of objects that share dependencies. These dependencies can be seen as a connectivity graph, where objects are situated in the vertices and edges represent references.

In prior work, Fang et Al[11] have proposed a global object space design based on object access behaviour and object connectivity. Thus, they need to profile extensively the object behaviour. Their work is restricted to the allocation phase and it is not related to the GC. The main weakness of this approach is that knowledge of the connectivity graph during the allocation phase requires a lot of profile code and extra meta-data, which results in significant overhead in both performance and space.

In our proposal the object placement is based on object connectivity as well, but it is managed by the GC and takes place during its reclaiming phase. Hence, we have eliminated the profiling phase and the extra needed code. Moreover, after extensive experiments, we have observed that the tracing GC family is the optimal candidate to implement such a distributed scheme. Our results with distributed GCs show significant gains in performance and reductions in amount of exchanged data in the distributed JVM implementation in comparison to state-of-the-art distributed JVM schemes.

The rest of the paper is organized as follows. We first describe related work on both distributed GC and JVM. Then, we overview the main topics in distributed tracing GC. Next, we present our proposal based on suitable object placement and node selection during GC. Then, we describe the experimental setup used in our experiments and the results obtained. Finally, we present an overview of our main conclusions and outline possible future research lines.

2 Related Work

Different works have been performed in the area of both distributed garbage collection and distributed JVM. Plainfosse and Shapiro[5] published a complete survey of distributed GC techniques. In addition, Lins details a good overview of distributed JVMs within the Jones's classical book about GC[3].

A very relevant work in this area of distributed JVMs on a cluster of computing elements is the dJVM framework by Zigman et Al[8], which presents the distributed JVM as a single system image to the programmer. Although the approach is very interesting and novel, it is relatively conservative because it does not reclaim objects with direct or indirect global references. Our work in this paper is particularly enhancing this proposal (see Section 5.1) by removing its previously mentioned limitations to improve performance of the distributed JVM. Similar approaches with more limited scope of application to dJVM are cJVM[15] and Jessica[14]. cJVM, from Aridor et Al[7] is a distributed JVM built on top of a cluster enabled infrastructure, which includes a new object model and thread implementation that are hidden to the programmer. Thus, cJVM uses a master-proxy model for accessing remote objects. Jessica, from the university of Hong-Kong, employs a master-slave thread model. The negative part of these approaches is that for each distributed thread another thread exists in the master node, which handles I/O redirection and synchronization. Recently, Daley et Al[10] have developed a solution to avoid the need of a master node in certain situations.

Another variation to distributed JVMs is proposed by JavaParty[16] from the university of Karlsruhe. JavaParty uses language extensions to implement a remote method invocation, which is the main communication method in the cluster environment. The extensions are precompiled into pure Java code and finally into bytecode. However, Java language augmentation does not provide good solutions as it does not provide a true single system image.

Finally, in Jackal[12] and Hyperion[13] it is proposed a direct compilation of multi-threaded Java bytecode into C code, and subsequently into native machine code. Therefore, the Java runtime system is not used when executing applications. The main negative point of this scheme is the lack of flexibility and need to recompile the system when it is ported to another underlying hardware architecture executing the JVM.

3 Distributed Tracing Garbage Collection

In our work, we have developed a new framework for the analysis and optimization of tracing-based distributed GC by using the dJVM approach[8] as initial starting point. However, conversely to dJVM our new framework does not include any reference counting GC mechanisms[4], which are very popular in mono-processor GC solutions, because of two reasons. First, the reference counting GC is not complete and needs a periodical tracing phase to reclaim cycles, which will create an unaffordable overhead in execution time since it needs to block all the processing nodes. Second, the update of short-lived references produces continuous messages to go back and forth between the different nodes of the distributed GC, which makes this algorithm not scalable within a cluster.

On the contrary, as we have observed in our experiments, tracing GCs[4] seem to be the more convenient option to create such distributed GCs. Conceptually, all consist in two different phases. First, the marking phase allows the GC to identify living objects. This phase is global and implies scanning the whole distributed heap. Second, the reclaiming phase takes care of recycling the unmarked objects (i.e., garbage). The reclaiming phase is local to each node and can be implemented as a non-moving or as a moving GC. Thus, if objects are moved, we need to reserve space. The amount of reserved space must be equal to the amount of allocated memory. Hence, the available memory is reduced by two.

4 Object Placement During Garbage Collection

Our goal is to distribute objects in the cluster nodes according to a way that the communication message volume can be minimized. Ideally, a new object should be placed on the node where it is mainly required. Keeping this idea in mind, our proposed distributed scheme has a main node in which all new objects are allocated. Then, when this node runs out of memory, a local collection is triggered. During the tracing phase it is possible to know the exact connectivity graph. Therefore, we can select a global branch of the references graph and move to a node a complete group of connected objects.

It becomes apparent that our scheme incurs in a penalty, as objects are not distributed as soon as they are created. However, in the long run, this possible penalty is compensated by three factors. First, a high percentage of Java objects are very short-lived. Therefore, in our proposal, we have a higher probability of distributing long-lived data, which means that we

avoid segregating objects that may die quickly. Second, as our experimental results show (Section 5.3), we achieve a considerable reduction in the number of inter-node messages due to the interval before the object distribution phase is applied. Finally, since all the objects that survive a collection in the main node migrate to others nodes, we do not need to reserve space for them. Thus, in the main node we have all the possible available memory, without the reduction by a factor of two that occurs in all the moving GCs (see Section 3).

5 Experimental Setup and Results

In this section we first describe the whole simulation environment used to obtain detailed memory access profiling of the JVM (for both the application and the GC phase). It is based on cycle-accurate simulations of the original Java code of the applications under study. Then, we summarize the representative set of GCs used in our experiments. Finally, we introduce the sets of applications selected as case studies and indicate the main results obtained with our experiments.

5.1 Jikes RVM and dJVM

Jikes RVM is a high performance JVM designed for research. It is written in Java and the components of the virtual machine are Java objects[2], which are designed as a modular system to enable the possibility of modifying extensively the source code to implement different GC strategies, optimizing techniques, etc. There are different compiling options in Jikes. The baseline compiler does not perform any analysis and translates Java bytecodes to a native equivalent. In addition, Jikes RVM has an optimizing compiler and an adaptive compiler. Jikes is a Java virtual machine that runs on itself producing competitive performance with production JVMs.

The dJVM approach[8,9] is an enhancement of Jikes RVM to construct a distributed VM. Several infra-structured components were altered including inter-node communication, the booting process and the use of system libraries. It provides a single system image to Java applications and so it is transparent to the programmer. The dJVM employs a master-slave architecture, where one node is the master and the rest are slaves. The boot process starts at the master node. This node is also responsible for the setting up of the communication channels between the slaves, holding global data and the loading of application classes. The class loader runs in the master.

In dJVM objects are available remotely and objects have only a node local instance. This is achieved by using a global and a local addressing schemes for objects. The global data is also stored in the master with a copy of its global identifier in each slave node per object. Each node has a set of universal identifiers. Instances of primitives types, array types and most class types are always allocated locally. The exceptions are class types which implement the *Runnable* interface.

The initial design of dJVM targets the Jikes RVM baseline compiler. dJVM uses version 2.3.0 along with the Java memory manager Toolkit (JMTk)[1]. dJVM comes with different node selection policies: Round Robin (RR), random, etc. In our experiments, RR slightly outperforms the others. Thus, we use RR in the reported results is best policy choice for distributed JVMs. Finally, Jikes RVM code is scattered with a lot of assertion checking code that we have disabled for our experiments.

5.2 Case Studies

We have applied the proposed experimental setup to dJVM running the most representative benchmarks in the suite SPECjvm98 and the SPECjbb2000 benchmark[6]. These benchmarks were launched as dynamic services and extensively use dynamic data allocation. The used set of applications is the following:

_201_compress, _202_Jess, _209_db, _213_javac, _222_mpegaudio and _227_Jack. These benchmarks are not real multi-threading.

_228_mtrt: it is the multi-threaded version of _205_raytrace. It works in a graphical scene of a dinosaur. It has two threads, which make the render of the scene removed from a file of 340 KB.

The suite SPECjvm98 offers three input sets(referred as s1, s10, s100), with different data sizes. In this study we have used the biggest input data size, represented as s100, as it produces a bigger amount of cross-references among objects.

SPECjbb2000 is implemented as a Java program emulating a 3-tier system with emphasis on the middle tier. All three tiers are implemented within the same JVM. These tiers mimic a typical business application, where users in Tier 1 generate inputs that result in the execution of business logic in the middle tier (Tier 2), which calls to a database on the third tier. In SPECjbb2000, the user tier is implemented as random input selection.

We have also used a variant of the SPECjbb2000 benchmark for our experiments. SPECjbb2000 simulates a wholesale company whose execution consists of two stages. During startup, the main thread sets up and starts a number of warehouse threads. During steady state, the warehouse threads execute transactions against a database (represented as in-memory binary trees). This variant of SPECjbb2000 that we have used is called pseudojbb: pseudojbb runs for a fixed number of transactions (120,000) instead of a fixed amount of time.

5.3 Experimental Results

In our experiments we have utilized as hardware platform a 32-node cluster with a fast Ethernet communication hardware between the nodes. The networking protocol is standard TCP/IP. Then, each node is a Pentium IV, 866 MHz with 1024 Mb and Linux Red Hat 7.3.

Our experiments cover different types of configurations of the 32-node cluster, in the range of 2 to 32 processors. We have executed the benchmarks presented in the previous section in different single- and multi-threaded configurations to have a complete design exploration space. Precisely, we have executed pseudo JBB, _228_mtrt as multi-threaded applications, in combination with the rest of the other SPEC jvm98 and jbb2000 benchmarks, which are all single-threaded.

In our first set of experiments, shown in Fig. 1, we report the number of messages with distributed data that need to be exchanged between the nodes using our approach and in the case of the original dJVM. The results are normalized to the volume of messages of the original dJVM framework. These results indicate that our proposed approach reduces the number of messages that need to be exchanged with distributed information in all the possible configurations of processors with respect to dJVM, even in the case of single-threaded benchmarks. In fact, in complex and real-life multi-threaded applications, the reduction is very significant and always is above 20%. Moreover, the best configuration of distributed JVM is four processing nodes in both cases, and in this case the reduction in the

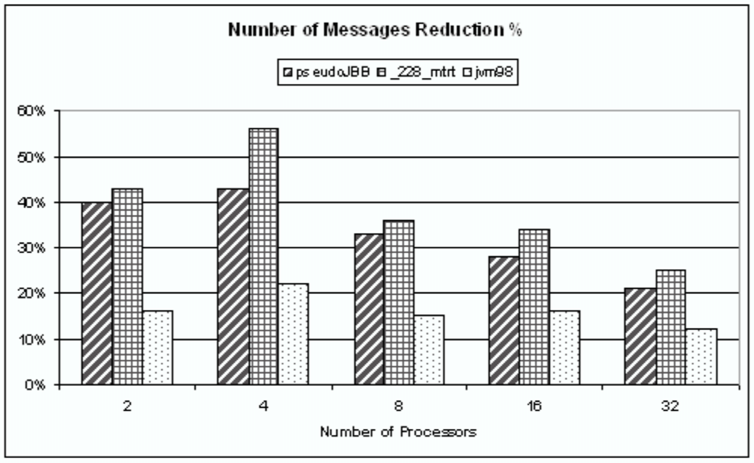


Figure 1. Reductions percentages in the number of messages exchanged between the 32 processing nodes of the cluster in our distributed JVM scheme for different sets of SPEC jvm98 and jbb2000 benchmarks. The results are normalized to the total amount of messages used in the dJVM

amount of exchanged data between nodes is approximately 40% on our side in comparison to dJVM.

In our second set of experiments, shown in Fig. 2, we have compared the execution time of our approach (node selection After Garbage Collection or AGC in Fig. 2) against Jikes RVM running on a uniprocessor system (or just one processor of the 32-node cluster). We have also compared the execution time of dJVM against Jikes RVM and the results are normalized in both cases to the single-processor Jikes RVM solutions. Thus, results in Fig. 2 greater than one mean that the distributed approaches are faster than the uniprocessor one.

Our results in Fig. 2 indicate that the previously observed reduction in the number of messages translates in a significant speed-up with respect to dJVM (up to 40% performance improvement) and single-processor Jikes RVM (up to 45% performance gains). In addition, in Fig. 2 we can observe that single-threaded applications (i.e., most of the jvm98 benchmarks) are not suited to distributed execution since the lack of multiple threads and continuous access to local memories severely affects the possible benefits of distributed JVM schemes. Thus, instead of achieving speed-ups, any distributed JVM suffers from performance penalties (up to 40%) with respect to monoprocessor systems, specially with a large number of processing nodes (i.e., 8 or more nodes) are used.

Finally, our results indicate that even in the case of multi-threaded Java applications of SPEC jvm98 and jbb2000 benchmarks, due to the limited number of threads available, the maximum benefits of distributed JVMs are with four processing nodes. Moreover, although pseudoJBB obtains forty percent speedups with eight processors, the difference in execution time related to four processors does not compensate for the amount of resources wasted in this computation and the observed computation efficiency of the overall cluster is lower in this case than with 4 processing nodes.

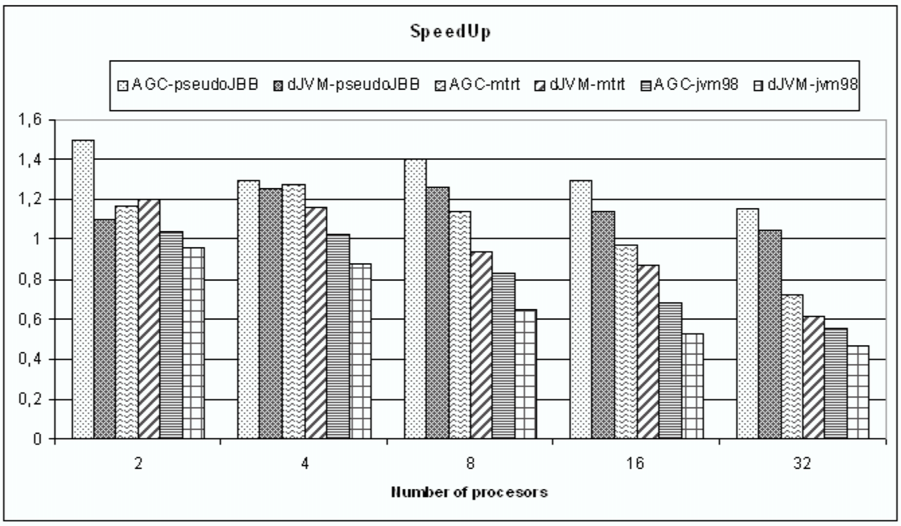


Figure 2. Performance speed-ups of different distributed JVMs for different SPEC jvm98 and jbb2000 benchmarks. The results are normalized to the Jikes RVM running locally in 1 cluster node

6 Conclusions and Future Work

In the last years software design has increased its complexity due to an extensive effort to exploit distributed computing. In this new working environment, portability while providing the programmer with the single system image of a classical JVM has become of the main challenges. Thus, it is really important to have efficient Java Virtual Machine (JVM) that can execute Java applications in a distributed fashion on the processing nodes of clusters, without incurring in significant processing efforts added to the software programmer. In this direction, JVMs need to evolve to provide an efficient and scalable automatic recycling mechanism or Garbage Collector (GC).

In this paper we have presented an object placement strategy based on the connectivity graph and executed by the GC. Our results show that our new scheme provides significant reductions in the amount of exchanged messages between the processing nodes of a 32-node cluster. Due to this reduction in the number of messages, our approach is faster than state-of-the-art distributed JVMs (up to 40% on average). In addition, our results indicate that distributed single-threaded applications (e.g., most benchmarks included in the SPEC jvm98) are not suited to distributed execution. Finally, our experiments indicate that the maximum gains are obtained with a limited number of processors, namely, 4 processors out of 32 in an homogeneous cluster.

In this work, our distributed JVM scheme places objects after the first GC in the master node occurs. As future work we intend to extend object migration after every global GC. Our belief is that that this feature can produce significant additional improvements in multi-threaded applications, both in execution time and number of messages reduction.

Acknowledgements

This work is partially supported by the Spanish Government Research Grant TIN2005-05619.

References

1. IBM, *The Jikes' research virtual machine user's guide 2.2.0.*, (2003). http://oss.software.ibm.com/developerworks/oss/jikesrvm/
2. The source for Java technology, (2003). http://java.sun.com
3. R. Jones, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, 4th edition, (John Wiley & Sons, 2000).
4. R. Jones and R. D. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, (John Wiley & Sons, 1996).
5. D. Plainfosse and M. Shapiro, *A survey of distributed garbage collection techniques*, in: Proc. International Workshop on Memory Management, (1995).
6. SPEC, Spec documentation, (2000). http://www.spec.org/
7. M. Factor, Y. Aridor and A. Teperman, *A distributed implementation of a virtual machine for Java.* Concurrency and Computation: Practice and Experience.
8. J. Zigman and R. Sankaranarayanara, *djvm - a distributed jvm on a cluster*, Technical report, Australia University, (2002).
9. J. Zigman and R. Sankaranarayanara, *djvm - a distributed jvm on a cluster*, in: 17th European Simulation Multiconference, Nottingham, UK, (2003).
10. A. Daley, R. Sankaranarayana and J. Zigman, *Homeless Replicated Objects*, in: 2nd International Workshop on Object Systems and Software Architectures (WOSSA'2006), Victor Harbour, South Australia, (2006).
11. W. Fang, C.-L. Wang and F. C. M. Lau, *On the design of global object space for efficient multi-threading Java computing on clusters*, J. Parallel Computing, 11–12 Elsevier Science Publishers, (2003)
12. R. Veldema, R. Bhoedjang and H. Bal, *Distributed Shared Memory Management for Java*, Technical report, Vrije Universiteit Amsterdam, (1999).
13. The Hyperion system: Compiling multi-threaded Java bytecode for distributed execution. http://www4.wiwiss.fu-berlin.de/dblp/page/record/journals/pc/AntoniuBHMMN01
14. JESSICA2 (Java-Enabled Single-System-Image Computing Architecture version 2). http://i.cs.hku.hk/~clwang/projects/JESSICA2.html
15. Cluster Virtual Machine for Java. http://www.haifa.il.ibm.com\/projects/systems/cjvm/index.html
16. JavaParty. http://svn.ipd.uni-karlsruhe.de/trac/javaparty/wiki/

Memory Debugging of MPI-Parallel Applications in Open MPI

Rainer Keller, Shiqing Fan, and Michael Resch

High-Performance Computing Center, University of Stuttgart,
E-mail: {keller, fan, resch}@hlrs.de

In this paper we describe the implementation of memory checking functionality based on instrumentation using `valgrind`. The combination of valgrind based checking functions within the MPI-implementation offers superior debugging functionality, for errors that otherwise are not possible to detect with comparable MPI-debugging tools. The functionality is integrated into Open MPI as the so-called `memchecker`-framework. This allows other memory debuggers that offer a similar API to be integrated. The tight control of the user's memory passed to Open MPI, allows not only to find application errors, but also helps track bugs within Open MPI itself.

We describe the actual checks, classes of errors being found, how memory buffers internally are being handled, show errors actually found in user's code and the performance implications of this instrumentation.

1 Introduction

Parallel programming with the distributed memory paradigm using the Message Passing Interface MPI[1] is often considered as an error-prone process. Great effort has been put into parallelizing libraries and applications using MPI. However when it comes to maintaining the software, optimizing for new hardware or even porting the code to other platforms and other MPI implementations, the developers face additional difficulties[2]. They may experience errors due to implementation-defined behaviour, hard-to-track timing-critical bugs or deadlocks due to communication characteristics of the MPI-implementation or even hardware dependent behaviour. One class of bugs, that are hard-to-track are memory errors, specifically in non-blocking communication.

This paper introduces a debugging feature based on instrumentation functionality offered by `valgrind`[3], that is being employed within the Open MPI-library. The user's parameters, as well as other non-conforming MPI-usage and hard-to-track errors, such as accessing buffers of active non-blocking operations are being checked and reported. This kind of functionality would otherwise not be possible within traditional MPI-debuggers based on the PMPI-interface.

This paper is structured as follows: Section 2 gives an introduction into the design and implementation, Section 3 shows the performance implications, Section 4 shows the errors, that are being detected. Finally, Section 5 gives a comparison of other available tools and concludes the paper with an outlook.

2 Design and Implementation

The tool suite `valgrind`[3] may be employed on static and dynamic binary executables on `x86`/`x86_64`/`amd64`- and PowerPC32/64-compatible architectures. It operates by intercepting the execution of the application on the binary level and interprets and instruments

the instructions. Using this instrumentation, the tools within the `valgrind`-suite then may deduce information, e. g. on the allocation of memory being accessed or the definedness of the content being read from memory. Thereby, the `memcheck`-tool may detect errors such as buffer-overruns, faulty stack-access or allocation-errors such as dangling pointers or double frees by tracking calls to `malloc`, `new` or `free`. Briefly, the tool `valgrind` shadows each byte in memory: information is kept whether the byte has been allocated (so-called A-Bits) and for each bit of each byte, whether it contains a defined value (so-called V-Bits).

As has been described on the web-page[4] for MPIch since version 1.1, `valgrind` can be used to check MPI-parallel applications. For MPIch-1[5] `valgrind` has to be declared as debugger, while for Open MPI, one only prepends the application with `valgrind` and any valgrind-parameters, e. g. `mpirun -np 8 valgrind --num_callers=20 ./my_app inputfile`.

As described, this may detect memory access bugs, such as buffer overruns and more, but also by knowledge of the semantics of calls like `strncpy`. However, `valgrind` does not have any knowledge of the semantics of MPI-calls. Also, due to the way, how `valgrind` is working, errors due to undefined data may be reported late, way down in the call stack. The original source of error in the application therefore may not be obvious.

In order to find MPI-related hard-to-track bugs in the application (and within Open MPI for that matter), we have taken advantage of an instrumentation-API offered by `memcheck`. To allow other kinds of memory-debuggers, such as `bcheck` or Totalview's memory debugging features[6], we have implemented the functionality as a module into Open MPI's Modular Component Architecture[7]. The module is therefore called `memchecker` and may be enabled with the configure-option `--enable-memchecker`.

The instrumentation for the `valgrind`-parser uses processor instructions that do not otherwise change the semantics of the application. By this special instruction preamble, `valgrind` detects commands to steer the instrumentation. On the x86-architecture, the right-rotation instruction `ror` is used to rotate the 32-bit register `edi`, by 3, 13, 29 and 19, aka 64-Bits, leaving the same value in `edi`; the actual command to be executed is then encoded with an register-exchange instruction (`xchgl`) that replaces a register with itself (in this case `ebx`):

```
#define __SPECIAL_INSTRUCTION_PREAMBLE                    \
            "roll $3,  %%edi ; roll $13,%%edi\n\t"        \
            "roll $29,%%edi ; roll $19,%%edi\n\t"         \
            "xchgl %%ebx,%%ebx\n\t"
```

In Open MPI objects such as communicators, types and requests are declared as pointers to structures. These objects when passed to MPI-calls are being immediately checked for definedness and together with `MPI_Status` are checked upon exit[a]. Memory being passed to Send-operations is being checked for accessibility and definedness, while pointers in Recv-operations are checked for accessibility, only.

Reading or writing to buffers of active, non-blocking Recv-operations and writing to buffers of active, non-blocking Send-operations are obvious bugs. Buffers being passed to non-blocking operations (after the above checking) is being set to undefined within the MPI-layer of Open MPI until the corresponding completion operation is issued. This set-

[a] E. g. this showed up uninitialized data in derived objects, e. g. communicators created using `MPI_Comm_dup`

ting of the visibility is being set independent of non-blocking `MPI_Isend` or `MPI_Irecv` function. When the application touches the corresponding part in memory before the completion with `MPI_Wait`, `MPI_Test` or multiple completion calls, an error message will be issued. In order to allow the lower-level MPI-functionality to send the user-buffer as fragment, the so-called lower layer Byte Transfer Layer (BTLs) are adapted to set the fragment in question to accessible and defined, as may be seen in Fig. 1. Care has been taken to handle derived datatypes and it's implications.

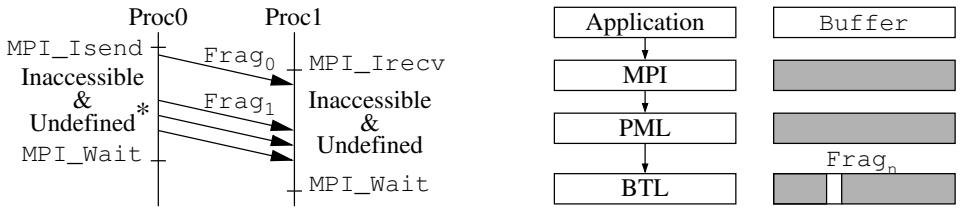


Figure 1. Fragment handling to set accessibility and definedness

For Send-operations, the MPI-1 standard also defines, that the application may not access the send-buffer at all (see[1], p. 30). Many applications do not obey this strict policy, domain-decomposition based applications that communicate ghost-cells, still read from the send-buffer. To the authors' knowledge, no existing implementation requires this policy, therefore the setting to undefined on the Send-side is only done with strict-checking enabled (see Undefined* in Fig. 1).

3 Performance Implications

Adding instrumentation to the code does induce a slight performance hit due to the assembler instructions as explained above, even when the application is not run under `valgrind`. Tests have been done using the Intel MPI Benchmark (IMB), formerly known as Pallas MPI Benchmark (PMB) and the BT-Benchmark of the NAS parallel benchmark suite (NPB) all on the `dgrid`-cluster at HLRS. This machine consists of dual-processor Intel Woodcrest, using Infiniband-DDR network with the OpenFabrics stack.

For IMB, two nodes were used to test the following cases: with&without `--enable-memchecker` compilation and with `--enable-memchecker` but without MPI-object checking (see Fig. 2) and with&without `valgrind` was run (see Fig. 3). We include the performance results on two nodes using the PingPong test. In Fig. 2 the measured latencies (left) and bandwidth (right) using Infiniband (not running with `valgrind`) shows the costs incurred by the additional instrumentation, ranging from 18 to 25% when the MPI-object checking is enabled as well, and 3-6% when memchecker is enabled, but no MPI-object checking is performed. As one may note, while latency is sensitive to the instrumentation added, for larger packet-sizes, it is hardly noticeable anymore (less than 1% overhead). Figure 3 shows the cost when additionally running with `valgrind`, again without further instrumentation compared with our additional instrumentation applied, here using TCP connections employing the IPoverIB-interface.

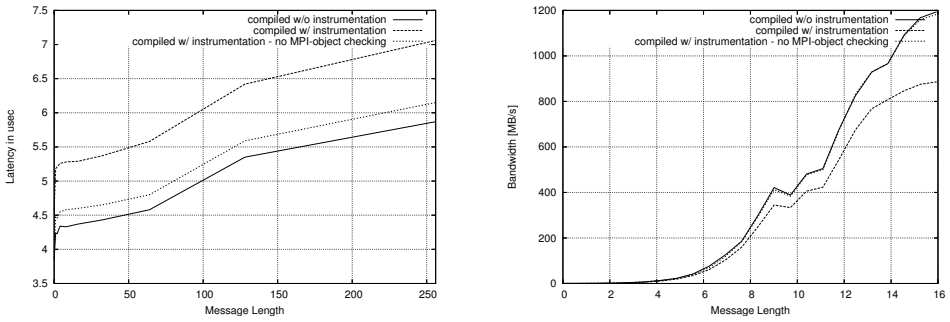


Figure 2. Latencies and bandwidth with&without memchecker-instrumentation over Infiniband, running without `valgrind`.

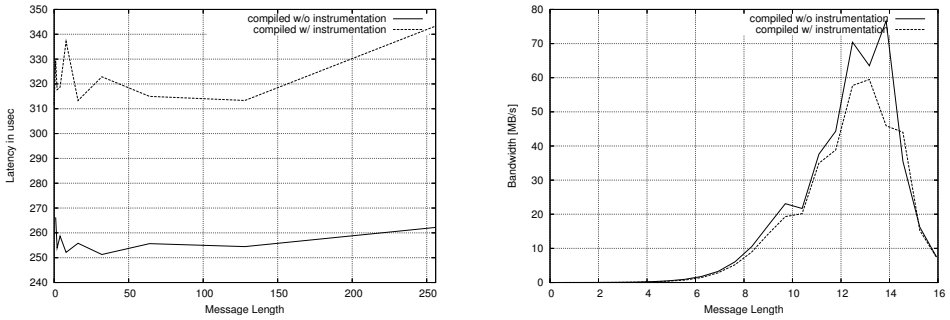


Figure 3. Latencies and bandwidth with&without memchecker-instrumentation using IPoverIB, running with `valgrind`.

The large slowdown of the MPI-object checking is due to the tests of every argument and its components, i. e. the internal data structures of an `MPI_Comm` consist of checking the definedness of 58 components, checking an `MPI_Request` involves 24 components, while checking `MPI_Datatype` depends on the number of the base types.

The BT-Benchmark has several classes, which have different complexity, and data size. The algorithm of BT-Benchmark solves three sets of uncoupled systems of equations, first in the x, then in the y, and finally in the z direction. The tests are done with sizes Class A and Class B. Figure 4 shows the time in seconds for the BT Benchmark. The Class A (size of 64x64x64) and Class B (size of 102x102x102) test was run with the standard parameters (200 iterations, time-step dt of 0.0008).

Again, we tested Open MPI in the following three cases: Open MPI without memchecker component, running under `valgrind` with the memchecker component disabled and finally with `--enable-memchecker`.

As may be seen and is expected this benchmark does not show any performance implications whether the instrumentation is added or not. Of course due to the large memory requirements, the execution shows the expected slow-down when running under valgrind, as every memory access is being checked.

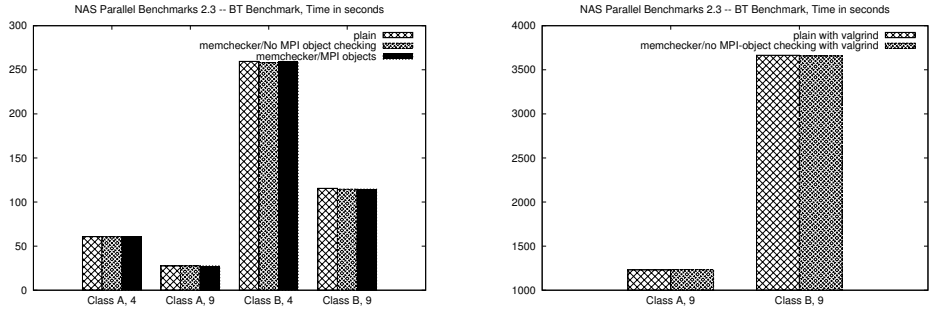


Figure 4. Time of the NPB/BT benchmark for different classes running without (left) and with (right) `valgrind`.

4 Detectable Error Classes and Findings in Actual Applications

The kind of errors, detectable with a memory debugging tool such as `valgrind` in conjunction with instrumentation of the MPI-implementation are:

- Wrong input parameters, e. g. undefined memory passed into Open MPI[b]:

```
char * send_buffer;
send_buffer = (char *) malloc (5);
memset (send_buffer, 0, 5);
MPI_Send (send_buffer, 10, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
```

- Wrong input parameters, wrongly sized receive buffers:

```
char * recv_buffer;
recv_buffer = (char*) malloc(SIZE-1);
memset (buffer, SIZE-1, 0);
MPI_Recv (buffer, SIZE, MPI_CHAR, ..., &status);
```

- Uninitialized input buffers:

```
char * buffer;
buffer = (char *) malloc (10);
MPI_Send (buffer, 10, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

- Usage of the uninitialized `MPI_ERROR`-field of `MPI_Status`[c]:

```
MPI_Wait (&request, &status);
if(status.MPI_ERROR != MPI_SUCCESS)
    return ERROR;
```

- Writing into the buffer of active non-blocking Send or Recv-operation or persistent communication:

[b] This could be found with a BTL such as TCP, however not with any NIC using RDMA.
[c] The MPI-1 standard defines the `MPI_ERROR`-field to be undefined for single-completion calls such as `MPI_Wait` or `MPI_Test` (p. 22).

```
int buf = 0;
MPI_Request req;
MPI_Status status;
MPI_Irecv (&buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
buf = 4711;                    /* Will produce a warning */
MPI_Wait (&req, &status);
```

- Read from the buffer of active non-blocking Send-operation in strict-mode:

```
int inner_value = 0, shadow = 0;
MPI_Request req;
MPI_Status status;
MPI_Isend (&shadow, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &req);
inner_value += shadow;         /* Will produce a warning */
MPI_Wait (&req, &status);
```

- Uninitialized values, e. g. MPI-objects from within Open MPI.

During the course of development, several software packages have been tested with the memchecker functionality. Among them problems showed up in Open MPI itself (failed in initialization of fields of the status copied to user-space), an MPI testsuite[8], where tests for the `MPI_ERROR` triggered an error. In order to reduce the number of false positives Infiniband-networks, the `ibverbs`-library of the OFED-stack[9] was extended with instrumentation for buffer passed back from kernel-space.

5 Conclusion

We have presented an implementation of memory debugging features into Open MPI, using the instrumentation of the `valgrind`-suite. This allows detection of hard-to-find bugs in MPI-parallel applications, libraries and Open MPI itself[2]. This is new work, up to now, no other debugger is able to find these kind of errors.

With regard to related work, debuggers such as Umpire[10], Marmot[11] or the Intel Trace Analyzer and Collector[2], actually any other debugger based on the Profiling Interface of MPI, may detect bugs regarding non-standard access to buffers used in active, non-blocking communication without hiding false positives of the MPI-library itself.

In the future, we would like to extend the checking for other MPI-objects, extend for MPI-2 features, such as one-sided communication, non-blocking Parallel-IO access and possibly other error-classes.

References

1. Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, (1995). `http://www.mpi-forum.org`
2. J. DeSouza, B. Kuhn and B. R. de Supinski, *Automated, scalable debugging of MPI programs with Intel message checker*, in: Proc. 2nd International Workshop on Software Engineering for High Performance Computing System Applications, vol. **4**, pp. 78–82, (ACM Press, NY, 2005).

3. J. Seward and N. Nethercote, *Using Valgrind to detect undefined value errors with bit-precision*, in: Proc. USENIX'05 Annual Technical Conference, Anaheim, CA, (2005).
4. R. Keller, *Using Valgrind with MPIch*, Internet. http://www.hlrs.de\/people/keller/mpich_valgrind.html
5. W. Gropp, E. Lusk, N. Doss and A. Skjellum, *A high-performance, portable implementation of the MPI message passing interface standard*, Parallel Computing, **22**, 789–828, (1996).
6. Totalview Memory Debugging capabilities, Internet. http://www.etnus.com/TotalView/Memory.html
7. T. S. Woodall, R. L. Graham, R. H. Castain, D. J. Daniel, M. W. Sukalski, G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett and A. Lumsdaine, *Open MPI's TEG Point-to-Point communications methodology: comparison to existing implementations*, in: Proc. 11th European PVM/MPI Users' Group Meeting, D. Kranzlmüller, P. Kacsuk, and J. J. Dongarra, (Eds.), vol. **3241** of *Lecture Notes in Computer Science (LNCS)*, pp. 105–111, (Springer, 2004).
8. R. Keller and M. Resch, *Testing the correctness of MPI implementations*, in: Proc. 5th Int. Symp. on Parallel and Distributed Computing (ISDP), Timisoara, Romania, (2006).
9. *The OpenFabrics project webpage*, Internet, (2007). https://www.openfabrics.org
10. J. S. Vetter and B. R. de Supinski, *Dynamic software testing of MPI applications with Umpire*, in: Proc. SC'00, (2000).
11. B. Krammer, M. S. Müller and M. M. Resch, *Runtime checking of MPI applications with Marmot*, in: Proc. PARCO'05, Malaga, Spain, (2005).

Hyperscalable Applications

Massively Parallel All Atom Protein Folding in a Single Day

Abhinav Verma[1], Srinivasa M. Gopal[2], Alexander Schug[2], Jung S. Oh[3], Konstantin V. Klenin[2], Kyu H. Lee[3], and Wolfgang Wenzel[2]

[1] Institute for Scientific Computing
Research Centre Karlsruhe, D-76344, Karlsruhe, Germany
E-mail: verma@int.fzk.de

[2] Institute for Nanotechnology
Research Centre Karlsruhe, D-76344, Karlsruhe, Germany
E-mail: {gopal, klenin, schug, wenzel}@int.fzk.de

[3] Supercomputational Materials Lab
Korean Institute for Science and Technology, Seoul, Korea
E-mail: {soo5, khlee}@kist.re.kr

The search for efficient methods for all-atom protein folding remains an important grand-computational challenge. We review predictive all atom folding simulations of proteins with up to sixty amino acids using an evolutionary stochastic optimization technique. Implementing a master-client model on an IBM BlueGene, the algorithm scales near perfectly from 64 to 4096 nodes. Using a PC cluster we fold the sixty-amino acid bacterial ribosomal protein L20 to near-native experimental conformations. Starting from a completely extended conformation, we predictively fold the forty amino acid HIV accessory protein in less then 24 hours on 2046 nodes of the IBM BlueGene supercomputer.

1 Introduction

Protein folding and structure prediction have been among the important grand computational challenges for more than a decade. In addition to obvious applications in biology, the life sciences and medicinal success, protein simulation strategies also impact the materials and an increasingly the nano-sciences.

It is important to develop methods that are capable of folding proteins and their complexes from completely unbiased extended conformations to the biologically relevant native structure. This problem is difficult to achieve by the presently most accurate simulation techniques, which follow the time evolution of the protein in its environment. Since the microscopic simulation step in such molecular-mechanics methods is of the order of femtoseconds, while the folding or association process takes place on the order of milliseconds, such simulations remain limited in the system size due to the large computational efforts[1,2]. Direct simulation studies were successful however for a number of small proteins, while unfolding simulations starting from the native conformation have given insight into several aspects of protein thermodynamics[3] and folding[4].

It has been a great hope for almost a decade that emerging massively parallel computers architectures, which are available now at the teraflop scale, and which will reach the petaflop scale in the foreseeable future, will be able to contribute to the solution of these problems. Unfortunately kinetic methods face enormous difficulties in the exploitation of

the full computational power of these architectures, because they impose a sequence of steps onto the simulation process, which must be completed one after the other. The parallelization of the energy and force evaluation of a single time-slice of the simulation requires a very high communication bandwidth when distributed across thousands of nodes. This approach alone is therefore unlikely to fully utilize many thousand processors of emerging petaflop-architectures.

In a fundamentally different approach we have developed models[5] and algorithms[6,7] which permit reproducible and predictive folding of small proteins from random initial conformations using free-energy forcefields. According to Anfinsen's thermodynamic hypothesis[8] many proteins are in thermodynamic equilibrium with their environment under physiological conditions. Their unique three-dimensional native conformation then corresponds to the global optimum of a suitable free-energy model. The free-energy model captures the internal energy of a given backbone conformation with the associated solvent and side-chain entropy via an implicit solvent model. Comparing just individual backbone conformations these models assess the relative stability of conformations[9] (structure prediction). In combination with thermodynamic simulation methods (Monte-Carlo or parallel tempering)[10], this approach generates continuous folding trajectories to the native ensemble.

Stochastic optimization methods[13], rather than kinetic simulations, can be used to search the protein free energy landscape in a fictitious dynamical process. Such methods explore the protein free-energy landscape orders of magnitude faster than kinetic simulations by accelerating the traversal of transition states[14], due to the directed construction of downhill moves on the free-energy surface, or the exploitation of memory effects or a combination of such methods. Obviously this approach can be generalized to use not just one, but several concurrent dynamical processes to speed the simulation further, but few scalable simulation schemes are presently available. In a recent investigation, we found that parallel tempering scales only to about 32 replicas[10,11]. The development of algorithms that can concurrently employ thousands of such dynamical processes to work in concert to speed the folding simulation remains a challenge, but holds the prospect to make predictive all-atom folding simulations in a matter of days a reality[12].

The development of such methods is no trivial task for a simple reason: if the total computational effort (number of function evaluations N) is conserved, while the number of nodes (n_p) is increased, each process explores a smaller and smaller region of the conformational space. If the search problem is exponentially complex, as protein folding is believed to be[15], such local search methods revert to an enumerative search, which must fail. It is only the 'dynamical memory' generated in thermodynamic methods such as simulated annealing[13], that permit the approximate solution of the search problem in polynomial time. Thus, massively parallel search strategies can only succeed if the processes exchange information.

Here we review applications of a recently developed evolutionary algorithm, which generalized the basin hopping or Monte-Carlo with minimization method[7,21] to many concurrent simulations. Using this approach we could fold the sixty amino acid bacterial ribosomal protein to its native ensemble[17]. For the largest simulations we used a 4096 processor IBM BlueGene computer to investigate the folding of the 40 amino acid HIV accessory protein. We find that the algorithm scales from 64 to 4096 nodes with less than 10% loss of computational efficiency. Using 2048 processors we succeed to fold the pro-

tein from completely extended to near-native conformations in less than a single day.

2 Methods

2.1 Forcefield

We have parameterized an all-atom free-energy forcefield for proteins (PFF01)[5], which is based on the fundamental biophysical interactions that govern the folding process. This forcefield represents each atom (with the exception of apolar CH_n groups) of the protein individually and models their physical interactions, such as electrostatic interactions, Pauli exclusion, on the bonds attraction and hydrogen bonding. The important interactions with the solvent are approximated by an implicit solvent model. All of these interactions have been optimized to represent the free-energy of a protein microstate corresponding to a particular backbone conformation. For many proteins we could show that the native conformations correspond to the global optimum of this forcefield. We have also developed, or specifically adapted, efficient stochastic optimization methods[14,11,6] (stochastic tunnelling, basin hopping, parallel tempering, evolutionary algorithms) to simulate the protein folding process. Forcefield and simulation methods are implemented in the POEM (Protein Optimization with free-Energy Methods) program package.

With this approach we were able to predictively and reproducibly fold more than a dozen proteins, among them the trp-cage protein (23 amino acids)[18], the villin headpiece (36 amino acids)[19], the HIV accessory protein (40 amino acids)[9], protein A (40 amino acids) as well as several toxic peptides and beta-hairpin proteins (14-20 amino acids) in simulations starting from random initial conformations. The largest protein folded de-novo to date is the 60 amino-acid bacterial ribosomal protein L20[17]. We could demonstrate that the free-energy approach is several orders of magnitude faster than the direct simulation of the folding pathway, but nevertheless permits the full characterization of the free-energy surface that characterizes the folding process according to the prevailing funnel-paradigm for protein folding[9,19].

2.2 Optimization Method

Most stochastic optimization methods map the complex potential energy landscape of the problem onto a fictitious dynamical process that is guided by its inherent dynamics toward the low energy region, and ultimately the global optimum, of the landscape. In many prior simulations the basin hopping technique proved to be a reliable workhorse for many complex optimization problems[7], including protein folding[9], but employs only one dynamical process. This method[21] simplifies the original landscape by replacing the energy of each conformation with the energy of a nearby local minimum. This replacement eliminates high energy barriers in the stochastic search that are responsible for the freezing problem in simulated annealing. In order to navigate the complex protein landscape we use a simulated annealing (SA) process for the minimization step[7]. Within each SA[13] simulation, new configurations are accepted according to the Metropolis criterion, while the temperature is decreased geometrically from its starting to the final value. The starting temperature and cycle length determine how far the annealing step can deviate from its starting conformation. The final temperature must be small compared to typical energy differences between competing metastable conformations, to ensure convergence to a local minimum.

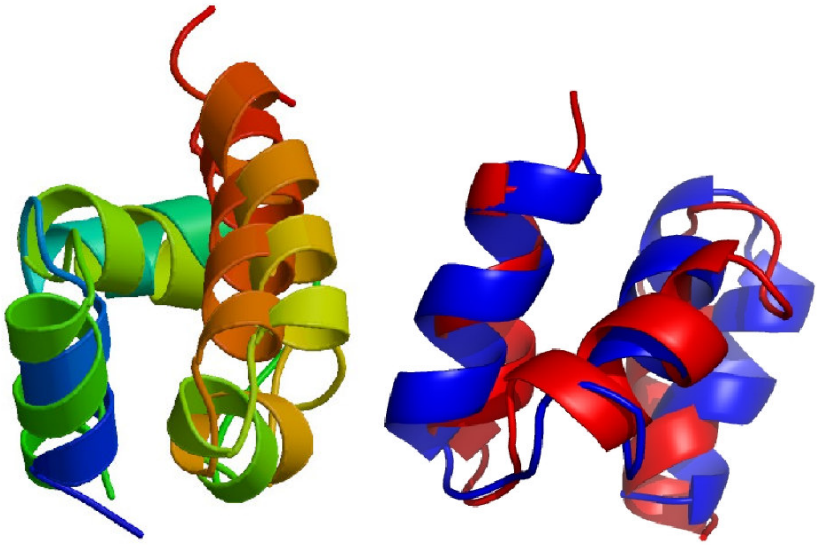


Figure 1. Overlay of the folded and the experimental conformation of the bacterial ribosomal protein L20 (left) and the HIV accessory protein (right)

We have generalized this method to a population of P interdependent dynamical processes operating on a population of N conformations[17]. The whole population is guided towards the optimum of the free energy surface with a simple evolutionary strategy in which members of the population are drawn and then subjected to a basin hopping cycle. At the end of each cycle the resulting conformation either replaces a member of the active population or is discarded. This algorithm was implemented on a distributed master-client model in which idle clients request a task from the master. Conformations are drawn with equal probability from the active population. The acceptance criterion for newly generated conformations must balance the diversity of the population against the enrichment low-energy decoys. We accept only new conformations which are different by at least 4 Å RMSB (root mean square backbone deviation) from all active members. If we find one or more members of the population within this distance, the new conformation replaces the all existing conformations if its energy is lower than the best, otherwise it is discarded. If the new conformation differs by at least the threshold from all other conformation it replaces the worst conformation of the population if it is better in total (free) energy. If a merge operation has reduced the size of the population, the energy criterion for acceptance is waived until the original number of conformations is restored.

This approach is easy to parallelize in a master client approach, where idle clients request tasks from the master. Since the new task is drawn at random from the population, new tasks can be distributed at any time. When the number of processors is much large than the size of the population, many processors attempt to improve the same conformation. However, because the dimensionality of the search space is so large (over 100 independent variables in the problems described here), most of these simulations fail to find good solutions. As a result, the algorithm remains effective even in this limit.

3 Simulations

3.1 Folding the Bacterial Ribosomal Protein L20

We have folded the 60 amino acid bacterial ribosomal protein(pdb-id:1GYZ). This simulation was performed in three stages: In the first stage we generate a large set of unfolded conformations, which was pruned to 266 conformations by energy and diversity. In stage two we perform 50 annealing cycles per replica on these conformation, after which the population was pruned to the best N=50 decoys (by energy). We then continued the simulation for another 5500 annealing cycles. At the end of the simulations, the respective lowest energy conformations had converged to 4.3 Å RMSB with respect to the native conformation. Six of the ten lowest structures had independently converged to near-native conformations of the protein. The first non-native decoy appears in position two, with an energy deviation of only 1.8 kcal/mol (in our model) and a significant RMSB deviation.

The good agreement between the folded and the experimental structure is evident from Fig. 1, which shows the overlay of the native and the folded conformations. The good alignment of the helices illustrates the importance of hydrophobic contacts to correctly fold this protein. Figure 2 demonstrates the convergence of both the energy and the average RMSB deviation as the function of the number of total iterations (basin hopping cycles). Both simulations had an acceptance ratio approximately 30 %.

3.2 Folding the HIV Accessory Protein

We have also folded the 40 amino acid HIV accessory protein(pdb-id: 1F4I). For timing purposes we have performed simulations using 64, 128, 256, 512, 1024, 2048 and 4096 processors on an IBM BlueGene in virtual processor mode. We find that the simulation scales perfectly with the number of processors, inducing less than 5% loss of efficiency when comparing P=64 with P=4096 processor simulations. The control loop is implemented employing a synchronous simulation protocol, where tasks are distributed to all processors of the machine. As the simulations finish, their conformations are transferred to the master, which decides whether to accept (average probability: 57%) the conformation into the active population or disregard the conformation. Then a new conformation is immediately given to the idle processor. Because the processors are processed sequentially some processors wait for the master before they get a new conformation. Fluctuations in the client execution times induce a waiting time before the next iteration can start. For the realistic simulation times chosen in these runs, the average waiting time is less than 10% of the execution time and nearly independent of the number of processors used.

We next performed a simulation using 2048 processors starting from a single completely stretched “stick” conformation. The seed conformation had an average RMSB deviation of 21.5Å to the experimental conformation. We then performed 20 cycles of the evolutionary algorithm described above. Figure 1 shows the overlay of the folded and the experimental conformation. The starting conformation has no secondary structure and no resemblance of the native conformation. In the final conformation, the secondary structure elements agree and align well with the experimental conformation. Figure 2 shows that the best energy converges quickly to a near-optimal value with the total number of basin hopping cycles. The average energy trails the best energy with a finite energy difference. This difference will remain indefinitely by construction, because the algorithm is designed

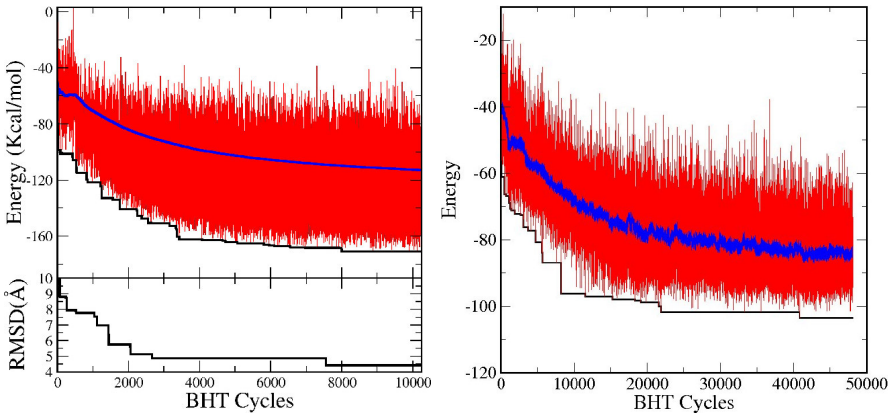


Figure 2. Instantaneous energy (red), mean energy (blue) and best energy (black) for the simulations of the bacterial ribosomal protein L20 (left) and the HIV accessory protein (right). For the bacterial ribosomal protein L20 the lower panel indicates the convergence of the RMS deviation of the lowest energy conformation.

to balance diversity and energy convergence. The acceptance threshold of 4 Å RMSB for the new population enforces that only one near-native conformation is accepted in the population, the average energy will therefore always be higher than the best energy.

4 Discussion

Using a scalable evolutionary algorithm we have demonstrated the all-atom folding of two proteins: Using 50 processors of a loosely connected PC cluster we succeeded to fold the 60 amino acid bacterial ribosomal protein to near-native conformations, Using 2048 processors of an IBM BlueGene we also folded the 40 amino acid HIV accessory protein from a completely extended conformation to within 4 Å of the native conformation in about 24 hours turnaround. The results of this study provide impressive evidence that all-atom protein structure prediction with free-energy forcefields is becoming a reality. The key to convergence of the method lies in the exploitation of the specific characteristics of the free energy landscape of naturally occurring proteins. According to the current funnel paradigm[22] the protein explores an overall downhill process on the energy landscape, where the conformational entropy of the unfolded ensemble is traded for enthalpic gain of the protein and free energy gain of the solvent[3]. Using one- or low-dimensional indicators the complex folding process appears for many small proteins as a two-state transition between the unfolded and the folded ensemble with no apparent intermediates. This transition has been rationalized in terms of the funnel paradigm, where the protein averages over average frictional forces on its downhill path on the free-energy landscape. In this context, the evolutionary algorithm attempts to improve many times each of the conformations of the active population. Because of the high dimensionality of the search problem ($D = 160$ free dihedral angles for 1F4I) most of these attempts fail, but those which succeed are efficiently distributed for further improvement by the evolutionary method.

The search for methods and models for *de novo* folding of small and medium size proteins from the completely extended conformation at atomic resolution has been a “holy

grail" and grand computational challenge for decades[23]. The development of multi-teraflop architectures, such as the IBM BlueGene used in this study, has been motivated in part by the large computational requirements of such studies. The demonstration of predictive folding of a 40 amino acid protein with less than 24 hours turnaround time, is thus an important step towards the long time goal to elucidate protein structure formation and function with atomistic resolution. The results reviewed above demonstrate that it is possible to parallelize the search process by splitting the simulation into a large number of independent conformations, rather than by parallelizing the energy evaluation. This more coarse-grained parallelization permits the use of a much larger number of weakly-linked processors. The algorithm scales very well, because each work-unit that is distributed to a client is very large. We do not parallelize over a single, but instead over thousands of energy evaluations. For this reason the algorithm can tolerate very large latency times and even collapse of part of the network. Since conformations are drawn randomly according to some probability distribution from the current population, client requests need never wait for the next task. In practice, however, redistribution of the same conformation to many processors may lead to duplication of work. Already for the study performed here (N=2048, N_{pop}=64), 32 processors attempt to improve each conformation on average. However, the success rate for improvement drops rapidly with the dimension of the search space. For this reason, the algorithm will perform well on even larger networks as the sizes of the proteins under investigation increase.

The present study thus demonstrates a computing paradigm for protein folding that may be able to exploit the petaflop computational architectures that are presently being developed. The availability of such computational resources in combination with free-energy folding methods can make it possible to investigate and understand a wide range of biological problems related to protein folding, mis-folding and protein-protein interactions.

Acknowledgements

This work is supported by grants from the German national science foundation (DFG WE1863/10-2), the Secretary of State for Science and Research through the Helmholtz-Society and the Kurt Eberhard Bode foundation. We acknowledge the use of facilities at the IBM Capacity on Demand Center in Rochester and KIST Supercomputional Materials Lab in Seoul. We are grateful for technical assistance from G. S. Costigan and C. S. Sosa from the IBM Capacity on Demand Center for technical assistance.

References

1. X. Daura, B. Juan, D. Seebach, W.F. van Gunsteren and A. E. Mark, *Reversible peptide folding in solution by molecular dynamics simulation*, J. Mol. Biol., **280**, 925, (1998).
2. C. D. Snow, H. Nguyen, V. S. Pande and M. Gruebele, *Absolute comparison of simulated and experimental protein folding dynamics*, Nature, **420**, 102–106, (2002).
3. T. Lazaridis, and M. Karplus, *"New view" of protein folding reconciled with the oldthrough multiple unfolding simulations*, Science, **278**, 1928–1931, (1997).

4. A. E. Garcia and N. Onuchic, *Folding a protein in a computer: An atomic description of the folding/unfolding of protein A*, Proc. Nat. Acad. Sci. (USA), **100**, 13898–13903, (2003).
5. T. Herges and W. Wenzel. *An all-atom force field for tertiary structure prediction of helical proteins*, Biophys. J., **87**, 3100–3109, (2004).
6. A. Schug, A. Verma, T. Herges, K. H. Lee and W. Wenzel, *Comparison of stochastic optimization methods for all-atom folding of the trp-cage protein*, ChemPhysChem, **6**, 2640–2646, (2005).
7. A. Verma, A. Schug, K. H. Lee and W. Wenzel, *Basin hopping simulations for all-atom protein folding* , J. Chem. Phys., **124**, 044515, (2006).
8. C. B. Anfinsen, *Principles that govern the folding of protein chains*, Science, **181**, 223–230, (1973).
9. T. Herges and W. Wenzel, *Reproducible in-silico folding of a three-helix protein and characterization of its free energy landscape in a transferable all-atom forcefield*, Phys. Rev. Lett., **94**, 018101, (2005).
10. A. Schug, T. Herges and W. Wenzel, *All-atom folding of the three-helix HIV accessory protein with an adaptive parallel tempering method*, Proteins, **57**, 792–798, (2004).
11. A. Schug, T. Herges, A. Verma and W. Wenzel, *Investigation of the parallel tempering method for protein folding*, J. Physics: Cond. Mat., **17**, 1641–1650, (2005).
12. A. Garcia and J. Onuchic, *Folding a protein on a computer: hope or reality*, Structure, **13**, 497–498, (2005).
13. S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, *Optimization by simulated annealing*, Science, **220**, 671–680, (1983).
14. W. Wenzel and K. Hamacher, *Stochastic tunneling approach for global optimization of complex potential energy landscapes*, Phys. Rev. Lett., **82**, 3003–3007, (1999).
15. C. Levinthal, *Are there pathways for protein folding ?*, J. Chim. Phys., **65**, 44–45, (1968).
16. Z. Li and H. A. Scheraga, *Monte carlo minimization approach to the multiple minima problem in protein folding*, Proc. Nat. Acad. Sci. U.S.A., **84**, 6611–6615, (1987).
17. A. Schug and W. Wenzel, *An evolutionary strategy for all-atom folding of the sixty amino acid bacterial ribosomal proein L20*, Biophys. J., **90**, 4273–4280, (2006).
18. A. Schug, T. Herges and W. Wenzel, *Reproducible protein folding with the stochastic tunneling method*, Phys. Rev. Letters, **91**, 158102, (2003).
19. T. Herges and W. Wenzel, *Free energy landscape of the villin headpiece in an all-atom forcefield*, Structure, **13**, 661, (2005).
20. T. Herges, H. Merlitz and W. Wenzel, *Stochastic optimization methods for biomolecular structure prediction*, J. Ass. Lab. Autom., **7**, 98–104, (2002).
21. A. Nayeem, J. Vila and H. A. Scheraga, *A comparative study of the simulated-annealing and monte carlo-with-minimization approaches to the minimum-energy structures of polypeptides: [met]-enkephalin*, J. Comp. Chem., **12(5)**, 594–605, (1991).
22. K. A. Dill and H. S. Chan, *From levinthal to pathways to funnels: The "new view" of protein folding kinetics*, Nature Structural Biology, **4**, 10–19, (1997).
23. Y. Duan and P. A. Kollman *Pathways to a protein folding intermediate observed in a 1-microsecond simulation in aqueous solution*, Science, **282**, 740–744, (1998).

Simulations of QCD in the Era of Sustained Tflop/s Computing

Thomas Streuer[1] **and Hinnerk Stüben**[2]

[1] Department of Physics and Astronomy
University of Kentucky, Lexington, KY, USA
E-mail: thomas.streuer@desy.de

[2] Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
Takustr. 7, 14195 Berlin, Germany
E-mail: stueben@zib.de

The latest machine generation installed at supercomputer centres in Germany offers a peak performance in the tens of Tflop/s range. We study performance and scaling of our quantum chromodynamics simulation programme BQCD that we obtained on two of these machines, an IBM Blue Gene/L and an SGI Altix 4700. We compare the performance of Fortran/MPI code with assembler code. The latter allows to exploit concurrency at more levels, in particular in overlapping communication and computation as well as prefetching data from main memory.

1 Introduction

The first computer delivering a performance of more than 1 Tflop/s peak as well as in the Linpack benchmark appeared on the Top500 list in June 1997[1]. For German QCD[a] researchers it has taken until the installation of the current generation of supercomputers at national centres until a sustained Tflop/s was available in everyday runs of their simulation programmes.

In this paper we report on how the sustained performance was obtained on these machines. There are two machines, the IBM BlueGene/L at NIC/ZAM Jülich and the SGI Altix 4700 at LRZ Garching/Munich. Both started user operation in 2006. The BlueGene/L has 16.384 CPUs (cores) and offers a peak performance of 45 Tflop/s. The Altix 4700 originally had 4096 cores delivering 26 Tflop/s peak. It was upgraded in 2007 to 9726 cores delivering 62 Tflop/s peak. The performance figures we present were measured on the upgraded system.

It is well known that the performance of QCD programmes can be significantly improved by using low level programming techniques like programming in assembler. In general compilers are not able to generate most efficient code for the multiplication of small complex matrices which is the typical operation in computational QCD (see Section 2), even if all data needed for the computations is in the data cache (see Table 2). In assembler one can in addition exploit concurrency at more levels. At one level there are low level communication calls on the BlueGene, by which one can achieve that communication and computation overlap. Another level is prefetching data from main memory, which will be important on the Altix[b].

[a] Quantum chromodynamics (QCD) is the theory of strongly interacting elementary particles.

[b] On the Altix there are two potential methods for overlapping communication and computation. (a) Since mem-

As will be explained in Section 2 simulations of QCD are communication intensive. Therefore overlapping communication and computation is an important issue. On systems with larger SMP-nodes one can overlap communication and computation by combining OpenMP and MPI[2]. The nodes of the machines we consider are too small nodes for this high level programming technique to work efficiently.

2 Computational QCD

The starting point of QCD is an infinite-dimensional integral. To deal with the theory on the computer space-time continuum is replaced by a four-dimensional regular finite lattice with (anti-) periodic boundary conditions. After this discretisation the integral is finite-dimensional but rather high-dimensional.

The standard algorithm employed today in simulations of QCD is Hybrid Monte Carlo[3] (HMC). HMC programmes have a pronounced kernel, which is an iterative solver of a large system of linear equations. In BQCD we use the standard conjugate gradient (cg) solver. Depending on the physical parameters 80 % or up to more than 95 % of the execution time is spent in the solver. The dominant operation in the solver is the matrix times vector multiplication. In the context of QCD the matrix involved is called *fermion matrix*. This paper is about optimising one part of fermions matrix multiplication which is the multiplication of a vector ψ with the *hopping matrix* D: $\phi(i) = \sum_{j=1}^{n} D(i,j)\psi(j)$, where n is the lattice volume. The hopping matrix is large and sparse. The entries in row i are the nearest neighbours of entry i of the vector ψ (for an illustration see Fig. 1). The entries of the hopping matrix are 3×3 complex matrices and for Wilson fermion, which are used in BQCD, the entries of the vectors are 4×3 complex matrices or with internal indices $s, s' = 1, 2, 3, 4$ and $c, c' = 1, 2, 3$ spelled out

$$\phi_{sc}(i) = \sum_{\mu=1}^{4} \left[(1+\gamma_\mu)_{ss'} U^{\dagger}_{\mu,cc'}(i-\hat{\mu})\psi_{s'c'}(i-\hat{\mu}) + (1-\gamma_\mu)_{ss'} U_{\mu,cc'}(i)\psi_{s'c'}(i+\hat{\mu}) \right] . \quad (2.1)$$

$U^{\dagger}$ denotes hermitian conjugation. The γ-matrices

$$\gamma_1 = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}, \gamma_2 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \gamma_3 = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}, \gamma_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

lead to different access patterns to the entries of ψ. No floating point operations are needed in their multiplications.

A static performance analysis of the hopping matrix multiplication yields that the ratio of floating point multiplications to floating point additions is about 85 % which gives the maximal theoretical performance on processors with fused multiply-adds. Per memory access 13 floating point operations have to be performed on the average.

QCD programmes are parallelised by decomposing the lattice into regular domains. The domains become relatively small. For example, the size of a CPU local domain is

ory is shared between all nodes, it is possible to exchange data simply by using loads or stores (via so-called *shmem pointers*), combined with prefetches as needed in order to hide latency. We have tried this promising method in assembler and in Fortran/C/MPI (without explicit prefetching). In both cases performance decreased. (b) One could try to employ *hyper-threading* where one would use on thread per core for computation and a second thread for communication. In principle there should be no hardware bottlenecks. However, hyper-threading is switched off on the machine we were using.

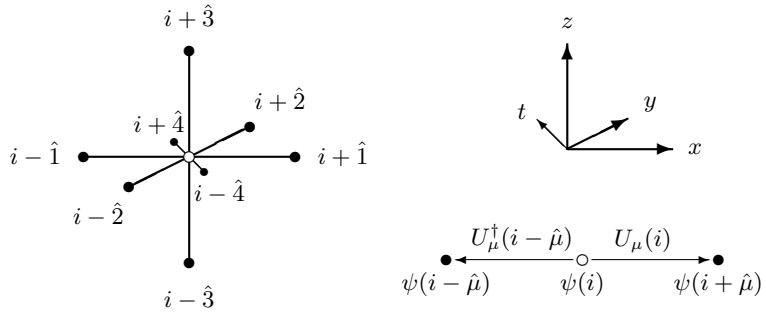


Figure 1. Nearest neighbour stencil underlying the hopping matrix D. The central point is i. On the righthand side the corresponding Cartesian coordinate system and the variables appearing in Eq. (2.1) are indicated for one dimension. U is called the *gauge field* which is defined on the links of the lattice. The field ψ is defined on the lattice sites.

$8^3 \times 4 = 2048$ lattice sites when putting a $32^3 \times 64$ lattice (a typical size used in today's simulations) on 1024 CPUs. The size of the surface of this local volume is 2560 sites, i.e. the surface to volume ratio is 1.25. Here is the challenge of QCD programmes. In every iteration of the solver data of the size of the input vector ψ has to be communicated to neighbouring processes.

The basic optimisation is to calculate the projections $(1 \pm \gamma_\mu)\psi(j)$ before the communication. Due to the symmetries of the projections the amount of data to be transfered can be halved. Even with this optimisation, the problem is communication intensive. Very good communication hardware is needed and overlapping communication and computation is desired to scale QCD programmes to large numbers of processes.

3 QCD on the IBM BlueGene/L

3.1 Hardware

The IBM BlueGene/L is a massively parallel computer. The topology of the network connecting the compute nodes is a three-dimensional torus. Each compute nodes has links to its six nearest neighbours. The hardware bandwidth per link is 175 MByte/s. The network latency is[4]:

$$\text{One way Latency} = (2.81 + .0993 \times \text{Manhattan Distance})\ \mu s$$

In addition to the torus network which is used for point-to-point communication there is a tree network for collective communications.

A BlueGene/L compute chip[5] contains two standard PowerPC 440 cores running at a clock speed of 700 MHz. Each core has a *double hummer* floating point unit[6] (FPU) which operates as a vector processor on a set of 2×32 registers. Besides pure vector arithmetic operations, there is a set of instructions which operates differently on both registers as it is needed for performing complex arithmetic. Since each FPU can perform one vector-multiply-add operation per clock cycle, the peak performance of the chip is 5.6 Gflop/s.

Each compute node contains 512 MByte of main memory. Each core has a 32 kByte L1 data cache. There is a 4 MByte L3 cache on the chip which is shared between the two cores. Coherency between the L1 caches of the two cores is not enforced by hardware, so software has to take care of it. To facilitate data exchange between the two cores, each chip contains 1 kByte of static ram (SRAM) which is not cached.

The torus network is accessed from the chips through a set of memory-mapped FIFOs. There are 6 injection FIFOs and 12 reception FIFOs on each chip[7]. Data can be written to or read from these FIFOs using some of the double-hummer load/store instructions. We used this feature in our code. We made no special use of the independent tree network.

The BlueGene/L operating systems supports two modes of operation: (a) *communication coprocessor mode*, where one of the cores is dedicated to communication, while the other does the computational work, and (b) *virtual node mode*, where both cores perform both communication and computation operations. We always run our programs in virtual node mode.

3.2 Assembler Kernel

The y-, z-, and t-directions of the lattice are decomposed in such a way that the decomposition matches the physical torus network exactly. The x-direction is split between the two cores of a node. Since the L1 caches are not coherent, communication in the x-dimension cannot be done via shared memory in the most straightforward way. Instead, we use the 1 kByte SRAM for communication between the two CPUs.

Ideally communication and computation should overlap. In a QCD programme on the BlueGene/L this can only be achieved by programming in assembler. For the floating pointing operations and communication double-hummer instructions are used. In the course of the computation, each node needs to receive part of the data from the boundary of its neighbouring nodes, and likewise it has to send part of the data from its boundary to neighbouring nodes. In order to hide communication latency, the assembler kernel always looks ahead a few iterations and sends data that will be needed by a remote node. Data is sent in packets of 96 bytes (plus 32 bytes for header and padding), which is the size of a projected spinor $(1 \pm \gamma_\mu)\psi(j)$ in double precision. When a CPU needs data from another node, it polls the respective reception FIFO until a data packet arrives. Since each node sends data packets in the same order in which they are needed on the receiving side, it is not necessary to do any reordering of the packets or to store them temporarily. For comparison with a similar implementation see Reference[8].

3.3 Performance Results

In scaling tests the performance of the *cg*-kernel was measured. For performance measurements the code was instrumented with timer calls and for the kernel all floating point operations were counted manually.

In order to get good performance it is important that the lattice fits the physical torus of the machine. In the assignment of MPI process ranks the four torus directions have to be permuted. On the BlueGene/L this can be accomplished by setting the environment variable `BGLMPI_MAPPING` appropriately. The settings of that variable were `TXYZ` on 1, 2, and 4 racks and `TYZX` on 8 racks.

implementation: Fortran/MPI		lattice: $48^3 \times 96$		
#racks	Mflop/s per core	overall Tflop/s	speed-up	efficiency
1	280	0.57	1.00	1.00
2	292	1.20	2.09	1.04
4	309	2.53	4.41	1.10
8	325	5.32	9.29	1.16
implementation: Fortran/MPI		lattice: $32^4 \times 64$		
#racks	Mflop/s per core	overall Tflop/s	speed-up	efficiency
1	337	0.69	1.00	1.00
2	321	1.32	1.91	0.95
4	280	2.30	3.33	0.83
8	222	3.65	5.28	0.66
implementation: assembler		lattice: $32^3 \times 64$		
#racks	Mflop/s per core	overall Tflop/s	speed-up	efficiency
1	535	1.10	1.00	1.00
2	537	2.20	2.01	1.00
8	491	8.05	7.34	0.92

Table 1. Performance of the conjugate gradient kernel on the BlueGene/L for two implementations and two lattices.

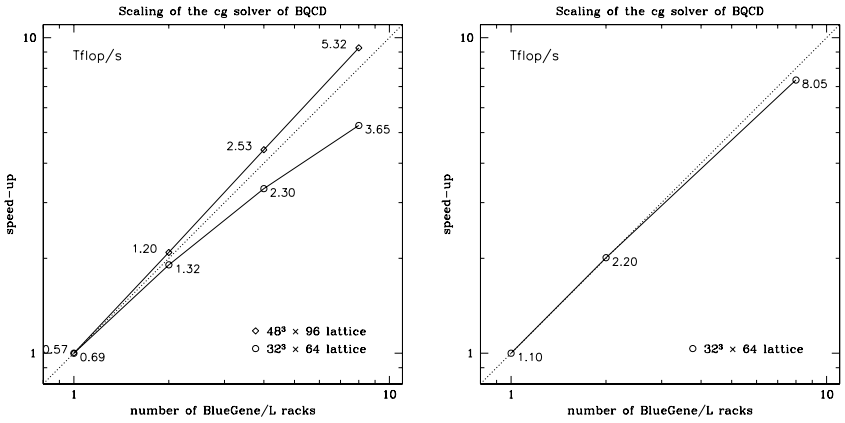


Figure 2. Scaling of the conjugate gradient kernel of BQCD on the BlueGene/L for the Fortran 90/MPI version (left) and for the assembler version (right). The dotted lines indicate linear scaling.

Performance results are given in Table 1. In Fig. 2 results are shown on double logarithmic plots. One can see from the table and the plots that the Fortran/MPI version exposes super-linear scaling on the $48^3 \times 96$ lattice. Even the $32^3 \times 64$ lattice scales quite well given the fact that the lattice volumes per core become tiny (down to 16×2^3). The scaling of the assembler version is excellent. For the same tiny local lattices the scaling is considerably better than for the Fortran/MPI version. This means that in the assembler version computation and communication really overlap.

4 QCD on the SGI Altix 4700

4.1 Hardware

The SGI Altix 4700 is a scalable ccNUMA parallel computer, i.e. its memory is physically distributed but logically shared and the memory is kept coherent automatically by the hardware. For the programmer (or a programme) all of the machine's memory is visible to all nodes, i.e. there is a global address space.

The compute nodes of the Altix 4700 consist of 256 dual core processors. One processor is reserved for the operating system, 255 processors can be used for computation. Inside a node processors are connected via the fat tree *NUMAlink 4* network with a theoretical bandwidth of 6.4 GB/s per link. The nodes are connected via a two-dimensional torus type of network. However, the network is not homogeneous, which *a priori* makes it difficult to scale our problem to very large numbers of cores. The machine at LRZ has the following bisection bandwidths per processor[9]:

intra-node	2×0.8 GByte/s.
any two 'vertical' nodes	2×0.4 GByte/s.
four nodes (shortest path)	2×0.2 GByte/s.
total system	2×0.1 GByte/s.

The processors of the Altix 4700 are Intel Itanium2 *Montecito* Dual Core CPUs, clocked at 1.6 GHz. Each core contains two floating point units, each of which is capable of performing one multiply-add operation per cycle, leading to a peak performance of 6.4 Gflop/s per core (12.8 Gflop/s per processor).

There are three levels of cache, but only two of them (L2 and L3) are used for floating point data. The L3 cache has a size of 9 MByte and a maximum bandwidth of 32 bytes/cycle, which is enough to feed the floating point units even for memory-intensive operations. The bandwidth to main memory is substantially lower.

4.2 Assembler Kernel

Because the memory bandwidth is so much lower than the L3 cache bandwidth, it is important that we partition our problem in such a way that we can keep the fields which we need during the conjugate gradient iterations in the L3 cache, so that in principle no access to local memory is required. From Table 2 one can see that lattices up to about 8^4 sites fit

lattice	#cores	Fortran [Mflop/s]	assembler [Mflop/s]
4^4	1	3529	4784
6^4	1	3653	4813
8^4	1	3245	4465
10^4	1	1434	3256
12^4	1	1329	2878
14^4	1	1103	2766
16^4	1	1063	2879

Table 2. Performance of the hopping matrix multiplication on a single core on the Altix 4700.

weak scaling for local 8^4 lattices			
lattice	#cores	Fortran [Mflop/s]	assembler [Mflop/s]
8^4	1	2553	3655
16^4	16	1477	2235
24^4	81	1273	1978
32^4	256	1251	1750
$32^3 \times 64$	512	1195	1619
$40^3 \times 64$	1000	1156	1485
strong scaling for the $32^3 \times 64$ lattice			
lattice	#cores	Fortran [Mflop/s]	assembler [Mflop/s]
$32^3 \times 64$	512	1195	1619
$32^3 \times 64$	1024	1395	1409
$32^3 \times 64$	2048	996	841

Table 3. Scaling on the Altix 4700 for the conjugate gradient solver. Performance figures are in Mflop/s per core.

into the L3 cache. When staying inside the L3 cache assembler code is roughly a factor of 1.3 faster. Outside the L3 cache the assembler is faster up to a factor of 2.7. The reason for this speed-up is prefetching. Prefetching is important in the parallel version even if the local lattice would fit into the cache, because data that stems form remote processes will not be in the cache but rather in main memory.

4.3 Performance Results

Performance results are given in Table 3 and plotted in Fig. 3. Weak scaling results are shown on the left hand side of Fig. 3. From the weak scaling we see that parallel performance is dominated by data communication overhead. When going from one core to the

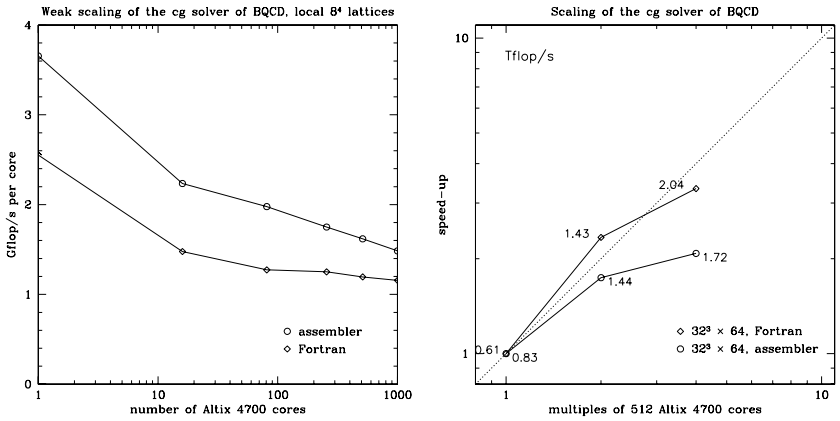


Figure 3. Weak (left) and strong scaling (right) of the conjugate gradient solver. The dotted line in the strong scaling plot indicates linear scaling.

general case of $3^4 = 81$ cores (on a single node) performance drops by a factor of about two and continues to decrease slowly when increasing the number of cores further.

Strong scaling results are shown on the right hand side of Fig. 3. The Fortran code scales super-linearly when going from 512 to 1024 cores which is clearly an effect of the large L3 cache. Remarkably, Fortran outperforms the assembler on 2048 cores. This indicates that the MPI calls, that are handled in the assembler part, lead in this case to an inefficient communication pattern.

Note that 1024 and 2048 cores do not fit into two or four nodes respectively. For a production run a 'sweet spot' had been searched and filling two nodes with local 8^4 lattices was chosen. The overall lattice size was $40^3 \times 64$ which was put onto $5^3 \times 8 = 1000$ cores. The average overall performance sustained was 1.485 Tflop/s which is 23 % of the peak performance.

5 Summary

Using lower level programming techniques improves the performance of QCD programmes significantly. The speed-up that can be achieved in comparison to programming in Fortran/MPI is a factor of 1.3–2.0.

We found that our code scales up to the whole Blue Gene/L in Jülich. The highest performance measured was 8.05 Tflop/s on the whole machine. In production typically one rack (2048 cores) is used on which a performance 1.1 Tflop/s or 19 % of peak is sustained. The high performance and scaling could be obtained by using *double hummer* instructions and techniques to overlap communication and computation.

On the Altix 4700 the large L3 data cache helps a lot to boost performance. Due to the hierarchical nature of the communication network performance measurement depend to some degree on the placement of programmes. In production an average sustained performance of 1.485 Tflop/s or 23 % of peak is achieved when using 1000 cores.

Acknowledgements. The performance measurements were done on the BlueGene/L at NIC/ZAM Jülich at the BlueGene Scaling Workshop, Jülich, 3–5 December 2006 and on the Altix 4700 at LRZ Garching/Munich. T.S. is supported by a Feodor-Lynen Fellowship.

References

1. www.top500.org
2. G. Schierholz and H. Stüben, *Optimizing the Hybrid Monte Carlo Algorithm on the Hitachi SR8000*, in: S. Wagner, W. Hanke, A. Bode and F. Durst (eds.), High Performance Computing in Science and Engineering, (2004).
3. S. Duane, A. Kennedy, B. Pendleton, D. Roweth, Phys. Lett. B, **195**, 216–222, (1987).
4. *Unfolding the IBM eServer Blue Gene Solution*, IBM Redbook, ibm.com/redbooks.
5. A. A. Bright *et al.*, IBM J. Res. & Dev., **49**, 277–287, (2005).
6. C. D. Wait *et al.*, IBM J. Res. & Dev., **49**, 249–254, (2005).
7. N. R. Adiga *et al.*, IBM J. Res. & Dev., **49**, 265–276, (2005).
8. P. Vranas *et al.*, Proceedings of the ACM/IEEE SC2006 Conference on High Performance Computing and Networking, Tampa, Florida, USA, November 2006.
9. www.lrz-muenchen.de/services/compute/hlrb/batch/batch.html

Optimizing Lattice QCD Simulations on BlueGene/L

Stefan Krieg

Department of Physics
Faculty of Mathematics and Natural Sciences
University of Wuppertal
Gaußstraße 20, D-42119 Wuppertal, Germany
Jülich Supercomputing Centre (JSC)
Research Centre Jülich
52425 Jülich, Germany
E-mail: krieg@fz-juelich.de

I describe how Lattice QCD simulations can be optimised on IBM's BlueGene/L eServer Solution computer system, focusing more on the software side of such simulations rather than on simulation algorithms. I sketch the requirements of Lattice QCD simulations in general and describe an efficient implementation of the simulation code on this architecture.

1 Introduction

The strong nuclear force binds protons and nucleons to form the atomic nucleus, countering the electric repulsion between the protons. However protons and nucleons are not fundamental particles, but are comprised of "up" and "down" quarks, two representatives of a family of 6 quarks. These are the fundamental degrees of freedom of the theory of the strong force, called Quantum Chromodynamics (QCD). The gluons are the gauge bosons in QCD, mediating the strong nuclear force, similar to photons in (Quantum) Electrodynamics. Because the coupling describing the strength of QCD interactions scales with the energy, and is large at low energy, Taylor expansions in this coupling are not valid, meaning that there is no known method of analytically calculating the low energy observables of the theory, such as particle masses. Thus it is difficult to compare the theory with experiment.

However a specific formulation of the theory, called Lattice QCD (LQCD), allows a numerical calculation of many of these observables, using a Monte-Carlo approach. A similarity between LQCD and Statistical Mechanics has allowed a fruitful interaction between the two fields, so that similar simulation algorithms are employed in both areas. The standard Monte Carlo methods used in LQCD are the Hybrid Monte Carlo algorithm[1] and its variants, comprised of a Molecular Dynamics evolution of the gluon fields, followed by a Metropolis accept/reject step. The main difference between LQCD and Statistical Mechanics is the dimensionality: QCD, being a relativistic theory, has four (space-time) dimensions. In LQCD, the quarks and gluons "live" on a four dimensional lattice and all "interactions" are essentially local, so that only nearest neighbouring (and in a few formulations next to neighbouring) lattice sites interact.

The Wilson Dirac Kernel

The run-time of a LQCD simulation is dominated by a small kernel, which multiplies the sparse Dirac Matrix with a vector. There are several different variants of the Dirac Matrix.

A commonly used option is the Wilson Dirac Matrix

$$M_{xy} = 1 - \kappa \sum_{\mu} (r - \gamma_\mu) \otimes U_{\mu\, x} \delta_{y\, x+\hat{\mu}} + (r + \gamma_\mu) \otimes U^{\dagger}_{\mu\, x-\hat{\mu}} \delta_{y\, x-\hat{\mu}}. \qquad (1.1)$$

It connects only lattice points which are nearest neighbours. γ^μ represent 4×4 complex matrices with only four nonzero entries each, the $U_\mu(x)$ matrices are 3×3 (dense complex) SU(3) matrices (these matrices are tensor multiplied in qE. 1.1) and κ encodes the quark mass. The γ matrices are fixed for every direction μ, whereas the U matrices are dynamical variables and thus different for every μ and lattice site x. The latter will therefore have to be stored in memory but are *the only* memory required for this matrix. Also apparent from Eq. 1.1 is the sparseness of the Wilson Dirac matrix: The number of flops required for a matrix-vector multiplication scales linearly with V, the number of lattice sites x in the simulation volume. The matrix dimension N is simply given by the dimension of the tensor product of a γ and a U matrix and the lattice volume V and is thus just $N = 12 \times V$. This matrix-vector multiplication (the kernel) is thus sufficiently small and simple to be optimised by hand.

2 The BlueGene/L System

The IBM Blue Gene/L eServer Solution (BGL) stems from a family of customized LQCD machines. The family history begins in 1993 with the development of QCDSP, built by a collaboration of different LQCD groups. The QCDSP systems were fully installed in 1998 and won the Gordon Bell prize for "Most Cost Effective Supercomputer" that year. Most members of the QCDSP collaboration, now joined by IBM, went on to design the QCDOC ("QCD on a Chip") computer. The development phase began in 1999 and the first machines were installed in 2004.

In December 1999, IBM announced a five-year effort to build a massively parallel computer, to be applied to the study of biomolecular phenomena such as protein folding. In 2004 the BGL, the first of a series of computers built and planned within this effort, took the position as the world's fastest computer system away from the "Earth Simulator" who held this position from 2002 to 2004.

The BGL shares several properties with the two dedicated QCD machines. It is a massively parallel machine, with the largest installation at Los Alamos National Laboratory being comprised of 64 racks containing 65,536 nodes, with 2 CPUs per node. These nodes are connected by a 3d torus network (QCDOC: 6d) with nearest neighbour connections. The two CPUs of a node can be operated in two ways: The so called "Virtual Node" (VN) mode divides the 512 MB and 4 MB 3^{rd} level cache between the two nodes which then work independently, whereas the so called "Coprocessor mode" has the first CPU doing all the computations and the second assisting with the communications. The VN mode proved to be more interesting for QCD, especially to match the 4d space-time lattice to the communication hardware topology. However, in VN mode, the communication cannot be offloaded to any communication hardware but has to be managed by the CPU itself. But the gain in peak performance by having both CPU's performing the calculations more than compensates for this.

The CPU integrates a ppc440 embedded processor and a "double" FPU optimised for complex arithmetic. The FPU operates on 128 bit registers with a 5 cycle pipeline latency,

but is capable of double precision calculations only. Single precision calculations can be performed, but they imply an on the fly conversion to double precision during the load and a conversion to single precision during the store operation. The FPU can perform one multiply-add instruction on two doubles stored in a 128 bit register, which results in 2.8 GFlops peak performance per CPU with the 700 MHz clock rate used in BGL. The double precision load/store operations always load two floating point numbers, with the first being aligned to a 16 Byte boundary, similar to the SSE2 instructions.

The compilers are capable of rearranging the code to satisfy this requirement ("auto-simdization"), but the simulation code usually is too complicated for this strategy to work without problems. When auto-simdization proves problematic, it is also possible to compile the code using the first FPU only making auto-simdization unnecessary.

3 Performance Optimization

The two cores of a BGL compute node can be run in two different modes: In Coprocessor (CO) mode the primary core does all calculations and a small part of the communications and the other core performs the remaining part of the communications. In Virtual Node (VN) mode both cores act as independent nodes. As far as MPI is concerned this doubles the number of available MPI tasks.

For LQCD with its rather simple communication patterns, the time spent in the communication part of the kernel is relatively small compared to the time spent in the serial part of the kernel. For that reason the VN mode is advantageous since it doubles the machine's peak performance. In other words: in order to achieve the same performance as in VN mode, the CO mode kernel has to have more than twice the performance per core of the parallelised VN mode kernel, since not all communications can be handed off to the secondary core. With a sustained performance of over 25.5% of machine peak (see below) that would require the CO mode kernel to reach over 51% of machine peak in order to be competitive. This is not likely since the kernel is memory bound and a single core cannot sustain the whole memory bandwidth. As a consequence most (if not all) LQCD applications including the Wilson Dirac kernel described here use VN mode.

In the remainder of this section I shall describe how the kernel's serial performance and communication part was optimised.

Optimising Single Core Performance

Since the kernel is sufficiently small and simple, it is possible and customary to optimise the serial part by hand. Several approaches are usually considered here:

- Writing the kernel in assembly
- Writing the kernel using GCC inline assembly
- Writing the kernel using compiler macros
- Writing the kernel using some assembly generator (eg. BAGEL)
- Writing some core parts of the Kernel using one of the above methods

I will focus here on a version of the kernel written using the ”intrinsics” of the IBM XLC compiler. These compiler macros are easy to use and the code generated this way performs almost equally well as assembly code.

All arithmetic operations in the kernel use complex numbers. The FPU is optimized for such calculations and is for example capable of performing a multiplication of two complex numbers (`a,b`) in only 2 clock cycles (`c` contains the result):

```
fxpmul a, b, tmp
fxcxnpma c, b, tmp, a
```

The same code using intrinsics (including load/store):

```
double _Complex alpha,beta,gamma, tmp;
alpha = __lfpd(&a);
beta = __lfpd(&b);
__fxpmul(alpha, beta, tmp);
__fxcxnpma(gamma, beta, tmp, alpha);
__stfpd(&c, gamma);
```

As can be seen from the second example, the intrinsics match the assembly instructions, but do not use explicit registers. The compiler will select the registers as well as schedule the instructions.

The most important part of the kernel is a multiplication of the (complex dense) 3×3 SU(3) matrix, representing the gluon fields, with a vector. The intrinsics code for the first component of the result vector is:

```
tmp  = __fxpmul(su3_00, __creal(vec0));
tmp  = __fxcxnpma(tmp, su3_00, __cimag(vec0));
tmp  = __fxcpmadd(tmp, su3_01, __creal(vec1));
tmp  = __fxcxnpma(tmp, su3_01, __cimag(vec1));
tmp  = __fxcpmadd(tmp, su3_02, __creal(vec2));
tmp  = __fxcxnpma(tmp, su3_02, __cimag(vec2));
```

This code could run at about 92% of peak, (not considering pipeline issues). In order to avoid stalling the pipeline, two complete matrix vector multiplications should be performed simultaneously.

Typically LQCD calculations in general and the kernel operations in particular are not only limited by the memory bandwidth but also feel the memory latency because of non-sequential memory access patterns. Thus prefetching is another very important ingredient for the kernel optimisation. The intrisic prefetch instruction (`__dcbt(*void)`) will prefetch one 32 byte cacheline including the specified pointer. All scheduling will be done by the compiler, so one only has to identify the data that has to be prefetched. A strategy that proved successful is to prefetch the next SU(3) matrix and next vector, while performing the previous calculation. This will optimize for the case that the data is already in the 3^{rd} level cache, which can be assumed since memory bandwidth is limited.

Optimising Communications

Since LQCD uses a 4d space-time lattice and only nearest neighbour interactions, the obvious strategy for communication is to match the lattice dimensions with the communication

hardware layout, that is to parallelise in all 4 dimensions (the 4th direction is along the two CPU's on a node). A typical MPI communicator will have the dimension $8 \times 8 \times 8 \times 2$, here matching the communication layout to the smallest partition of BGL (being a torus), the so called "mid-plane" (512 nodes/1024 CPUs). The latest versions of MPI will usually make sure that a node's coordinates in a communicator agree with its hardware coordinates in the job's partition. This is rather important, since a mismatch can severely impact performance[a]. A useful tool to check whether this really is the case are the run-time system calls "`rts_get_personality(..)`", which returns a structure containing the node's coordinates, and "`rts_get_processor_id()`", which returns the calling CPU's ID inside the node (0 or 1).

Thin interplay between communication and calculation can be highly optimised if the special LCQD communication APIs are used. In that case many different communication strategies can be applied. Since for single precision the performance of the kernel is the same with the QCD API or MPI, I will focus on the MPI implementation here. For MPI in general there are two different communication strategies that can be used (on a machine that cannot offload communications):

- Either first communicate the boundaries of the local sub-lattices and then do all the calculations
- or first carry out some part of the calculation, communicate intermediate results cutting the amount of data to be communicated by half, and then do the remaining calculations.

On the BGL the second strategy proved to be best, and all results quoted are for a kernel using this method. The communication buffers needed within this approach can be located in an area of memory with a special caching strategy: Since they are only written during the preparation of the communication and only read once, it is advantageous to put them in a chunk of memory with "store-without-allocate" caching strategy. The run-time call for this is "`rts_get_dram_window(.., RTS_STORE_WITHOUT_ALLOCATE, ..)`". Data stored to memory allocated in this way will not pollute the L1 cache but be directly stored to the lower memory hierarchies, in this case the L3.

Since in the case of the kernel all pointers to memory that are involved in the communications do never change, the use of persistent sends and receives ("`MPI_PSend, MPI_PRecv, MPI_Startall, MPI_Waitall`") has proved to be the best choice[b]. This also allows the MPI to make sure that all send/receive fifos are kept busy simultaneously, thus optimizing the bandwidth.

Making Use of Single Precision Arithmetics

Since the FPU of the BGL compute node supports only double precision arithmetics both single and double precision calculations have the same peak performance. The compute kernel however is memory bandwidth limited. Thus the single precision kernel will reach

[a] The mapping of the MPI Cartesian communicator to the hardware torus can be easily set with the "`BGLMPI_MAPPING=TZYX`" environment variable by changing the order of the default "`XYZT`".

[b] The environment variable "`BGLMPI_PACING=N`" can be set to switch off the packet pacing, since the dominant communication is nearest neighbours only. This results in slightly smaller communication time.

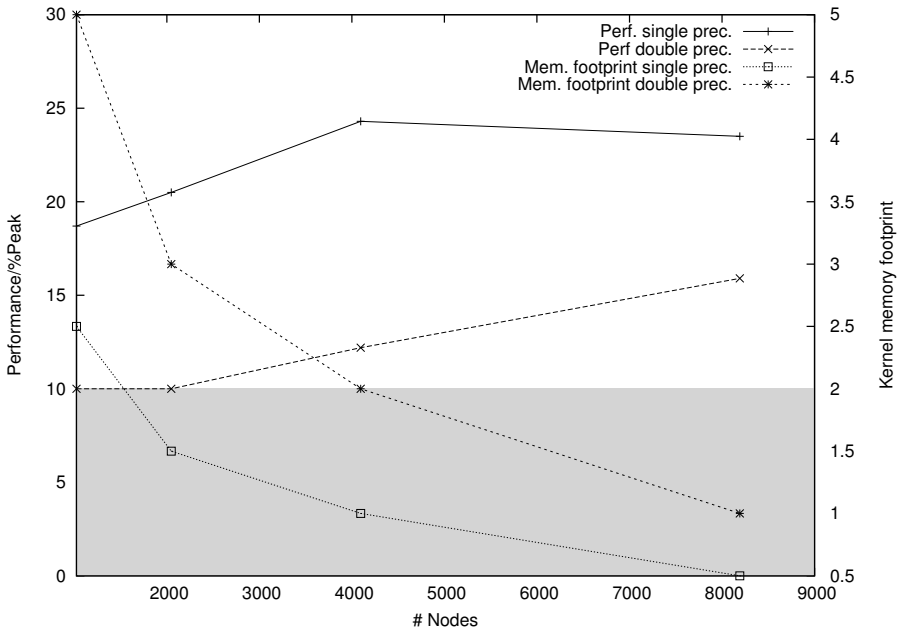


Figure 1. Performance and memory footprint of the single and the double precision kernels as a function of the number of nodes in a strong scaling analysis (global $48^3 \times 96$ lattice). The grey shaded area is the L3 cache size.

a higher performance since it has half the memory and communication bandwidth requirements. Since it also has half the memory footprint of the double precision kernel, scaling will also be improved (see Fig.1). This improvement comes at virtually no coding costs: the only difference between the single precision and the double precision kernel is the load/store instructions and prefetches.

Making use of the single precision kernel is not trivial from a mathematical point of view, since most LQCD calculation require double precision accuracy. For our simulations with dynamical overlap fermions however we can use our inverter scheme, the relaxed GMRESR with SUMR preconditioner[2] and replace the double precision with a single precision preconditioner. Depending on the lattice size almost all computer time will be spent in this preconditioner and thus the performance of the single precision kernel will, to a good accuracy, dominate the performance of the whole code. There certainly is a price to be paid by switching to the single precision SUMR: the total number of calls to the kernel increases slightly compared to the full double precision calculation. Up to lattice sizes of $48^3 \times 64$ this increase however always stayed well below 10% and is thus much smaller than the gain of using double precision (comp. figures 1 and 2).

4 Performance Results and Scaling

Since all communications are constrained to a node's nearest neighbours, and the BGL's network is a torus, weak scaling is perfectly linear. In Fig.2 both the performance for

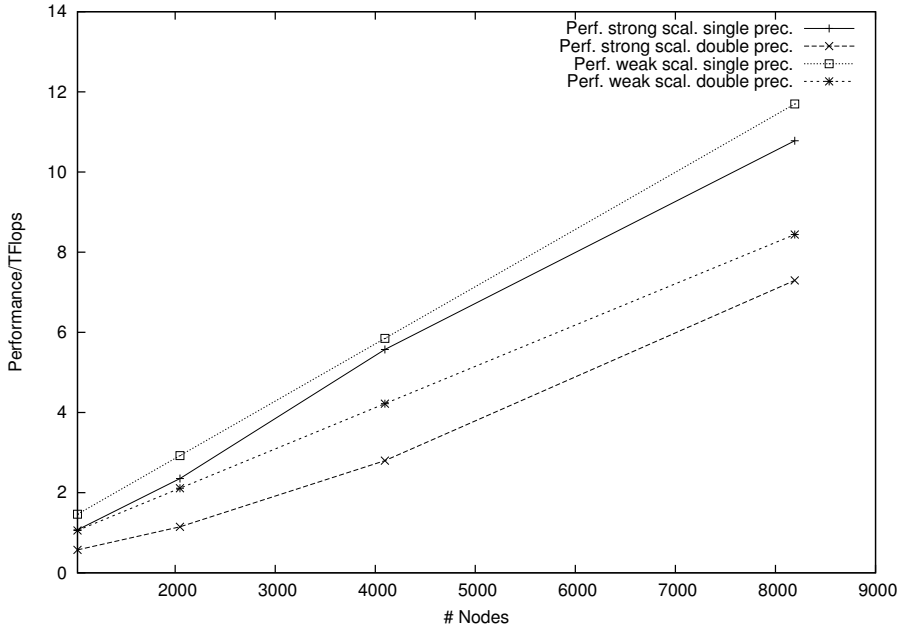


Figure 2. Performance of the single and the double precision kernels as a function of the number of nodes in a weak (local $4^3 \times 8$ lattice) and strong scaling analysis (strong $48^3 \times 96$).

a weak and strong scaling analysis of the kernel are shown. For weak scaling, the kernel reaches 25.5% of machine peak for single and 18.4% of machine peak for double precision accuracy. The performance numbers for the strong scaling analysis are plotted in Fig.1 and reach 24.5% of machine peak single and 16.0% of machine peak in double precision accuracy.

As long as the kernel memory footprint fits into the L3 cache, the performance of the kernel matrix always exceeds 20% of machine peak. (The relative dip of performance at 2k nodes is due to a suboptimal local lattice layout.) The scaling region is therefore quite large and reaches up to 8k nodes for the global lattice chosen here.

5 Conclusions and Outlook

I have shown how to optimise the Wilson Dirac kernel for the IBM Blue Gene/L architecture with the IBM XLC compiler macros and MPI. The kernel scales up to the whole machine and reaches a performance of over 11 TFlop/s in single precision and over 8.4 TFlop/s in double precision on the whole 8 Rack Blue Gene/L "JUBL" at the Jülich Supercomputing Centre (JSC) of the Research Centre Jülich.

The JSC will install a 16 Rack Blue Gene/P (BGP) with 220 TFlop/s peak performance end of 2007. A first implementation of the (even/odd preconditioned) Wilson Dirac kernel shows that this too is a particularly well suited architecture for LQCD: the sustained per node performance of the kernel went up from 1.3 GFlop/s (1.0 GFlop/s) on BGL to

4.3 Gflop/s (3.3 GFlop/s) or 31.5% (24.3%) of machine peak on BGP in single precison (double precision) accuracy.

Acknowledgements

I would like to thank Pavlos Vranas, then IBM Watson Research, for many valuable discussions, Bob Walkup of IBM Watson Research for his help during the first stages of the project, Charles Archer of IBM Rochester for his help with the BGL MPI and Nigel Cundy for his ongoing collaboration. I am indebted to Jutta Docter and Michael Stephan, both JSC, for their constant help with the machine.

References

1. S. Duane, A. Kennedy, B. Pendelton and D. Roweth, Phys. Lett. B, **195**, 216 (1987).
2. N. Cundy, J. van den Eshof, A. Frommer, S. Krieg, Th. Lippert and K. Schäfer, *Numerical methods for the QCD overlap operator. III: Nested iterations*, Comp. Phys. Comm., **165**, 841, (2005). hep-lat/0405003.
3. P. Vranas et al., The Blue Gene/L Supercomputer and Quantum Chromo Dynamics
4. N. R. Adiga et al, *BlueGene/L torus interconnect network*, IBM Journal of Research and Development Vol. **49**, Number 2/3, (2005).
5. IBM Redbook ”Blue Gene/L: Application Development”
6. IBM J. Res. & Dev. Vol. **49**, March/May, (2005)

Parallel Computing with FPGAs

IANUS: Scientific Computing on an FPGA-Based Architecture

Francesco Belletti[1,2], Maria Cotallo[3,4], Andres Cruz[3,4], Luis Antonio Fernández[5,4], Antonio Gordillo[6,4], Andrea Maiorano[1,4], Filippo Mantovani[1,2], Enzo Marinari[7], Victor Martín-Mayor[5,4], Antonio Muñoz-Sudupe[5,4], Denis Navarro[8,9], Sergio Pérez-Gaviro[3,4], Mauro Rossi[10], Juan Jesus Ruiz-Lorenzo[6,4], Sebastiano Fabio Schifano[1,2], Daniele Sciretti[3,4], Alfonso Tarancón[3,4], Raffaele Tripiccione[1,2], and Jose Luis Velasco[3,4]

[1] Dipartimento di Fisica, Università di Ferrara, I-44100 Ferrara (Italy)

[2] INFN, Sezione di Ferrara, I-44100 Ferrara (Italy)

[3] Departamento de Física Teórica, Facultad de Ciencias
Universidad de Zaragoza, 50009 Zaragoza (Spain)

[4] Instituto de Biocomputación y Física de Sistemas Complejos (BIFI), 50009 Zaragoza (Spain)

[5] Departamento de Física Teórica, Facultad de Ciencias Físicas
Universidad Complutense, 28040 Madrid (Spain)

[6] Departamento de Física, Facultad de Ciencia
Universidad de Extremadura, 06071, Badajoz (Spain)

[7] Dipartimento di Fisica, Università di Roma "La Sapienza", I-00100 Roma (Italy)

[8] Departamento de Ingenieria Electrónica y Comunicaciones
Universidad de Zaragoza, CPS, Maria de Luna 1, 50018 Zaragoza (Spain)

[9] Instituto de Investigación en Ingenieria de Aragón (I3A)
Universidad de Zaragoza, Maria de Luna 3, 50018 Zaragoza (Spain)

[10] ETH Lab - Eurotech Group, I-33020 Amaro (Italy)

This paper describes the architecture and FPGA-based implementation of a massively parallel processing system (IANUS), carefully tailored to the computing requirements of a class of simulation problems relevant in statistical physics. We first discuss the system architecture in general and then focus on the configuration of the system for Monte Carlo simulation of spin-glass systems. This is the first large-scale application of the machine, on which IANUS achieves impressive performance. Our architecture uses large-scale on chip parallelism ($\simeq$ 1000 computing cores on each processor) so it is a relevant example in the quickly expanding field of many-core architectures.

1 Overview

Monte Carlo simulations are widely used in several fields of theoretical physics, including, among others, Lattice Gauge Theories, statistical mechanics[1], optimization (such as, for instance, K-sat problems).

In several cases, the simulation of even *small* systems, described by simple dynamic equations requires huge computing resources. Spin systems - discrete systems, whose

variables (that we call *spins*) sit at the vertices of a $D-$dimensional lattice - fall in this category[2]. Reaching statistical equilibrium for a lattice of $48 \times 48 \times 48$ sites requires $10^{12} \ldots 10^{13}$ Monte Carlo steps, to be performed on $O(100)$ copies of the system, corresponding to $\simeq 10^{20}$ spin updates. Careful programming on traditional computers (introducing some amount of SIMD processing) yields an average spin-update time of $\simeq 1$ ns: a typical simulation would use 1 PC for $\simeq 10^4$ years (or $\simeq 10000$ PCs for 1 year, optimistically and unrealistically assuming that perfect scaling takes place!) The structure of simulation algorithms for spin system are however very well suited to some unconventional computer architectures that are clearly emerging at this point in time (see, e.g.[3]).

This paper describes the architecture and the FPGA-based implementation of a massively parallel system carefully tailored for the class of applications discussed above. The system, that we call IANUS, has been developed by a collaboration of the Universities of Ferrara, Roma I and Zaragoza, and of Instituto de Biocomputación y Física de Sistemas Complejos (BIFI), with Eurotech as industrial partner. Our text is structured as follows: Section 2 builds a bridge between the structure of the physical problem and its computing requirements; Section 3 describes our FPGA-based architecture and discusses how we configure it for our application. Section 4 reports on performance and compares with corresponding figures for traditional PC clusters and for at least one recently appeared architecture. Section 5 contains our concluding remarks.

2 Sample Application: Spin Systems on a Lattice

We briefly describe here discrete spin systems, the application that has triggered IANUS development. Consider $N = L \times L \times L$ nodes labeled by $i = \{0, \ldots, N-1\}$ and arranged on a 3D cubic grid. On each node we place a *spin* variable s_i that only takes discrete values $\{+1, -1\}$. An *energy cost* is associated to every pair of *nearest neighbour* sites on the grid (each site has six nearest neighbors, two for each direction x, y and z), which is proportional to the product of the values of the two spins.

$$\epsilon_{ij} = -J_{ij} s_i s_j$$

the proportionality constants J_{ij}, usually called *couplings*, are in general different for each pair of sites, and take the values $+1$ or -1. For a positive coupling $J_{ij} = 1$, the situation in which two neighboring spins are *parallel* (they takes the same value), is energetically favoured. Negative couplings favour *anti-parallel* (mutually opposite) neighboring spins. Periodic boundary conditions are usually applied so the system is a 3D discrete torus. The sum of energy contributions for all pairs of neighboring sites is by definition the energy function of the system.

$$H = \sum_{\langle i,j \rangle} \epsilon_{ij} = -\sum_{\langle i,j \rangle} J_{ij} s_i s_j \tag{2.1}$$

The notation $\langle i, j \rangle$ means that the summation is done only on neighboring sites. The set of all J_{ij} is kept fixed during a simulation (the set is extracted from distribution probabilities that fix the physical properties of the systems described by the model: if $J_{ij} = 1 \; \forall i, j$ we have the Ising model of ferromagnetism, while random equiprobable values $+1, -1$ are considered in the Edwards-Anderson Spin Glass model).

The variables s_i evolve in time according to some specified rules, that we now briefly describe. Given a system with energy function H, one is usually interested in its properties at constant temperature $T = 1/\beta$ (system in equilibrium with a heat source); every configuration of spins $\{S\}$ contributes to thermodynamic properties with a weight factor given by the Boltzmann probability $P \propto \exp(-\beta H[\{S\}])$.

The Monte Carlo Metropolis algorithm is a standard tool to generate configurations according to the Boltzmann probability. As such, it is a key computational algorithm in this field. It may be described by the following steps:

1. choose a site at random in the lattice, and compute the local energy E of the corresponding spin (containing only contributions of nearest neighbour couplings);

2. perform the move: flip the chosen spin and compute the new local energy E' and the energy variation of the move

$$\Delta E = E' - E$$

3. accept the move with probability $P = \min[1, \exp(-\beta \Delta E)]$;

4. go back to 1;

A Monte Carlo *sweep* is usually taken as a number of iterations of the elementary steps described above equal to the volume N of the lattice. Several (in practice, a huge number of) Monte Carlo sweeps are needed in order to reach the equilibrium distribution. It can be shown that the asymptotic behaviour is the same if we choose to visit sites in any given lexicographic order, so a sweep is usually performed by sequentially visiting sites.

Inspection of the algorithm and of equation (2.1) shows that the procedure has several properties relevant for an efficient implementation:

- a large amount of parallelism is available: each site interacts with its nearest neighbors only, so up to one-half of all sites, organized in a checkerboard structure, can be handled in parallel (two neighboring spins may not be updated simultaneously since local transition probability must be well-defined at each step), provided enough random numbers are available;

- the computational kernel has a regular loop structure. At each iteration the same set of operations is performed on data words whose addresses can be computed in advance;

- data processing is associated to bit-manipulation (as opposed to arithmetics performed on long data words), since bit valued variables are involved ;

- the data base associated to the computation is very small, just a few bytes for each site (e.g., $\simeq 100$ KBytes for a grid of 48^3 sites).

In brief, the algorithm is an obvious target for aggressive parallelization, and, for a given budget of logical resources, parallelization can be pursued with greater efficiency if very simple processing elements are available.

3 IANUS Architecture

We try to match the architectural features described in the previous section with an FPGA-based architecture, following earlier attempts[4,5] and leveraging on technology advances. We use latest generation FPGAs, each accommodating several hundreds (see later for details) processing engines. A dramatic memory access bottleneck follows, as all engines process one lattice site each, since several thousands data bits must be moved from memory to the processing engines at each clock cycle. This bandwidth can only be sustained by memory available on-chip that, in current generation FPGAs, is large enough for the data-set.

The system that we have built uses two hierarchical levels of parallelism:

- our system is based on a 4×4 grid of processing elements (that we call SPs) with nearest-neighbour links (and periodic boundary conditions). All SPs are also connected to a so-called IO-Processor (IOP), that merges and moves data to a host computer via 2 Gigabit-Ethernet links. Both the IOP and the SP are implemented by Xilinx Virtex4-LX160 or Virtex4-LX200 FPGAs.
- The SP processor contains uncommitted logic that can be configured at will (fast on the fly reconfiguration of all SPs is handled by the IOP). In our typical application, the SP becomes a *many*-core processor (we use the terminology proposed recently in[3]), each core performing the same algorithm on a subset of the spins (see later for details).

A block diagram of the system is shown in Fig. 1, while a picture of the IANUS prototype is shown in Fig. 2

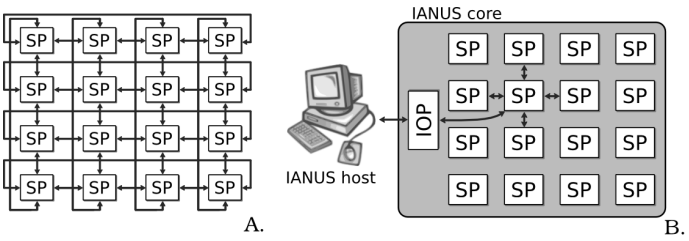


Figure 1. IANUS topology (A) and overview of the IANUS prototype implementation (B), based on a processing board (PB) with 16 SPs and one I/O module (IOP).

The computational architecture that we configure on each SP for the first large-scale IANUS application - the Monte Carlo simulation described above - is shown in Fig. 3.

As a guide to understand the structure, we have a set of processing engines (also called update cells, UC), that receive all the variables and parameters needed to update one spin and perform all needed arithmetic and logic operations, producing updated values of the spin variable. Data (variables and parameters) are kept in memories and fed to the appropriate UC. Updated values are written back to memory, to be used for later updates.

The choice of an appropriate storage structure for data and the provision of appropriate data channels to feed all UCs with the value they need is a complex challenge; designing

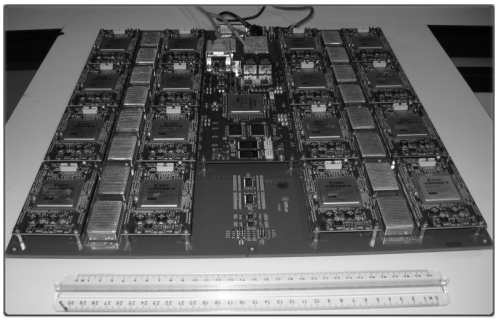


Figure 2. IANUS board prototype: SPs are accommodated in 4 lines of 4 processors each while the I/O processor is placed at the centre.

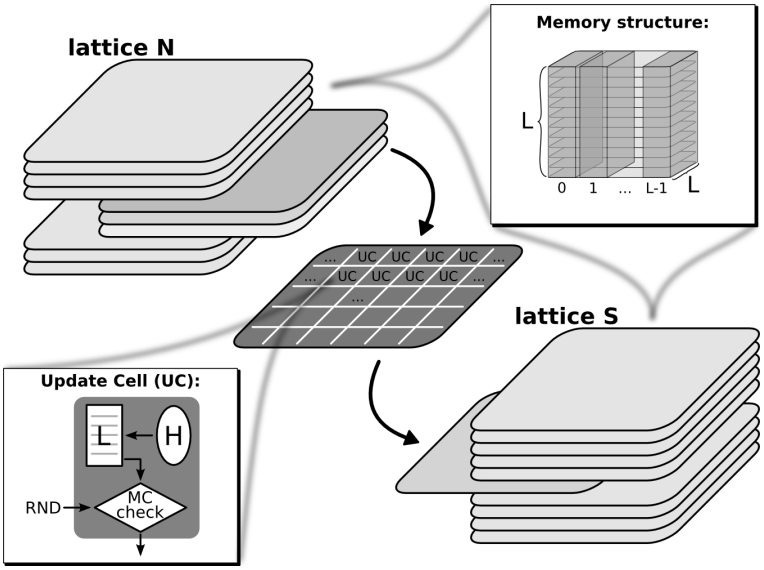


Figure 3. Parallel update scheme. The spins (lattice S) and their neighbors (lattice N) are stored in RAM blocks. All $L \times L$ spins in a plane in the S lattice of vertical coordinate (memory address) z are updated simultaneously. A set of $L \times L$ Update Cells (UC) are implemented in logic; at each clock cycle they receive all needed neighboring spins (the lattice planes of vertical coordinates z, $z-1$, $z+1$). Note that at regime we only need to fetch one plane from the N lattice, as the other two are available from the previous iteration. The outputs of the UCs are the updated spins, stored into the S memories at address z. The structure is slightly more complex than described here as we also need to feed UCs with the coupling values. When all planes in S have been processed, S and N interchange their roles.

the update cells is a comparatively minor task, since these elements perform rather simple logic functions.

We now briefly describe our data structure (the reader may find reference to Fig. 3 helpful to follow the discussion). As pointed out before, we can arrange for parallel update of up to one half of the lattice, processing all *white* (*black*) sites in a checkerboard scheme

at once. In order to maximize performance, we resort to a simple trick, by replicating the system twice: the two *replicas* share the same coupling data base, but follow independent dynamics, doubling the overall statistics. We arrange all white sites of replica 1 and black sites of replica 2 in one lattice (stored in *S*, the *updating spin* block, see Fig. 3) and all black sites of replica 1 and white sites of replica 2 in another lattice (the *neighbors block*, called *N* in Fig. 3). With this arrangement, all spins in the updating block have their neighbors in the neighbors block only. These artificial lattices have the same topology as the original ones, and are mapped directly onto embedded RAM blocks on the FPGA. Consider the simple case of an $L = 16$ system. The updating spin lattice (and of course the neighbors lattice) may be stored in 16 RAM blocks (whose labels are the x coordinates), each 16 bit wide (bit index is the y coordinate) and 16 word deep (memory addresses are z coordinates). We need three similar structures to store the couplings J_{ij} one per each coordinate axis (each site has two links to neighbors in each direction). Fetching one 16 bit word for each RAM block at the same address z corresponds to fetching a full (xy) plane of the updating spin block (or the neighbors or coupling blocks in one direction). This scheme is easily generalized to several values of lattice size L. Of course only fractions of entire planes may be updated for very large lattices (since in this case there are not enough logical resources within the FPGA), but this can be easily managed.

Each update step consumes one (pseudo)-random number. Random number generators use a large fraction of the available hardware resources. We implemented the Parisi-Rapuano shift-register wheel[7], defined by the rules

$$\begin{aligned} I(k) &= I(k-24) + I(k-55) \\ R(k) &= I(k) \otimes I(k-61)\ , \end{aligned} \tag{3.1}$$

where $I(k-24)$, $I(k-55)$ and $I(k-61)$ are elements (32-bit wide in our case) of a shift register initialized with externally generated values. $I(k)$ is the new element of the wheel, and $R(k)$ is the generated pseudo-random value. This generation rule is easily arranged on configurable logic only, so, by exploiting cascade structures, each wheel produces about one hundred values per clock cycle.

After spins are fed to an update cell, the local (integer) energy values are computed and used to address a look-up table whose entries are pre-calculated Boltzmann probability values (normalized to $2^{32}-1$). The probability is compared with the 32 bit random number in order to decide the spin's fate. Look-up tables (LUTs) are implemented in distributed RAM whose availability is one limiting factor in the number of possible parallel updates. On the other side, LUTs are small data sets so arranging them on RAM blocks would be a waste of precious storage. Since two reads are possible from each LUT in one clock cycle, each one is shared between two update cells. A more detailed description can be found in in[8].

We accommodate all memory structures, 512 update cells and a matching number of random generators and LUTs on the XILINX-LX160 FPGA available on the SP. A larger FPGA (the XILINX-LX200 that will be used for the full-scale IANUS system) doubles all values. Independent simulations are carried out on each SP belonging to the IANUS system, since several simulations associated to different parameters of the system are needed anyway. The number of update cells corresponds to the number of updates per clock cycle (512 on the LX160 and 1024 on the LX200). Note that at 1024 updates per clock cycle and 62.5 MHz clock frequency, the random number generator alone runs at 64 Gops (32-bit)

on each node (1 Tops per IANUS board), even neglecting the cost of XOR operations.

In short, we are in the rewarding situation in which: i) the algorithm offers a large degree of allowed parallelism, ii) the processor architecture does not introduce any bottleneck to the actual exploitation of the available parallelism, iii) performance of the actual implementation is only limited by the hardware resources contained in the FPGAs.

4 Performance

The algorithms described above run on our FPGA with a system clock of 62.5 MHz, corresponding to an average update time of $1/(512 \times 62.5 \times 10^6) = 32$ ps/spin (16 ps/spin in the LX200 version, as the number of update cells doubles) per FPGA, that is 2 ps/spin (1 ps/spin) for a full IANUS processing board (16 FPGAs).

It is interesting to compare these figures with those appropriate for a PC. Understanding what exactly has to be compared is not fully trivial. Popular PC codes update in parallel the *same* site of a large number of (up to 128) replicas of the system, each mapped onto one bit of the processor word. The same random number is shared by all replicas. This scheme (we call it Asynchronous Multi Spin Coding, AMSC) extracts reasonable performance from architectures with large word sizes in a context in which the natural variable size is just one bit. This approach is useful for statistical analysis on large samples. On the other hand, it is less appropriate when a *small* number of replicas of a large system must be updated *many* (e.g., $10^{12} \ldots 10^{13}$) times. In this case an algorithm that updates in parallel many spins of the same lattice (exactly what we do in IANUS) is a much better choice. Such codes (that we call Synchronous Multi Spin Coding, SMSC) were proposed several years ago, but never widely used. We put a rather large effort in developing efficient AMSC and SMSC codes on a high-end PC (an Intel Core2Duo - 64 bit - 1.6 GHz processor). A comparison - see Table 1 - shows that one IANUS processing board has a performance of hundreds (or even thousands) of PCs.

Preliminary measurements of these codes ported to the IBM Cell Broadband Engine (CBE) (work is still in progress in this area) show that one CBE (using all 8 synergistic processors) is approximatively $10 - 15$ times faster than an Intel Core 2 Duo PC.

	LX160	LX200	PC (SMSC)	PC (AMSC)
Update Rate	2 ps/spin	1 ps/spin	3000 ps/spin	700 ps/spin

Table 1. Comparing the performances of one IANUS processing board and two different PC codes.

Finally note that the mismatch between processing power available on the system and bandwidth to the host is such that a full simulation run must be executed on IANUS; this is at variance with recent architectures in which FPGA co-processors are directly attached to a traditional CPU and executes relatively fine-grained tasks.

Our planned IANUS installation (16 sub-systems of 16 processors each, expected for fall 2007) has a processing power equivalent to ≥ 10000 PCs or approximately 1000 CBEs, making it possible to carry out the physics program outlined above in about one year time. IANUS will be housed in one standard rack; power dissipation will be $\simeq 4$ KW, a huge improvement with respect to a large PC farm.

5 Conclusions

Our performance are admittedly obtained by carefully handcrafting an appropriate architecture for a parallel-friendly algorithm. This experience teaches, in our opinion, some lessons relevant in a more general context:

- we have put in practice the potential for performance of a *many*-core architecture, exposing all parallelization opportunities available in the application.
- The huge performances that we obtain depend on huge bandwidth to/from memory, only achievable with *on-chip* embedded memories.
- FPGAs can be configured to perform functions poorly done by traditional CPUs, exploiting a large fraction of available resources. This compensates the overheads associated to reconfigurability and inherent to any FPGA structures.

At present we are fine-tuning for IANUS a Monte Carlo procedure for random graph colouring (a prototype K-sat problem), for which we also expect large performance gains with respect to PCs. This code, that we describe elsewhere, will assess the performance potential of FPGA-based processors for algorithms with irregular memory access patterns.

Acknowledgements

IANUS has been partially funded by the UE (FEDER funds) and by Diputación General de Aragón (Spain) and supported by the Spanish MEC (FIS2006-08533 and TEC2004-02545).

References

1. D. P. Landau and K. Binder, *A Guide to Monte Carlo Simulations in Statistical Physics*, (Cambridge University Press 2005).
2. See for instance *Spin Glasses and Random Fields*, edited by P. Young, (World Scientific, Singapore, 1998).
3. K. Asanovic et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. Report UCB/EECS-2006-183, (2006).
4. J. H. Condon and A. T. Ogielski, *Fast special purpose computer for Monte Carlo simulations in statistical physics.*, Rev. Sci. Instruments, **56**, 1691–1696, (1985).
5. A. Cruz, et al., *SUE: A Special Purpose Computer for Spin Glass Models*, Comp. Phys. Comm., **133**, 165, (2001) .
6. F. Belletti et al., *IANUS: An Adaptive FPGA Computer*, Computing in Science and Engineering, **71**, 41, (2006).
7. G. Parisi and F. Rapuano, *Effects of the random number generator on computer simulations*, Phys. Lett. B, **157**, 301–302,(1985).
8. F. Belletti et al., *Simulating spin systems on IANUS, an FPGA-based computer*, http://arxiv.org/abs/0704.3573, Comp. Phys. Comm., (2008, in press).

Optimizing Matrix Multiplication on Heterogeneous Reconfigurable Systems

Ling Zhuo and Viktor K. Prasanna

Ming Hsieh Department of Electrical Engineering
University of Southern California
Los Angeles, USA
E-mail: {*lzhuo, prasanna*}*@usc.edu*

With the rapid advances in technology, FPGAs have become an attractive option for acceleration of scientific applications. In particular, reconfigurable computing systems have been built which combine FPGAs and general-purpose processors to achieve high performance. Previous work assumes the nodes in such systems are homogeneous, containing both processors and FPGAs. However, in reality, the nodes can be heterogeneous, based on either FPGAs, processors, or both. In this paper, we model these heterogeneous reconfigurable systems using various parameters, including the computing capacities of the nodes, the size of memory, the memory bandwidth, and the network bandwidth. Based on the model, we propose a design for matrix multiplication that fully utilizes the computing capacity of a system and adapts to various heterogeneous settings. To illustrate our ideas, the proposed design is implemented on Cray XD1. Heterogeneous nodes are generated by using only the FPGAs or the processors in some nodes. Experimental results show that our design achieves up to 80% of the total computing capacity of the system and more than 90% of the performance predicted by the model.

1 Introduction

Field-Programmable Gate Arrays (FPGAs) are a form of reconfigurable hardware. They offer the design flexibility of software, but with time performance closer to Application Specific Integrated Circuits (ASICs). With the increasing computing power of FPGAs, high-end systems have been built that employ them as application-specific accelerators. Such systems include SRC 6 and 7[1], Cray XD1 and XT3[2], and SGI RASC[3]. These systems contain multiple nodes that are connected through an interconnect network. Each node is based on either FPGAs, general-purpose processors, or both. It has been shown that such systems can achieve higher performance than systems with processors only[4,5].

Reconfigurable computing systems contain multiple forms of heterogeneity. For example, the nodes of these systems can be based on either processors, FPGAs, or both. Also, due to its reconfigurability, the computing capacity of the FPGA in a node varies based on the designs implemented on it. Moreover, the nodes have access to multiple levels of memory with various sizes and various bandwidths. To fully utilize the available computing capacity of the system, all these heterogeneity factors need to be considered.

In this paper, we propose a model to facilitate workload allocation and load balancing in heterogeneous reconfigurable systems. Each node of the system is characterized by various parameters, including its computing capacity, the available memory size, and memory bandwidth. The model also considers the time for transferring data from the main memory of the processors to the FPGAs and the network communication costs of executing an application over heterogeneous nodes.

Using the proposed model, we develop a design for floating-point matrix multiplication. The input matrices are partitioned into blocks and the block multiplications are distributed among the nodes. Multiple levels of memory available to the FPGAs are employed to reduce memory transfer time. In addition, the design scales across multiple heterogeneous nodes by overlapping the network communications with computations.

To illustrate our ideas, we implemented the proposed design on 6 nodes of Cray XD1[2]. The nodes of XD1 contain both AMD Opteron processors and Xilinx Virtex-II Pro FPGAs. In our experiments, we chose to use only the FPGAs, or only the processors, or both to generate heterogeneous nodes. Experimental results show that in various heterogeneous settings, our design achieves up to 80% of the total computing capacity of the system. In addition, our design achieves more than 90% of the performance predicted by the model.

The rest of the paper is organized as follows. Section 2 introduces related work and presents several representative reconfigurable computing systems. Section 3 proposes our model. Section 4 proposes a design for matrix multiplication based on our model. Section 5 presents the experimental results. Section 6 concludes the paper.

2 Related Work and Background

Heterogeneous systems have been studied extensively[7,8]. Such systems contain processors with various computing capacities which are interconnected into a single unified system. Task scheduling algorithms have been proposed to minimize the execution time of an application[7,8]. In these systems, it is assumed that all tasks can be executed on any processors. However, this may not be true for reconfigurable computing systems because not all tasks are suitable for hardware implementation.

Heterogeneous distributed embedded systems combine processors, ASICs and FPGAs using communication links. Hardware/software co-synthesis techniques have been proposed to perform allocation, scheduling, and performance estimation[9,10]. However, such techniques cannot be applied to heterogeneous reconfigurable systems straightforwardly due to the complex memory hierarchy and multiple nodes in these systems.

Many reconfigurable computing systems have become available. One representative system is Cray XD1[2]. The basic architectural unit of XD1 is a compute blade, which contains two AMD 2.2 GHz processors and one Xilinx Virtex-II Pro XC2VP50. Each FPGA has access to four banks of QDR II SRAM, and to the DRAM memory of the processors. In SRC 7[1], the basic architectural unit contains one Intel microprocessor and one reconfigurable logic resource called MAP processor. One MAP processor consists of two FPGAs and one FPGA-based controller. Each FPGA has access to six banks of SRAM memory. The FPGA controller facilitates communication and memory sharing between the processor and the FPGAs. SGI has also proposed Reconfigurable Application Specific Computing (RASC) technology, which provides hardware acceleration to SGI Altix servers[3]. In an SGI RASC RC1000 blade, two Xilinx Virtex-4 FPGAs are connected to 80 GB SRAM memory. Each blade is directly connected to the shared global memory in the system through the SGI NUMAlink4 interconnect.

Hybrid designs have been proposed for reconfigurable computing systems that utilize both the processors and the FPGAs[4,11]. In some work, the computationally intensive part of a molecular dynamics simulation is executed on the FPGAs, while the remaining part runs on the processors[4]. In our prior work, the workload of several linear algebra applications

is partitioned between the processors and the FPGAs[11] so that they are both effectively utilized. However, all these works assume that the nodes are homogeneous and none of them addresses the heterogeneity in the systems. The authors of[12] investigated scheduling algorithms for heterogeneous reconfigurable systems. However, that work only utilizes a single node while our work considers multiple nodes.

3 Model for Reconfigurable Computing Systems

Reconfigurable computing systems can be seen as distributed systems with multiple nodes connected by an interconnect network. The nodes are based on either general-purpose processors, FPGAs, or both. Each node has its own local memory, and may have access to the memory of a remote node. Suppose the system contains p nodes, including p_1 P(Processor)-nodes, p_2 F(FPGA)-nodes, and p_3 H(Hybrid)-nodes. The architectural model of such a system is shown in Fig. 1. The nodes are denoted as N_0, N_1, ..., N_{p-1}. "GPP" in the figure stands for "general-purpose processor".

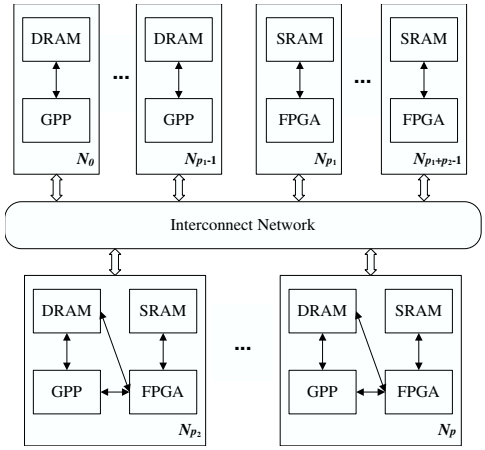


Figure 1. Architectural model of heterogeneous reconfigurable computing systems

In our model, we use C_i to refer to the computing capacity of N_i ($i = 0, \ldots, p-1$). As we target scientific computing applications, the computing capacity of a node equals its floating-point sustained performance for a given application, measured in GFLOPS (one billion floating-point operations per second). If N_i is a P-node, C_i is obtained by executing a program for the application. If N_i is an F-node, C_i is determined by the FPGA-based design for the application. Suppose the design performs O number of floating-point operations in each clock cycle and its clock speed is F. Thus, $C_i = O \times F$. For an H-node, C_i depends not only on the processor-based program and the FPGA-based design, but also on the workload partitioning and coordination between the processor and the FPGA.

Each node in the system has its own memory hierarchy. As we focus on FPGA-based designs, our model considers only the storage capacity and memory bandwidth available to the FPGAs. Designs on F-nodes have access to multiple levels of memory. The first level is the on-chip memory of the FPGA, usually Block RAMs (BRAMs). The second level is off-chip but on-board memory, which is usually SRAM. The FPGAs also have access to the DRAM memory of the adjacent processors.

In our model, the FPGA-based design reads the source data from the DRAM memory and stores the intermediate results in the SRAM memory. Thus, two important parameters are the size of the SRAM memory and the bandwidth between the FPGA and the DRAM memory. These two parameters are denoted as S and bw_d, respectively. Note that our model does not consider memory access latency because data is streamed into and out of the FPGAs for the applications considered. Therefore, the memory access latency is incurred only once for each task and is negligible.

We assume that the nodes are interconnected by a low latency, high bandwidth network. The network bandwidth between any two node is denoted as bw_n. In the system, only the processors in the P-nodes and the H-nodes can access the network directly. An F-node can only communicate with other nodes through the DRAM memory of a P-node or an H-node. The bandwidth of such access is denoted as bw_f, which is bounded by $\min\{bw_n, bw_d\}$. In this case, we say the F-node is "*related*" to the P-node or H-node. Without loss of generality, we assume that each F-node is related to only one P-node or H-node.

In our model, an F-node reads data from the DRAM memory of the adjacent processor. Because of spatial parallelism, the design on FPGA can continue to execute while the data is streaming in. However, because the processor moves the data from the DRAM to the FPGA, the computations on the processor cannot continue until the data transfer is complete. Also, the computations on the processor have to be interrupted while the processor receives the data sent to the F-node from the network. Note that if multiple threads run on a processor, the memory transfer and the network communications can also be overlapped with the computations. However, in this case, additional overheads such as the cost of context switch and thread synchronization arise. As we are concerned with providing a simple yet effective model, we assume the processors are single-threaded.

4 Design for Matrix Multiplication

Consider computing $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are $n \times n$ matrices. Each element of the matrices is a floating-point word.

4.1 Workload Allocation

To distribute the workload among the nodes in the system, we partition $\mathbf{B}$ into row stripes of size $k \times n$. The value of k depends on the FPGA-based matrix multiplier, which will be explained in Section 4.2. The tasks in the application are identified as $(n \times n) \times (n \times k)$ block multiplications. Each task contains n^2k floating-point multiplications and n^2k floating-point additions. The tasks contain few data dependencies so that they can be executed on any node. Suppose x_i tasks are assigned to N_i. N_i performs an $(n \times n) \times (n \times kx_i)$ matrix multiplication and is in charge of computing kx_i columns of $\mathbf{C}$. For load balance, we have $\frac{x_0}{C_0} \approx \frac{x_1}{C_1} \approx \cdots \approx \frac{x_{p-1}}{C_{p-1}}$.

We next adjust the workload allocation by overlapping memory transfer and network communications with the computations, as discussed in Section 3. To do so, we partition matrix $\mathbf{A}$ into column stripes of size $n \times k$. Thus, we overlap the multiplication of one stripe of $\mathbf{A}$ and one stripe of $\mathbf{B}$ with the memory transfer and network transfer of the subsequent stripes.

Suppose N_u is an F-node, and N_v is the P-node or H-node related to N_u. During the computations, N_v gets one stripe of $\mathbf{A}$ and one stripe of $\mathbf{B}$. The $\mathbf{A}$ stripe and $x_u \times k \times k$ words of the $\mathbf{B}$ stripe are then sent to N_u. Using these two stripes, N_v and N_u performs $2nk^2x_v$ and $2nk^2x_u$ floating-point operations, respectively. Except for the first stripes of $\mathbf{A}$ and $\mathbf{B}$, the transferring of $2nk$ words over the network and the transferring of $nk+k^2x_u$ words from the DRAM memory to the FPGA are both overlapped with the computations on the FPGA. If N_v is a P-node, to maintain the load balance between N_u and N_v, we have

$$\frac{2nk^2x_v}{C_v} + \frac{2nk}{bw_n} + \frac{nk + k^2x_u}{bw_f} \approx \frac{2nk^2x_u}{C_u} \tag{4.1}$$

If N_v is an H-node, the task allocated to it is partitioned between its processor and its FPGA. A simple partitioning method is to assign n_p rows of $\mathbf{A}$ to the processor and n_f rows to the FPGA. $n_p + n_f = n$ and $\frac{n_p}{n_f} \approx \frac{C_{vp}}{C_{vf}}$, where C_{vp} and C_{vf} are the computing capacities of the processor and the FPGA within N_v, respectively. Within an H-node, the time for transferring data from the DRAM to the FPGA and the time for sending data to other processors are overlapped with the computations on the FPGA. If no F-node is related to N_v, the partitioning is determined by

$$\frac{2n_pk^2x_v}{C_{vp}} + \frac{2nk}{bw_n} + \frac{n_fk + k^2x_v}{bw_d} \approx \frac{2n_fk^2x_v}{C_{vf}} \tag{4.2}$$

When N_u is related to N_v, we have:

$$\frac{2n_pk^2x_v}{C_{vp}} + \frac{2nk}{bw_n} + \frac{n_fk + k^2x_v}{bw_d} + \frac{nk + k^2x_u}{bw_f} \approx \frac{2n_fk^2x_v}{C_{vf}} \approx \frac{2nk^2x_u}{C_u} \tag{4.3}$$

4.2 Proposed Design

In our design, matrices $\mathbf{A}$ and $\mathbf{B}$ are distributed among the nodes initially. In particular, $\frac{n}{p_1+p_3}$ columns of $\mathbf{A}$ and $\frac{n}{p_1+p_3}$ rows of $\mathbf{B}$ are stored in the DRAM memory of N_i, where N_i is either a P-node or H-node. During the computations, matrix $\mathbf{A}$ is read in column-major order, and matrix $\mathbf{B}$ is read in row-major order. The stripes are further partitioned into $k \times k$ submatrices. For each row stripe of $\mathbf{B}$, N_i node stores x_i such submatrices into its DRAM memory. When a submatrix in $\mathbf{A}$ is transferred to a node, it is multiplied with all the stored submatrices of $\mathbf{B}$ whose row indices are the same as its column index. The result matrix $\mathbf{C}$ is transferred back to N_0, and is written back to N_0.

Each FPGA employs our FPGA-based design for matrix multiplication[13]. This design has k PEs. Each PE performs one floating-point multiplication and one floating-point addition during each clock cycle. The BRAM memory on the FPGA device serves as the internal storage of the PEs. Using an internal storage of size $\Theta(k^2)$ words, the design achieves the optimal latency of $\Theta(k^2)$ for $k \times k$ matrix multiplication[13]. This matrix multiplier utilizes the SRAM memory to minimize the amount of data exchanged between the DRAM memory and the FPGA.

As F-node N_u calculates kx_u columns of $\mathbf{C}$, it needs a storage of nkx_u words for the intermediate results of $\mathbf{C}$. In particular, $nkx_u \leq S$, where S is the size of the SRAM memory of the FPGA. On the other hand, the FPGA in H-node N_v is assigned n_{vf} rows

of $\mathbf{A}$ and kx_v columns of $\mathbf{B}$. It needs to store $n_{vf}kx_v$ words of intermediate results of $\mathbf{C}$. Thus, we have $n_{vf}kx_v \leq S$.

When the matrix size is large, block matrix multiplication is performed. Matrices $\mathbf{A}$ and $\mathbf{B}$ are partitioned into blocks of size $b \times b$. The value of b is determined using $bkx_u \leq S$ and $b_{vf}kx_v \leq S$.

5 Experimental Results

To illustrate our ideas, we implemented our design on 6 nodes of XD1. In each node, we used at most one AMD 2.2 GHz processor and one Xilinx Virtex-II Pro XC2VP50. The FPGA-based designs were described in VHDL and were implemented using Xilinx ISE 7.1i[14]. Our own 64-bit floating-point adders and multipliers that comply with IEEE-754 standard were employed[15]. A C program was executed on the processor, and was in charge of file operations, memory transfer and network communications.

5.1 Detailed Model for XD1

Based on the proposed model in Section 3, we developed a detailed model for XD1 and used it for our design. In XD1, $bw_n = 2$ GB/s. As the FPGA-based design gets one word from the DRAM memory in each clock cycle, $bw_d = 1.04$ GB/s. On each node, 8 MB of SRAM memory is allocated to store the intermediate results of $\mathbf{C}$, hence $S = 2^{23}$ bytes. To satisfy the memory requirement in Section 4.2, we set $n = 3072$.

When our FPGA-based matrix multiplier[13] is implemented on one FPGA in XD1, at most 8 PEs can be configured. Each PE performs two floating-point operations in each clock cycle and our design achieves a clock speed of 130 MHz. Thus, the computing capacity of an F-node is $16 \times 130 \times 10^6 = 2.08$ GFLOPS. To obtain the sustained performance of the processor for matrix multiplication, *dgemm* subroutine in AMD Core Math Library (ACML)[16] was executed. Using this subroutine, the computing capacity of a P-node is approximately 3.9 GFLOPS. According to Equation 4.2, an H-node achieves a sustained performance of 5.2 GFLOPS for matrix multiplication.

5.2 Workload Allocation

In our experiments, we chose to use only the FPGAs or the processors in some nodes to achieve heterogeneity.

In Setting 1, all the nodes are H-nodes. As the nodes are homogeneous, the number of tasks assigned to N_i, x_i, equals $\frac{n}{6k}$, $(i = 0, 1, \ldots, 5)$. Using Equation 4.2, we partition the workload among the processor and the FPGA so that $n_{if} = 1280$ and $n_{ip} = 1792$.

In Setting 2, there are three P-nodes and three F-nodes. Each F-node is related to a P-node. The source matrices are sent to the P-nodes first. Each P-node then sends the data to the related F-node. The computations on the P-nodes do not continue until these network communications are complete. The task allocation is determined according to Equation 4.1. If N_i is a P-node, $x_i = 204$; if it is an F-node, $x_i = 180$.

In Setting 3, there are two P-nodes, two F-nodes and two H-nodes. The source matrices are sent to the P-nodes and the H-nodes first. P-nodes then start their computations while each H-node sends the data to the F-node related to it. After that, the processor of each

H-node transfers part of the data to the FPGA within the node. We perform the partitioning according to Equation 4.3. For a P-node, $x_i = 54$; for an F-node, $x_i = 46$. If N_i is an H-node, $x_i = 92$, $n_{if} = 1440$, and $n_{ip} = 1632$.

5.3 Performance Analysis

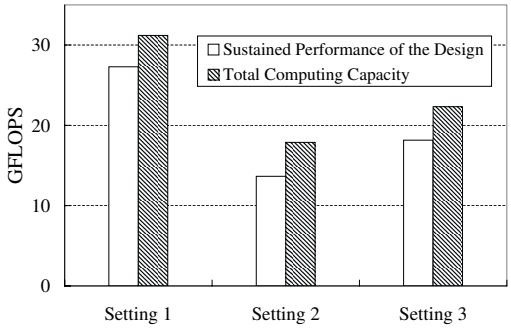


Figure 2. Performance of the proposed design under various heterogeneity

Figure 2 shows the floating-point performance achieved by our design for the three settings defined above. It also shows the sum of the computing capacities of the nodes in each setting, which is denoted as C_{sum}. $C_{sum} = \sum_{i=0}^{5} C_i$. In Setting 1, our design overlaps more than 90% of the memory transfer time and more than 80% of the network communication time with the computations. It achieves more than 85% of C_{sum}. In Setting 3, because of the related F-nodes, the processors in the H-nodes spend more time on network communications. Thus, the design achieves 80% of C_{sum}. In Setting 2, the P-nodes cannot perform computations when they are receiving data and sending data to the F-nodes. Thus, only 70% of the network communication time is overlapped with the computations. In this case, our design achieves about 75% of C_{sum}.

Fig. 2 shows that our design can adapt to various heterogeneous settings. When the total computing capacity in the system increases, the sustained performance of our design also increases. In XD1, our design achieves up to 27 GFLOPS for double-precision floating-point matrix multiplication.

We also used the proposed model for performance prediction. To do so, we use the same system parameters and the same workload allocation as in the experiments. However, we assume all the communication costs and memory transfer time are overlapped with the computations on the FPGAs. As our design efficiently hides the memory transfer and network communication costs, it achieves more than 90% of the predicted performance in all three settings. The figure is not shown here due to page limitation.

6 Conclusion

In this paper, we proposed a model for reconfigurable computing systems with heterogeneous nodes. The model describes a system using various parameters, including the computing capacity of the nodes, the size of the memory available to the nodes, the available memory bandwidth, and the network bandwidth among the nodes. Matrix multiplication

was used as an example and was implemented on Cray XD1. Experimental results show that our design for matrix multiplication can adapt to various heterogeneous settings, and its performance increases with the total computing capacity in the system. The proposed model can also provide a fairly accurate prediction for our design. In the future, we plan to develop detailed models for more reconfigurable computing systems, such as Cray XT3 and SGI Altix350. We also plan to use the model for more complex applications.

Acknowledgements

This work is supported by the United States National Science Foundation under grant No. CCR-0311823, in part by No. ACI-0305763 and CNS-0613376. NSF equipment grant CNS-0454407 and equipment grant from Xilinx Inc. are gratefully acknowledged.

References

1. SRC Computers, Inc. http://www.srccomp.com/.
2. Cray Inc. http://www.cray.com/.
3. Silicon Graphics, Inc. http://www.sgi.com/.
4. R. Scrofano, M. Gokhale, F. Trouw, and V. K. Prasanna, *A Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers*, in: Proc. FCCM 2006, (2006).
5. L. Zhuo and V. K. Prasanna, *Scalable Hybrid Designs for Linear Algebra on Reconfigurable Computing Systems*, in: Proc. of ICPADS 2006, (2006).
6. Cray Inc., Cray XD1™. http://www.cray.com/products/xd1/.
7. B. Hamidzadeh, D. J. Lilja, and Y. Atif, *Dynamic Scheduling Techniques for Heterogeneous Computing Systems*, Concurrency: Practice & Experience, **7**, (1995).
8. H. Siegel and S. Ali, *Techniques for Mapping Tasks to Machines in Heterogeneous Computing Systems*, Journal of Systems Architecture, **46**, (2000).
9. H. Oh and S. Ha, *A Hardware-Software Cosynthesis Technique Based on Heterogeneous Multiprocessor Scheduling*, in: Proc. CODES'99, (1999).
10. B. P. Dave, G. Lakshminarayana and N. K. Jha, *COSYN: Hardware-Software Co-Synthesis of Heterogeneous Embedded Systems*, IEEE TVLSI, **7**, (1999).
11. L. Zhuo and V. K. Prasanna, *Hardware/Software Co-Design on Reconfigurable Computing Systems*, in: Proc. IPDPS 2007, (2007).
12. P. Saha and T. El-Ghazawi, *Applications of Heterogeneous Computing in Hardware/-Software Co-Scheduling*, in: Proc. AICCSA-07, (2007).
13. L. Zhuo and V. K. Prasanna, *Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs*, in: Proc. IPDPS 2004, (2004).
14. Xilinx Incorporated. http://www.xilinx.com.
15. G. Govindu, R. Scrofano and V. K. Prasanna, *A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing*, in: Proc. of ERSA 2005, (2005).
16. AMD Core Math Library. http://developer.amd.com/acml.aspx.

Mini-Symposium

“The Future of OpenMP in the Multi-Core Era”

The Future of OpenMP in the Multi-Core Era

Barbara Chapman[1] **and Dieter an Mey**[2]

[1] Computer Science Department
University of Houston
501 Phillip G. Hoffman
4800 Calhoun, Houston, TX 77204-3010
E-mail: chapman@cs.uh.edu

[2] RWTH Aachen University
Center for Computing and Communication
Seffenter Weg 23
52074 Aachen, Germany
E-mail: anmey@rz.rwth-aachen.de

OpenMP is an Application Programming Interface (API) which is widely accepted as a standard for high-level shared-memory parallel programming. It is a portable, scalable programming model that provides a simple and flexible interface for developing shared-memory parallel applications in Fortran, C, and C++. By November 2007 the upcoming Version 3.0 of the OpenMP specification will be in its public reviewing phase. It will include a novel tasking concept and improved support for nested parallelism. Features to make OpenMP ccNUMA-aware are under intense discussion. Another important approach to broaden the scope of OpenMP is Intel's Cluster OpenMP implementation.

Mini-Symposium Contributions

In the first session, Marc Bull of EPCC reported on OpenMP Version 3.0 which is in the final stages of preparation. He described the new features which are likely to be included in the new specification, including the tasking constructs, item loop collapsing, stack size control, thread wait policy control, improved C++ compatibility, additional loop scheduling features, revisions to the memory model, and additional routines to support nested parallelism. He discussed the rationale for these new features, and presented some examples of how they can be used.

In his presentation "OpenMP for Clusters", Larry Meadows of Intel introduced Intel's Cluster OpenMP product. Using DVSM technology (descended from the Treadmarks software developed at Rice University), Cluster OpenMP provides the user with a full implementation of OpenMP that runs on any cluster of 64 bit Intel Architecture nodes (Intel64 or Itanium). Cluster OpenMP is included with the Intel compiler suite for C, C++, and Fortran. In this talk he introduced Cluster OpenMP, gave a short review of the implementation and some technical details, discussed the kinds of applications that are appropriate for Cluster OpenMP, gave an overview of the tools that Intel provides for porting and tuning Cluster OpenMP applications, and showed performance evaluations for several Cluster OpenMP applications.

In his talk "Getting OpenMP Up to Speed", Ruud van der Pas of SUN Microsystems covered the basic rules how to get good performance out of an OpenMP program. OpenMP provides for a very powerful and flexible programming model, but unfortunately there is a

persistent misconception that it does not perform well. Surely one may have performance issues with an OpenMP program, but that does not mean these can not be addressed. The talk included a detailed coverage of false sharing, a potentially nasty inhibitor of scalable performance and several case studies, illustrating several of the points made.

In the next presentation, Barbara Chapman of the University of Houston reported on PerfOMP, a runtime performance monitoring API for OpenMP. The OpenMP Architecture Review Board (ARB) recently sanctioned an interface specification for profiling/tracing tools that defines a protocol for two-way communications and control between the OpenMP runtime library and performance tools. To evaluate this approach to performance measurement, the PerfOMP interface has been designed to operate as an intermediary software layer to support unidirectional communications from the OpenMP runtime library to performance tools. PerfOMP can support the implementation of the OpenMP ARB sanctioned profiling interface by providing the underlying infrastructure for tracking OpenMP events/states inside the OpenMP runtime and satisfying specific queries made by the performance tool to the OpenMP runtime. Alternatively, PerfOMP can be implemented to directly profile or trace the performance of an OpenMP application through the OpenMP runtime library. PerfOMP has been integrated into an existing open source compilation and performance tool environment and successfully tested with benchmark kernels and a production quality parallel computational fluid dynamics application. More details can be found in the following paper.

The last talk "Affinity Matters! OpenMP on Multicore and ccNUMA Architectures" was given by Dieter an Mey of RWTH Aachen University. So far OpenMP was predominantly employed on large shared memory machines. With the growing number of cores on all kinds of processor chips and with additional OpenMP implementations e.g. the GNU and Visual Studio compilers, OpenMP is available for use by a rapidly growing, broad community. Upcoming multicore architectures make the playground for OpenMP programs even more diverse. The memory hierarchy will grow, with more caches on the processor chips. Whereas applying OpenMP to Fortran and C programs on machines with a flat memory (UMA architecture) is straight forward in many cases, there are quite some pitfalls when using OpenMP for the parallelization of C++ codes on one hand and on ccNUMA architectures on the other hand. The increasing diversity of multicore processor architectures further introduces more aspects to be considered for obtaining good scalability. Approaches to improve the support for memory and thread affinity within the upcoming OpenMP specification are still under discussion. So far operating system and compiler dependent calls to pin threads to processor cores and to control page allocation have to be employed to improve the scalability of OpenMP applications on ccNUMA and multicore architectures.

Towards an Implementation of the OpenMP Collector API

Van Bui[1], **Oscar Hernandez**[1], **Barbara Chapman**[1],
Rick Kufrin[2], **Danesh Tafti**[3], **and Pradeep Gopalkrishnan**[3]

[1] Department of Computer Science
University of Houston
Houston, TX 77089
E-mail: {vtbui, oscar, chapman}@cs.uh.edu

[2] National Center for Supercomputing Applications
University of Illinois
Urbana, IL, 61801
E-mail: rkufrin@ncsa.uiuc.edu

[3] Department of Mechanical Engineering,
Virginia Tech
Blacksburg, VA, 24061
E-mail: {dtafti, pradeepg}@vt.edu

Parallel programming languages/libraries including OpenMP, MPI, and UPC are either in the process of defining or have already established standard performance profiling interfaces. The OpenMP Architecture Review Board (ARB) recently sanctioned an interface specification for profiling/tracing tools that defines a protocol for two-way communications and control between the OpenMP runtime library and performance tools, known as the collector API. Reference implementations of the collector are sparse and are primarily *closed-source*. We provide a description of our efforts towards a full implementation of an *open-source* performance monitoring tool for OpenMP based on the collector API. This effort[a] evaluates the collector's approach to performance measurement, assesses what is necessary to implement a performance tool based on the collector interface, and also provides information useful to performance tool developers interested in interfacing with the collector for performance measurements.

1 Introduction

Many scientific computing applications are written using parallel programming languages/libraries. Examples include applications in mechanical engineering, neuroscience, biology, and other domains[1,2,3]. Software tools, such as integrated development environments (IDEs), performance tools, and debugging tools are needed that can support and facilitate the development and tuning of these applications. Parallel programming libraries/languages such as OpenMP[4], MPI[5], and UPC[6] are in the process of drafting or have already standardized/sanctioned a performance monitoring interface[7,8,9,10]. Performance tools must be able to present performance information in a form that captures the user model of the parallel language/library and a performance monitoring interface enables this. The task of relating performance data in terms of the user model can be challenging for some languages. In the case of OpenMP, the compiler translates OpenMP directives inserted into the source code into more explicitly multi-threaded code using the OpenMP runtime library. Measuring the performance of an OpenMP application is complicated

[a] This work is supported by the National Science Foundation under grant No. 0444468, 0444319, and 0444407.

by this translation process, resulting in performance data collected in terms of the implementation model of OpenMP. Performance tool interfaces have emerged for OpenMP to provide a *portable* solution to implementing tools to support presentation of performance data in the user model of the language[10,9].

The performance monitoring API for OpenMP, known as the collector interface, is an event based interface requiring *bi-directional* communications between the OpenMP runtime library and performance tools[11]. The OpenMP Architecture Review Board (ARB) recently sanctioned the collector interface specifications. Reference implementations to support development of the collector are sparse. The only known *closed-source* prototype implementation of the collector is provided by the Sun Studio Performance Tools[10]. To the best of our knowledge, there are currently no *open-source* implementations of the collector API. Our goal is to provide such a reference implementation for the research community. We have designed and implemented an API that leverages the OpenMP runtime library to support performance monitoring of OpenMP applications. Our performance monitoring system consists of an OpenMP runtime library, a low-level performance collector that captures hardware performance counters, and a compiler to support mapping performance back to the source level.

In the remainder sections of this paper, we provide a more implementation focused discussion of the OpenMP collector API, detail our own experiences in implementing and evaluating a performance monitoring system that leverages the OpenMP runtime library, present a performance case study to demonstrate the usefulness of our performance monitoring system, and finally end with our conclusions from this study.

2 The Collector Runtime API for OpenMP

The collector interface specifies a protocol for communication between the OpenMP runtime library and performance tools. A prior proposal for a performance tool interface for OpenMP, known as POMP[9], was not approved by the committee. POMP is an interface that enables performance tools to detect OpenMP events. POMP supports measurements of OpenMP constructs and OpenMP API calls. The advantages of POMP includes the following: the interface does not constrain the implementation of OpenMP compilers or runtime systems, it is compatible with other performance monitoring interfaces such as PMPI, and it permits multiple instrumentation methods (e.g. source, compiler, or runtime). Unfortunately, if the compiler is unaware of these instrumentation routines, they can interfere with static compiler analysis and optimizations, which affects the accuracy of gathered performance data.

In contrast, the collector interface is independent of the compiler since it resides and is implemented inside the OpenMP runtime. The primary advantage of the collector interface is that the application source code remains unmodified since instrumentation is not required at the source or compiler level. Consequently, data collection will interfere much less with compiler analysis/optimizations and a more accurate picture of the performance of the application is possible. The design also allows for the collector and OpenMP runtime to evolve independently. However, overheads must be carefully controlled from both the side of the OpenMP runtime and from collecting the measured data in order to obtain the most accurate results. A second drawback is that the interface provides little support for distinguishing between different loops inside a parallel region and corresponding barrier

regions. Given these drawbacks, much additional software support and efficient algorithms are required to properly address these concerns.

2.1 Rendezvous Between OpenMP Runtime and Collector

The OpenMP runtime and collector communicate via the collector interface. The collector initiates communications and also makes queries to the OpenMP runtime via the collector interface. The collector interface consists of a single routine that takes the following form: *int __omp_collector_api (void *arg)*. The **arg** parameter is a pointer to a byte array containing one or more requests. For example, the collector can request that the OpenMP runtime notifies it when a specific event occurs. The input parameters sent in this case would include the name of the event and the callback routine to be invoked from inside the OpenMP runtime when this specific event occurs. Figure 1 depicts this request more precisely along with other requests that can be made by the collector. The collector may make requests for the current state of the OpenMP runtime, the current/parent parallel region ID (PRID), and to pause/resume generation of events.

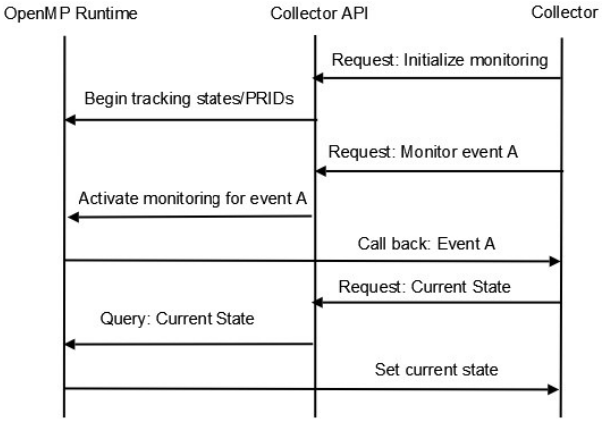


Figure 1. Example of sequence of requests made by collector to OpenMP runtime.

2.2 Performance Tool Support

Performance tools may use the collector interface to support performance measurements of OpenMP codes. The performance tool typically queries the OpenMP runtime for information (e.g. event notifications, thread state, etc), records performance information, and captures the runtime call stack. The call stack that is captured at runtime is based on the implementation model of OpenMP since the translated code includes calls to the OpenMP runtime library. Performance data should be presented in the user model of OpenMP since the OpenMP specification is written in terms of the user model. Using the support provided by the OpenMP runtime and realizing the support required on the tools side, the user

model call stack can be reconstructed from the implementation model call stack for each thread.

There are certain expectations that must be met by supporting system and auxiliary user-level software to allow the approach we describe to provide the most accurate performance feedback. At the lowest level, the operating system should maintain the performance data of interest in a manner such that the data is virtualized on a per-thread basis (that is, it must be possible to extract performance measurements isolated to a single thread, not across an entire process, group, or the entire system). The optimal situation is one in which the underlying thread model is "one-to-one", where every user level thread is bound to a single kernel thread. In this case, the kernel can maintain the required virtualized performance data as part of the overall thread context and return this data as-is in response to a query from a performance tool. Similarly, it must be possible to retrieve at runtime the call stack associated with a single thread in an efficient manner.

2.3 Support Inside the OpenMP Runtime

The OpenMP runtime is primarily responsible for implementation of the collector interface. Performance tools may make several requests to the OpenMP runtime library via the collector interface. To satisfy these requests, the OpenMP runtime library needs to implement support to (1) initiate/pause/resume/stop event generation, (2) respond to queries for the ID of the current/parent parallel region, and (3) respond to queries for the current state of the calling thread.

The collector initiates interactions with the OpenMP runtime by making a request to the OpenMP runtime to create and initiate the data structures required to store the state of each thread and also to track the current/parent parallel region ID. Updates and accesses to these data structures by each of the threads must be properly managed to minimize overhead.

When the collector makes a request for notification of a specified event (s), the OpenMP runtime will activate monitoring for this event inside its environment. The collector may also make requests to pause, resume, or stop event generation. To make a request for notification of a specific event, the collector passes along the name of the event to monitor as well as a callback routine to be invoked by the OpenMP runtime to notify the collector each time the event occurs. Examples of events include fork, join, entry/exit into a barrier, entry/exit into a master region, entry/exit into an idle state, and several others. The collector interface specification requires that the OpenMP runtime provide support for generating the fork event and specifies support for the other events as optional to support tracing. Some issues to consider regarding implementing this functionality includes the overhead associated with activating/inactivating event generation and overheads from invoking the call back routine. These overheads should be minimized in order to obtain the most accurate performance measurements.

3 Building a Performance Tool Inside the OpenMP Runtime

The process of building a performance tool based on the collector interface consists of a series of steps. We first select the software components that will form the infrastructure for our tool. Second, we assess whether any changes or extensions are needed in order for the software components to communicate with one another to perform the specified tasks of

the end tool. Third, we build a prototype tool to assess the overheads associated with data collection and to also evaluate the accuracy of the generated data for performance tuning. Finally, we move forward to build the end tool.

3.1 Software Components

The set of software components needed to implement a performance profiling tool based on the collector API includes a compiler's OpenMP runtime library, a performance monitoring interface for OpenMP, and a performance collection tool. We use the OpenUH[12] compiler's OpenMP runtime library and PerfSuite's[13] measurement libraries as the collector for our profiling tool. Extensions are needed for both the OpenMP runtime library and the collector in order to realize our final prototype tool. The software components that we employ in this development effort are open-source and freely available to the research community.

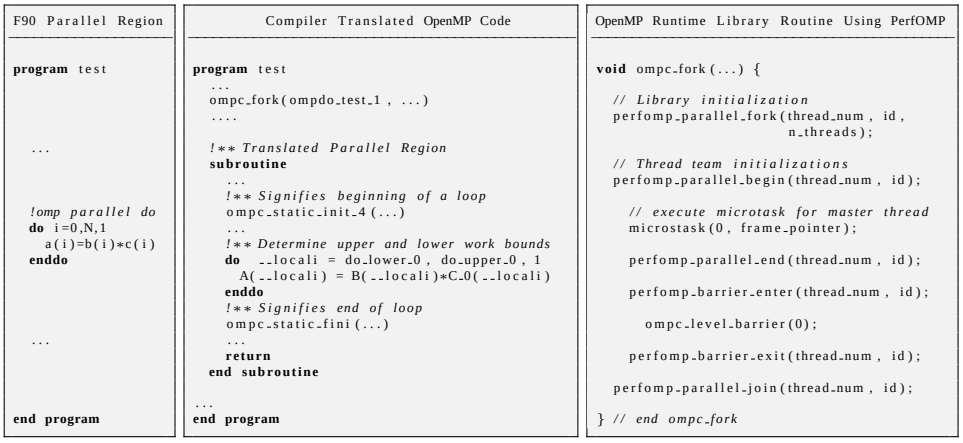
To support building a prototype performance tool based on the collector interface, we designed a performance monitoring interface for OpenMP, called PerfOMP[14]. PerfOMP was designed to monitor the runtime execution behavior of an OpenMP application with the help of the OpenMP runtime library. The current PerfOMP event based interface includes support for monitoring fork, join, loop, and barrier regions. Each interface routine takes only a few parameters including the thread number, thread count, and parallel region ID. PerfOMP maps OpenMP events with identifiers such as the thread number and parallel region number. PerfOMP supports the development of performance profiling and tracing tools for OpenMP applications in a manner that is transparent to the user.

3.2 Instrumentation

The compiler translates OpenMP directives with the help of its OpenMP runtime library. An example of this transformation process in the OpenUH compiler is shown in Fig. 2a and Fig. 2b. Figure 2c shows an example of PerfOMP instrumentation points inside the OpenMP runtime library of OpenUH. Figure 2c shows instrumentation points inside the *ompc_fork* operation that captures the following events: fork, join, parallel begin/end for the master thread, and the implicit barrier entry/exit for the master thread. To capture events for the slave threads requires instrumentation of a separate routine in the runtime library. The entire instrumentation of the OpenMP runtime library of the OpenUH compiler required minimal programming effort. This instrumentation enables collecting measurements of several events including the overall runtime of the OpenMP library, of individual parallel regions and several events inside a parallel region for each thread.

3.3 Mapping Back to Source

Mapping the performance data back to the source code was accomplished with the help of the OpenUH compiler. Modifications were made in the OpenMP translation by the OpenUH compiler so that it dumps to an XML file source level information. Each parallel region is identified by a unique ID. Unique IDs can also be generated for other parallel constructs. This required changes in the OpenMP compiler translation phase to add an additional integer parameter variable to the fork operation to hold and pass along the parallel region ID to the OpenMP runtime. A map file containing the source level mappings is generated for each program file and is created statically at compile time.



(a) (b) (c)

Figure 2. (a) OpenMP parallel region in Fortran. (b) Compiler translation of the piece of OpenMP code from part (a). (c) A PerfOMP instrumented OpenMP runtime library routine.

As an alternative approach to support mapping back to source, we have extended PerfSuite's library routines to also return the runtime call stack. The advantage of obtaining the source level mappings via the compiler is that we can get a direct mapping between the performance data and the compiler's representation of the program. This will enable the compiler to more easily utilize the generated data to improve its optimizations. Although this mapping is not context sensitive in that it does not distinguish between different calling contexts of a parallel region. Gathering source level mappings at runtime using the call stack will allow us to distinguish the different calling contexts and this additional capability motivates our use of runtime stack information gathered through PerfSuite. A performance tool should be able to reconstruct the user model call stack using the retrieved implementation model call stack. The extensions being made to PerfSuite will provide a parameter variable allowing the caller to specify the number of stack frames to skip or ignore to facilitate this reconstruction. To control the overheads associated with retrieving large stack frames, PerfSuite will also provide a parameter for specifying the maximum number of instruction pointers to retrieve. These extensions to PerfSuite will be used in our implementation of a performance tool based on the collector interface.

4 Experimental Results

To demonstrate the usefulness of the data gathered from our prototype measurement system, we present a case study where we analyze the performance of GenIDLEST[1]. GenIDLEST uses a multiblock structured-unstructured grid topology to solve the time-dependent Navier-Stokes and energy equations. We use its 45rib input data set. All the experiments were performed on an SGI Altix 3700 distributed shared memory system and using eight threads. The OpenMP version of GenIDLEST is compiled with the OpenUH compiler with optimization level 3 and OpenMP enabled.

4.1 Case Study: Bottleneck Analysis of GenIDLEST

Several different methodologies exist for utilizing performance hardware counters for performance analysis. We apply the methodology for detecting bottlenecks using the Itanium 2 performance counters as described by Jarp[15]. This technique applies a drill down approach, where the user starts counting the most general events and drills down to more fine grained events. Applying this measurement technique, GenIDLEST shows an estimated 8% of additional overheads when we activate performance measurements with PerfOMP for all OpenMP parallel regions in the code.

First, we measure the percentage of stall cycles and find that stall cycles account for ∼70% of the total cycles. The data also shows that the parallel region inside the **diff_coeff** subroutine is taking up about 20% of the execution time and spending about 3.34% of time waiting inside the barrier at the end of a parallel region. We collect additional performance counters for the **diff_coeff** subroutine to identify and resolve the bottleneck for that region of code.

The performance data for the parallel region inside **diff_coeff** shows ∼35% data cache stalls, ∼23% instruction miss stalls, and ∼28% of the stalls are occurring in the floating point unit (FLP). Relating the memory counters to the stall cycles, we found that ∼60% of the memory stalls result from L2 hits and ∼38% from L3 misses.

We apply transformations to the parallel region inside **diff_coeff** to better utilize the memory hierarchy. We apply modifications of variable scope from shared to private to this subroutine. Privatizing the data should relieve the bottlenecks associated with remote memory accesses since privatization ensures that each thread will have its own private local copy of the data.

After privatization of the data in the **diff_coeff** subroutine, we observe a ∼25% improvement in overall runtime of GenIDLEST. The parallel region inside the **diff_coeff** subroutine now only takes up ∼5.5% of the execution time compared to the previous estimated 20%. Furthermore, the wait time spent inside the barrier region also significantly improved by a factor of 10X. Performance counting data from the parallel region inside **diff_coeff** also shows a significant drop in stall cycles (∼80-90% decrease).

5 Conclusions and Future Work

An implementation-focused discussion of the OpenMP ARB sanctioned collector interface is presented to support tool developers interested in implementing support for the collector API. We describe the necessary software components and extensions for implementing a performance tool based on the collector API. A prototype performance profiling tool that emulates the functionality of a tool based on the collector API is also presented and shown to generate useful and accurate performance data for performance tuning purposes.

Building on top of the current software infrastructure we have laid out, plans are currently underway to provide a full implementation of an open-source performance tool based on the collector interface. The remaining tasks include finalizing the extensions being made to PerfSuite that enable runtime call stack access, implementing the collector interface inside the OpenMP runtime library of OpenUH, and designing/implementing efficient algorithms for mitigating overheads that can incur both inside the OpenMP runtime library and from the collector. Our goal is to encourage more compiler developers to im-

plement this support inside their respective OpenMP runtime libraries, thereby enabling more performance tool support for OpenMP users.

References

1. D. K. Tafti, *GenIDLEST - A Scalable Parallel Computational Tool for Simulating Complex Turbulent Flows*, in: Proc. ASME Fluids Engineering Division, (2001).
2. H. Markram, *Biology—The blue brain project*, in: SC '06: Proc. 2006 ACM/IEEE conference on Supercomputing, p. 53, (ACM Press, NY, 2006).
3. C. Chen and B. Schmidt, *An adaptive grid implementation of DNA sequence alignment*, Future Gener. Comput. Syst., **21**, 988–1003, (2005).
4. L. Dagum and R. Menon, *OpenMP: an industry-standard API for shared-memory programming*, IEEE Comput. Sci. Eng., **5**, 46–55, (1998).
5. Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Tech. Rep. UT-CS-94-230, (1994).
6. T. El-Ghazawi, W. Carlson, T. Sterling and K. Yelick, *UPC: Distributed Shared Memory Programming*, (John Wiley & Sons, 2005).
7. M. Snir, S. Otto, S. Huss-Lederman, D. Walker and J. Dongarra, *MPI: The Complete Reference*, (MIT Press, Cambridge, Mass., 1996).
8. H. Su, D. Bonachea, A. Leko, H. Sherburne, M. Billingsley III and A. George, *GASP! A standardized performance analysis tool interface for global address space programming models*, Tech. Rep. LBNL-61659, Lawrence Berkeley National Lab, (2006).
9. B. Mohr, A. Malony, H. C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger and S. Shah, *A performance monitoring interface for OpenMP*, in: Proc. 4th European Workshop on OpenMP, (2002).
10. G. Jost, O. Mazurov and D. an Mey, *Adding new dimensions to performance analysis through user-defined objects*, in: IWOMP, (2006).
11. M. Itzkowitz, O. Mazurov, N. Copty and Y. Lin, *White paper: an OpenMP runtime API for profiling*, Tech. Rep., Sun Microsystems, (2007).
12. C. Liao, O. Hernandez, B. Chapman, W. Chen and W. Zheng, *OpenUH: an optimizing, portable OpenMP compiler*, in: 12th Workshop on Compilers for Parallel Computers, (2006).
13. R. Kufrin, *PerfSuite: an accessible, open source performance analysis environment for Linux*, in: 6th International Conference on Linux Clusters: The HPC Revolution 2005, (2005).
14. V. Bui, *Perfomp: A runtime performance/event monitoring interface for OpenMP*, Master's thesis, University of Houston, Houston, TX, USA, (2007).
15. S. Jarp, *A Methodology for using the Itanium-2 Performance Counters for Bottleneck Analysis*, Tech. Rep., HP Labs, (2002).

Mini-Symposium

“Scaling Science Applications on Blue Gene”

Scaling Science Applications on Blue Gene

William D. Gropp[1], Wolfgang Frings[2], Marc-André Hermanns[2], Ed Jedlicka[1], Kirk E. Jordan[3], Fred Mintzer[3], and Boris Orth[2]

[1] Argonne National Laboratory
Mathematics and Computer Science Division
9700 South Cass Avenue, Argonne, IL 60439, USA
E-mail: {gropp, jedlicka}@mcs.anl.gov

[2] Forschungszentrum Jülich,
John von Neumann Institute for Computing, 52425 Jülich, Germany
E-mail: {w.frings, m.a.hermanns, b.orth}@fz-juelich.de

[3] Deep Computing, IBM Systems and Technology Group
1 Rogers Street, Cambridge, MA 02142, USA
E-mail: {kjordan, mintzer}@us.ibm.com

Massively parallel supercomputers like IBM's Blue Gene/L offer exciting new opportunities for scientific discovery by enabling numerical simulations on an unprecedented scale. However, achieving highly scalable performance is often not straightforward as the system's extraordinary level of parallelism and its specialized nodes present challenges to applications in many areas, including: communication efficiency, memory usage, and I/O.

This mini-symposium aimed to highlight some of the remarkable scaling and performance results achieved, and bring together users and system experts to discuss possible solutions to yet unresolved issues. It featured speakers whose applications have run at large scale on the 8K node system at John von Neumann Institute for Computing at Forschungszentrum Jülich and the 20K node system at IBM's Watson Research Center. The talks provided a sampling of the different applications and algorithms that have successfully run on Blue Gene. The speakers discussed best practices, particularly with respect to scaling to tens of thousands of processes, and challenges faced in using BlueGene/L to massive scale, and they showcased some of the breakthrough science that has already been achieved.

Mini-Symposium Contributions

Jörg Schumacher of Ilmenau Technical University reported on his work on turbulence in laterally extended systems. He has studied three-dimensional turbulence in a very flat Cartesian cell using periodic boundaries in the lateral directions and free-slip boundaries in the vertical one. The pseudospectral simulations were conducted on Blue Gene/L, the memory limitations of which required a two-dimensional parallelization of the numerical scheme. Schumacher reported on performance and scaling tests on up to eight racks and presented first physical results for the case of convective turbulence.

Kevin Stratford from Edinburgh Parallel Computing Centre presented an implementation of Lees-Edwards sliding periodic boundary conditions used to perform simulations of sheared binary fluids. In conjunction with domain decomposition and message passing he was able to perform shear simulations of significant size and duration. Stratford discussed the scaling and performance of these large calculations on the Blue Gene/L architecture.

Ilian T. Todorov talked about DL_POLY_3, a new generation software package for molecular dynamics simulations developed at Daresbury Laboratory. After an overview of

the package he reported on the weak scaling behaviour of DL_POLY_3 and discussed its dependence on force field complexity. Todorov showed benchmark results on massively parallel architectures like BlueGene/L and Cray XT3, and concluded with a review of the challenges and successes in applying DL_POLY_3 in two different simulation studies: (i) radiation damage cascades in minerals and oxides, where the problem size (length scale) is of importance, and (ii) biochemical simulations, where long time scale simulations are required.

Erik Koch from the Institute for Solid State Research at Forschungszentrum Jülich reported on simulations of materials with strong correlations. In these simulations the Lanczos method was used to calculate ground state and dynamical properties of model Hamiltonians approximating the Hamiltonian of the full, non-perturbative many-body problem. Koch presented an efficient implementation for Blue Gene that overcomes the method's principal problem of non-local memory access and scales very well. Besides correlated electronic systems it may also be applied to quantum spin systems, which are relevant for quantum computing.

Jeffrey Fox of Gene Network Sciences reported on massively parallel simulations of electrical wave propagation in cardiac tissue. The software of his group is being used in an on-going study to connect drug-induced modifications of molecular properties of heart cells to changes in tissue properties that might lead to a rhythm disorder. The performance of the code was studied by simulating wave propagation in an anatomically realistic model of the left and right ventricles of a rabbit heart on Blue Gene partitions of up to 4,096 processors. Judging from these results, the Blue Gene architecture seems particularly suited for cardiac simulation, and offers a promising platform for rapidly exploring cardiac electrical wave dynamics in large spatial domains.

Hinnerk Stüben from Zuse Institute Berlin reported on how he achieved, for the first time, a sustained performance of more than 1 TFlop/s in everyday simulations of Lattice QCD on the IBM BlueGene/L system at Jülich and the SGI Altix 4700 at LRZ, Garching. He compared the performance of Fortran/MPI code with assembler code, the latter of which allows for a better overlapping of communication and computation, and prefetching of data from main memory. Lower-level programming techniques resulted in a speed-up of 1.3-2.0 compared to the Fortran code. His program scales up to the full eight Blue Gene/L racks at Jülich, at a remarkable maximum performance of 8.05 Tflop/s measured on the whole machine.

Henry M. Tufo from the National Center for Atmospheric and Climate Research (NCAR) presented HOMME, the High-Order Method Modeling Environment, developed by NCAR in collaboration with the University of Colorado. HOMME is capable of solving the shallow water equations and the dry/moist primitive equations and has been shown to scale to 32,768 processors on BlueGene/L. Tufo reported that the ultimate goal for HOMME was to provide the atmospheric science community a framework upon which to build a new generation of atmospheric general circulation models for CCSM based on high-order numerical methods that efficiently scale to hundreds-of-thousands of processors, achieve scientifically useful integration rates, provide monotonic and mass conserving transport of multiple species, and can easily couple to community physics packages.

Last but not least, Kirk E. Jordan from IBM presented interesting early experiences with Blue Gene/P, the next generation of the IBM Blue Gene system.

Turbulence in Laterally Extended Systems

Jörg Schumacher[1] **and Matthias Pütz**[2]

[1] Department of Mechanical Engineering
Technische Universität Ilmenau
98684 Ilmenau, Germany
E-mail: joerg.schumacher@tu-ilmenau.de

[2] Deep Computing – Strategic Growth Business
IBM Deutschland GmbH
55131 Mainz, Germany
E-mail: mpuetz@de.ibm.com

We study three-dimensional turbulence in a very flat Cartesian cell with periodic boundaries in the lateral directions and free-slip boundaries in the vertical one. The pseudospectral simulations are conducted on the massively parallel Blue Gene/L system. The small amount of memory per core requires a two-dimensional parallelization of the numerical scheme. We report performance and scaling tests on up to 16384 CPUs and present first physical results for the case of convective turbulence.

1 Introduction

Turbulent flows appear frequently for lateral extensions that exceed the vertical ones by orders of magnitude. Examples can be found in planetary and stellar physics or in atmospheric and oceanographic science[1]. For example, atmospheric mesoscale layers or the Gulf stream evolve on characteristic lengths L of 500 - 5000 km in combination with vertical scales H of about 1 - 5 km. The resulting aspect ratios follow to $\Gamma = L/H = 100 - 1000$[1].

This is one main reason why many turbulence studies in geophysical context are conducted in two dimensions from beginning, i.e., the vertical variations of the fields under consideration are neglected. The description of the physical processes is done then only with respect to the remaining lateral coordinates (for a review see Ref. 2). However, the reduction of the space dimension from three to two alters the turbulence physics in a fundamental way. In three dimensions, kinetic energy is cascading from the largest eddies down to smaller ones. This regime is known as the *direct* cascade. In two dimensions, energy which is injected at small scales flows in an *inverse* cascade of the kinetic energy towards larger eddies. Such large-scale vortex structures are observed, e.g. as big storm systems or as the Red Spot on planet Jupiter. The reason for this cascade reversal is a topological one. The fundamental mechanism of vortex stretching which generates energy at finer scales is absent in two dimensions[2].

Theoretical studies on the crossover from three- to (quasi-)two-dimensional turbulence are rare. They are based on cascade models which are derived from the Fourier space formulation of the Navier-Stokes equations of motion. The space dimension d is included as a varying parameter. Fournier and Frisch[3] found a reversal of the turbulent energy cascade from direct to inverse when the space dimension becomes smaller than $d = 2.06$. Such crossover studies are not possible in a direct numerical simulation which uses a spatial

grid. However, one can extend a rectangular box in the two lateral directions successively while keeping the spectral resolution with respect to all three space coordinates the same. Theoretically, this allows for a horizontal rearrangement of flow structures on scales significantly larger than the vertical height of the cell. Turbulence can thus become quasi-two-dimensional for $\Gamma \gg 1$. The flat geometry will differently affect the turbulent transport of substances in the lateral directions compared to the vertical one. Frequently, such layers are in a state of convective turbulence that is caused by the temperature dependence of the fluid density. Furthermore, planets and stars rotate at constant angular frequencies, i.e. in addition to buoyancy effects Coriolis forces will appear[5]. This is exactly the system which we want to study. In the following, we describe preparatory steps for the implementation and first tests of the simulation program on the Blue Gene/L system.

2 Equations and Numerical Integration Scheme

The Boussinesq equations, i.e. the Navier-Stokes equations for an incompressible flow with an additional buoyancy term $\alpha g\theta \mathbf{e}_z$ and the advection-diffusion equation for the temperature field, are solved numerically for the three-dimensional case[6]. In addition, the Coriolis force term $2\Omega\mathbf{e}_z\times\mathbf{u}$ is added for cases with rotation about the vertical z-axis. The equations are then

$$\nabla \cdot \mathbf{u} = 0\,, \tag{2.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\mathbf{u} = -\nabla p - 2\Omega\mathbf{e}_z\times\mathbf{u} + \nu\nabla^2\mathbf{u} + \alpha g\theta\mathbf{e}_z\,, \tag{2.2}$$

$$\frac{\partial \theta}{\partial t} + (\mathbf{u}\cdot\nabla)\theta = \kappa\nabla^2\theta + u_z\frac{\Delta T}{H}\,. \tag{2.3}$$

Here, $\mathbf{u}$ is the turbulent velocity field, p the pressure field and θ the temperature fluctuation field. The system parameters are: rotation frequency Ω, gravity acceleration g, kinematic viscosity ν, thermal diffusivity κ, aspect ratio $\Gamma = L/H$, vertical temperature gradient $\Delta T/H$, and thermal expansion coefficient α. The temperature field is decomposed into a linear mean profile (which causes the scalar driving term in (2.3)) and fluctuations θ

$$T(\mathbf{x},t) = -\frac{\Delta T}{H}(z - H/2) + \theta(\mathbf{x},t)\,. \tag{2.4}$$

Since T is prescribed at boundaries $z = 0$ and π, the condition $\theta = 0$ follows there. Here, $\Delta T > 0$. The dimensionless control parameters are the Prandtl number Pr, the Rayleigh number Ra, the Taylor number Ta, and the aspect ratio Γ,

$$Pr = \frac{\nu}{\kappa}\,, \quad Ra = \frac{\alpha g H^3 \Delta T}{\nu\kappa}\,, \quad Ta = \frac{4\Omega^2 H^4}{\nu^2}\,, \quad \Gamma = \frac{L}{H}\,. \tag{2.5}$$

The simulation domain is $V = L\times L\times H = [0,\Gamma\pi]\times[0,\Gamma\pi]\times[0,\pi]$. In lateral directions x and y, periodic boundary conditions are taken. In the vertical direction z, free-slip boundary conditions are used which are given by

$$u_z = \theta = 0 \quad \text{and} \quad \frac{\partial u_x}{\partial z} = \frac{\partial u_y}{\partial z} = 0\,. \tag{2.6}$$

The numerical method which is used is a pseudospectral scheme[7]. The partial differential equations of motion (2.1) – (2.3) are transformed to an infinite-dimensional system

of nonlinear ordinary differential equations in Fourier space. In a compact notation, the equations for a Fourier mode $\phi(\mathbf{k},t) = (u_i(\mathbf{k},t), \theta(\mathbf{k},t))$ and $\gamma = (\nu, \kappa)$ with a given wavevector $\mathbf{k}$ result then to

$$\frac{\partial \phi(\mathbf{k},t)}{\partial t} = -\gamma k^2 \phi(\mathbf{k},t) + F(\phi(\mathbf{q},t),t) \tag{2.7}$$

where the last term of (2.7) contains all mode couplings that arise due to the nonlinearity of the original equations. In order to avoid the explicit calculation of very large convolution sums, one transforms the respective terms into the physical space. Here, the products of the fields can be evaluated gridpointwise. The back- and forward transformations between physical and Fourier space are established by Fast Fourier Transformations (FFT) using the IBM-ESSL[8] or fftw[9] libraries.

The first term on the r.h.s. of (2.7) can be treated by the integrating factor. The substitutions $\tilde{\phi} = \exp(\gamma k^2 t)\phi$ and $\tilde{F} = \exp(\gamma k^2 t)F$ lead to

$$\frac{\partial \tilde{\phi}(\mathbf{k},t)}{\partial t} = \tilde{F}(\phi(\mathbf{q},t),t)\,. \tag{2.8}$$

This translates into the following second-order predictor-corrector scheme for the time integration[10]. We define the integrating factor $I_n = \exp(\gamma k^2 t_n)$. The predictor step is then given by

$$\phi^* = \frac{I_n}{I_{n+1}}[\phi_n + \Delta t F_n]\,. \tag{2.9}$$

The corrector step with $F^* = F(\phi^*)$ follows then to

$$\phi_{n+1} = \frac{I_n}{I_{n+1}}\left[\phi_n + \frac{\Delta t}{2}F_n\right] + \frac{\Delta t}{2}F^*\,. \tag{2.10}$$

3 3D-FFT Packages for Blue Gene/L

One of the main building blocks of the present numerical method is the Fast Fourier Transform (FFT). The classical parallel implementation of three-dimensional FFTs uses a slabwise decomposition of the simulation domain. For a simulation with N^3 grid points, the method allows a parallelization on up to N processors. Due to the rather small memory size per core, the Blue Gene/L requires a *volumetric* FFT which decomposes the three-dimensional volume into so-called pencils and hence allows a parallelization degree of N^2. The prime requirement for being able to run a simulation with a large grid is that the domain fractions of the grid (including buffers and temporary storage) fit into the 512 MB of memory on a single Blue Gene/L node. At same time, of course, the FFT algorithm should also be scalable, i.e. increasing the number of CPUs to solve the problem should also substantially decrease the time-to-answer.

To check this we compared[11] three FFT packages on a small test case: the old slabwise method, the BGL3DFFT package by M. Eleftheriou *et al.*[12] and the P3DFFT package by D. Pekurovsky[13]. The second package is written in C++ and contains complex-to-complex FFTs only (Fortran- and C-bindings are available from IBM). The first and third package are available in Fortran. The results on strong scaling are summarized in Fig. 1. For the

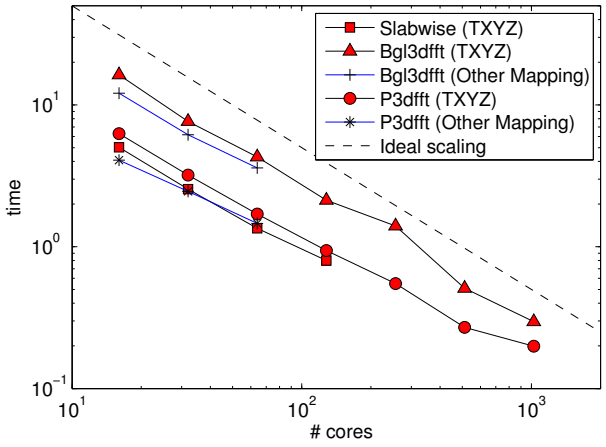


Figure 1. Strong scaling tests for the three different FFT packages on up to 1024 cores. The test function is $f(x, y, z) = \sin(x)\cos(2y)\sin(z)$ which is resolved on an equidistant 128^3 cubic grid of side length 2π. Other mapping means that we have edited map files by hand.

volumetric FFT packages, we have varied the mapping of the MPI tasks onto the torus grid network of Blue Gene/L . By default, Blue Gene/L maps MPI tasks on all primary cores of a partition first, i.e. the corresponding environment variable is set to `BGLMPI_MAPPING = XYZT`. Beside the fact that the P3DFFT interface and implementation supports our needs in an optimal way (real-to-complex/complex-to-real FFTs), it also turned out to be the best solution in terms of performance. For example, the symmetry $\mathbf{u}(\mathbf{k}) = \mathbf{u}^*(-\mathbf{k})$ for real-to-complex transforms is explicitely implemented already. Additionally, the in-place transformation flag allows the storage of physical and Fourier space fields in the same array.

4 Simulation Results

4.1 Scaling Rests on Blue Gene/L

The strong scaling of the whole simulation code is shown in Fig. 2 for 16 to 2048 cores. The resolution is $512 \times 512 \times 64$ grid points with an aspect ratio $\Gamma = L/H = 16$. All tests have been done in the virtual node (VN) mode. For more than 256 cores the `mpirun` option for the processor topology, `BGLMPI_MAPPING = XYZT`, brought the shortest computation times. The analysis of the code with the `mpitracelib` indicated that after these steps had been made almost all of the communication overhead remained in the `MPI_alltoallv` communication task which is required in the FFT algorithm for the transposition of the pencils. Hence it is the only factor which limits strong scaling on a larger number of cores eventually. Future improvements of the implementation of `MPI_alltoallv` will therefore have an immediate impact on the performance of our code.

Table 1 summarizes our findings for grid resolutions in production runs, i.e. $N_x \times N_y \times N_z = 4096 \times 4096 \times 256$ at $\Gamma = 32$. For such large grids it became necessary to implement

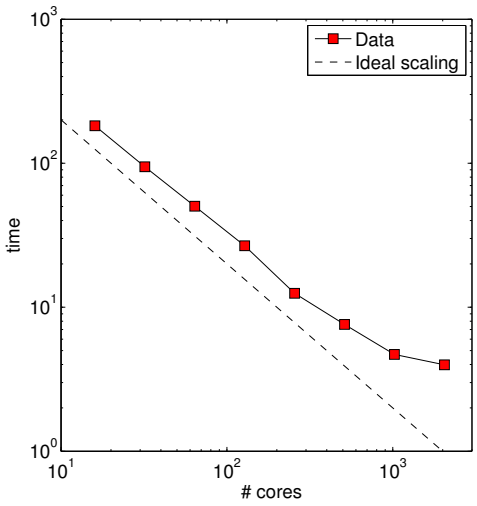


Figure 2. Strong scaling test for the code from 16 to 2048 CPUs cores. Note that the computational volume for this test case is a very flat cell with an aspect ratio of $\Gamma = 16$.

some internal summation loops in double precision. All of these loops appear in the calculation of the right-hand-side of the equations of motion which is the most expensive part of the program. The number of grid points per MPI task was of the order of 10^6 which is simply too large to sum over with 32-bit floating point precision. We have studied the two

# tasks	2048	1024	4096	4096	8192	4096	16384	8192	8192
Mode	VN	CO	VN	VN	VN	CO	VN	CO	VN
$iproc$	32	32	64	128	128	64	128	128	512
$jproc$	64	32	64	32	64	64	128	64	16
time(s)	183.3	205.7	118.6	109.6	77.8	90.66	62.6	60.5	74.0

Table 1. Runtime tests for the run with an aspect ratio $\Gamma = 32$. The CPU time was measured with the MPI_Wtime() task for a loop of 5 integration steps.

modes of the dual core chip, the virtual node mode (VN) and the co-processor mode (CO). The parameters $iproc$ and $jproc$ in the table represent the two dimensions of the processor grid that has to be generated for the volumetric FFTs. Note, that the ratio $iproc/jproc$ can be tuned as long as $iproc \times jproc = N_{core}$. In general, a ratio of one would be best, but other choices can be better, if they improve the mapping of the MPI tasks onto the partition grid. For the large problem we observe that the differences between the VN and CO modes are small. This is due to the fact that large local grid fractions do no longer fit into the L3 cache of a Blue Gene/L node and have to be streamed from memory. If the second core is used for computation (VN mode) the two cores are actually competing for the memory bandwidth of the node and the gain in computing time is fairly small. The

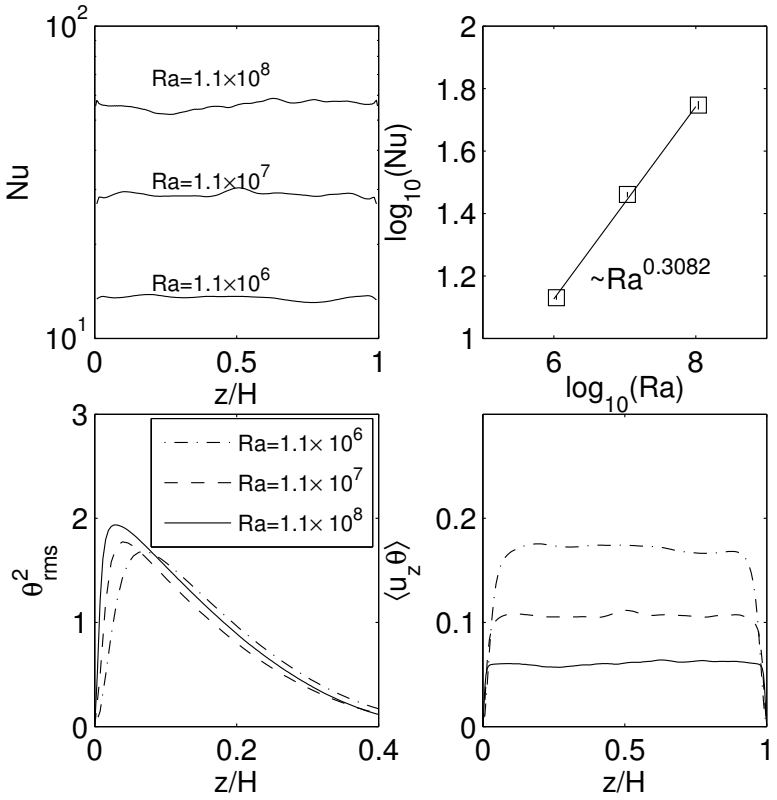


Figure 3. Turbulent heat transport in convective turbulence. Left upper panel: Vertical profiles $Nu(z)$ for three runs. The mid line is the Blue Gene/L run. Upper right panel: Nusselt number scaling with respect to Ra. The resulting scaling exponent is also indicated in the figure. Lower left panel: Vertical profiles of the mean square temperature fluctuations. Lower right panel: Vertical profiles of the mixed turbulent stress $\langle u_z \theta \rangle$. Linestyles are the same as in the lower left panel.

communication time in VN mode is however bigger than in CO mode, such that these two effects almost compensate each other. The table indicates that the runtime starts to saturate for $N_{core} \geq 8192$. There is no chance to improve the cache locality of the code to avoid this, because the FFTs will always require the full grid. There may however be some further optimization potential by using explicit task mappings. The main problem here is to find an embedding of a 2d grid decomposition into a 3d torus communication grid, such that no communication hot spots occur on the point-to-point network of the Blue Gene/L. In general, this is a hard problem.

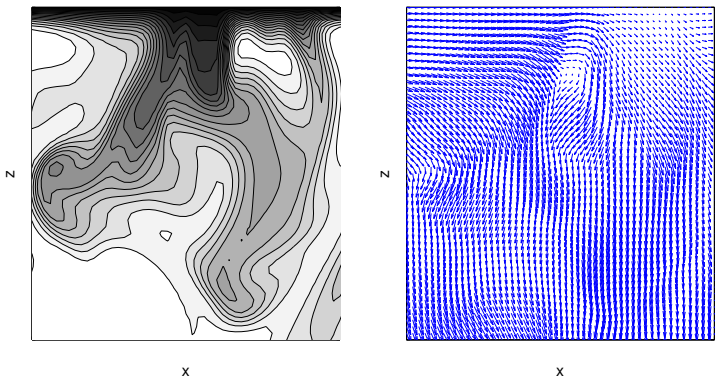


Figure 4. Magnification of a small fraction of the vertical cut through the three-dimensional volume at $y = \pi$. The region in the vicinity of the cold top plane is shown for a simulation at $Ra = 1.1 \times 10^8$. Left panel: Full temperature field T. Right panel: Corresponding velocity field (u_x, u_z). The figures illustrate a cold finger-like plume which falls down into the turbulent bulk and the corresponding fluid motion.

4.2 Comparison of First Physical Results

First physical results for convective turbulence without rotation are presented now for an aspect ratio $\Gamma = 2$. The simulation was conducted at $Ra = 1.1 \times 10^7$ and $Pr = 0.7$. The results are compared with former simulation runs in the same parameter range and for the same Γ. The resolution for this benchmark run is $N_x \times N_y \times N_z = 256 \times 256 \times 256$. It was run on 32 cores. Figure 3 presents our results for the turbulent heat transport, the mean square fluctuations of the temperature field and the mixed turbulent stress of temperature and vertical velocity. The turbulent heat transport is given by the dimensionless Nusselt number Nu which is defined as

$$Nu(z) = \frac{\langle u_z \theta \rangle_A - \kappa \partial \langle T \rangle_A / \partial z}{\kappa \Delta T / H}, \tag{4.1}$$

where $\langle \cdot \rangle_A$ denotes an statistical average over lateral $x - y$ planes. The upper left panel of Fig. 3 demonstrates that the relation $Nu(z) = const.$ holds approximately for all cases. The mean of $Nu(z)$ over z is Nu which is shown as a function of the Rayleigh number in the upper right panel. The algebraic scaling $Nu \sim Ra^{0.31}$ corresponds with findings from other simulations. The vertical profiles of the mean square temperature fluctuations are shown in the lower left panel. The distance of the maximum of the profile to the free-slip wall corresponds with the thermal boundary layer thickness. It decreases with increasing Ra and puts thus stronger resolution requirements on the simulation. The lower right panel shows the turbulent stress $\langle u_z \theta \rangle$. The turbulence becomes homogeneous for the lateral planes that form the plateau in the profiles. We also found that very long time series are necessary for a satisfactory turbulence statistics. The reason for a long time series lies in coherent structures that are formed in the vicinity of both free-slip planes as shown in Fig. 4. They go in line with strong local correlations between the fluctuating temperature field and the turbulent vertical velocity component. To summarize, the results of the BG/L test run fit consistently into the former studies.

5 Concluding Remarks

Significant efforts had to be taken in order to port the pseudospectral turbulence simulation code on the massively parallel Blue Gene/L system, including the switch to fast Fourier transformations that are parallelized in two dimensions. Two factors are coming together in our problem: the small amount of memory per core and the very flat geometry that causes strongly asymmetric pencils in the domain decomposition. The use of the P3DFFT package and a number of changes in the communication overhead improved the performance of the code on the Blue Gene/L system significantly and provide now a basis for further studies of turbulence in this specific flat geometry. The latter is of interest for many geophysical and astrophysical problems.

Acknowledgements

JS wishes to thank the John von Neumann-Institute for Computing for their steady support with supercomputing ressources. We thank J.C. Bowman, R. Grauer, D. Pekurovsky and P.K. Yeung for helpful discussions. This work is supported by the Deutsche Forschungsgemeinschaft under Grant No. SCHU/1410-2.

References

1. B. Cushman-Roisin, *Introduction to Geophysical Fluid Dynamics*, (Prentice Hall, Englewood Cliffs, NJ, 1994).
2. P. Tabeling, *Two-dimensional turbulence: a physicist approach*, Phys. Rep., **362**, 1–62, (2002).
3. J.-D. Fournier and U. Frisch, *d-dimensional turbulence*, Phys. Rev. A, **17**, 747–762, (1978).
4. L. P. Kadanoff, *Turbulent heat flow: structures and scaling*, Phys. Today, **54**, 34–39, (2001).
5. K. Julien, S. Legg, J. C. McWilliams and J. Werne, *Rapidly rotating turbulent Rayleigh-Benard convection*, J. Fluid Mech., **322**, 243–273, (1996).
6. J. Schumacher, K. R. Sreenivasan and V. Yakhot, *Asymptotic exponents from low-Reynolds-number flows*, New J. Phys., **9**, art. no. 89, (2007).
7. C. Canuto, M. Y. Hussaini, A. Quarteroni and T. A. Zang, *Spectral Methods Methods in Fluid Dynamics*, (Springer, Berlin, 1988).
8. http://www.www.ibm.com/systems/p/software/essl.html
9. http://www.fftw.org/
10. P. K. Yeung and S. B. Pope, *An algorithm for tracking fluid particles in numerical simulations of homogeneous turbulence*, J. Comp. Phys., **79**, 373-416, (1988).
11. J. Schumacher and M. Pütz, *Report on the Blue Gene/L Scaling Workshop 2006*, Editors: W. Frings, M.-A. Hermanns, B. Mohr and B. Orth, Technical Report FZJ-ZAM-IB-2007-02, , 15–18, (2007).
12. M. Eleftheriou, B. G. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. S. Germain, *Scalable framework for 3D FFTs on the Blue Gene/L supercomputer: implementation and early performance measurements*, IBM J. Res. & Dev., **49**, 457–464, (2005).
13. http://www.sdsc.edu/us/ressources/p3dfft.php

Large Simulations of Shear Flow in Mixtures via the Lattice Boltzmann Equation

Kevin Stratford[1] **and Jean Christophe Desplat**[2]

[1] Edinburgh Parallel Computing Centre, The University of Edinburgh, Edinburgh, United Kingdom
E-mail: kevin@epcc.ed.ac.uk

[2] Irish Centre for High-End Computing, Dublin, Ireland
E-mail: j-c.desplat@ichec.ie

In this paper we describe the implementation of Lees-Edwards sliding periodic boundary conditions used to perform simulations of sheared binary fluids. In conjunction with domain decomposition and parallelism using the message passing interface, we are able to perform shear simulations of significant size and duration. We discuss the scaling and performance of these large calculations on the IBM Blue Gene/L architecture.

1 Introduction

Fluid dynamics presents many computational challenges, particularly in the area of complex fluids, where microscopic/mesoscopic details of the fluid components are important in addition to the bulk properties such as the viscosity. One useful method for studying such systems is based on the lattice Boltzmann equation (LBE) for the incompressible Navier-Stokes equations (for a review see e.g., ref. 1). The LBE provides a natural way for the microscopic details — e.g., composition, liquid crystal ordering, and so on — to be coupled to the fluid flow. In addition, by relaxing the constraint of exact incompressibility the LBE allows the fluid pressure to be computed locally, and thus is extremely well suited to parallel computation.

One application where the LBE is useful is flow involving a mixture of fluids, where the fluid-fluid interface can evolve on the lattice in a natural way without the need for explicit interface tracking. One example of such a problem is *spinodal decomposition* involving two symmetric liquids. If at high temperature the liquids are miscible, a drop in temperature can cause spinodal decomposition, where the liquids separate and form domains which grow continuously in time. The growth in domain size occurs in a well-understood fashion and is ultimately limited by the container size in experiment, or the system size in simulation. However, experiments on sheared systems report saturation in the length scales after a period of (anisotropic) domain growth. The extreme elongation of the domains in the flow direction means that finite-size effects cannot be excluded as the reason for saturation even in experiments. Evidence for steady-states from simulations large enough to be uncontaminated by finite-size effects is therefore of great interest. Recently, a number of works have addressed this type of problem using simulation[5,6,12,13]; here, we describe the steps we take to implement the calculation.

In two dimensions, our recent results indicate unambiguously that non-equilibrium steady states do exist in sheared binary mixtures[6]. Such two-dimensional calculations, based typically on 1024x512 lattices run for 500,000 time steps, are tractable with relatively modest computational effort. In three dimensions, the situation is slightly more

complex as the domains can grow in the vorticity direction allowing the possibility of different physics[12]. To avoid finite-size effects here, systems of up to 1024x512x512 have been used, which must also be run for a similar length of time to probe possible steady states. These three-dimensional calculations then present a more substantial computational challenge for which parallel computing is essential.

In our LBE calculations, a shear flow is driven by block-wise introduction of Lees-Edwards sliding periodic boundary conditions (first used in molecular dynamics[2]). When using the LBE, the system is divided in the shear direction into a number of slabs which translate (conceptually) relative to each other as the simulation proceeds[10]. The translation of the slabs sets up a linear shear profile across the system. In practice, this subdivision of the system requires a refined version of the halo exchange between parallel sub-domains in the appropriate coordinate direction. We discuss two approaches to the implementation of the sliding periodic boundaries: (1) using point-to-point communication, and (2) using MPI-2 single-sided communication.

In the following section we give a brief overview of the LBE as used for binary fluids and its related performance and scalability issues. We include a number of indicative results in the final section and comment on their significance.

2 Short Description of the LBE

2.1 The LBE

The central computational object in the lattice Boltzmann approach is the distribution function $f_i(\mathbf{r};t)$, which can be thought of as representing the contributions to the density of fluid at given lattice site $\mathbf{r}$ having discrete velocity $\mathbf{c}_i$, for $i = 1,\ldots,N$. A set of N discrete velocities is chosen to ensure the appropriate symmetry properties of the Navier-Stokes equations are maintained on the lattice. In three dimensions, N is typically 15 or 19, which includes the null velocity $\mathbf{0}$[3]. The non-zero $\mathbf{c}_i$ are such that $\mathbf{r}+\mathbf{c}_i\Delta t$ is a neighbouring lattice site, i.e., each element of the distribution *propagates* one lattice spacing in a different direction in a discrete time step Δt.

The N elements of the distribution function are complemented by N modes $m_i(\mathbf{r};t)$, which include the important hydrodynamic quantities of the system. In three dimensions, there are ten hydrodynamic modes: the density, three components of the momentum, and six independent components of the stress. The remaining modes are non-hydrodynamic, or "ghost" quantities, which do not have direct physical interpretation. Using Greek subscripts to denote Cartesian directions, the hydrodynamic modes may be written as moments of the distribution function: the density is $\rho(\mathbf{r};t) = \sum_i f_i(\mathbf{r};t)$; the momentum is $\rho u_\alpha(\mathbf{r};t) = \sum_i c_{i\alpha} f_i(\mathbf{r};t)$; the stress is $\Pi_{\alpha\beta}(\mathbf{r};t) = \sum_i c_{i\alpha}c_{i\beta} f_i(\mathbf{r};t)$. (In what follows we assume the repeated index is summed over.) More generally, a transformation can be made between the modes and the distributions by writing $m_i(\mathbf{r};t) = E_{ij} f_j(\mathbf{r};t)$, where E_{ij} is a known matrix of eigenvectors for a given choice of velocity set $\mathbf{c}_i$. Likewise, the inverse transformation can be used to obtain the distributions from the modes, $f_i(\mathbf{r};t) = E^{-1}_{ij} m_j(\mathbf{r};t)$.

The time evolution of the distribution is then described by a discrete Boltzmann equation, i.e., the LBE:

$$f_i(\mathbf{r}+\mathbf{c}_i\Delta t; t+\Delta t) - f_i(\mathbf{r};t) = C_{ij}\left(f_j(\mathbf{r};t) - f^{eq}_j(\mathbf{r};t)\right). \qquad (2.1)$$

The left hand side can be viewed as the propagation stage, where each element of the distribution is moved one lattice spacing in the appropriate direction. The right hand side represents a *collision* stage which introduces the appropriate physics through the collision operator C_{ij} and some choice of equilibrium distribution $f_j^{eq}(\mathbf{r};t)$. In practice, it is convenient to perform the collision stage not on the distribution function but on the modes, where the physical interpretation is clear. Both the density and the momentum are conserved by the collision, and so are unchanged. The stress is relaxed toward the equilibrium value $\Pi_{\alpha\beta}^{eq} = \rho u_\alpha u_\beta + p\delta_{\alpha\beta}$ at a rate related to the fluid viscosity, with p being the fluid pressure. Treatment of the remaining ghost modes may vary but they can, for example, be eliminated directly at each time step. As the fluid obeys an ideal gas equation of state, with $p = \rho c_s^2$ relating the pressure to the density and the speed of sound on the lattice c_s, it can be seen that the collision process is entirely local at each lattice site.

The propagation and collision stages are then performed in turn by repeated transformations between the distribution and mode representations. Computationally, the propagation stage involves essentially no floating point operations, while the collision stage requires the two matrix-vector multiplications $E_{ij}f_i$ and $E_{ij}^{-1}m_i$ needed for the transformations. This gives approximately $2N^2$ floating point operations per lattice site. The LBE (2.1) approximates the Navier-Stokes equation for incompressible flow provided a low Mach number constraint is satisfied. This means the local fluid velocity u must remain small compared with the speed of sound on the lattice c_s or, alternatively, that the density does not deviate significantly from the mean. The Mach number constraint may be thought of as analogous to the Courant-like condition in finite difference approximations to the equations of motion where information must not propagate beyond one lattice spacing in one discrete time step. In a parallel domain decomposition, one round of communication is required before the propagation stage at each step to allow neighbouring distributions to move to adjacent lattice sites. Owing to the local nature of the pressure calculation, no global communication is required.

2.2 Binary Mixtures

A number of different methods are available to represent mixtures which use a lattice kinetic equation analogous to the LBE[7,8]. Here, we use a method which employs a compositional order parameter to represent the relative concentration of the two fluids, along with the appropriate dynamics near the interface. The coupling between the fluids introduces a force in the Navier-Stokes equations from the bending of the interface, while the order parameter is advected at the local fluid velocity. The order parameter introduces a new distribution to represent the composition of the fluid. This distribution undergoes the same propagation stage to the density distribution $f_i(\mathbf{r};t)$, and a similar collision process to introduce the required interfacial physics. In what follows, we only refer specifically to the density distribution; an analogous description holds for the distribution representing the composition.

2.3 Lees Edwards Sliding Periodic Boundaries

There are a number of different ways in which shear can be introduced into a system in the LBE picture. For example, it is particularly easy to represent solid walls with the LBE

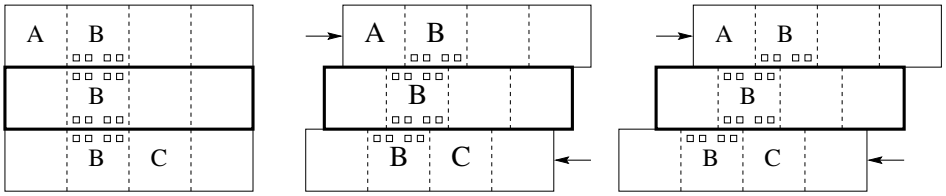


Figure 1. Schematic diagram showing the blockwise decomposition of the system with different slabs translating relative to each other at a constant velocity $\pm u_{LE}$ in the horizontal direction. Here, the sliding boundaries are represented by the horizontal lines, and four processor subdomains (A, B, C, D) by the vertical dashed lines. Elements of the distribution f_i which cross the sliding boundaries (i.e., those at adjacent lattice sites, small squares) undergo a transformation to account for the relative change in velocity. As adjacent blocks translate, communication is required between different subdomains to obtain the neighbouring lattice sites as a function of the displacement $u_{LE}t$. For example, subdomain B requires no direction communication at $t = 0$ (left), but communication with A at the upper edge and C at the lower edge at the later times (middle, right).

by using reflection boundary conditions for the distributions ("bounce-back on links"). Two side walls, or planes, moving in opposite directions at velocity u_w and separated by the length of the system in the shear direction L can then impart shear to fluid with rate $\dot{\gamma} = 2u_w/L$. While straightforward, this approach does not allow high shear rates to be achieved when the distance between the walls becomes large. Owing to the low Mach number constraint, the maximum fluid velocity near the walls is c_s, and hence the overall shear rate is limited to no more than $2c_s/L$. Likewise, using a body force on the bulk of the fluid to drive, for example, a Couette flow, is subject to the same sort of limitation.

To overcome this constraint, block-wise Lees-Edwards sliding periodic boundary conditions have been introduced[10]. Here, the system is decomposed into two or more slabs which translate relative to each other at a fixed velocity $\pm u_{LE}$. While the value of u_{LE} is subject to the same Mach number constraint as for the solid walls, a large number of slabs N_{LE} may be used to increase the maximum available shear rate to $2N_{LE}c_s/L$. In no block does the flow locally exceed approximately u_{LE} relative to the lattice, so the Mach number constraint is always obeyed. The situation is represented schematically in Fig. 1. Three horizontal slabs are shown, the central one of which can be considered at rest (only relative motion of the blocks is important). At its upper boundary the central block sees the neighbouring block translate to the right at speed u_{LE}, while the block at its lower boundary translates in the opposite direction. Conventional periodic boundary conditions are applied at the limits of the system in all dimensions. The resulting fluid flow is represented in Fig. 2. The velocity or flow direction is parallel to the sliding boundaries, while a velocity gradient is set up in the direction normal to the boundaries.

As shown in Fig. 1, the sliding boundaries separate lattice sites in adjacent planes of the system but do not coincide. This means the collision stage can take place locally at each lattice site unaffected by the presence of the sliding boundaries. However, at the propagation stage, elements of the distribution associated with velocities $\mathbf{c}_i$ which have a component crossing the boundary must undergo a Galilean transformation to take account of the velocity difference between the slabs. As the relative velocity u_{LE} is not a lattice vector $\mathbf{c}_i$, the transformation required is best understood in terms of the modes rather than the distributions[9]. We denote the hydrodynamic quantities in the upper slab with a star. In

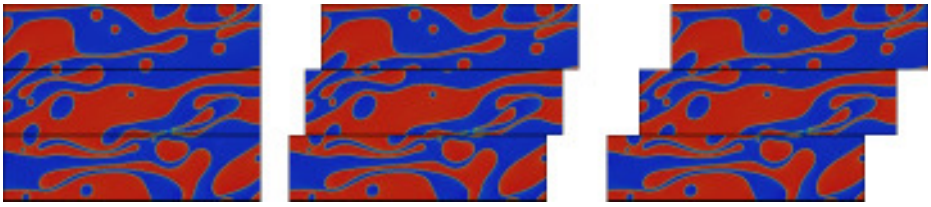


Figure 2. A schematic representation of the result from a binary mixture undergoing shear as a function of time. As the simulation proceeds, the shear causes domains of the two fluids (here shaded differently) to stretch and elongate in the flow direction. As the slabs translate relative to each other, the fluid motion remains continuous across the sliding boundaries. A coherent picture can be obtained by displacing the slabs $u_{LE}t$ and using the periodic boundary conditions in the horizontal direction. The shear direction is in the vertical; the flow direction is horizontal; in three dimensions the vorticity direction is perpendicular to the page.

crossing the boundary from the central block to the upper block the density is unchanged $\rho^\star = \rho$, the momentum is incremented $\rho u^\star_\alpha = \rho u_\alpha + \rho u_{LE\alpha}$, and the stress $\Pi^\star_{\alpha\beta} = \Pi_{\alpha\beta} + \rho u^\star_\alpha u_{LE\beta} + \rho u^\star_\beta u_{LE\alpha} + \rho u_{LE\alpha} u_{LE\beta}$. There is no change in the ghost modes if they are constrained to zero at each step. Having the appropriate modes $m^\star_i$ allows the corresponding transformed distributions $f^\star_i = E^{-1}_{ij} m^\star_j$ to be computed.

In the case of a binary mixture, the interfacial force in the Navier-Stokes equations requires information on the spatial gradient of the order parameter. The method used for the exchange of information required for gradients normal to the sliding boundaries is the same as that for the distributions.

2.4 Implementation

The starting point for the implementation of the sliding periodic boundary conditions in parallel is a regular three-dimensional domain decomposition. While sliding boundaries are required at regular intervals in the shear direction to drive the flow, their placement should not constrain the processor decomposition and so affect the parallel scaling of a large calculation. However, to avoid unnecessary complication, it is convenient to ensure the sliding boundaries coincide with neither the processor subdomain boundaries nor the true periodic boundary in the shear direction. This does not, in practice, place any serious constraint on the number of slabs which can be used, as the slab boundaries and the processor subdomain boundaries can be interleaved. This means that the halo exchange of distribution values required at the processor boundaries (and the actual periodic boundaries) is unchanged in the presence of shear. In practice, the slabs are typically 8–32 lattice sites in thickness; smaller slabs increase the communication requirement and open the possibility of artifacts at the boundaries.

While the blocks translate conceptually in relation to one another, there is clearly no movement of memory associated with the blocks. For lattice sites at the edge of a slab, the neighbouring locations *in memory* are fixed at time $t = 0$. As time proceeds, the location in memory associated with a neighbouring physical lattice site moves in the opposite direction to the translating slab (see Fig. 3). At times $t > 0$, an extra communication is therefore required in the along-boundary direction to make neighbouring information available; potentially two different processor subdomains are involved at each boundary

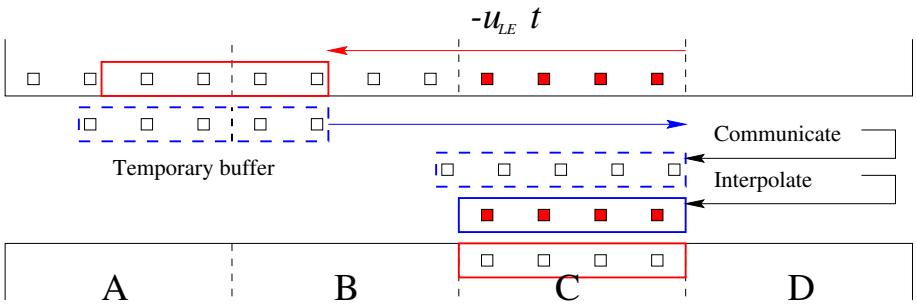


Figure 3. Schematic diagram showing the pattern of communication and interpolation required in the fixed frame of reference of memory to compute appropriate boundary-crossing distribution values for the propagation stage. For the given displacement $u_{LE}t$, the values required by processor subdomain C in the lower slab are found split between processor subdomains A and B in the upper slab. Communication of transformed distributions $f_i^\star$ takes place to a temporary buffer, where linear interpolation to the final values (shaded lattice sites) required by C can also take place. An analogous exchange takes place in the opposite direction for values required in the upper slab.

as a function of time. The processor subdomains involved are determined by the current displacement $u_{LE}t$; periodic boundaries mean this displacement is modulo the system size in the direction of the sliding boundaries.

Further, as the displacement $\pm u_{LE}t$ is not necessarily a whole number of lattice spacings, an interpolation between neighbouring memory locations is required to get the appropriate cross-boundary distribution values $f_i^\star$ so that the propagation stage can take place as normal. To do this, communication of all values required for a linear interpolation are received by a temporary buffer. The interpolation, determined by the displacement modulo the lattice spacing takes place in the temporary buffer. The boundary-crossing distributions are finally copied to the memory locations neighbouring the destination lattice sites.

2.5 MPI Communications

Two different implementations of the along-boundary communication have been used. The first uses MPI point-to-point communication[14] via `MPI_Send` and `MPI_Recv` between the relevant subdomains. The second uses MPI 2[15] single-sided communication via `MPI_Get`. This requires that appropriate memory allocations are made with `MPI_Alloc_mem` and appropriate remote memory access windows are identified with `MPI_Win_create`. In both cases, the source process for the exchanged data can be computed as a function of time from the slab displacement $u_{LE}t$ modulo the system size in the along-boundary direction.

Where MPI-2 implementations are available, for example on the IBM p690+, we have found that both implementations give very similar results in terms of performance. The results described use the point-to-point implementation. MPI-2 implementation of single-sided communication is not universally available at the time of writing.

3 Results

A recent work[12] has examined the characteristics of shear flow in three-dimensional binary fluids using the methods described here. The requirement that finite size effects be

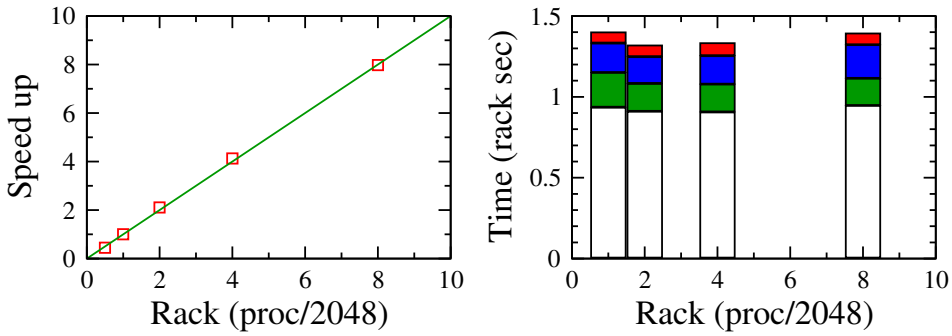


Figure 4. Benchmark results for a 1024x512x512 lattice with shear driven by 32 equally spaced Lees-Edwards sliding boundaries, performed on the IBM Blue Gene/L system at Thomas J. Watson Research Center. The left panel shows the scaling of the calculation as a function of the number of racks (2048 processors, all virtual node mode) normalised to one rack. The right panel shows rack number multiplied by time in seconds for various parts of the calculation: from the bottom, collision, propagation, lattice halo swaps, and Lees-Edwards transformation (including along-boundary communication).

excluded means that large system sizes of up to 1024x512x512 lattice sites have been used. Shear was generated by 32 equally spaced slabs in one of the shorter directions. The calculations were performed using the IBM Blue Gene/L machine at the University of Edinburgh, where a single rack is available (2048 MPI tasks in virtual node mode — all computations take place in this mode). To investigate the scaling of this type of calculation on larger machines, we have also run the same 1024x512x512 problem on the BG/L system at IBM Thomas J. Watson Research Center on up to 8 racks (16,384 MPI tasks). The scaling of the calculation and breakdown of the LBE algorithm are illustrated in Fig. 4. The scaling of the calculation is extremely good, while the absolute performance is is in the range of 5–10% of theoretical peak performance.

4 Summary

We have described the implementation of Lees-Edwards sliding periodic boundary conditions in parallel to allow simulations of large systems under shear. In conjunction with high performance computing this method, based on the LBE, provides an excellent way to investigate the properties of complex fluids.

Acknowledgements

We thank IBM Thomas J. Watson Research Center for the chance to use their Blue Gene/L system, along with Ronojoy Adhikari, Mike Cates and Paul Stansell for many useful discussions. KS acknowledges support from the UK EPSRC Grant EP/C536452 (RealityGrid). JCD is grateful to the Irish Centre for High-End Computing for access to its facilities.

References

1. S. Succi, *The Lattice Boltzmann Equation and Beyond*, (Clarendon Press, Oxford 2000).
2. A. W. Lees and S. F. Edwards, *The computer study of transport processes under extreme conditions*, J. Phys. C: Solid State Phys., **5**, 1921, (1972).
3. Y. H. Qian, D. d'Humières and P. Lallemand, *Lattice BGK models for Navier-Stokes Equation*, Europhys. Lett., **17**, 479, (1992).
4. M. E. Cates, V. M. Kendon, P. Bladon and J.-C. Desplat, *Inertia, coarsening and fluid motion in binary mixtures*, Faraday Disc., **112**, 1, (1999).
5. F. Corberi, G. Gonnella and A. Lamura, *Spinodal decomposition of binary mixtures in uniform shear flow*, Phys. Rev. Lett., **81**, 3852, (1998); *Two-scale competition in phase separation with shear*, Phys. Rev. Lett., **83**, 4057, (1999).
6. P. Stansell, K. Stratford, J.-C. Desplat, R. Adhikari and M. E. Cates, *Nonequilibrium steady states in sheared binary fluids*, Phys. Rev. Lett., **96**, 085701, (2006).
7. M. R. Swift, W. R. Osborn and J. M. Yeomans, *Lattice Boltzmann simulation of non-ideal fluids*, Phys. Rev. Lett., **75**, 830, (1995); M. R. Swift, E. Orlandini, W. R. Osborn, and J. M. Yeomans, *Lattice Boltzmann simulations of liquid-gas and binary fluid systems*, Phys. Rev. E, **54**, 5041, (1996).
8. A. K. Gunstensen, D. H. Rothman, S. Zaleski and G. Zanetti, *Lattice Boltzmann model of immiscible fluids*, Phys. Rev. A, **43**, 4320, (1991); X. Shan and H. Chen, *Lattice Boltzmann model for simulating flows with multiple phases and components*, Phys. Rev. E, **47**, 1815, (1993); L.-S. Luo and S. S. Girimaji, *Theory of the lattice Boltzmann method: Two-fluid model for binary mixtures*, Phys. Rev. E, **67**, 036302, (2003).
9. R. Adhikari, J.-C. Desplat and K. Stratford, *Sliding periodic boundary conditions for the lattice Boltzmann and lattice kinetic equations*, cond-mat/0503175, (2005).
10. A. J. Wagner and I. Pagonabarraga, *Lees-Edwards boundary conditions for lattice Boltzmann*, J. Stat. Phys., **107**, 521, (2002).
11. K. Stratford, R. Adhikari, I. Pagonabarraga and J.-C. Desplat, *Lattice Boltzmann for binary fluids with suspended colloids*, J. Stat. Phys., **121**, 163, (2005).
12. K. Stratford, J.-C. Desplat, P. Stansell and M. E. Cates, *Binary fluids under shear in three dimensions*, Phys. Rev. E, **76**, 030501(R), (2007).
13. J. Harting, G. Giupponi and P. V. Coveney, *Structural transitions and arrest of domain growth in sheared binary immiscible fluids and microemulsions*, Phys. Rev. E, **75**, 041504, (2007).
14. Message Passing Interface Forum (1995), *A message passing interface standard*. http://www.mpi-forum.org/
15. Message Passing Interface Forum (1997). *MPI-2: Extensions to the message passing interface*. http://www.mpi-forum.org/

Simulating Materials with Strong Correlations on BlueGene/L

Andreas Dolfen, Yuan Lung Luo, and Erik Koch

Institut für Festkörperforschung
Forschungszentrum Jülich, 52425 Jülich, Germany
E-mail: {a.dolfen, y.luo, e.koch}@fz-juelich.de

Understanding the physics of strongly correlated materials is one of the grand-challenges in condensed-matter physics. Simple approximations such as the local density approximation fail, due to the importance of the Coulomb repulsion between localized electrons. Instead we have to resort to non-perturbative many-body techniques. Such calculations are, however, only feasible for quite small model systems. This means that the full Hamiltonian of a real material has to be approximated by a model Hamiltonian comprising only the most important electronic degrees of freedom, while the effect of all other electrons can merely be included in an average way. Realistic calculations of strongly correlated materials need to include as many of the electronic degrees of freedom as possible.

We use the Lanczos method for calculating ground state and dynamical properties of such model Hamiltonians. In the method we have to handle the full many-body state of the correlated system. The method is thus limited by the available main memory. The principal problem for a distributed-memory implementation is that the central routine of the code, the application of the Hamiltonian to the many-body state, leads, due to the kinetic energy term, to very non-local memory access. We describe a solution for this problem and show that the new algorithm scales very efficiently on BlueGene/L. Moreover, our approach is not restricted to correlated electrons but can also be used to simulate quantum spin systems relevant for quantum computing.

1 Motivation

Essentially all of condensed matter physics is described by the non-relativistic Schrödinger equation $i\hbar\frac{\partial}{\partial t}\left|\Psi\right\rangle = H\left|\Psi\right\rangle$, with the Hamiltonian

$$H = -\sum_{\alpha=1}^{N_n}\frac{\vec{P}_\alpha^2}{2M_\alpha} - \sum_{j=1}^{N_e}\frac{\vec{p}_j^{\,2}}{2m} - \sum_{j=1}^{N_e}\sum_{\alpha=1}^{N_n}\frac{Z_\alpha e^2}{|\vec{r}_j - \vec{R}_\alpha|} + \sum_{j<k}^{N_e}\frac{e^2}{|\vec{r}_j - \vec{r}_k|} + \sum_{\alpha<\beta}^{N_n}\frac{Z_\alpha Z_\beta e^2}{|\vec{R}_\alpha - \vec{R}_\beta|}$$

where Z_α is the atomic number, M_α the mass, $\vec{R}_\alpha$ the position and $\vec{P}_\alpha$ the momentum of nucleus α. $\vec{p}_j$ and $\vec{r}_j$ denote the j^{th} electron's momentum and position and N_e, N_n the number of electrons, nuclei respectively. To accurately describe materials of technological interest and design new ones with superior properties, all we have to do is solve this equation. There is, however, a severe problem which makes a brute-force approach to the many-body Schrödinger equation infeasible. To illustrate this, let us for consider a single iron atom. With its 26 electrons the total electronic wave function depends on 26 times 3 spatial coordinates. Thus, even without spin, specifying the electronic wave function on a hypercubic grid with merely 10 points per coordinate, we would have to store 10^{78} numbers. This is impossible in practice: Even if we could store a number in a single hydrogen atom, the required memory would weight 10^{51} kg – far more than our home-galaxy, the milky way.

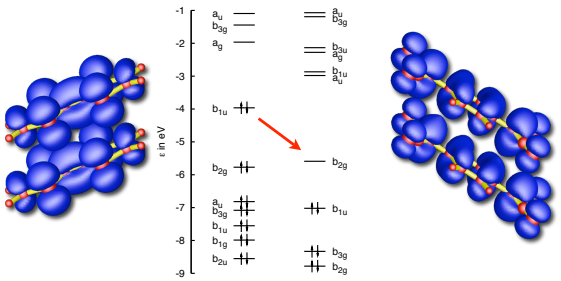


Figure 1. The molecular metal TTF-TCNQ. Centre: molecular levels of the isolated molecules; left: two TTF molecules with the electron density of their highest occupied molecular orbital (HOMO); right: TCNQ with the electron density of the lowest unoccupied molecular orbital (LUMO). The red arrow denotes the charge transfer of 0.6 electrons from the TTF-HOMO to the TCNQ-LUMO.

Still, the quantitative description of solids is not an entirely hopeless enterprise. Even though an exact treatment is a practical impossibility, there are successful approximations that work for wide classes of materials. The most prominent examples are approximations to density functional theory.[1] They effectively map the hard many-body problem to an effective single-particle problem that can be efficiently solved numerically. Essential to these approximations is that the Coulomb repulsion is described on a mean-field level. Such an approximation fails, however, to capture the physics in systems with strong correlations. In these systems the Coulomb repulsion between the electrons is so strong that the motion of a single electron depends on the position of all the others. The electrons thus lose their individuality and the single-electron picture breaks down. To accurately model this, we have to solve the many-electron problem exactly. Clearly we cannot do this for the full Hamiltonian. Instead, we consider a simplified Hamiltonian, which describes only those electrons that are essential to the correlation effects.[2] We illustrate this for the example of TTF-TCNQ, a quasi one-dimensional organic metal.

As shown in Fig. 1, TTF and TCNQ are stable molecules with completely filled molecular orbitals. The highest molecular orbital (HOMO) of TTF is, however, significantly higher in energy than the lowest unoccupied molecular orbtial (LUMO) of TCNQ. Thus in a crystal of TTF and TCNQ, charge is transferred from the TTF-HOMO to the TCNQ-LUMO. This leads to partially filled bands and thus metallic behaviour. In the TTF-TCNQ crystal, like molecules are stacked on top of each other. Electrons can move along these stacks, while hopping between different stacks is extremely weak. Thus the material is quasi one-dimensional.

As pointed out above, we cannot treat all the electrons in the molecular solid. Instead, we focus our efforts on the most important electronic states: the partially filled TTF-HOMO and TCNQ-LUMO. The effects of the other electrons are included by considering their screening effects.[3] The simplest model Hamiltonian which captures both effects, the itinerancy of the electrons as well as the strong Coulomb interaction is the Hubbard model

$$H = -\sum_{\sigma, i \neq j} t_{ij} c^{\dagger}_{i\sigma} c_{j\sigma} + U \sum_{i} n_{i\uparrow} n_{i\downarrow} \; . \tag{1.1}$$

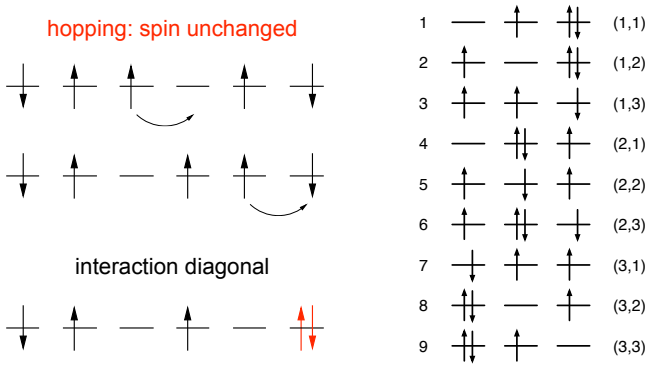


Figure 2. Illustration of the Hubbard model and the configuration basis of the Hilbert space. Upper left: next neighbour hopping (kinetic term); lower left: Coulomb repulsion U; right: basis for a three site system with two up and one down spin electron. The left label denotes the index of the configuration. Equivalently, a state is also unambiguously pointed to by a tuple of up- and down-configuration index.

The first term gives the kinetic energy, where t_{ij} is the amplitude for an electron to hop from the molecule at site i to lattice site j. Note that hopping does not change the spin σ. The second term represents the Coulomb repulsion between two electrons in the same molecular orbital. Both terms are illustrated in Fig. 2.

Since the Hamiltonian does neither change the number of electrons nor their spin, we need to consider (1.1) only on Hilbert spaces with a fixed number of electrons of spin up $N_\uparrow$ and spin down $N_\downarrow$. For a finite system of L orbitals there are $\binom{L}{N_\sigma}$ different ways to arrange N_σ electrons of spin σ. Thus the dimension of the Hilbert space is given by $\binom{L}{N_\uparrow} \cdot \binom{L}{N_\downarrow}$. For three orbitals, two up- and one down-spin electron, there are 9 different configurations shown in Fig. 2. Even though we significantly simplified the problem, we still have to face the many-body problem: increasing system size, the dimension of the Hilbert space increases steeply. A system with 20 orbitals and 10 electrons of either spin already contains more than 34 billion ($34\,134\,779\,536$) different configurations. Storing a single many-body state for this system takes about 254 GB.

2 Lanczos Method

We have seen that even for small systems the full Hamiltonian matrix is so large that it cannot be stored on a computer. Fortunately the matrix is sparse in real space, since each configuration is only connected to few others by hopping and the Coulomb repulsion part of the Hamiltonian is diagonal. To obtain the ground-state eigenvalue and -vector for our sparse Hamiltonian H we use the Lanczos method: We start from a random vector $|\phi_0\rangle$, leading to a first energy value $E_0 = E[\phi_0] = \langle\phi_0|H|\phi_0\rangle$. In analogy to the gradient descent method we look for the direction of steepest descent in the energy functional which is given by $H|\phi_0\rangle$. The resulting vector is orthogonalized with respect to the starting vector, i.e.

$$\langle\phi_1|H|\phi_0\rangle\,|\phi_1\rangle = H\,|\phi_0\rangle - \langle\phi_0|H|\phi_0\rangle\,|\phi_0\rangle \quad .$$

Now the same step is done with $|\phi_1\rangle$ leading to

$$\langle\phi_2|H|\phi_1\rangle\,|\phi_2\rangle = H\,|\phi_1\rangle - \langle\phi_1|H|\phi_1\rangle\,|\phi_1\rangle - \langle\phi_1|H|\phi_0\rangle\,|\phi_0\rangle \ ,$$

or in general,

$$\beta_{n+1}\,|\phi_{n+1}\rangle = H\,|\phi_n\rangle - \alpha_n\,|\phi_n\rangle - \beta_n\,|\phi_{n-1}\rangle \ ,$$

where $n \in 2, \ldots, m$ and

$$\alpha_n = \langle\phi_n|H|\phi_n\rangle\,, \quad \beta_{n+1} = \langle\phi_{n+1}|H|\phi_n\rangle\,.$$

From this equation we see that the Hamiltonian H is tridiagonal in the basis of the Lanczos vectors. In practice we need only of the order of 100 iterations to converge to the ground-state. If we are only interested in the ground-state energy, we only have to store two vectors of the size of the Hilbert space. In order to also get the ground state vector, a third vector is needed. As discussed above the dimension grows swiftly and thus the applicability of the method is limited by the maximum available main memory.

The Lanzcos method also provides a way to calculate Green functions, i.e.

$$G_k(\omega) = \sum_n \frac{\left|\left\langle \psi_n^{N\pm1}\middle|c_k^{(\dagger)}\middle|\psi_0\right\rangle\right|^2}{\omega \mp (E_n^{N\pm1} - E_0^N)} \ .$$

Having calculated the ground-state vector as described above we apply the $c_k^{(\dagger)}$ operator to it, normalize, and use the result as initial vector for a Lanczos iteration. The α_i and β_i then give the Green function

$$G_k(\omega) = \cfrac{\beta_0^2}{\omega - \alpha_0 - \cfrac{\beta_1^2}{\omega-\alpha_1-\cfrac{\beta_2^2}{\omega-\alpha_2-\cfrac{\beta_3^2}{\omega-\alpha_3-\cdots}}}} \ .$$

3 Computational Aspects

The key ingredient of the Lanczos algorithm described above is the sparse matrix vector multiplication. Already for quite small systems this operation takes most of the execution time, and increasing the size of the many-body vector it dominates ever more. Thus it will be the focus our parallelization efforts. On shared memory systems this matrix-vector multiplication is embarrassingly simple: The resulting vector elements can be calculated independently. Thus different threads can work on different chunks of this vector. The factor vector as well as the matrix elements are only read, so that there is no need for locking. An OpenMP parallelization thus needs only a single pragma. Similarly parallelizing also the scalar products, we obtain almost ideal speedup on an IBM p690 frame of JUMP in Jülich. The implementation is however limited to a single node. To use significantly more memory we need to find an efficient distributed memory implementation.

The kinetic energy term of the Hamiltonian (1.1) has non-diagonal terms and therefore leads to non-local memory access patterns. A naive distributed memory parallelization is to emulate a shared memory by direct remote memory access via MPI one-sided communication. This approach leads, however, to a severe speed-down, i.e. the more processors we use, the longer we have to wait for the result. This is shown in the inset of Fig. 4.

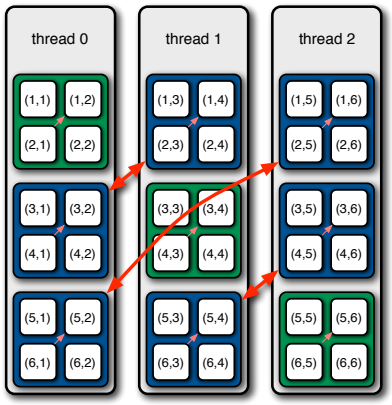


Figure 3. Transpose operation that makes memory access thread-local when calculating the operation of the Hamiltonian on the state-vector. The communication (red arrows) is realized by a call to `MPI_Alltoall`, which is very efficiently implemented on BlueGene/L. The small grey arrows indicate the local operations needed to complete the matrix-transpose.

To obtain an efficient distributed memory implementation we use a simple yet important observation: As pointed out above, the kinetic energy term conserves spin. Thus, performing the up-electron hopping takes only different up-hopping configurations into account while the down-electron configuration remains unchanged. If we group all up configurations for a fixed down configuration together in a single thread this hopping can thus be carried out locally. Fig. 2 illustrates this: for a fixed index $i_\downarrow$, all $i_\uparrow$ configurations follow and can be stored in a thread. We see, that this basis can be naturally indexed by a tuple $(i_\downarrow, i_\uparrow)$ (right labels in Fig. 2) instead of the global index (left labels). We can therefore equivalently regard the vectors as matrices $v(i_\downarrow, i_\uparrow)$ with indices $i_\downarrow$ and $i_\uparrow$. Now it is easy to see that a matrix transpose reshuffles the data elements such that the down configurations are sequentially in memory and local to the thread.

Therefore, the efficiency of the sparse matrix-vector multiplication rests on the performance of the matrix transpose operation. We implement it with `MPI_Alltoall`. This routine expects, however, the data packages which will be sent to a given process to be stored contiguously in memory. This does not apply to our case, since we would like to store the spin-down electron configurations sequentially in memory. Thus, the matrix is stored column wise. For `MPI_Alltoall` to work properly, we would have to bring the data elements in row-major order. This could be done by performing a local matrix transpose. The involved matrices are, however, in general rectangular, leading to expensive local-copy and reordering operations. We can avoid this by calling `MPI_Alltoall` for each column separately. After calling `MPI_Alltoall` for each column (red arrows in Fig. 3) only a local strided transposition has to be performed (small pink arrows) to obtain the fully transposed matrix or Lanczos vector.[4,5] The speed-up (left plot of Fig. 4) shows that collective communication is indeed very efficient, particularly on the BlueGene/L.

The implementation described so far uses `MPI_Alltoall` which assumes that the matrix to be transposed is a square matrix and that the dimension $dim_\uparrow = dim_\downarrow$ is divisible by the number of MPI processes. To overcome these restrictions we have generalized

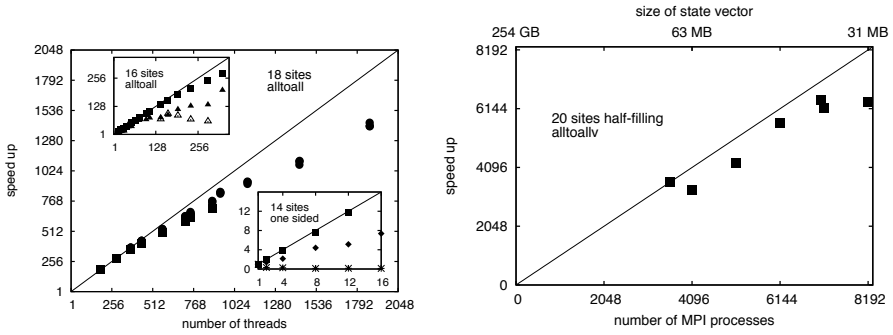


Figure 4. Speed-up of the Lanczos code. Left: IBM BlueGene/L JUBL (squares (CO mode) and circles (VN mode)) and IBM Regatta JUMP (triangles and diamonds: collective comm., stars: 1-sided comm.) for different problem sizes; right: speedup of the Lanczos code on IBM BlueGene/L JUBL in CO mode for 20 sites half-filled Hubbard model.

the algorithm to `MPI_Alltoallv`.[4] The right plot of Fig. 4 shows the speed-up of the generalized algorithm for a 20 sites system with half-filling (10 electrons of either spin). The corresponding size of the many-body vector is about 254 GB. The plot shows that we can efficiently use about 8192 threads and suggests that for larger system sizes we should be able to exploit even more processors.

4 Cluster Perturbation Theory and Spin-Charge Separation

Our parallel implementation of the Lanczos method enables us to efficiently calculate angular-resolved spectral functions for quite large systems. However, we still can have at most as many different momenta as we have sites. To resolve exciting physics like spin-charge separation we need, however, a much higher resolution. A way to achieve this is cluster perturbation theory (CPT).[6] The general idea is to solve a finite cluster with open boundary conditions exactly and then treat hopping between clusters in strong coupling perturbation theory, leading to an effective infinite chain.

We use the CPT technique in combination with the Lanczos method to study the quasi one-dimensional molecular metal TTF-TCNQ. Its low dimensionality in tandem with strong Coulomb repulsion compared to the kinetic energy leads to strong correlations and interesting many-body effects. Fig. 5 shows the angular-resolved spectral function for TCNQ in a CPT calculation for a 20 sites t-U Hubbard model. At the Γ-point we observe signatures of spin-charge separation: The electron dispersion splits into a holon and a spinon branch. To generate this plot about three BlueGene/L rack-days are needed: The calculation of the ground state is negligible and takes considerably less than half an hour on 2048 processors in VN mode on a BlueGene/L system. To calculate the Green's function for photoemission and inverse photoemission about 400 Green's functions each have to be calculated, where the former calculation takes a total of about 15 hours whereas the latter one takes about two days.

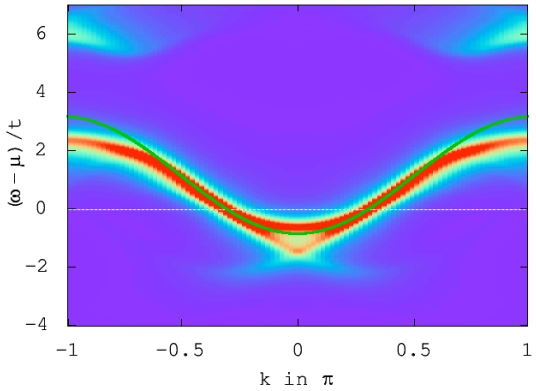


Figure 5. Angular-resolved spectral function obtained by CPT for a 20 sites TCNQ-like t-U Hubbard model with 6 electrons of either spin ($U = 1.96$eV, $t = 0.4$eV). The white line shows the chemical potential, the green cosine the independent-particle band. Signatures of spin-charge separation are clearly observed in the vicinity of the Γ-point.

5 Quantum Spins and Decoherence

A simplified version of the transposition scheme using `MPI_Alltoall` can be used to simulate quantum spin systems. We study the decoherence of two 1/2-spins (qubits) coupled to independent baths (a and b) of 1/2-spins. The Hamiltonian of this system is

$$H = \sum_{i \in \mathrm{BATH}_a} J_i \vec{s}_0 \vec{s}_i + \sum_{j \in \mathrm{BATH}_b} J_j \vec{s}_1 \vec{s}_j \ .$$

Since the Hilbert space of this system grows like 2^L where L is the total number of spins, we face similar problems as in the Lanczos method. In general, for a time-independent Hamiltonian, the time evolution operator is given by $U(t) = \exp(-iHt)$. Using a Suzuki-Trotter decomposition we can rewrite the operator as a product of time evolution operators for small time steps $U(t) \approx \prod^N U(\Delta t)$, where $t = N\Delta t$. We further decompose the time evolution operator such that we have a product of time evolution operators for two-spin systems, i.e. treating each term in the Hamiltonian separately, yielding

$$U(\Delta t) = \prod_{k=1}^{N} \exp(-iH_k \Delta t/2) \cdot \prod_{k=N}^{1} \exp(-iH_k \Delta t/2) \ + \mathcal{O}(\Delta t^3) \ ,$$

where N denotes the terms in the Hamiltonian and $\exp(-iH_k\Delta t/2)$ is given by

$$e^{iJ_k\Delta t/8} \begin{pmatrix} \exp(-iJ_k\Delta t/4) & 0 & 0 & 0 \\ 0 & \cos(J_k\Delta t/4) & -i\sin(J_k\Delta t/4) & 0 \\ 0 & -i\sin(J_k\Delta t/4) & \cos(J_k\Delta t/4) & 0 \\ 0 & 0 & 0 & \exp(-iJ_k\Delta t/4) \end{pmatrix} .$$

Thus we reduce calculating the exponential of the spin-Hamiltonian to simple matrix products. As before the matrix elements are not necessarily local to the threads, but can be made so by appropriate calls to `MPI_Alltoall`. To test this approach we have so far considered systems of up to 26 spins for which we obtain almost ideal speed-up for up to 1024

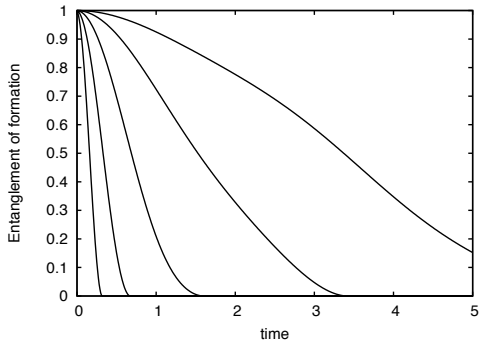


Figure 6. Entanglement-of-formation of a two qubit system, where each qubit couples to a bath of 10 spins. The coupling constants are chosen such that $\sum_i |J_i|^2 = C$ (see text). The curves from left to right show C = 1,1/2,1/4,1/8,1/16. The initial state for the qubits is a Bell state.

processors in VN mode. As a first application we study the sudden death of entanglement. We start with the two qubits in the Bell state $(|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle)/\sqrt{2}$ and a random, but unpolarized bath. The coupling constants are chosen randomly, but normalized to $\sum_i |J_i|^2 = C$. We find that the entanglement-of-formation[7] goes to zero after a finite time. As can be seen from Fig. 6, the time to this sudden-death of entanglement[8] is roughly proportional to the strength of the coupling to the bath C.

References

1. W. Kohn, *Nobel Lecture: Electronic structure of matter: wave functions and density functionals*, Rev. Mod. Phys., **71**, 1253, (1999).
2. E. Koch and E. Pavarini, *Multiple Scales in Solid State Physics*, Proceedings of the Summer School on Multiscale Modeling and Simulations in Science, (Springer,2007).
3. L. Cano-Cortés, A. Dolfen, J. Merino, J. Behler, B. Delley, K. Reuter and E. Koch, *Coulomb parameters and photoemission for the molecular metal TTF-TCNQ*, Eur. Phys. J. B, **56**, 173, (2007).
4. A. Dolfen, *Massively parallel exact diagonalization of strongly correlated systems*, Diploma Thesis, RWTH Aachen, (2006).
5. A. Dolfen, E. Pavarini and E. Koch, *New horizons for the realistic description of materials with strong correlations*, Innovatives Supercomputing in Deutschland, **4**, 16, (2006).
6. D. Sénéchal, D. Perez and M. Pioro-Ladrière, *Spectral weight of the Hubbard model through cluster perturbation theory*, Phys. Rev. Lett., **84**, 522, (2000).
7. W. K. Wooters, *Entanglement of formation of an arbitrary state of two qubits*, Phys. Rev. Lett., **80**, 2245, (1998).
8. M. P. Almeida, *et al. Environment-induced sudden death of entanglement*, Science, **316**, 579, (2007).

Massively Parallel Simulation of Cardiac Electrical Wave Propagation on Blue Gene

Jeffrey J. Fox[1], **Gregery T. Buzzard**[2], **Robert Miller**[1], **and Fernando Siso-Nadal**[1]

[1] Gene Network Sciences
53 Brown Rd, Ithaca, NY, USA
E-mail: {jeff, robert, siso}@gnsbiotech.com

[2] Department of Mathematics, Purdue University
W. Lafayette, Indiana, USA
E-mail: buzzard@math.purdue.edu

Heart rhythm disorders are a leading contributor to morbidity and mortality in the industrialized world. Treatment and prevention of cardiac rhythm disorders remains difficult because the electrical signal that controls the heart rhythm is determined by complex, multi-scale biological processes. Data-driven computer simulation is a promising tool for facilitating a better understanding of cardiac electrical properties. Conducting detailed, large-scale simulations of the cardiac electrical activity presents several challenges: the heart has a complicated 3D geometry, conduction of the electrical wave is anisotropic, and cardiac tissue is made up of cells with heterogeneous electrical properties. Our group has developed a software platform for conducting massively parallel simulations of wave propagation in cardiac tissue. The code is being used in an on-going study to connect drug-induced modifications of molecular properties of heart cells to changes in tissue properties that might lead to a rhythm disorder. The platform uses a finite difference method for modelling the propagation of electrical waves in cardiac tissue using the cable equation with homogeneous Neumann boundary conditions. We use a novel algorithm which is based on the phase field method for handling the boundary conditions in complicated geometries. To map grid points within the spatial domain to compute nodes, an optimization process is used to balance the trade-offs between load balancing and increased overhead for communication. The performance of the code has been studied by simulating wave propagation in an anatomically realistic model of the left and right ventricles of a rabbit heart on Blue Gene partitions of up to 4,096 processors. Based on these results, the Blue Gene architecture seems particularly suited for cardiac simulation, and offers a promising platform for rapidly exploring cardiac electrical wave dynamics in large spatial domains.

1 Introduction

The primary motivation for developing a detailed 3D model of the electrical activity of the heart is to create a tool that can be used to help identify the underlying electrical mechanisms for dangerous ventricular arrhythmias and to determine the effects of interventions, such as drugs, that may prevent or exacerbate these arrhythmias. High-performance computing systems such as the Blue Gene are a promising resource for conducting the large-scale simulations required for elucidating how molecular modifications to heart cells might lead to dangerous alterations to electrical wave propagation in the heart.

Our group has developed a software platform for conducting massively parallel simulations of wave propagation in cardiac tissue. The motivation for this work is discussed in Section 2. The mathematical framework that the platform uses is described in Section 3. Next, several aspects of the code are outlined in Section 4, including methods for paral-

lelization. Finally, the performance of the code on up to 4096 processors is studied in Section 5.

2 Motivation

The electrical signal that controls the mechanical activity of the heart originates in the sinus node. This signal travels down a specialized conduction system and then propagates through the ventricles, the main pumping chambers of the heart. At the cellular level, the electrical signal is a steep rise and then gradual fall in the cellular membrane potential called the action potential (AP). This electrical activity is manifest on the surface of the body in the form of the electrocardiogram (ECG). The QT interval, the time between the "QRS" complex and the "T" wave in the ECG, roughly corresponds to the period of time between the upstroke of the AP and re-polarization.

Heart rhythm disorders, or arrhythmias, refer to a disruption of the normal electrical signal. Catastrophic rhythm disturbances of the heart are a leading cause of death in many developed countries. For example, one type of rhythm disorder called ventricular fibrillation results in about 350K deaths per year in the US alone. In addition, cardiac electrical activity can be unintentionally disrupted by drugs. Unfortunately this problem does not only occur in compounds designed to treat the heart; it can happen in any therapeutic area. Through a number of studies scientists have identified a link between drugs that block certain ion currents (particularly I_{Kr}, the rapid component of the delayed rectifier potassium current), that prolong the QT interval in the ECG, and a deadly type of arrhythmia called Torsades de Pointes. Yet, it remains difficult to determine what specific modifications in the function of these proteins can lead to the initiation of a complex and life-threatening cardiac electrical disturbance. For example, some I_{Kr} inhibitors are widely prescribed and safely used. Thus, the mechanism by which a modification in the functional properties of one or more ion channels might lead to the induction of a dangerous ventricular arrhythmia is still poorly understood.

3 Mathematical Framework

A standard model[1] for the dynamics of the membrane potential, V (measured in mV), in cardiac tissue is given by the continuous cable equation:

$$\frac{\partial V}{\partial t} = \nabla \cdot D \nabla V - \frac{1}{C_{mem}} \sum I_{ion} \tag{3.1}$$

C_{mem} is the membrane capacitance (measured in pF/cm^2), which is usually taken to be 1, I_{ion} is the sum of all ionic currents (measured in pA/cm^2), ∇ is the 3D gradient operator, and D is the effective diffusion coefficient (measured in cm^2/s), which is in general a 3x3 matrix to allow for anisotropic coupling between cells. The equations for the ionic currents are ordinary differential equations (ODEs) and contain the details of the AP models. For isolated tissue, the current flow must be 0 in directions normal to the tissue boundary. This zero-flux condition is represented by the condition $n \cdot (D \nabla V) = 0$, where n is the unit vector normal to the heart surface, which implies that in the absence of the I_{ion} term, voltage is conserved. Fenton, et al[2] provide a method of deriving D from knowledge of

the fibre direction at each point in the heart. The phase field method[3] can be used to deal with irregular geometries characteristic of cardiac anatomy. In this method ϕ is a smooth function so that the region of interest (the ventricle), R, is given by $R = \phi > 0$ so that $\phi = 0$ outside of R and $\phi = 1$ on a large portion of R. The region $0 < \phi < 1$ is called the transition layer and should be thought of as a thin layer on the inside of the boundary of R. Then the modified equation is

$$\phi \frac{\partial V}{\partial t} = \nabla \cdot (\phi D \nabla V) - \phi \frac{1}{C_{mem}} \sum I_{ion}. \tag{3.2}$$

It can be shown that voltage is conserved in the limit as the transition layer width tends to 0. However, there are still some delicate issues in terms of implementation. In order to recover exactly the voltage conservation property of the cable equation with homogeneous Neumann conditions, we developed a finite-difference algorithm[4] by first discretizing the phase-field modified cable equation on a regular grid, then taking a limit as the width of the phase field tends to 0. This is discussed further in the next section.

4 Overview of the Software

The platform uses a finite difference method for modelling the propagation of electrical waves in cardiac tissue using the cable equation (a reaction-diffusion type partial differential equation, or PDE) with homogeneous Neumann boundary conditions. We use a novel algorithm that is based on a limiting case of the phase field method for handling the boundary conditions in complicated geometries. To map grid points within the spatial domain to compute nodes, an optimization process is used to balance the trade-offs between load balancing and increased overhead for communication. The code is written in C++, and uses several standard libraries, including SUNDIALS, MPI, BOOST, BLAS/LAPACK, and SUPERLU.

4.1 Numerical Methods

Our software uses a novel finite difference scheme for evaluating $\nabla \cdot (\phi D \nabla V)$ on a regular grid in 2 or 3D[4]. The scheme is obtained as a limiting case of the phase field method using a sharp transition layer given by $\phi = 1$ in R and $\phi = 0$ outside of R. As a result, this scheme gives exact voltage conservation over the cells in R. This scheme is motivated by the formula for summation by parts that is the analog of integration by parts[5]. We assume that the problem lies on a regular grid containing a bounded region of interest, R, with grid points $p = (x_1, x_2, x_3)$ (the 2D case will be obtained simply by ignoring the third dimension). The grid space in the i^{th} coordinate is Δx_i, and e_i represents the vector that is Δx_i in the i^{th} coordinate and 0 in the other coordinates. Given a function f on the grid

and using $V_t(p)$ to denote the change in $V(p)$ during a single time step, define:

$$\Delta_i f(p) = \frac{f(p+e_i) - f(p)}{\Delta x_i} \tag{4.1}$$

$$\bar{\Delta}_i f(p) = \frac{f(p+e_i) - f(p-e_i)}{2\Delta x_i} \tag{4.2}$$

$$\Delta^i_j f(p) = \begin{cases} \Delta_i f(p) & \text{if } i = j \\ \frac{1}{2}(\bar{\Delta}_j f(p) + \bar{\Delta}_j f(p+e_i)) & \text{if } i \neq j \end{cases} \tag{4.3}$$

$$\Delta_{ij} f(p) = \begin{cases} \Delta_i \Delta_i f(p-e_i) & \text{if } i = j \\ \bar{\Delta}_i \bar{\Delta}_j f(p) & \text{if } i \neq j. \end{cases} \tag{4.4}$$

Given this definition, we propose that the discrete analog of the spatial part of the modified equation cable equation is:

$$\begin{aligned} \phi(p)V_t(p) = \sum_{i,j=1}^{3} [&\frac{\phi(p)}{2}(\Delta_i D_{ij}(p)\Delta^i_j V(p) + \Delta_i D_{ij}(p-e_i)\Delta^i_j V(p-e_i)) \\ &+ \frac{D_{ij}(p)}{2}(\Delta_i \phi(p)\Delta^i_j V(p) + \Delta_i \phi(p-e_i)\Delta^i_j V(p-e_i)) \\ &+ \phi(p)D_{ij}(p)\Delta_{ij}V(p))]. \end{aligned} \tag{4.5}$$

By taking a limit as the phase field width tends to 0, we obtain what we call the sharp interface limit. In this case, either $\phi(p) = 0$ or $\phi(p) = 1$. When $\phi(p) = 1$, the expression above defines $V_t(p)$ as a weighted sum of the values of V at the neighbors of p. When $\phi(p) = 0$, the expression is interpreted by multiplying both sides by $\phi(p)$ and requiring that the remaining sum on the right be 0. In this case, the first and third lines drop out due to $\phi(p) = 0$, so we are left with an equation determined by setting the remaining part of the sum to 0. This gives a system of linear equations corresponding to the Neumann boundary conditions. Solving this system, which need be done only once for a given (nonmoving) boundary, produces a scheme that gives exact voltage conservation (up to roundoff error). This allows simulation of complex geometries without the need to include a transition layer that may introduce numerical artifacts. Employing this algorithm has several additional advantages. First, the implementation of this method is straightforward. Second, it enables the use of a simple finite difference scheme on a regular cubic grid without the need for a finite element representation or for additional computation of boundary values. This ensures that the complexity of the numerical scheme does not increase substantially and also facilitates parallelization of the code. Moreover, the outer layer of the ventricle, known as the epicardial layer, may be on the order of 1 mm. Since cell heterogeneity plays an important role in arrhythmias[6], an accurate representation of tissue requires reducing or eliminating the transition layer. Additionally, from a computational point of view, a wider transition layer is very costly. In a model of rabbit ventricles, with a grid spacing of 0.25 mm, the phase field method with a transition layer of 4 grid points requires 742,790 copies of the ionic model, versus 470,196 for the method described here, a reduction of nearly 40 percent. The bulk of simulation time is devoted to the ionic terms, so this corresponds roughly to the same reduction in simulation time. Third, an important benefit of choosing this method over the use of finite elements is that the 3D code is essentially the same for all geometries considered, so that only the input file containing the anatomic structure and

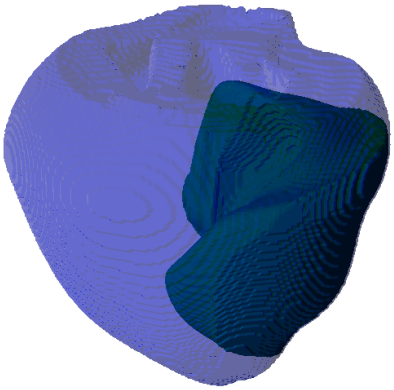


Figure 1. Visualization of the spreading activation wave front from a simulation of wave propagation in a model of the rabbit left and right ventricles using the Fox et al AP model.

conductivity tensor changes from one geometry to the next, thereby avoiding the complex and time-consuming task of 3D mesh generation. Each grid point in the 3D model can be assigned a different ionic model, thus allowing the specification of the locations of epicardial, endocardial, and M cells. This flexibility also allows the effect of variations in the distribution of these cell types on global wave propagation to be studied.

To integrate the discretized cable equation, we use one of two methods. The simplest method, and that used in the scaling study below, is operator splitting. For this method we solve the ionic term for a fixed time interval using CVODE and then solve the diffusion term using the Euler method. More recently we have investigated the method of lines, which performs significantly better (2 to 10 times speed up depending on the problem). For this method we produce a coupled system of ODEs, then integrate with the stiff solver in the SUNDIALS package (from the CVODE subset of http://acts.nersc.gov/sundials/ using backwards differentiation). Each grid point corresponds to a separate set of equations given by the ionic terms plus the terms from spatial coupling as described above. We use Krylov subspace preconditioning with a diagonal approximation to the Jacobian. To define the fibre directions at boundary points, we start with fibre directions at points in R, then determine the fibre direction at p in the boundary, $B(R)$, by taking an average of the directions of nearest neighbors. In the simplified case described above, we then project this onto the nearest coordinate axis (or select one axis in the case of a tie). Figure 1 shows an example of wave propagation in a model of the rabbit ventricle using the sharp interface method.

4.2 Parallelization

We use MPI for parallelization of the code. Parallelization of the 3D simulation requires almost exclusively nearest neighbour communication between compute nodes. Specifically, each node is assigned a portion of the 3D spatial domain, and the nodes pass information

about the boundary of the domain to nodes that are responsible for neighboring domains. In order to create simulations to match experimental canine ventricular wedge preparations (these preparations are typically $2 \times 1.5 \times 1$ to $3 \times 2 \times 1.5$ cm^3 in size), 192,000 to 576,000 grid points will be required. On the other hand, a typical human heart has dimensions on the order of 10 cm, so directly simulating wave propagation in the human ventricle, a primary goal of electrophysiological modelling, requires 10 to 100 million grid points. Since the bulk of computation time is used on evaluation of the ionic terms and local integration and since communication between grid points is nearest neighbour in form and minimal in size, this problem is well suited to a parallel implementation using a large number of processors. Nevertheless, there are many interesting problems about how best to distribute the cells to the processors to maximize speed; improving load balancing for a large number of processors is an ongoing project. Load balancing is a particularly important issue to consider when using spatial domains that are anatomically realistic. In this case, because the problem is defined on a regular grid, only a portion of the grid nodes correspond to active tissue; the rest are empty space. For example, an anatomical model of a rabbit ventricular wedge uses a grid that is $119 \times 109 \times 11$, so there are 142,681 nodes, but only 42,457 are active. All 142,681 are assigned to tasks, so some tasks get more active nodes than others. To map nodes to tasks, an optimization process is used to balance the trade-offs between load balancing and increased overhead for communication.

To limit the complexity of the message-passing scheme, we consider the tissue as embedded in a larger rectangular domain (a fictitious domain in the language of numerical PDE) and subdivide this rectangular domain into a product grid. This results in a collection of disjoint boxes which together cover the fictitious domain. E.g., if the fictitious domain has dimensions 110 by 150 by 120, the product domain might consist of boxes of size 10 by 5 by 6. A single processor may be assigned more than one box in the resulting decomposition. In the current implementation of the algorithm, the size of the boxes in the product grid is the same for all the boxes and is chosen using an optimization algorithm in order to distribute the number of active nodes nearly evenly among all the processors while at the same time limiting the amount of message passing that is required among processors. In particular, the boxes are assigned to processors in such a way that nearest neighbour boxes are assigned to nearest neighbour processors in order to take advantage of the torus topology of the communications network. In most cases of interest, this algorithm distributes the active nodes to within 1 percent of the optimal distribution. Moreover, given the data to determine the active nodes and the size of the network, the optimization algorithm can be run in advance of the simulation to produce the correct division of labour. Further improvements in this algorithm are possible.

5 Parallel Performance

We have carried out a number of simulations on the Blue Gene/L system to illustrate the parallel performance of the cardiac simulation application. We simulated wave propagation in a 3D model of the rabbit ventricular anatomy[8] using the Fox et al AP model[7]. We conducted two different comparisons. First, we compared performance between two different sized spatial domains using the same number of compute nodes (32 nodes, 64 processors). We compared the time to simulate 10 msec of wave propagation in the full rabbit ventricle versus a wedge of the ventricle. The results are shown in the table be-

low. The improved performance of the code on the rabbit ventricle is likely explained by

Tissue	Total grid pts.	Active	% diff. from opt.	Run time (sec)	% int.
wedge	142,000	42,000	13	80	67
ventricle	2,846,000	470,000	3	659	93

looking at the column "% diff. from opt.". This quantity measures the percent difference between the optimal number of spatial grid points assigned to each compute node and the actual number of grid points assigned to the "most loaded" compute node. Ideally, each compute node would be responsible for an identical number of spatial grid points. However, the task assignment process must strike a balance between achieving optimal division of labour and minimizing communication time. Most likely, the simple explanation as to why the full ventricle has a more homogeneous load is the fact that it has more nodes than the wedge so percent differences will be smaller. In addition, it may be that specific geometrical features play a role.

The second comparison is a strong scaling performance example. We compared simulation times using the same anatomy as a function of number of processors for 10 msec of wave propagation. The table below shows results for the full rabbit ventricle. Figure 2

Nproc	Run time (sec)	Ideal (sec)	% int.	Speed up	Ideal speed up
32	1203	1203	93	32	32
64	620	602	93	62.08	64
128	316	301	88	121.92	128
256	166	150	85	232	256
512	87	75	82	442.56	512
1024	49	38	77	785.6	1024
4096	18	9	54	2138.56	4096

shows how the time to solution as a function of processor number compares to an ideal scaling. The performance of the code is quite good for the partition sizes studied here.

6 Concluding Remarks

To conclude, large scale resources like Blue Gene allow researchers to explore qualitatively different problems. For the example application described above, conducting a dose-response simulation of the effect of a compound on a 3D canine ventricle at a variety of pacing frequencies would require many months of simulation time on a standard computer cluster, but with Blue Gene one could conduct several such simulations in a day. Thus, using the software described in this study on Blue Gene is a promising platform for conducting multi-scale biomedical simulations.

Acknowledgements

This work was supported by NIH grants R44HL077938 and R01HL075515. We are grateful to IBM for providing computer time for the parallel performance studies.

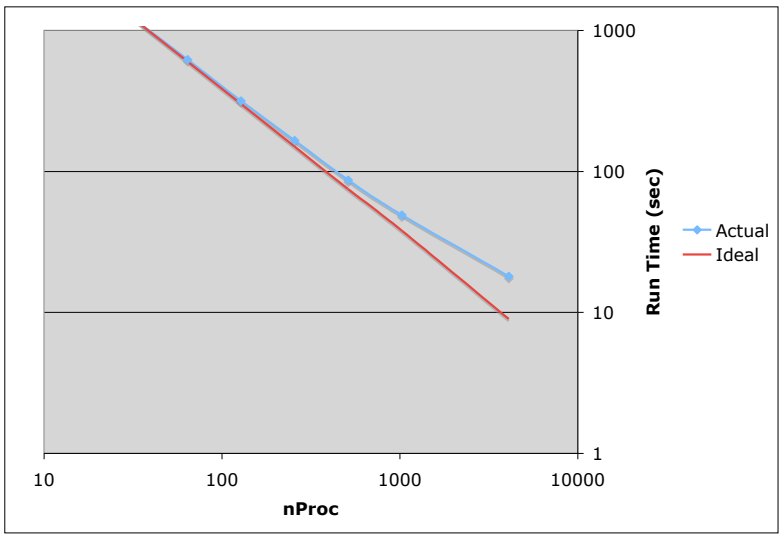


Figure 2. Run time as a function of processor number. The diamond points are from scaling runs on the Blue Gene/L system, and the solid line is the ideal run time assuming perfect scaling.

References

1. *Special issue on cardiac modeling*, Chaos, **8**, (1998).
2. F. H. Fenton, E. M. Cherry, A. Karma and W. J. Rappel, *Modeling wave propagation in realistic heart geometries using the phase-field method*, Chaos, **15**, 013502, (2005).
3. A. Karma and W. J. Rappel, *Numerical simulation of three-dimensional dendritic growth*, Phys Rev Lett, **77**, 450–453, (1996).
4. G. T. Buzzard, J. J. Fox and F. Siso-Nadal, *Sharp interface and voltage conservation in the phase field method: application to cardiac electrophysiology*, SIAM J. Sci Comp., (2007, in press).
5. R. L. Graham, D. E. Knuth and O. Patashnik, *Concrete Mathematics*, (Addison-Wesley, New York, 1989).
6. C. Antzelevitch and W. Shimizu, *Cellular mechanisms underlying the long QT syndrome*, Curr. Opin. Cardiol., **17**, 43–51, (2002).
7. J. J. Fox, J. L. McHarg and R. F. Gilmour, Jr., *Ionic mechanism of electrical alternans*, Amer. J. Physiol., **282**, H516–H530, (2002).
8. F. J. Vetter and A. D. McCulloch, *Three-dimensional analysis of regional cardiac function: a model of rabbit ventricular anatomy*, Progress Biophys. Mol. Bio., **69**, 157–183, (1998).

Mini-Symposium

“Scalability and Usability of HPC Programming Tools”

Scalability and Usability of HPC Programming Tools

Felix Wolf[1], **Daniel Becker**[1], **Bettina Krammer**[2], **Dieter an Mey**[3], **Shirley Moore**[4], **and Matthias S. Müller**[5]

[1] Jülich Supercomputing Centre
Forschungszentrum Jülich, 52425 Jülich, Germany
E-mail: {f.wolf, d.becker}@fz-juelich.de

[2] HLRS - High Performance Computing Center Stuttgart
Nobelstrasse 19, 70569 Stuttgart, Germany
E-mail: krammer@hlrs.de

[3] RWTH Aachen University, Center for Computing and Communication
Seffenter Weg 23, 52074 Aachen, Germany
E-mail: anmey@rz.rwth-aachen.de

[4] University of Tennessee
Innovative Computing Laboratory, Knoxville, TN 37996-3450, USA
E-mail: shirley@cs.utk.edu

[5] ZIH (Center for Information Services and HPC)
Technische Universität Dresden, 01162 Dresden, Germany
E-mail: matthias.mueller@tu-dresden.de

Facing increasing power dissipation and little instruction-level parallelism left to exploit, computer architects are realizing further performance gains by placing multiple "slower" processor cores on a chip rather than by building faster uni-processors. As a consequence, numerical simulations are being required to harness much higher degrees of parallelism in order to satisfy their growing demand for computing power. However, writing code that runs correctly and efficiently on large numbers of processors and cores is extraordinarily challenging and requires adequate tool support for debugging and performance analysis. Unfortunately, increased concurrency levels place higher scalability demands not only on applications but also on software tools. When applied to larger numbers of processors, familiar tools often cease to work in a satisfactory manner (e.g., due to escalating memory requirements or limited I/O bandwidth). In addition, tool usage is often complicated and requires the user to learn a separate interface. This minisymposium served as a forum to discuss methods and experiences related to scalability and usability of HPC programming tools, including their integration with compilers and the overall development environment.

Presentations

Benchmarking the Stack Trace Analysis Tool for BlueGene/L by Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Barton P. Miller and Martin Schulz presents a novel approach to emulating tool scaling behaviour using minimal resources and demonstrates how this technique was used to identify and to fix a scalability bug in the Stack Trace Analysis Tool.

Scalable, Automated Parallel Analysis with TAU and PerfExplorer by Kevin A. Huck, Allen D. Malony, Sameer Shende and Alan Morris presents the latest design changes to

PerfExplorer, a scalable parallel performance analysis framework which integrates performance data, metadata and expert knowledge to analyze large parallel application performance profiles.

Developing Scalable Applications with Vampir, VampirServer and VampirTrace by Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel presents scalability studies of Vampir and VampirTrace using the SPEC MPI 1.0 benchmarks. It covers instrumentation, tracing and visual trace analysis and investigates data volumes, overhead and visualization performance.

Scalable Collation and Presentation of Call-Path Profile Data with CUBE by Markus Geimer, Björn Kuhlmann, Farzona Pulatova, Felix Wolf, and Brian J.N. Wylie explains how the scalability of CUBE, a presentation component for runtime summaries and trace-analysis results in the SCALASCA toolkit, has been improved to more efficiently handle data sets from very large numbers of processes.

Coupling DDT and Marmot for Debugging of MPI Applications by Bettina Krammer, Valentin Himmler, and David Lecomber describes first steps towards combining those two tools – a parallel debugger and an MPI correctness checker – to provide application developers with a powerful and user-friendly environment that covers different aspects of correctness debugging.

Compiler Support for Efficient Instrumentation by Oscar Hernandez, Haoqiang Jin, and Barbara Chapman presents the benefits of using compiler analyses to provide an automated, scalable instrumentation infrastructure that can direct performance tools which regions of code to measure. Our approach has shown that it can significantly reduce instrumentation overheads to acceptable levels.

Comparing Intel Thread Checker and Sun Thread Analyzer by Christian Terboven compares two software tools that help the programmer in finding errors like data races and deadlocks. Experiences using both tools on multithreaded applications will be presented together with findings on the strengths and limitations of each product.

Continuous Runtime Profiling of OpenMP Applications by Karl Fürlinger and Shirley Moore investigates a technique to combine the advantages of tracing and profiling in order to limit data volume to enable manual interpretation, while retaining temporal information about the program execution characteristics of an application.

Understanding Memory Access Bottlenecks on Multicore by Josef Weidendorfer focuses on memory issues found both on single- and multi-core processors. The talk proposes metrics got via simulation as well as appropriate visualizations for deeper understanding of given issues, and guiding at code changes to avoid them.

Benchmarking the Stack Trace Analysis Tool for BlueGene/L

Gregory L. Lee[1], **Dong H. Ahn**[1], **Dorian C. Arnold**[2], **Bronis R. de Supinski**[1], **Barton P. Miller**[2], **and Martin Schulz**[1]

[1] Computation Directorate
Lawrence Livermore National Laboratory, Livermore, California, U.S.A.
E-mail: {lee218, ahn1, bronis, schulzm}@llnl.gov

[2] Computer Sciences Department
University of Wisconsin, Madison, Wisconsin, U.S.A.
E-mail: {darnold, bart}@cs.wisc.edu

We present STATBench, an emulator of a scalable, lightweight, and effective tool to help debug extreme-scale parallel applications, the Stack Trace Analysis Tool (STAT). STAT periodically samples stack traces from application processes and organizes the samples into a call graph prefix tree that depicts process equivalence classes based on trace similarities. We have developed STATBench which only requires limited resources and yet allows us to evaluate the feasibility of and identify potential roadblocks to deploying STAT on entire large scale systems like the 131,072 processor BlueGene/L (BG/L) at Lawrence Livermore National Laboratory.

In this paper, we describe the implementation of STATBench and show how our design strategy is generally useful for emulating tool scaling behaviour. We validate STATBench's emulation of STAT by comparing execution results from STATBench with previously collected data from STAT on the same platform. We then use STATBench to emulate STAT on configurations up to the full BG/L system size – at this scale, STATBench predicts latencies below three seconds.

1 Introduction

Development of applications and tools for large scale systems is often hindered by the availability of those systems. Typically, the systems are oversubscribed and it is difficult to perform the tests needed to understand and to improve performance at large scales. This problem is particularly acute for tool development: even if the tools can be run quickly, they do not directly produce the science for which the systems are purchased. Thus, we have a critical need for strategies to predict and to optimize large scale tools based on smaller scale tests. For this reason, we have developed STATBench, an innovative emulator of STAT, the Stack Trace Analysis Tool[4].

STAT is a simple tool to help debug large scale parallel programs. It gathers and merges multiple stack traces across space, one from each of a parallel application's processes, and across time through periodic samples from each process. The resulting output is useful in characterizing the application's global behaviour over time. The key goal for STAT is to provide basic debugging information efficiently at the largest scale.

From its inception, STAT was designed with scalability in mind, and was targeted at machines such as Lawrence Livermore National Laboratory's (LLNL's) BlueGene/L (BG/L), which employs 131,072 processor cores. Existing tools have come far short of

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (UCRL-CONF-235241).

debugging at such scale. For example, our measurements show that TotalView[1], arguably the best parallel debugger, takes over two minutes to collect and to merge stack traces from just 4096 application processes. STAT, a lightweight tool that employs a tree-based overlay network (TBON), merges stack traces at similar scale on Thunder, an Intel Itanium2 cluster at LLNL, in less than a second. However, gathering the data to demonstrate this scalability was an arduous task that required significant developer time to obtain the system at this large of a scale. Gaining time on the full BG/L system is even more difficult.

As an intermediate point in STAT's evolution and to determine the feasibility of running STAT at full scale on BG/L, we designed STATBench. STATBench allows specification of various parameters that model a machine's architecture and an application's profile. This emulation runs on the actual system but generates artificial application process stack traces, which allows the emulation of many more application processes than the actual processor count of the benchmark run. Our design is general: any TBON-based tool could employ a similar emulation strategy to understand scaling issues. Our STATBench results demonstrate that STAT will scale well to the full BG/L size of 131,072 processors.

In the remainder of this paper, we present STATBench and explain how it accurately models the scalability of STAT. In the following section, we first review the design of STAT and discuss why this type of tool is needed for large scale debugging. We then present the design of STATBench and discuss how it validates our large scale debugging strategy and can help to identify roadblocks before final implementation and deployment in Section 3. Section 4 then compares results from STATBench to observed performance of STAT at moderate scales and presents results from scaling studies using STATBench as well as improvements for STAT derived from those results. Finally, we discuss how to generalize this approach to support optimization of any TBON-based tool in Section 5.

2 The Stack Trace Analysis Tool

The Stack Trace Analysis Tool[4] (STAT) is part of an overall debugging strategy for extreme-scale applications. Those familiar with using large-scale systems commonly experience that some program errors only show up beyond certain scales and that errors may be non-deterministic and difficult to reproduce. Using STAT, we have shown that stack traces can provide useful insight and that STAT overcomes the performance and functionality deficiencies that prevent previous tools from running effectively at those scales. Namely, STAT provides a scalable, lightweight approach for collecting, analyzing, and rendering the spatial and temporal information that can reduce the problem search space to a manageable subset of processes.

STAT identifies process equivalence classes, groups of processes that exhibit similar behaviour, by sampling stack traces over time from each task of the parallel application. STAT processes the samples, which profile the application's behaviour, to form a call graph prefix tree that intuitively represents the application's behaviour classes over space and time. An example call graph prefix tree can be seen in Fig. 1. Users can then apply full-featured debuggers to representatives from these behaviour classes, which capture a reduced exploration space, for root cause problem analysis.

STAT is comprised of three main components: the front-end, the tool daemons, and the stack trace analysis routine. The front-end controls the collection of stack trace samples by the tool daemons, and our stack trace analysis routine processes the collected traces.

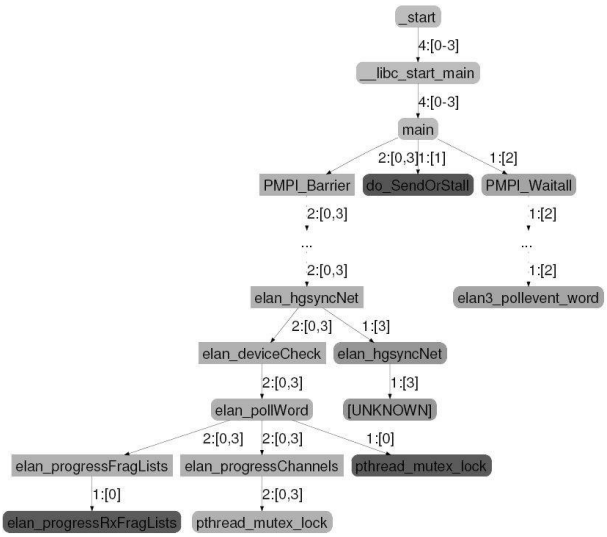


Figure 1. An example 3D-Trace/Space/Time call graph prefix tree from STAT.

The front-end also colour-codes and renders the resulting call graph prefix tree. The STAT front-end and back-ends use an MRNet[3] process tree for scalable communication and MRNet filters for efficiently executing the stack trace analysis routines, which parent nodes execute on input from children nodes. Back-ends merge local samples, which the TBŌN combines into a whole-program view. STAT uses the DynInst[2] library for lightweight, third-party stack tracing of unmodified applications.

3 Benchmark Implementation

STATBench inherits much of its implementation from STAT itself, using MRNet for scalable communication between the STATBench front-end and back-ends. STATBench also uses the same trace analysis algorithm as STAT to merge multiple traces into a single call prefix tree. The back-ends use this algorithm to perform local analysis and send the result through the MRNet tree to the front-end. The same algorithm is used in the MRNet filter function that is executed by the communication processes as the traces propagate through the communication tree.

STATBench does not actually gather traces from an application. Rather, the back-ends generate randomized traces controlled by several customizable parameters. These parameters include the maximum call breadth and the maximum call depth. The maximum breadth determines how many functions the generator can choose from, while the depth determines the number of function calls to generate for a given trace. STATBench prepends *__libc_start_main* and *main* function calls to the generated traces. STATBench also has a pair of parameters to control the spatial and temporal aspects of the trace generation. The spatial parameter governs how many processes each back-end emulates, equivalent to the number of processes a STAT daemon must debug. For the temporal aspect, STATBench also inputs the number of traces to generate per emulated process. With just these parameters, the generated traces can vary widely across tasks, which after spatial and temporal

merging yields a call prefix tree with a higher branching factor than is characteristic of real parallel applications. STATBench controls this variation through a parameter that specifies the number of equivalence classes to generate. Specifically, this parameter uses the task rank *modulo* the number of equivalence classes to seed the random number generator. Thus, all of the tasks within an equivalence class generate the exact same set of stack traces. These equivalence classes do not capture the hierarchical nature of the equivalence classes of real applications. However, our results, shown in Section 4, indicate that this simplification does not impact emulated performance significantly. With all of the possible parameters, STATBench can generate traces for a wide range of application profiles and can emulate various parallel computer architectures.

There is one main difference that allows STATBench to emulate larger scales than STAT, even given the same compute resources. This gain comes from the ability to utilize all of the processors on a Symmetric Multiprocessor (SMP) node's processing cores. Take, for example, a parallel machine with n-way SMP nodes, for some number n. With STAT, a single daemon is launched on each compute node and is responsible for gathering traces from all n of the application processes running on that node. STATBench, on the other hand, can launch its daemon emulators on all n of the SMP node's processing cores and have each daemon emulator generate n traces. The net effect is that STATBench only requires $1/n$ of the machine to emulate a STAT run on the full machine. Alternatively, we can use the full machine to emulate the scale of a machine that has n times as many compute nodes. The daemon emulators can also generate an arbitrary amount of traces, thus providing further insight for machines with more processing cores per compute node than the machine running STATBench.

4 Results

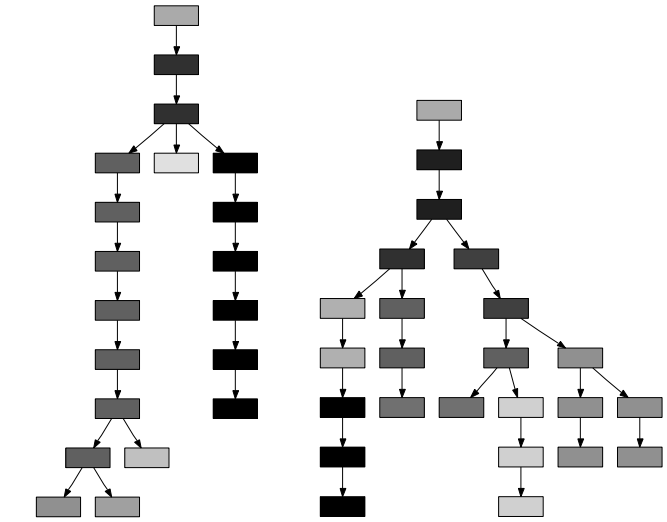
We first evaluate STATBench by comparing its performance to our STAT prototype. Next, we present the results from scaling STATBench to BG/L scales. Finally, we use those results to guide optimization of the tool's scalability.

4.1 STAT Benchmark vs. STAT

In order to evaluate STATBench's ability to model the performance of STAT, we compare results previously gathered from STAT with STATBench results. Both sets of results were gathered on Thunder, a 1024 node cluster at LLNL with four 1.4 GHz Intel Itanium2 CPUs and 8 GB of memory per node. The nodes are connected by a Quadrics QsNet^{II} interconnect with a Quadrics Elan 4 network processor.

The results from STAT were gathered from debugging an MPI ring communication program with an artificially injected bug. In this application, each MPI task performs an asynchronous receive from its predecessor in the ring and an asynchronous send to its successor, followed by an MPI_Waitall that blocks pending the completion of those requests. All of the tasks then synchronize at an MPI_Barrier. A bug is introduced that causes one process to hang before its send. An example 3D-Trace/Space/Time call graph prefix tree for this program, with node and edge labels removed, can be seen in Fig. 2(a)

STATBench was run with a set of parameters designed to match the STAT output from the MPI message ring program, particularly with respect to the depth, breadth, and node



(a) STAT (b) STATBench

Figure 2. Example structure (node and edge labels removed) of 3D-Trace/Space/Time call prefix trees from (a) STAT and (b) STATBench.

count of the call prefix tree. To achieve this goal, we specify three traces per simulated task, a maximum call depth past *main()* of seven, a call breadth of two, and five equivalence classes. An example output can be seen in Fig. 2(b). While this output does not exactly model STAT's output with the ring program, it does provide an approximation that is conservative; it is slightly more complex than the real application with respect to call graph topology and node count.

More than the visual comparison, our goal was to model the performance of STAT. We ran STATBench at various scales, using the same MRNet topologies that were used for STAT. Specifically, we employ a 2-deep tree, with one layer of communication processes between the STATBench front-end and back-ends. At all scales, we employ a balanced tree: all parent processes have the same number of children. Furthermore, on Thunder each STAT daemon gathered traces from four application processes, hence STATBench emulates four processes per daemon. Figure 3 shows that STATBench's performance closely models STAT, taking a few hundredths of a second longer on average. These results are not too surprising as the communication tree topologies of STATBench are equivalent to those of STAT. The slight increase in time could come from the fact that the STATBench parameters chosen result in a few more call graph nodes in the output than the STAT counterpart.

4.2 Simulating STAT on BlueGene/L

Having validated STATBench's ability to model STAT on Thunder, we next emulate STAT on BG/L. BG/L is a massively parallel machine at LLNL with 65,536 compute nodes. Each compute node has two PowerPC cores and runs a custom lightweight kernel (Compute

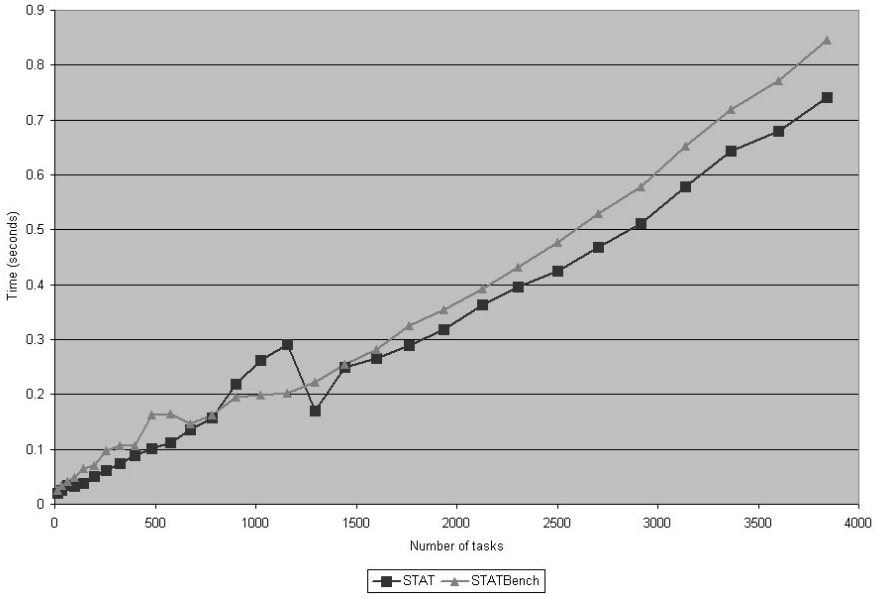


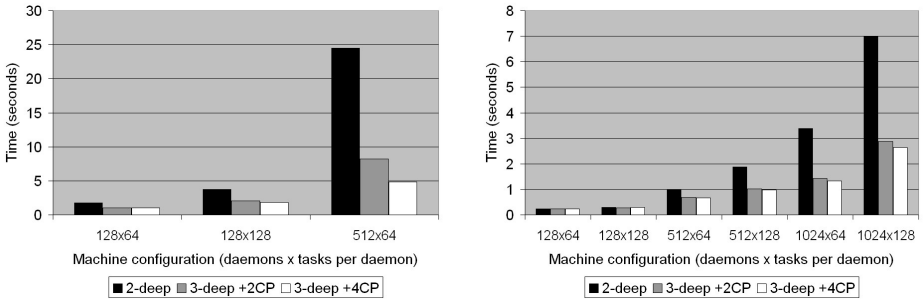
Figure 3. STAT versus STATBench on Thunder.

Node Kernel) that does not support multi-threading and only implements a subset of the standard Linux system calls. I/O operations for the compute nodes are executed by an associated I/O node. LLNL's BG/L configuration has 1024 I/O nodes, each responsible for 64 compute nodes. The I/O nodes are also responsible for running any debugger daemons. BG/L has two modes of operation, co-processor mode and virtual node mode. In co-processor mode one of the two compute node cores runs an application task, while the other core handles communication. Virtual node mode, on the other hand, utilizes both compute node cores for application tasks, scaling up to 131,072 tasks.

4.2.1 Initial Results

Our emulation was performed on uBG/L, a single rack BG/L machine. The architecture of uBG/L is similar to BG/L, with the main difference being the compute node to I/O node ratio. Instead of a sixty-four to one ratio, uBG/L has an eight to one ratio with a total of 128 I/O nodes and 1024 compute nodes. By running STATBench on all of uBG/L's 1024 compute nodes, we are able to emulate a full scale run of STAT on BG/L, where a daemon is launched on each of BG/L's 1024 I/O nodes and debugs 64 compute nodes.

We ran STATBench on uBG/L with similar parameters that were used on Thunder. One major change was that we ran tests with both 64 and 128 tasks per daemon to emulate co-processor mode and virtual node mode respectively. We tested 2-deep and 3-deep MRNet topologies that were dictated by the machine architecture. For the 2-deep tree, the STATBench front-end, running on the uBG/L front-end node, connects to all communication processes on the I/O nodes, each communication processes in turn connects to eight STATBench daemon emulators on the compute nodes. The 3-deep tree employs an



(a) BG/L Scale Results (b) BG/L Bit Vector Scale Results

Figure 4. BG/L scale STATBench performance results with (a) string-based task lists and (b) bit vector task lists.

additional layer of communication processes residing between the STATBench front-end process and the I/O node communication processes. We tested configurations with two and four processes in this layer, in both cases running on the four CPU uBG/L front-end node.

Using these parameters, we attempted to scale STATBench to 128, 512, and 1024 daemon emulators. The results of these tests can be seen in Fig. 4(a). STATBench modestly scaled when emulating up to 512 daemons with 64 tasks per daemon, especially when using a 3-deep tree with the additional layer of four communication processes. However, STATBench was only able to run up to this scale, half of the full BG/L machine in co-processor mode, which only represents 25% of BG/L's maximum process count. At larger scales, STATBench failed because of the quantity and the size of the task lists used for the edge labels. The task lists were implemented as strings in the STAT prototype. For example, the set $\{1,3,4,5,6,9\}$ would be translated into the string "1,3-6,9". At greater scales, such a representation can grow prohibitively large, up to 75 KB per edge label when emulating 32,768 tasks. Doubling this string length to 150K in order to represent twice as many tasks overloaded the communication processes, which received graphs from multiple children. Even if STAT could run at these scales with the string representation, the amount of data being sent and processed would have taken an intolerable amount of time.

4.2.2 Optimizing STAT

To overcome the scaling barrier caused by the string representation of the task lists, we instead implement each task list as a bit vector. In this implementation, one bit is allocated per task and its respective bit is set to *1* if the task is in the list, *0* otherwise. The bit vector benefits not only from requiring only one bit per task but also from merging the task lists with a simple *bitwise or* operation.

The results for using the bit vector at BG/L scales can be seen in Fig. 4(b). These results show a substantial improvement over the string implementation of the task lists. At the largest successful scale with the string implementation, 512 daemons with 64 tasks each, the best time was nearly five seconds, compared to two-thirds of a second with the bit vector implementation. The emulation of the full BG/L machine in virtual node mode, with 131,072 processes, required just over two and a half seconds to merge all of the traces. It is worth noting that going from a 2-deep tree to a 3-deep tree resulted in a substantial

improvement, especially at larger scales. Although much less significant, going from a layer of 2 communication processes to 4 communication processes also had some benefit.

5 Conclusion and Future Work

We have presented STATBench, an intermediate step in the development of the Stack Trace Analysis Tool. Our evaluation of STATBench indicates that it provides accurate results in its emulation of STAT. Using STATBench, we were able to emulate scaling runs of STAT on BlueGene/L, which allowed us to identify and to fix a scalability bug in the STAT prototype implementation. With this fix, STATBench estimates that STAT will be able to merge stack traces from 131,072 tasks on BG/L in under three seconds.

Running STAT on BG/L with the same MRNet topology as our STATBench tests will require additional computational resources, particularly for the layer of 128 communication processes that were run on the I/O nodes during our STATBench tests on uBG/L. We hypothesize that STAT performance will not suffer much by reducing this layer to 64 or 32 communication processes, however this remains to be tested once STAT has been deployed on BG/L. In any case, STAT can utilize the computational resources of BG/L's 14 front-end nodes and the visualization cluster connected to BG/L.

Our design of STATBench is general for addressing the scalability of TBŌN-based tools. In such tools, a single back-end daemon is typically responsible for analyzing multiple application processes on a compute node. A benchmark for such a tool does not need to restrict itself to a single back-end daemon per compute node; rather, it can run one back-end daemon on each of a compute node's processing cores. Each daemon can then emulate any number of processes by generating artificial, yet representative tool data (i.e., stack traces in the case of STATBench). The ability of a TBŌN-based tool benchmark to run on all of a compute node's processing cores, combined with the emulation of an arbitrary number of application processes, allows for tests to be performed at larger scales than the actual tool running on the same compute resources.

References

1. TotalView Technologies, TotalView `http://www.totalviewtech.com/productsTV.htm`
2. B. R. Buck and J. K. Hollingsworth, *An API for runtime code patching*, The International Journal of High Performance Computing Applications, **14**, 4, (317–329)2000.
3. P. Roth, D. Arnold and B. Miller, *MRNet: A software-based multicast/reduction network for scalable tools*, in: Proc. IEEE/ACM Supercomputing '03, (2003).
4. D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller and M. Schulz, *Stack trace analysis for large scale debugging*, in: 21st International Parallel and Distributed Processing Symposium 2007, (2007).

Scalable, Automated Performance Analysis with TAU and PerfExplorer

Kevin A. Huck, Allen D. Malony, Sameer Shende and Alan Morris

Performance Research Laboratory
Computer and Information Science Department
University of Oregon, Eugene, OR, USA
E-mail: {khuck, malony, sameer, amorris}@cs.uoregon.edu

Scalable performance analysis is a challenge for parallel development tools. The potential size of data sets and the need to compare results from multiple experiments presents a challenge to manage and process the information, and to characterize the performance of parallel applications running on potentially hundreds of thousands of processor cores. In addition, many exploratory analysis processes represent potentially repeatable processes which can and should be automated. In this paper, we will discuss the current version of PerfExplorer, a performance analysis framework which provides dimension reduction, clustering and correlation analysis of individual trails of large dimensions, and can perform relative performance analysis between multiple application executions. PerfExplorer analysis processes can be captured in the form of Python scripts, automating what would otherwise be time-consuming tasks. We will give examples of large-scale analysis results, and discuss the future development of the framework, including the encoding and processing of expert performance rules, and the increasing use of performance metadata.

1 Introduction

Parallel applications running on high-end computer systems manifest a complexity of performance phenomena. Tools to observe parallel performance attempt to capture these phenomena in measurement datasets rich with information relating multiple performance metrics to execution dynamics and parameters specific to the application-system experiment. However, the potential size of datasets and the need to assimilate results from multiple experiments makes it a daunting challenge to not only process the information, but discover and understand performance insights. In order to perform analysis on these large collections of performance experiment data, we developed PerfExplorer[1], a framework for parallel performance data mining and knowledge discovery. The framework architecture enables the development and integration of data mining operations that can be applied to large-scale parallel performance profiles. PerfExplorer is built on a performance data management framework called PerfDMF[2], which provides a library to access the parallel profiles and save analysis results in a relational database.

A performance data mining framework should support both advanced analysis techniques as well as extensible *meta analysis* of performance results. The use of *process control* for analysis scripting, *persistence* and *provenance* mechanisms for retaining analysis results and history, *metadata* for encoding experiment context, and support for *reasoning* about relationships between performance characteristics and behavior all are important for productive performance analytics. However, the framework must also be concerned about how to interface with application developers in the performance discovery process. The ability to engage in process programming, knowledge engineering (metadata and inference rules), and results management opens the framework toolset for creating data mining

environments specific to the developer's concerns.

We have re-engineered our integrated framework for performing meta analysis to incorporate parallel performance data, performance context metadata, expert knowledge, and intermediate analysis results. New methods were needed for correlating context metadata with the performance data and the analysis results, in order to provide the capability to generate desired empirical performance results from accurate suggestions on how to improve performance. Constructing this framework also required methods for encoding expert knowledge to be included in the analysis of performance data from parametric experiments. Knowledge about subjects such as hardware configurations, libraries, components, input data, algorithmic choices, runtime configurations, compiler choices, and code changes will augment direct performance measurements to make additional analyses possible.

The remainder of the paper is as follows. We discuss our analysis approach for the framework and our prototype implementation in Section 2. We will present some recent analysis examples which demonstrate some new PerfExplorer features in Section 3 and present future work and concluding remarks in Section 4.

2 PerfExplorer Design

PerfExplorer[2] is a Java application developed for performing data mining analyses on multi-experiment parallel performance profiles. Its capabilities include general statistical analysis of performance data, dimension reduction, clustering, and correlation of performance data, and multi-experiment data query and management. These functions were provided, in part, by the existing analysis toolkits (R[3] and Weka[4]), and our profile database system PerfDMF[2].

While PerfExplorer is a step forward in the ability to automatically process complex statistical functions on large amounts of multi-dimensional parallel performance data, its functionality was limited in two respects. First, the tool only allows a user to select single analysis operations via a graphical user interface. Multi-step analysis processes are not possible. Second, PerfExplorer only provides new descriptions of the data – it does not *explain* the performance characteristics or behavior observed (i.e., meta analysis). Scripting and support for retaining intermediate results can help to address the first shortcoming. The second is more challenging.

For example, an analyst can determine that on average, application X spent 30% of its total execution time in function `foo()`, and that when the number of processors is increased, the percentage of time may go up or down, and so on. However, PerfExplorer did not have the capability to explain *why* the change happened. The explanation may be as simple as the fact that the input problem doubled in size, but without that contextual knowledge, no analysis tool could be expected to come to any conclusions about the cause of the performance change without resulting to speculation.

As we discussed our enhancements to PerfExplorer, we will consider two analysis cases: 1) we have collected parallel performance data from multiple experiments, and we wish to compare their performance, or 2) we have collected performance data from one experiment, and would like to compare the performance between processes or threads of execution. Like other tools, PerfExplorer can provide the means for an analyst to determine which execution is the "best" and which is the "worst", and can even help the analyst investigate further into which regions of code are most affected, and due to which met-

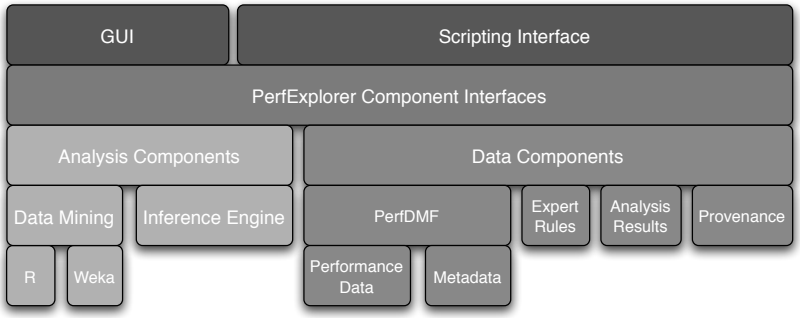


Figure 1. The redesigned PerfExplorer components.

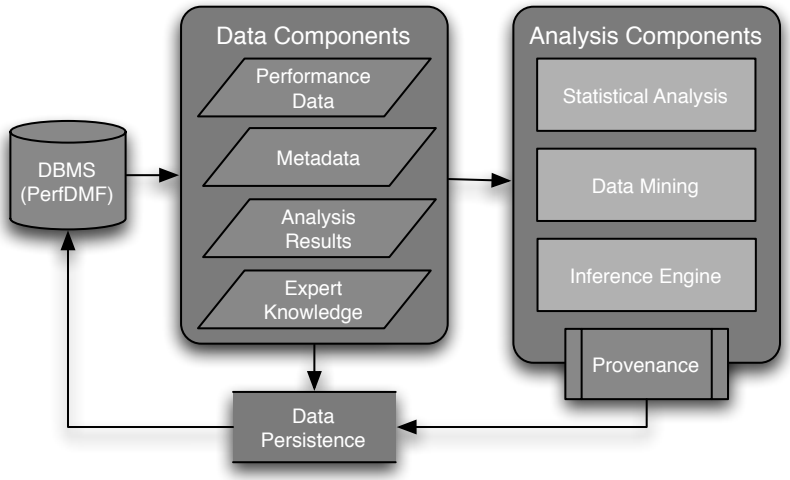


Figure 2. PerfExplorer components and their interactions.

rics. However, there is no explicit *process control*, nor is there *higher-level* reasoning or analysis of the performance result to explain what caused the performance differences. In addition, process control is required in order to perform repeated analysis procedures or non-interactive analysis automation. In order to perform this type of meta-analysis, several components are necessary to meet the desired goals.

Figure 1 shows the PerfExplorer components, and Fig. 2 shows the interaction between components in the new PerfExplorer design. The performance data and accompanying metadata are stored in the PerfDMF database. Performance data is used as input for statistical analysis and data mining operations, as was the case in the original version of PerfExplorer. The new design adds the ability to make all intermediate analysis data persistent, not just the final summarization. Expert knowledge is incorporated into the analysis, and these new inputs allow for higher-level analysis. An inference engine is also added to combine the performance data, analysis results, expert knowledge and execution metadata into a performance characterization. The provenance of the analysis result is stored with the result, along with all intermediary data, using object persistence. The whole pro-

cess is contained within a process control framework, which provides user control over the performance characterization process.

2.1 Process Control

One of the key aspects of the new PerfExplorer design is the requirement for process control. While user interfaces and data visualization are useful for interactive data exploration, the user will need the ability to control the analysis process as a discrete set of operations. In order to synthesize analysis results, expert knowledge and metadata into higher-level meta-analysis process, PerfExplorer needs an extension mechanism for creating higher-order analysis procedures. One way of doing this is through a scripting interface, such as Jython[a]. With such an interface, the analysis process is under the control of the analyst, who is able to reliably produce the characterization with minimal effort. This new scripting support was used to generate the results in Section 3.2.

2.2 Collecting and Integrating Metadata

Performance instrumentation and measurement tools such as TAU[5] collect context metadata along with the application performance data. This metadata potentially contains information relating to useful analysis input such as the build environment, runtime environment, configuration settings, input and output data, and hardware configuration. Metadata examples which are automatically collected by the profiling provided by TAU include fields such as processor speed, node hostname, and cache size. It should be easy to see how fields such as *CPU MHz*, *Cache Size* or *Memory Size* would be useful in explaining the differences between executions. By integrating these fields into the analysis process, the analysis framework can reason about potential causes for performance failures. This new metadata support was used to generate the results in Section 3.1.

2.3 Inference Engine

Because the needs of the meta-analysis are dynamic, why should a performance analysis tool hard-code the complex rules that guide the decision making process of a performance analyst? Is it even possible to translate the subtleties of analysis reasoning to source code? In order to provide the type of higher-level reasoning and meta-analysis we require in our design, we have integrated a JSR-94[b] compliant rule engine, JBoss Rules[c]. The strategic selection of an inference engine and processing rules allows another method of flexible control of the process, and also provides the possibility of developing a domain specific language to express the analysis.

2.4 Provenance and Data Persistence

In order to rationalize analysis decisions, any explanation needs to include the data provenance, or the full chain of evidence and handling from raw data to synthesized analysis result. The new design will include the ability to make all intermediate analysis data persistent, not just the final summarization. The provenance of the analysis result is stored with the results and all intermediary data, using object persistence[d]. Any scientific endeavour is considered to be of "good provenance" when it is adequately documented in

[a] Jython: http://www.jython.org/
[b] Java Rule Engine API: http://jcp.org/en/jsr/detail?id=94
[c] JBoss Rules: http://www.jboss.com/products/rules
[d] Relational Persistence for Java and .NET: http://www.hibernate.org/

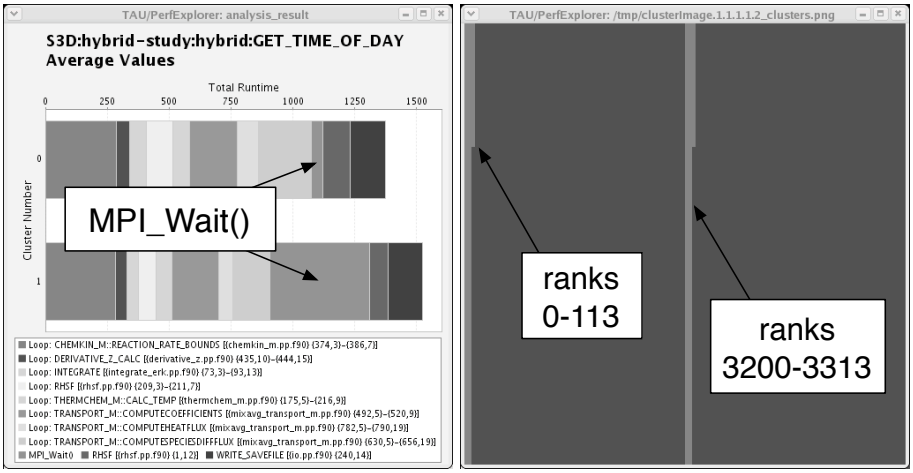


Figure 3. S3D cluster analysis. The figure on the left shows the difference in (averaged mean) execution behaviour between the two clusters of processes. The figure on the right shows a virtual topology of the MPI processes, showing the locations of the clustered processes. The lighter processes ran on XT3 nodes, and the darker processes ran on XT4 nodes.

order to allow reproducibility. For parallel performance analysis, this includes all raw data, analysis methods and parameters, intermediate results, and inferred conclusions.

3 Analysis Examples

3.1 S3D

S3D[6] is a multi-institution collaborative effort with the goal of creating a terascale parallel implementation of a turbulent reacting flow solver. S3D uses direct numerical simulation (DNS) to model combustion science which produces high-fidelity observations of the micro-physics found in turbulent reacting flows as well as the reduced model descriptions needed in macro-scale simulations of engineering-level systems. The examples described here were run on the hybrid Cray XT3/XT4 system at Oak Ridge National Laboratory (ORNL).

During scalability tests of S3D with TAU, it was observed that as the number of processors exceeded 1728, the amount of time spent in communication began to grow significantly, and `MPI_Wait()` in particular composed a significant portion of the overall run time (approximately 20%). By clustering the performance data in PerfExplorer, it was then observed that there were two natural clusters in the data. The first cluster consisted of a majority of the processes, and these nodes spent less time in main computation loops, but a long time in `MPI_Wait()`. The other cluster of processes spent slightly more time in main computation loops, and far less time in `MPI_Wait()`.

By adding Cray-specific `TAU_METADATA()` instrumentation calls, we were able to determine the names of the nodes on which the processes ran. In the case of a 6400 process run, as shown in Fig. 3, there were again two clusters, with 228 processes in one cluster having very low `MPI_Wait()` times (about 40 seconds), and the remainder of the processes in one cluster having very high `MPI_Wait()` times (over 400 seconds). The

metadata collected, manually correlated with information about the hardware characteristics of each node, identified the slower nodes as XT3 nodes, and the faster nodes as XT4 nodes. There are two primary differences between the XT3 and XT4 partitions. The XT3 nodes have slower DDR-400 memory (5986 MB/s) than the XT4 nodes' DDR2-667 memory (7147 MB/s), and the XT3 partition has a slower interconnection network (1109 MB/s v. 2022 MB/s). Because the application is memory intensive, the slower memory modules have a greater effect on the overall runtime, causing the XT3 nodes to take longer to process, and subsequently causing the XT4 nodes to spend more time in `MPI_Wait()`.

Running S3D on an XT4-only configuration yielded roughly a 12% time to solution reduction over the hybrid configuration, primarily by reducing `MPI_Wait()` times from an average of 390 seconds down to 104 seconds. If this application is to be run on a heterogeneous configuration of this machine or any other, load balancing should be integrated which takes into consideration the computational capacity of each respective processor. The use of metadata will also be important for this optimization.

3.2 GTC

The Gyrokinetic Toroidal Code (GTC) is a particle-in-cell physics simulation which has the primary goal of modelling the turbulence between particles in the high energy plasma of a fusion reactor. Scalability studies of the original large-scale parallel implementation of GTC (there are now a small number of parallel implementations, as the development has fragmented) show that the application scales very well - in fact, it scales at a better than linear rate. However, discussions with the application developers revealed that it had been observed that the application gradually slows down as it executes[7] - each successive iteration of the simulation takes more time that the previous iteration.

In order to measure this behaviour, the application was auto-instrumented with TAU, and manual instrumentation was added to the main iteration loop to capture dynamic phase information. The application was executed on 64 processors of the Cray XT3/XT4 system at ORNL for 100 iterations, and the performance data was loaded into PerfExplorer. A PerfExplorer analysis script was constructed in order to examine the dynamic phases in the execution. The script was used to load the performance data, extract the dynamic phases from the profile, calculate derived metrics (i.e. cache hit ratios, FLOPS), calculate basic statistics for each phases, and graph the resulting data as a time series showing average, minimum and maximum values for each iteration.

As shown in Fig. 4, over a 100 iteration simulation, each successive iteration takes slightly more time than the previous iteration. Over the course of the test simulation, the last iteration takes nearly one second longer than the first iteration. As a minor observation, every fourth iteration results in a significant increase in execution time. Hardware counters revealed that the L2 (and to a lesser extent, L1) cache hit-to-access ratio decreases from 0.92 to 0.86. Subsequently, the GFLOPS per processor rate decreases from 1.120 to 0.979. Further analysis of the two main routines in the iterations, `CHARGEI` and `PUSHI`, show that the decrease in execution is limited to these two routines. In the `CHARGEI` routine, each particle in a region of the physical subdomain applies a charge to up to four cells, and in the `PUSHI` routines, the particle locations are updated by the respective cells after the forces are calculated. The increase in time every fourth iteration was discovered to be due to a diagnostic call, which happens every `ndiag` iterations, an input parameter captured as metadata.

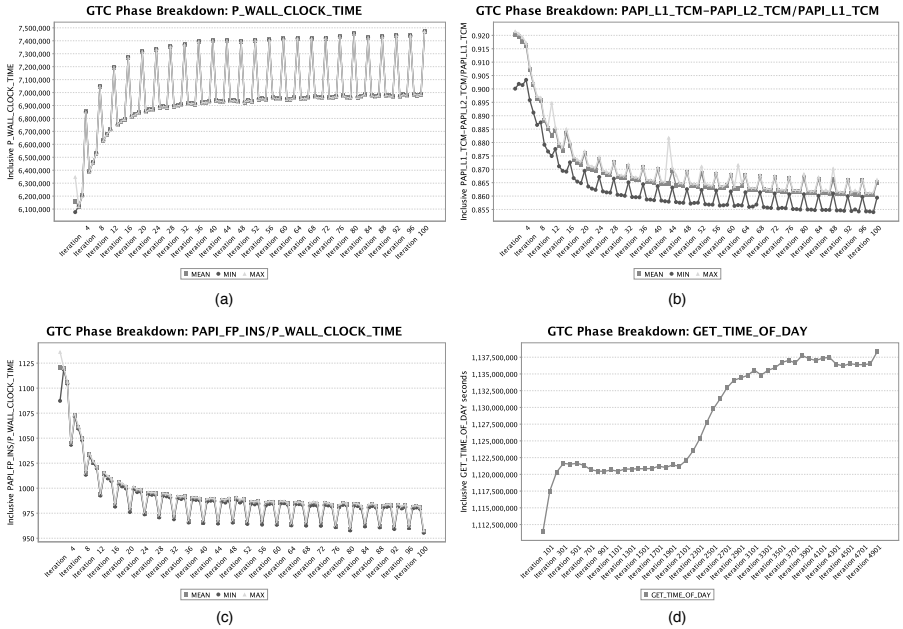


Figure 4. GTC phase analysis. (a) shows the increase in runtime for each successive iteration, over 100 iterations. (b) shows the decrease in L2 hit ratio, from 0.92 to 0.86, and (c) shows the decrease in GFLOPS from 1.120 to 0.979. (d) shows the larger trend when GTC is run for 5000 iterations, each data point representing an aggregation of 100 iterations.

Discussions with other performance analysis experts on the project revealed that the `CHARGEI` and `PUSHI` routines have good spatial locality when referencing the particles, however over time, they have poor temporal locality when referencing the grid cells. As a result, access to the grid cells becomes random over time. Further analysis is necessary to determine whether the expense of re-ordering the particles at the beginning of an iteration could be amortized over a number of iterations, and whether this added cost would yield a benefit in the execution time. While it appears that the performance degradation levels out after roughly 30 iterations, it should be pointed out that a full run of this simulation is at least 10,000 iterations, and as the 5,000 iteration execution shows in Fig. 4 (d), the performance continues to degrade. Assuming a 10,000 iteration execution would take an estimated 20 hours to complete on the Cray XT3/XT4, potentially 2.5 hours of computation time per processor could be saved by improving the cache hit ratios. Further analysis is ongoing.

4 Future Work and Concluding Remarks

In this paper, we have discussed the new design of PerfExplorer, including components for scripting, metadata encoding, expert rules, provenance and data persistence. In our examples, we have discussed how features such as metadata encoding and scripting aid in the analysis process. However, we have significant work to do in extending the capabilities of PerfExplorer. While the rudimentary metadata support in PerfExplorer allows

for some manual correlation between contextual information and performance results, sophisticated analysis rules to interpret the results with respect to the contextual information would aid us in our long term goal of a performance tool which would summarize performance results and link them back to the actual causes, which are essentially the context metadata relating to the application, platform, algorithm, and known related parallel performance problems. Encoding this knowledge in some form that our performance tool can use would be instrumental in developing new analysis techniques that capture more information about the experiment than simply the raw performance data. While full automation may prove difficult, we feel that a useful amount of automatic performance correlation is possible.

Acknowledgements

The research was funded by the Department of Energy, Office of Science under grant DE-FG02-05ER25680 and DE-FG02-07ER25826, and the National Science Foundation, High-End Computing program under grant NSF CCF 0444475. The authors would like to thank PERI, SciDAC, ORNL, NERSC and RENCI for including us in the PERI SciDAC project, and a special thanks to John Mellor-Crummey for a better understanding of the locality issues in GTC.

References

1. Kevin A. Huck and Allen D. Malony, *PerfExplorer: a performance data mining framework for large-scale parallel computing*, in: Conference on High Performance Networking and Computing (SC'05), Washington, DC, USA, IEEE, (2005).
2. K. Huck, A. Malony, R. Bell and A. Morris, *Design and implementation of a parallel performance data management framework*, in: Proc. International Conference on Parallel Computing, 2005 (ICPP2005), pp. 473–482, (2005).
3. The R Foundation for Statistical Computing, *R Project for Statistical Computing*, (2007). http://www.r-project.org
4. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd Ed., (Morgan Kaufmann, San Francisco, 2005). http://www.cs.waikato.ac.nz/~ml/weka/
5. S. Shende and A. D. Malony, *The TAU parallel performance system*, International Journal of High Performance Computing Applications, **20**, 287–331, (2006).
6. J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummy, N. Podhorski, R. Sankaran, S. Shende and C. S. Yoo, *Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D*, Institute of Physics Journal, (in press, 2007).
7. N. Wichmann, M. Adams and S. Ethier, *New advances in the gyrokinetic toroidal code and their impact on performance on the Cray XT series*, in: *Cray Users Group*, (2007).

Developing Scalable Applications with Vampir, VampirServer and VampirTrace

Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel

ZIH (Center for Information Services and HPC)
Technische Universität Dresden, 01162 Dresden
E-mail: {*matthias.mueller, andreas.knuepfer, matthias.jurenz*}*@tu-dresden.de*
E-mail: {*matthias.lieber, holger.brunst, hartmut.mix, wolfgang.nagel*}*@tu-dresden.de*

Abstract

This paper presents some scalability studies of the performance analysis tools Vampir and VampirTrace. The usability is analyzed with data collected from real applications, i.e. the thirteen applications contained in the SPEC MPI 1.0 benchmark suite. The analysis covers all phases of performance analysis: instrumenting the application, collecting the performance data, and finally viewing and analyzing the data. The aspects examined include instrumenting effort, monitoring overhead, trace file sizes, load time and response time during analysis.

1 Introduction

With (almost) petaflop systems consisting of hundreds of thousands of processors at the high end and multi-core CPUs entering the market at the low end application developers face the challenge to exploit this vast range of parallelism by writing scalable applications. Any performance analysis tool targeting this audience has to provide the same scalability. Even more so, as many performance problems and limitations are only revealed at high processors counts. Furthermore, massively parallel applications tend to be especially performance critical – otherwise they would not employ so many resources.

The remainder of this paper is organized as follows: Section 2 provides background information for the Vampir family of performance analysis tools. The next Sections 3 and 4 look at the overhead involved in the performance analysis process. Firstly, at tracing time to estimate the impact of the measurement infrastructure onto the dynamic behaviour of parallel programs. Secondly, at analysis time when interactive visualization for very huge amounts of data is required to allow a convenient work flow. Section 5 gives a summary and an outlook on future features of scalable performance analysis tools.

2 Description of Vampir and VampirTrace

For the analysis of MPI parallel applications there are a number of performance analysis tools available[1]. Vampir, which is being developed at ZIH, TU Dresden, is well known in the HPC community. Today, there are two versions of Vampir. Firstly, the workstation

based classical application with a development history of more than 10 years[2]. Secondly, the more scalable distributed version called VampirServer[3,4].

In addition there is the instrumentation and measurement software VampirTrace. Although not exclusively bundled to Vampir and VampirServer, this is the preferred way to collect the input data for both analysis tools.

2.1 Vampir

Vampir[2,5] is a performance analysis tool that allows the graphical display and analysis of program state changes, point-to-point messages, collective operations and hardware performance counters together with appropriate statistical summaries. It is designed to be an easy to use tool, which enables developers to quickly display program behaviour at any level of detail. Different timeline displays show application activities and communication along a time axis, which can be zoomed and scrolled. Statistical displays provide quantitative results for arbitrary portions of the timelines. Powerful zooming and scrolling allows to pinpoint the real cause of performance problems. Most displays have context-sensitive menus which provide additional information and customization options. Extensive filtering capabilities for processes, functions, messages or collective operations help to reduce the information to the interesting spots. The implementation is based on standard X-Windows and Motif and works on desktop workstations as well as on parallel production systems. It is available for nearly all 32- and 64-bit platforms like Linux-based PCs and Clusters, IBM, SGI, SUN, and Apple.

2.2 VampirServer

VampirServer[3,4] is the next generation, parallel implementation of Vampir with much higher scalability. It implements a client/server architecture. The server is a parallel program which uses standard communication methods such as MPI, pthreads, and sockets. The complex preparation of performance data is carried out by the server component. The server itself consists of one master process and a variable number of worker processes. The visualization of performance results is done by a small client program connecting to the server, more precisely to its master process[3].

VampirServer implements parallelized event analysis algorithms and customizable displays which enable fast and interactive rendering of very complex performance monitoring data. Trace information are kept in distributed memory on the parallel analysis machine. Therefore, ultra large data volumes can be analyzed without copying huge amount of data. Visualization can be carried out from any laptop or desktop PC connected to the Internet.

The implementation is based on standard MPI and X-Windows and works on most parallel production systems (server) and desktop Unix workstations (client). Currently, the list of supported vendor platforms includes: IBM, SGI, SUN, NEC, HP, and Apple. The current version 1.7 was used for all experiments shown below.

2.3 VampirTrace

VampirTrace provides a convenient measurement infrastructure for collecting performance data. VampirTrace supports the developer with instrumentation and tracing facilities tai-

lored towards HPC applications. It covers MPI and OpenMP as well as user code. Instrumentation modifies a target application in order to detect and record run-time events of interest, for example a MPI communication operation or a certain function call. This can be done at source code level, during compilation or at link time with various techniques. The VampirTrace library takes care of data collection within all processes. This includes user events, MPI events, OpenMP events as well as timing information and location (Cluster node, MPI rank, Thread, etc.). Furthermore, it queries selected hardware performance counters available on all platforms supported by PAPI[6].

Automatic instrumentation of the source code using the compiler is available with compilers from GNU, Intel (version 10), IBM, PGI and SUN (Fortran only). Binary instrumentation is performed with DynInst[7]. An analysis of the MPI calls made by the application is made using the so called profile interface of the MPI library.

The collected performance data is written to file using the Open Trace Format (OTF). OTF[9] is a fast and efficient trace format library with special support for parallel I/O. It provides a convenient interface and is designed to achieve good performance on single processor workstations as well as on massive parallel super computers. Transparent blockwise ZLib compression allows to reduce storage size.

VampirTrace is available at http://www.tu-dresden.de/zih/vampirtrace under BSD Open Source license for various platforms, e.g. Linux, IBM, SGI, SUN. The implementation is based on the Kojak tool-set[8]. Development is done at ZIH, TU Dresden, Germany in cooperation with ZAM, Research Center Jülich, Germany, and the Innovative Computing Laboratory at University of Tennessee, US. The current version is 5.3 which was used for the experiments below.

3 Overhead of Instrumentation

The data collection and measurement phase is the first occasion where additional overhead is introduced by the tracing infrastructure. There are two parts involved: instrumentation before execution and measurement during run-time. However, the former has no significant impact at all, because instrumentation does not depend on the number of processes or run time. Yet, the latter imposes notable overhead during run-time. There are four individual contributions:

- initialization at program start-up
- per-event overhead (in event handlers)
- storage of trace data to disk
- finalization

The initialization sets up internal data structures, get symbol information etc. and normally does not add noticeable overhead to the program start-up. Instead, calling event handlers contributes the most part of the critical overhead of tracing. The per-event overhead does not depend on the duration of the recorded event, thus significant overhead is produced for very frequent events, especially frequent short function calls.

Storing trace data on disk produces considerable overhead as well. Therefore, the trace data is first written to memory buffers and afterwards flushed to permanent storage. If

Code	LOC	Language	MPI call sites	MPI calls	Area
104.milc	17987	C	51	18	Lattice QCD
107.leslie3d	10503	F77,F90	43	13	Combustion
113.GemsFDTD	21858	F90	237	16	Electrodynamic simulation
115.fds4	44524	F90,C	239	15	CFD
121.pop2	69203	F90	158	17	Geophysical fluid dynamics
122.tachyon	15512	C	17	16	Ray tracing
126.lammps	6796	C++	625	25	Molecular dynamics
127.wrf2	163462	F90,C	132	23	Weather forecast
128.GAPgeofem	30935	F77,C	58	18	Geophysical FEM
129.tera_tf	6468	F90	42	13	Eulerian hydrodynamics
130.socorro	91585	F90	155	20	density-functional theory
132.zeusmp2	44441	C,F90	639	21	Astrophysical CFD
137.lu	5671	F90	72	13	SSOR

Table 1. Size of Applications (measured in Lines of Code) and programming language

possible, VampirTrace tries to flush only at the finalization phase. In this case there is no interference with the timing of events. Otherwise, the program execution needs to be interrupted eventually. This is marked within the trace but nevertheless it is a severe perturbation. During the finalization phase some post-processing of the trace data is required. This costs additional effort (in computation and I/O) but does not influence the quality of the measurement.

In the following, the overhead will be measured with several experiments on a SGI Altix 4700 machine with 384 Intel Itanium II Montecito cores (1.6 GHz) and 512 GB memory. As a first approach we measured the overhead of instrumenting a function call with a simple test program that calculates π and calls a simple function ten million times. We measured an overhead for each function call of 0.89μs using source code instrumentation and 1.12μs using binary instrumentation. The overhead increases to 4.6μs when hardware performance counters are captured.

For a thorough analysis how this overhead affects the analysis of real applications we instrumented the thirteen applications of the SPEC MPI2007 benchmark[10]. Table 1 contains a short description of the all codes. The analysis consists of two scenarios: first we do full source code instrumentation capturing each function call using a smaller dataset (the so called *train* data set) with 32 CPUs. Second, we do an analysis of the MPI behaviour, capturing only MPI calls made by the application using a large production like data set (called *mref*) with 256 CPUs[10].

In order to get useful performance measurements the overhead introduced by the monitoring must not be too high. In Table 2 the two main contributions to the overhead are distinguished if possible. The overhead during tracing (trace) dominates the total overhead, while the overhead from writing trace data to disk (flush) after tracing is smaller. In some cases, intermediate flusing occurs. Then, the most part of the flushing overhead is brought forward to the tracing phase and is indistinguishable. The missing entries show where no valid trace could be created. For this cases only the sum is given Table 2. Note,

Code	original (train) 32 CPUs	Fully instrumented trace	flush	original (mref) 256 CPUs	MPI instrumented trace	flush
104.milc	9.1s	1081s		267s	265.9s	8.2s
107.leslie3d	24.5s	57.0s	43.0s	192s	192.2s	17.7s
113.GemsFDTD	88.9s	-	-	1281s	1259s	32s
115.fds4	18.7s	34.6s	20.7s	605s	597s	21s
121.pop2	57.2s	-	-	444s	5598s	
122.tachyon	12.7s	2595s		264s	271.3s	13.5s
126.lammps	36.1s	(?) 15.6s	-	493s	498s	13s
127.wrf2	24.3s	992s		331s	343s	20s
128.GAPgeofem	4.3s	-	-	106s	-	-
129.tera_tf	89.1s	169.8s	56.1s	290s	287.1s	18s
130.socorro	29.3s	1755s		195s	(?) 177.4s	19s
132.zeusmp2	25.7s	26.0s	3.9s	160s	160.8s	17s
137.lu	12.4s	15.7s	6.4s	92s	96.4s	18.5s

Table 2. Runtime and overhead of fully instrumented and MPI instrumented codes. The overhead is divided in *trace* overhead and overhead from *flush* operations. Entries marked with only on number for both suffer from very large trace sizes that cause intermediate flushing. Entries marked with (?) show reproducible inconsistent results. Missing entries indicate erroneous behaviour where no valid trace could be generated.

that this happens mostly for full traces but only once for MPI-only traces[a].

Figure 1 shows that this effect is directly coupled with total trace volumes. Here, the size occupied in internal buffers is essential. This is not equal to the trace file sizes but proportional. In the same way, different file formats show proportional trace sizes.

As soon as memory buffers are exceeded, flushing becomes inevitable, triggering the unsatisfactory outcome. The memory buffer size is limited by the available memory and by the application's consumption. Only their difference is available for VampirTrace. Here, a quite comfortable buffer size of 2 GB per process was allowed.

In real world performance analysis, intermediate flushing should be avoided by all means! There are two standard solutions which can also be combined: Firstly, limited tracing, i.e. tracing a selected time interval of a subset of all processes only. Secondly, filtering of symbols, i.e. omitting certain functions from tracing.

4 Scalability of Performance Analysis

Once performance data is available from a large scale parallel test-run, one wants to analyze it in order to unveil performance flaws. There are several approaches for automatic, semi-automatic or manual analysis[1] complementing each other. Vampir and VampirServer provide an interactive visualization of dynamic program behaviour. Therefore, the main challenges are firstly, coping with huge amounts of trace data and secondly, accomplishing quick responses to user interactions. Both is vital for a convenient work flow.

Vampir and VampirServer rely on loading the trace data to main memory completely. This is the only way to achieve a quick evaluation on user requests. As a rule of thumb,

[a] In this example there is a tremendous point to point message rate of up to 3 million per second.

VampirServer needs about the memory size of the uncompressed OTF trace. For the VampirServer distributed main memory is good enough, thus, the memory requirements can be satisfied by just using enough distributed peers with local memory capacity each.

In the beginning of a performance analysis session, the trace data needs to be loaded into memory. The distribution across multiple files (i.e. OTF streams) enables parallel reading. This is another important advantage over the sequential counterpart. So, the input speed is only limited by the parallel file system and the storage network. Still, reading rather large amounts of trace data requires some time, compare Fig. 1.

However, this is required only once at the beginning and allows quicker responses during the whole performance analysis session. See first column of Table 3 for the times VampirServer requires to load the SpecMPI traces with 256 streams – corresponding to the number of MPI processes writing their own streams. The first row of Table 4 shows how load time decreases with a growing number of worker nodes for VampirServer with another example. This shows almost perfectly linear scaling.

After loading, the user wants to browse the data with various timeline displays and statistics windows using zooming, unzooming and scrolling. Table 3 shows that this is quite fast for all examples of the SpecMPI MPI-only traces, with the exception of *121.pop2* and *128.GAPgeofem* due to the extremely huge volume. All response times are $\leq 5s$ except for the call tree computation, which needed up to $14s$ for some larger traces. The experiments have been performed with 32+1 VampirServer nodes, i.e. 32 workers plus one master, running on an SGI Altix 4700 with a maximum I/O rate of 2 GB/s.

More detailed experiments are shown in Table 4 for varying numbers of nodes. It reveals that some tasks are always fast regardless of the number of nodes, for example the *timeline* and the *summary timeline*. At the same time, these are the most frequently used ones. The *timeline unzoom* is always fast because it uses internal caching, i.e. it requires no re-computation. For some tasks there is a notable increase in response time as the number of nodes becomes too small (4+1), in particular for the *counter timeline*. However, with enough analysis nodes, there is a sufficient responsiveness for all tasks.

So the evaluation and visualization with VampirServer looks quite promising. Provided enough distributed memory, i.e. distributed worker nodes, it allows a truly interactive work

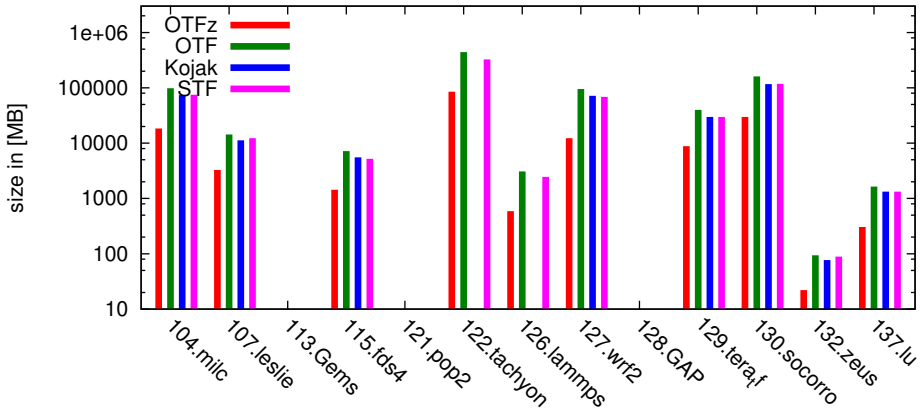


Figure 1. Filesize of the fully instrumented runs with different compressed and uncompressed trace formats.

code	startup & load	Timeline	Process TL	Summary TL	Counter TL	Message Statistics	Call Tree	Trace Size
104	5s	1s	2s	0s	1s	0s	0s	190 MB
107	29s	1s	1s	1s	4s	1s	2s	1.7 GB
113	10s	2s	1s	1s	9s	0s	1s	348 MB
115	7s	1s	1s	1s	3s	0s	0s	67 MB
121								128 GB
122	7s	3s	1s	3s	8s	0s	0s	1.6 MB
126	7s	2s	1s	1s	10s	0s	0s	64 MB
127	68s	4s	1s	4s	14s	1s	6s	4.1 GB
128								
129	14s	5s	2s	4s	9s	2s	2s	710 MB
130	43s	5s	1s	5s	5s	2s	3s	1.4 GB
132	12s	3s	1s	3s	3s	1s	1s	419 MB
137	79s	3s	2s	2s	4s	2s	4s	3.1 GB

Table 3. Vampir Server response times for MPI-only traces of the SpecMPI benchmarks with 32+1 processes.

CPUs for Analysis	Load	Timeline (TL)	TL Zoom	TL Unzoom	Summary TL	Counter TL	Message Statist.	Call Tree
1+1	957	≤ 1	5	≤ 1	≤ 1	29	≤ 1	148
4+1	217	≤ 1	2	≤ 1	≤ 1	8	≤ 1	43
16+1	58	≤ 1	≤ 1	≤ 1	≤ 1	2	≤ 1	12
32+1	29	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1	≤ 1	7

Table 4. Response times for the *107.leslie3d* full trace (3.2 GB compressed / 13.6 GB uncompressed in OTF).

flow even for huge traces. The software architecture is suitable for distributed memory platforms, which are most common today and allows quite easy extensibility.

5 Summary and Conclusion

We examined the scalability for the Vampir tools family with an extensive set of example applications from the SpecMPI benchmark suite. The experiments for tracing overhead showed reasonable overhead for most cases but also quite substantial overhead for very large traces. The solution is twofold: Either to reduce trace size with existing methods. Or to extend the tracing infrastructure for even lower disturbance with huge traces in the future. The analysis of the resulting traces with VampirServer turned out to be very reliable. Most traces can be visualized and analyzed with reasonable hardware resources in an interactive and convenient way. Only very few gigantic traces were critical. Instead of investing more and more resources for them (and even larger ones in the future) alternative methods might be better suited, which do not require main memory in the order of magnitude of the trace size[11]. Another promising project is to add existing and newly developed evaluation procedures into the existing VampirServer framework. Especially semi-automatic and fully-automatic evaluation may support the user in finding the actual spots that need close manual inspection. Hopefully, those can profit from VampirServer's scalable architecture as much as the ones examined in this paper.

References

1. S. Moore, D. Cronk, K. London and J. Dongarra, *Review of performance analysis tools for MPI parallel programs*, in: Y. Cotronis and J. Dongarra, eds., Recent Advances in Parallel Virtual Machine and Message Passing Interface, 8th European PVM/MPI Users' Group Meeting, LNCS **2131**, pp. 241—248, Santorini, Greece, (Springer, 2001).
2. W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe and K. Solchenbach. *VAMPIR: Visualization and analysis of MPI resources*, in: Supercomputer, vol. **1**, pp. 69–80, SARA Amsterdam, (1996).
3. H. Brunst and W. E. Nagel, *Scalable performance analysis of parallel systems: Concepts and experiences*, in: G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, eds., Parallel Computing: Software, Alghorithms, Architectures Applications, pp. 737–744, (Elsevier, 2003).
4. H. Brunst, D. Kranzlmüller and W. E. Nagel, *Tools for scalable parallel program analysis – VAMPIR NG and DEWIZ.* in: Distributed and Parallel Systems, Cluster and Grid Computing, vol. **777** of *International Series in Engineering and Computer Science*, pp. 93–102, (Kluwer, 2005).
5. H. Brunst, M. Winkler, W. E. Nagel and H.-C. Hoppe, *Performance optimization for large scale computing: The scalable VAMPIR approach*, in: V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner and C. J. Kenneth, eds., *International Conference on Computational Science – ICCS 2001*, vol. **2074**, pp. 751–760, (Springer, 2001).
6. S. Browne, C. Deane, G. Ho and P. Mucci, *PAPI: A portable interface to hardware performance counters*, in: Proc. Department of Defense HPCMP Users Group Conference, (1999).
7. J. K. Hollingsworth, B. P. Miller and J. Cargille, *Dynamic program instrumentation for scalable performance tools*, in: Proc. Scalable High Performance Compitung Conference, Knoxville, TN, (1994).
8. F. Wolf and B. Mohr, *KOJAK – a tool set for automatic performance analysis of parallel applications*, in: Proc. European Conference on Parallel Computing (Euro-Par), vol. **2790** of *LNCS*, pp. 1301–1304, Klagenfurt, Austria, (Springer, 2003).
9. A. Knüpfer, R. Brendel, H. Brunst, H. Mix and W. E. Nagel, *Introducing the open trace format (OTF)*, in: V. N. Alexandrov, G. D. Albada, P. M. A. Slot and J. Dongarra, eds., 6th International Conference for Computational Science (ICCS), vol. **2**, pp. 526–533, Reading, UK, (Springer, 2006).
10. M. S. Müller, M. van Waveren, R. Liebermann, B. Whitney, H. Saito, K. Kalyan, J. Baron, B. Brantley, Ch. Parrott, T. Elken, H. Feng and C. Ponder, *SPEC MPI2007 – an application benchmark for clusters and HPC systems*. in: ISC2007, (2007).
11. A. Knüpfer and W. E. Nagel, Compressible memory data structures for event-based trace analysis, Future Generation Computer Systems, **22**, 359–368, (2006).

Scalable Collation and Presentation of Call-Path Profile Data with CUBE

Markus Geimer[1], **Björn Kuhlmann**[1,3], **Farzona Pulatova**[1,2], **Felix Wolf**[1,3], **and Brian J. N. Wylie**[1]

[1] Jülich Supercomputing Centre
Forschungszentrum Jülich, 52425 Jülich, Germany
E-mail: {m.geimer, f.wolf, b.wylie}@fz-juelich.de, bjoern.kuhlmann@sap.com

[2] University of Tennessee
Innovative Computing Laboratory, Knoxville, TN 37996-3450, USA
E-mail: farzona@gmail.com

[3] RWTH Aachen University
Department of Computer Science, 52056 Aachen, Germany

Developing performance-analysis tools for parallel applications running on thousands of processors is extremely challenging due to the vast amount of performance data generated, which may conflict with available processing capacity, memory limitations, and file system performance especially when large numbers of files have to be written simultaneously. In this article, we describe how the scalability of CUBE, a presentation component for call-path profiles in the SCALASCA toolkit, has been improved to more efficiently handle data sets from thousands of processes. First, the speed of writing suitable input data sets has been increased by eliminating the need to create large numbers of temporary files. Second, CUBE's capacity to hold and display data sets has been raised by shrinking their memory footprint. Third, after introducing a flexible client-server architecture, it is no longer necessary to move large data sets between the parallel machine where they have been created and the desktop system where they are displayed. Finally, CUBE's interactive response times have been reduced by optimizing the algorithms used to calculate aggregate metrics. All improvements are explained in detail and validated using experimental results.

1 Introduction

Developing performance-analysis tools for applications running on thousands of processors is extremely challenging due to the vast amount of performance data usually generated. Depending on the type of performance tool, these data may be stored in one or more files or in a database and may undergo different processing steps before or while they are shown to the user in an interactive display. Especially when some of these steps are carried out sequentially or make use of shared resources, such as the file system, the performance can be adversely affected by the huge size of performance data sets. Another scalability limit is posed by memory capacity, which may conflict with the size of the data to be held at some point in the processing chain. As a consequence, performance tools may either fail or become so slow that they are no longer usable. One aspect where this is particularly obvious is the visual presentation of analysis results. For example, a viewer may prove unable to load the data or long response times may compromise interactive usage. And even when none of the above applies, limited display sizes may still prevent a meaningful presentation.

In this article, we describe how CUBE[1], a presentation component for call-path profiles that is primarily used to display runtime summaries and trace-analysis results in the SCALASCA performance tool set[2] for MPI applications, has been enhanced to meet the requirements of large-scale systems. While transitioning to applications running on thousands of processors, scalability limitations appeared in the following four areas:

1. Collation of data sets: Originally, the data corresponding to each application process was written to a separate file and subsequently collated using a sequential program – a procedure that performed poorly due to the large numbers of files being simultaneously created and sequentially processed.

2. Copying data sets between file systems: When the user wanted to display the data on a remote desktop, the relatively large data sets first had to be moved across the network.

3. Memory capacity of the display: As the data sets grew larger with increasing processor counts, they could no longer be loaded into the viewer.

4. Interactive response times: The on-the-fly calculation of aggregate metrics, such as times accumulated across all processes, consumed increasingly more time.

To overcome these limitations, we have redesigned CUBE in various ways. The elements of our solution include (i) a parallel collation scheme that eliminates the need to write thousands of files, (ii) a client-server architecture that avoids copying large data sets between the parallel machine and a potentially remote desktop, (iii) an optimization of the display-internal data structures to reduce the memory footprint of the data, and (iv) optimized algorithms for calculating aggregate metrics to improve interactive response time. The overall result is substantially increased scalability, which is demonstrated with realistic data sets from target applications running on up to 16,384 processors.

The article is organized as follows: In Section 2, we briefly review the CUBE display component. Then, in Section 3, we describe the parallel collation scheme and how it is used in SCALASCA, before we explain the client-server architecture in Section 4. The optimized internal data structures along with the improved aggregation algorithms are presented in Section 5. Finally, we conclude the paper and point to future enhancements in Section 6. Measurement results appear in the text along with the contents they refer to.

2 CUBE

The CUBE (CUBE Uniform Behavioral Encoding) display component is a generic graphical user interface for the presentation of runtime data from parallel programs, which may include both performance data but also data on, for example, runtime errors. It has been primarily designed for use in SCALASCA, but is also used in combination with the TAU performance tool suite[3] and the MPI error detection tool MARMOT[4].

Data model. CUBE displays data corresponding to a single run of an application, which is called an *experiment*. The internal representation of an experiment follows a data model consisting of three dimensions: a metric dimension, a call-tree dimension, and a system

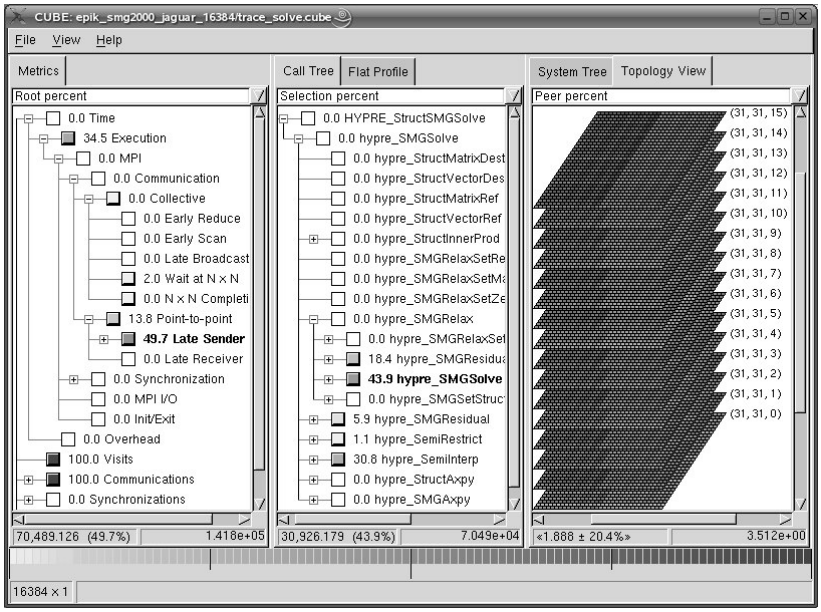


Figure 1. CUBE display showing a SCALASCA trace-analysis report for the ASC SMG2000 benchmark on 16,384 processors. The display shows the hierarchy of performance metrics (left pane), the call tree of the program (middle pane), and the application processes arranged in a three-dimensional virtual topology (right pane).

dimension. Motivated by the need to represent performance behaviour on different levels of granularity as well as to express natural hierarchical relationships among metrics, call paths, or system resources, each dimension is organized in a hierarchy.

The metric hierarchy is intended to represent metrics, such as times or events, and may provide more general metrics at the top (e.g., execution time) and more specialized metrics closer to the bottom (e.g., communication time). The call-tree hierarchy contains the different call paths a program may visit during execution (e.g., main() $\rightarrow$ foo() $\rightarrow$ bar()). In the context of pure message-passing applications, the system hierarchy consists of the three levels machine, (SMP) node, and process. However, in general it can include an additional thread level to represent analysis results from multi-threaded programs. Besides the hierarchical organization, the processes of an application can also be arranged in physical or virtual process topologies, which are part of the data model. A severity function determines how all the tuples (metric m, call path c, process p) of an experiment are mapped onto the accumulated value of the metric m measured while the process p was executing in call path c. Thus, an experiment consists of two parts: a definition part that defines the three hierarchies and a data part representing the severity function.

File format. CUBE experiments can be stored using an XML file format. A file representing a CUBE experiment consists of two parts: the definitions and the severity function values. The severity values are stored as a three-dimensional matrix with one dimension for the metric, one for the call path, and one for the process. CUBE provides a C++ API for creating experiments and for writing them to or loading them from a file.

Display. CUBE's display consists of three tree browsers representing the metric, the program, and the system dimension from left to right (Fig. 1). Since the tree representation of the system dimension turned out to be impractical for thousands of processes, CUBE provides a more scalable two- or three-dimensional Cartesian grid display as an alternative to represent physical or virtual process topologies.

Essentially, a user can perform two types of actions: selecting a tree node or expanding/collapsing a tree node. At any time, there are two nodes selected, one in the metric tree and another one in the call tree. Each node is labeled with a severity value (i.e., a metric value). A value shown in the metric tree represents the sum of a particular metric for the entire program, that is, across all call paths and the entire system. A value shown in the call tree represents the sum of the selected metric across the entire system for a particular call path. A value shown in the system tree or topology represents the selected metric for the selected call path and a particular system entity. To help identify high severity values more quickly, all values are ranked using colours. Due to the vast number of data items, the topological display shows only colours. Exploiting that all hierarchies in CUBE are inclusion hierarchies (i.e., that a child node represents a subset of the parent node), CUBE allows the user to conveniently choose between inclusive and exclusive metrics by collapsing or expanding nodes, respectively. Thus, the display provides two aggregation mechanisms: aggregation across dimensions (from right to left) by selecting a node, and aggregation within a dimension by collapsing a node.

3 Parallel Collation of Input Data

In SCALASCA, the trace analyzer, a parallel program in its own right, scans multiple process-local trace files in parallel to search for patterns of inefficient behaviour. During the analysis, each analysis process is responsible for the trace data generated by one process of the target application. In the earlier version, at the end of the analysis, all analysis processes created their own CUBE file containing the process-local analysis results, which then had to be collated into a single global result file in a sequential postprocessing step. This initial approach had a number of disadvantages. First, every local result file contained exactly the same definition data, causing the analyzer to write large amounts of redundant information. Second, sequentially collating the local results scaled only linearly with the number of processes. This was aggravated by the fact that first writing the local results to disk and then reading them back for collation during the postprocessing phase incurred expensive but actually unnecessary I/O activity.

To overcome this situation, we devised a new mechanism that lets every analysis process send its local result data across the network to a single master process that writes only a single CUBE file containing the collated results from all processes. This has the advantage that creating a large number of intermediate files and writing redundant definition data can be avoided. Although this idea seems trivial, it has to cope with the challenge that the size of the global data set may easily exceed the memory capacity of the collating master. Therefore, to ensure that the master never has to hold more data at a time than its capacity allows, the data is incrementally collected in very small portions and immediately written to the global result file as it arrives.

The CUBE data model stores the severity function in a three-dimensional matrix indexed by the triple (metric, call path, process). Accordingly, our new collation algorithm

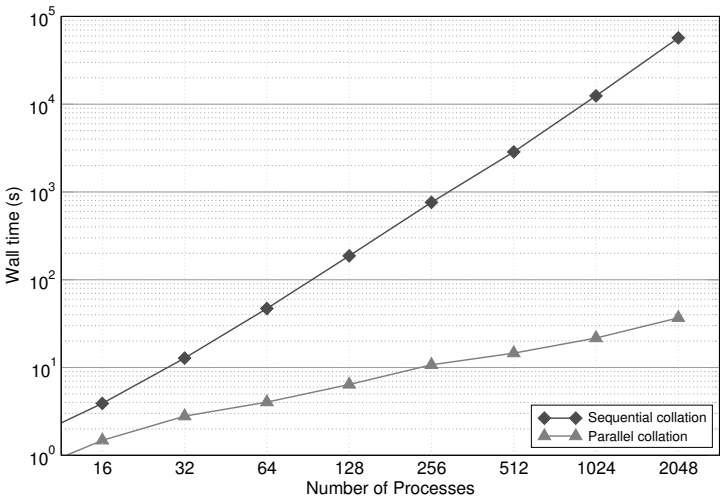


Figure 2. Comparison of trace-analysis result collation times for the ASC SMG2000 benchmark code on Blue Gene/L. The sequential collation was carried out on the front-end node, whereas the parallel collation was carried out on compute nodes.

consists of two nested loops, where the outer one iterates over all defined performance metrics and the inner one iterates over individual call paths. Since we assume that each analysis process stores only local results, the local process dimension for a given (metric, call path) combination consists only of a single entry. During an iteration of the inner loop, just this single value is collected from all analysis processes using an MPI gather operation. Then, the master process immediately collates all these values into the full (metric, call path) submatrix and writes it to the global file before the next iteration starts. In this way, even for 100,000 processes not more than 1 MB of temporary memory is required at a time. To prevent the master process from running out of buffer space when using an MPI implementation with a non-synchronizing gather, we placed a barrier behind each gather operation. To support the incremental write process, a special CUBE writer library is provided, which is implemented in C to simplify linkage to the target application if result files have to be written by a runtime library.

Figure 2 compares analysis result collation times of the initial sequential version and the new parallel version for the ASC SMG2000 benchmark code[5] running on up to 2,048 processes on Blue Gene/L. As can be seen, the sequential collation approach becomes more and more impractical at larger scales, whereas the improved algorithm scales very well. Even for 16,384 processes, the parallel scheme took less than five minutes.

4 Client-Server Architecture

To avoid copying potentially large CUBE data sets from the supercomputer where they have been generated across the network to a remote desktop for visualization, the previously monolithic CUBE display was split into a client and a server part. In this arrangement, the server is supposed to run on a machine with efficient access to the supercomputer's

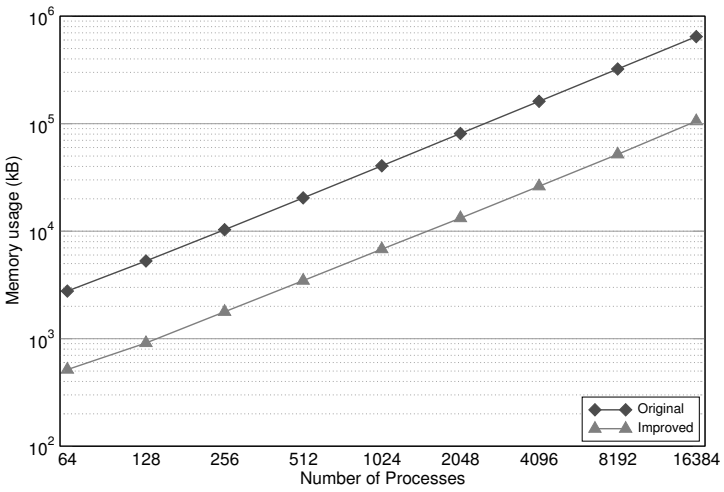


Figure 3. Comparison of the memory footprint of CUBE on a Linux workstation for trace analyses obtained from runs of the ASC SMG2000 benchmark code at a range of scales using the original (map) and the revised (vector) internal data representation.

file system (e.g., the front-end node), while the client, a lightweight display component, runs on the user's desktop system, querying the required data from the server over a BSD socket connection and displaying it without substantial further processing. The data is transferred in relatively small chunks corresponding to the values needed to populate one tree in the display (e.g., the metric tree or the call tree). Whenever the user performs an action requiring the update of one or more trees, the server performs all the necessary calculations including the calculation of aggregate metrics before sending the results to the client. To ensure security of the transmission, we rely on tunnelling the connection through SSH, which is widely available and usually works smoothly even with restrictive firewall configurations.

In addition to solving the problem of copying large amounts of data, the server can also take advantage of a more generous hardware configuration in terms of processing power and memory available on the machine where it is installed. In this way, it becomes possible to hold larger data sets in memory and to perform the relatively compute-intensive calculation of aggregate metrics quicker than on the user's desktop. In a later stage, even a moderate parallelization of the server is conceivable.

The underlying idea of separating the display from the actual processing of the data has also been used in the design of Vampir Server[6], where the trace data is accessed and prepared by a parallel server program, before it is presented to the user in a pure display client.

5 Optimized Internal Data Structures and Algorithms

Since a substantial number of entries in the three-dimensional severity matrix are usually zero (e.g., no communication time in non-MPI functions), the original C++ GUI imple-

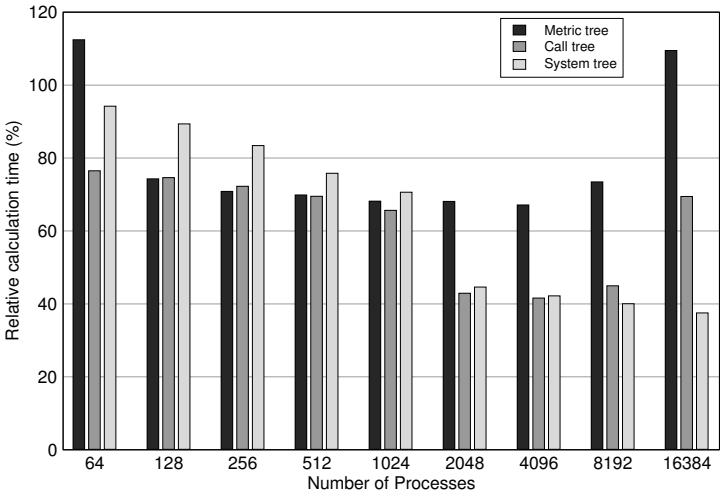


Figure 4. Calculation times of the improved aggregation algorithm for the three different hierarchies in relation to the original implementation (=100%) for trace analyses obtained from runs of the ASC SMG2000 benchmark code at a range of scales. The measurements were taken on a Linux workstation.

mentation used the associative STL container class `std::map` to store only the non-zero entries in the manner of a sparse matrix. The data structure employed a three-level nesting of maps using metric, call-path, and process pointers as keys.

However, experiments showed that in practice the third level, which stores the severity values of all processes for a particular tuple (metric, call path), is usually densely populated. Because STL maps are often implemented using some form of self-balancing binary search tree, this observation implied an opportunity for a substantial reduction of the memory overhead and at the same time for an improvement of the access latency for individual entries. Hence, the lowest nesting level was replaced by `std::vector`. However, this change required the global enumeration of all processes so that the process number could be used as vector index.

Figure 3 compares the memory consumption of the original implementation to the consumption of the revised version on a Linux workstation for trace-analysis data sets obtained from runs of the ASC SMG2000 benchmark code at various scales (similar results have been obtained for other data sets). As can be seen, changing from map to vector at the process level led to a significant reduction of the memory footprint. Depending on the data set and the number of processes, a reduction by factors between three and six could be observed. As a result, the CUBE viewer is now able to display analysis results at much larger scales.

In the course of the above modification, we enumerated also metrics and call paths to improve the performance of some of the algorithms used to calculate aggregate metrics, such as the accumulated time spent in a specific call path including all its children. In particular, the calculation of the inclusive and exclusive severity values for the metric, call path, and system hierarchy was revised by replacing recursions in depth-first order with iterations over the enumerated objects in both directions. The reuse of already calculated

sums at deeper levels of the hierarchy was preserved by making sure that child nodes received a higher index than their parents did.

Figure 4 compares the iterative aggregation algorithms to their recursive counterparts on a Linux workstation, again using the SMG2000 data sets as an example. As can be seen, the new algorithm significantly reduced the aggregation time in the majority of the cases. The fact that the calculation of the metric tree was slower in two cases can be ignored because this tree is calculated only once during the entire display session, whereas the call tree and the system tree have to be frequently re-calculated. In the end, the previously often slow response times that were noticeable especially at large scales could be substantially reduced.

6 Conclusion

In this paper, we have presented a number of measures to improve the scalability of the CUBE display. Although none of the changes required truly novel algorithms, they nonetheless led to tangible results in the form of a much quicker generation of input data sets, a simplified usage on remote desktops, significantly reduced memory requirements, and noticeably improved interactive response times. While certainly adding to the convenience of the overall user experience, the most important accomplishment, however, is that CUBE and the tools depending on it can now be used to study the runtime behaviour of applications running at substantially larger scales.

References

1. F. Song, F. Wolf, N. Bhatia, J. Dongarra and S. Moore, *An algebra for cross-experiment performance analysis*, in: Proc. International Conference on Parallel Processing (ICPP), pp. 63–72, IEEE Society, Montreal, Canada, (2004).
2. M. Geimer, F. Wolf, B. J. N. Wylie and B. Mohr, *Scalable parallel trace-based performance analysis*, in: Proc. 13th European PVM/MPI Users' Group Meeting, vol. **4192** of *LNCS*, pp. 303–312, Bonn, Germany, (Springer, 2006).
3. S. Shende and A. D. Malony, *The TAU parallel performance system*, International Journal of High Performance Computing Applications, **20**, 287–331, (2006).
4. B. Krammer, M. S. Müller and M. M. Resch, *Runtime checking of MPI applications with MARMOT*, in: Proc. of Parallel Computing (ParCo), pp. 893–900, Málaga, Spain, (2005).
5. Accelerated Strategic Computing Initiative, *The SMG2000 benchmark code*, (2001). http://www.llnl.gov/asc/purple/benchmarks/limited/smg/
6. H. Brunst and W. E. Nagel, *Scalable performance analysis of parallel systems: concepts and experiences*, in: Proc. Parallel Computing Conference (ParCo), pp. 737–744, Dresden, Germany, (2003).

Coupling DDT and Marmot for Debugging of MPI Applications

Bettina Krammer[1], **Valentin Himmler**[1], **and David Lecomber**[2]

[1] HLRS - High Performance Computing Center Stuttgart
Nobelstrasse 19, 70569 Stuttgart, Germany
E-mail: {krammer, himmler}@hlrs.de

[2] Allinea Software,
The Innovation Centre, Warwick Technology Park, Gallows Hill
Warwick, CV34 6UW, UK
E-mail: david@allinea.com

Parallel programming is a complex, and, since the multi-core era has dawned, also a more common task that can be alleviated considerably by tools supporting the application development and porting process. Existing tools, namely the MPI correctness checker Marmot and the parallel debugger DDT, have so far been used on a wide range of platforms as stand-alone tools to cover different aspects of correctness debugging. In this paper we will describe first steps towards coupling these two tools to provide application developers with a powerful and user-friendly environment.

1 Introduction

The Message Passing Interface (MPI) has been a commonly used standard[1,2] for writing parallel programs for more than a decade, at least within the High Performance Computing (HPC) community. With the arrival of multi-core processors, parallel programming paradigms such as MPI or OpenMP will become more popular among a wider public in many application domains as software needs to be adapted and parallelised to exploit fully the processor's performance. However, tracking down a bug in a distributed program can turn into a very painful task, especially if one has to deal with a huge and hardly comprehensible piece of legacy code.

Therefore, we plan to couple existing tools, namely the MPI correctness checker, Marmot, and the parallel debugger, DDT, to provide MPI application developers with a powerful and user-friendly environment. So far, both tools have been used on a wide range of platforms as stand-alone tools to cover different aspects of correctness debugging. While (parallel) debuggers are a great help in examining code at source level, e.g. by monitoring the execution, tracking values of variables, displaying the stack, finding memory leaks, etc., they give little insight into *why* a program actually gives wrong results or crashes when the failure is due to incorrect usage of the MPI API. To unravel such kinds of errors, the Marmot library has been developed. The tool checks at run-time for errors frequently made in MPI applications, e.g. deadlocks, the correct construction and destruction of resources, etc., and also issues warnings in the case of non-portable constructs.

In the following sections, we will shortly describe both tools and discuss our first considerations and results towards integrating the two of them.

1.1 DDT - Distributed Debugging Tool

Allinea Software's DDT[21] is a source-level debugger for scalar, multi-threaded and large-scale parallel C, C++ and Fortran codes. It provides complete control over the execution of a job and allows the user to examine in detail the state of every aspect of the processes and threads within it.

Control of processes is aggregated using groups of processes which can be run, single stepped, or stopped together. The user can also set breakpoints at points in the code which will cause a process to pause when reaching it.

When a program is paused, the source code is highlighted showing which lines have threads on them and, by simply hovering the mouse, the actual threads present are identified. By selecting an individual process, its data can be interrogated - for example to find the local variables and their values, or to evaluate specific expressions.

There are many parts of the DDT interface that help the user get a quick understanding of the code at scale - such as the parallel stack view which aggregates the call stacks of every process in a job and displays it as tree, or the cross-process data comparison tools.

Specifically to aid MPI programming, there is a window that examines the current state of MPI message queues - showing the send, receive and unexpected receive buffers. However, this can only show the current state of messages - and bugs due to the historic MPI behaviour are not easy to detect with only current information.

1.2 Marmot - MPI Correctness Checker

Marmot[9,10,11,14] is a library that uses the so-called PMPI profiling interface to intercept MPI calls and analyse them during runtime. It has to be linked to the application in addition to the underlying MPI implementation, not requiring any modification of the application's source code nor of the MPI library. The tool checks if the MPI API is used correctly and checks for errors frequently made in MPI applications, e.g. deadlocks, the correct construction and destruction of resources, etc. It also issues warnings for non-portable behaviour, e.g. using tags outside the range guaranteed by the MPI-standard. The output of the tool is available in different formats, e.g. as text log file or html/xml, which can be displayed and analysed using a graphical interface. Marmot is intended to be a portable tool that has been tested on many different platforms and with many different MPI implementations.

Marmot supports the complete MPI-1.2 standard for C and Fortran applications and is being extended to also cover MPI-2 functionality[12,13].

Figure 1 illustrates the design of Marmot. Local checks including verification of arguments such as tags, communicators, ranks, etc. are performed on the client side. An additional MPI process (referred to as *debug server*) is added for the tasks that cannot be handled within the context of a single MPI process, e.g. deadlock detection. Another task of the debug server is the logging and the control of the execution flow. Every client has to register at the debug server, which gives its clients the permission for execution in a round-robin way. Information is transferred between the original MPI processes and the debug server using MPI.

In order to ensure that the debug server process is transparent to the application, we map `MPI_COMM_WORLD` to a Marmot communicator that contains only the application processes. Since all other communicators are derived from `MPI_COMM_WORLD` they will also automatically exclude the debug server process. This mapping is done at start-up time

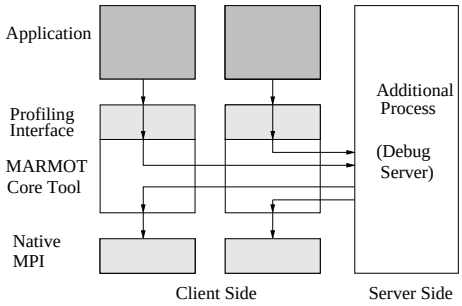


Figure 1. Design of Marmot.

in the `MPI_Init` call, where we also map all other predefined MPI resources, such as groups or datatypes, to our own Marmot resources. When an application constructs or destructs resources during run-time, e.g. by creating or freeing a user-defined communicator, the Marmot maps are updated accordingly. Having its own book-keeping of MPI resources, independently of the actual MPI implementation, Marmot can thus verify correct handling of resources.

1.3 Related Work

Debugging MPI programs can be addressed in various ways. The different solutions can be roughly grouped in four different approaches:

1. *classical debuggers:* Among the best-known parallel debuggers are the commercial debuggers DDT[21] and Totalview[22]. It is also possible to attach the freely available debugger gdb[20] to single MPI processes.
2. The second approach is to provide a *special debug version* of the MPI library, for instance for checking collective routines, e.g. mpich[7] or NEC MPI[18].
3. Another possibility is to develop tools for *run-time analysis* of MPI applications. Examples of such an approach are MPI-CHECK[16], Umpire[19] and Marmot[9]. MPI-CHECK is currently restricted to Fortran code and performs argument type checking or finds problems such as deadlocks. Like Marmot, Umpire[19] uses the profiling interface.
4. The fourth approach is to collect all information on MPI calls in a trace file for *post-mortem analysis*, which can be analysed by a separate tool after program execution[15,17]. A disadvantage with this approach is that such a trace file may be very large. However, the main problem is guaranteeing that the trace file is written in the presence of MPI errors, because the behaviour after an MPI error is implementation defined.

2 Coupling DDT and Marmot

The ultimate goal of coupling both tools is to create a common software development environment with debugging and correctness checking facilities at the same time, comple-

menting each other and, thus, offering the opportunity to detect a maximum number of bugs on different levels, for example, bugs being due to incorrect usage of the MPI API, memory leaks, etc. In order to do so, a number of considerations have to be taken into account and adaptations have to be implemented in both tools.

- **Marmot's `MPI_Init`:** As described in Section 1.2, Marmot requires one additional process for the debug server, which is always running on the process with highest rank. That means while the first n processes call our redefined `MPI_Init`, i.e. call `PMPI_Init`, take part in the special Marmot start-up and return from the `MPI_Init` call to proceed with the application's code, the debug server (process $n + 1$) always remains in `MPI_Init` to execute Marmot's debug server code. When executing an application with DDT and Marmot, DDT's environment variable `DDT_MPI_INIT` has to be set to the value `PMPI_Init` to circumvent problems with attaching DDT also to this last process, because the DDT start-up procedure normally requires a process to return from the `MPI_Init` call.

  While the debug server process is transparent to the application in the Marmot-only approach, i.e. from the user's point of view, showing silent signs of life by announcing errors and producing a logfile without being visible as additional MPI process, it is in the coupled approach currently also displayed in DDT's graphical interface and is, thus, visible to the user in the same way as one of the original application's MPI processes. As this may be confusing to users, the ultimate goal will be to hide Marmot's debug server process within DDT completely and to allow steering of Marmot through a plugin.

- **Marmot's breakpoints:** One reason for currently still displaying this debug server process – though it has nothing to do with the actual application itself – is that it allows users to set breakpoints that will be hit when Marmot finds an error or issues a warning. For this purpose, we implemented simple functions, named e.g. `insertBreakpointMarmotError`, that are called by Marmot's debug server in such an event, see Fig.2 left. The screenshot on the bottom shows the application's code pausing at the erroneous line, with the last process pausing in `MPI_Init` as explained above. As it has control over the execution flow, hitting one of the debug server's special breakpoints implies that the application processes cannot continue with the erroneous MPI call nor perform any further MPI calls, respectively.

  As part of the integration with Marmot, DDT will automatically add breakpoints into these "stub" functions and will thus alert the user to a Marmot-detected error or warning by pausing the program.

- **Error dialogues:** Having detected an error with Marmot, the question is how to get the warning message across to the user within DDT. Currently, a user will find such messages in Marmot's log file, or in DDT's variables or stderr windows at runtime, as shown in Fig.3.

  Simple text and error code are shown in DDT's local variables panel at the breakpoints inside the "stub" error functions - these will be passed as arguments to those functions. Extra integration is planned, such as displaying more detailed information and providing extra windows to display the context of errors more clearly.

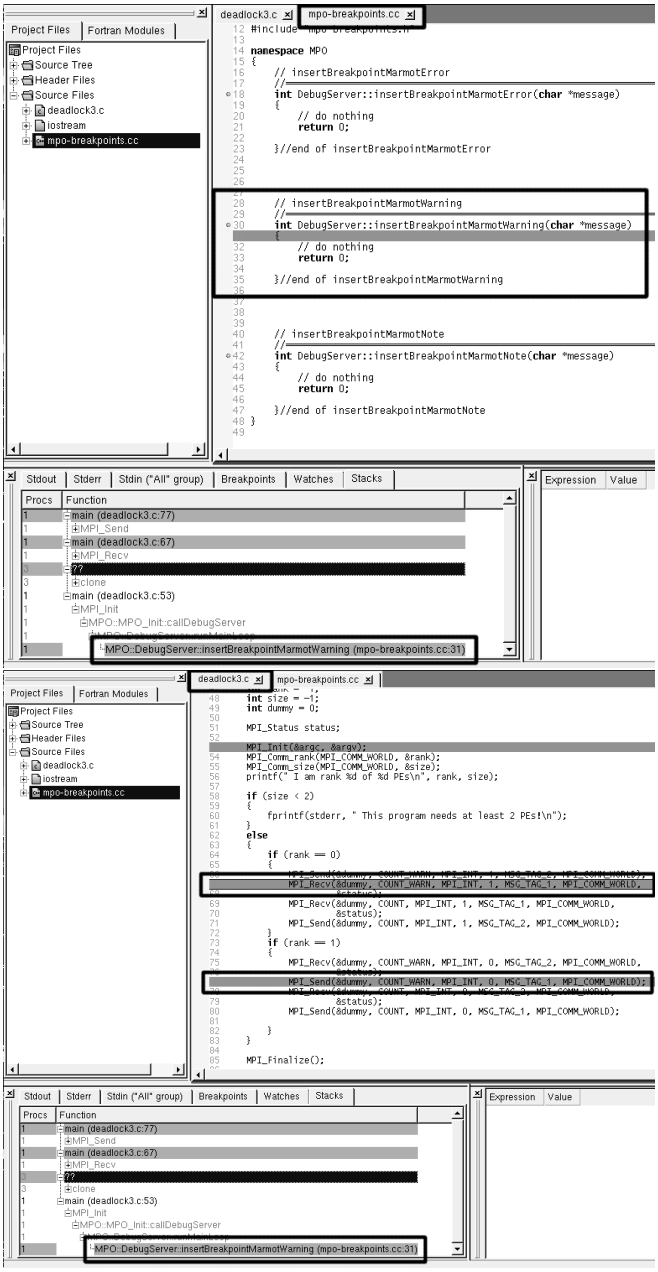


Figure 2. Setting breakpoints in Marmot's debug server process (top) causes program to stop at an error/warning detected by Marmot - in this case, a warning for using `COUNT_WARN`$= 0$ in send/recv calls (bottom)

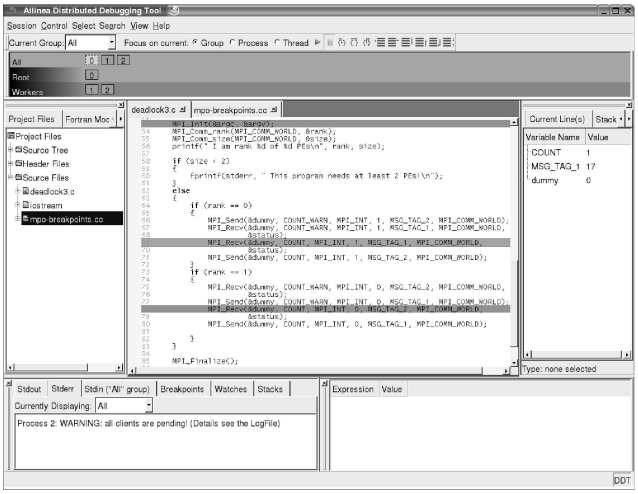


Figure 3. Deadlock warning from Marmot

- **Dynamic linking:** Currently, Marmot is built as static libraries per default. While this is an advantage concerning the performance, it is a drawback in the integration process of Marmot and DDT. Beside the fact that the size of statically linked applications can be very large compared to those dynamically linked, a more severe issue is, that as soon as an application is linked statically against the Marmot libraries, this application cannot be run without Marmot. Our goal is to be able to activate Marmot for a given run within DDT, and to disable it for another, using one and the same binary. The trick is to preload the dynamic Marmot libraries by using the environment variable `LD_PRELOAD`.

  This will tell the dynamic linker to first look in the Marmot libraries for any MPI call, provided that also the MPI libraries themselves are dynamic. For switching off Marmot, the variable will simply be unset. Note that in this case neither at compile time nor at link time the application has to be aware of the very existence of Marmot. Only at runtime the dynamic linker decides whether the MPI calls will be intercepted by Marmot or not. Switching Marmot on and off could then be as easy as ticking a checkbox in the DDT GUI, which consequently takes care of the preloading and automatically adds one extra process for the debug server.

  Although not all MPIs export the `LD_PRELOAD` setting to all processes in a job seamlessly, DDT already has the mechanism to do this for every MPI - and therefore extending it to preload the Marmot libraries will be a simple task.

3 First Results

Running applications with Marmot in conjunction with DDT has been tested on various HLRS clusters using different MPI Implementations, e.g. different vendor MPIs or Open Source implementations such as mpich[3,4] or Open MPI[5,6].

As a proof of concept, some simple programs were used for testing. For instance, the example shown in Fig.2 and 3 first triggers a warning by using send and receive calls with count 0 and then an error by using send and receive calls in wrong order (deadlock).

Building and preloading shared libraries were successfully tested on an Intel Xeon cluster with Open MPI 1.2.3 and IntelMPI 3.0, using the Intel Compiler 10.0. For both MPI implementations the build process generally involves the following steps: generating objects with *position independent code* and building shared libraries from these object files.

A given MPI application is invoked e.g. by the command `mpirun -np <n> ./myMPIApp`, where n denotes the number of processes. The same application can be run with Marmot by setting the appropriate environment variable and adjusting the number of processes:
`env LD_PRELOAD="<marmotlib>" mpirun -np <n+1> ./myMPIApp`.

4 Concluding Remarks

We have presented the DDT parallel debugger and the Marmot MPI correctness checker, which have been used successfully as stand-alone tools so far. Recently we have made first efforts to combine both tools so that MPI application developers get better insight into *where* and *why* their programs crash or give wrong results. The approach taken looks promising and could be verified using simple test programs.

Future work includes tests with real applications and technical improvements, e.g. a tighter integration of the display of warnings and message queues, thus being more user-friendly, or enhancements in Marmot's build process.

Acknowledgements

The research presented in this paper has partially been supported by the European Union through the IST-031857 project "int.eu.grid"[14] (May 2006 – April 2008) and by the German Ministery for Research and Education (BMBF) through the ITEA2 project "ParMA"[8] (June 2007 – May 2010). Marmot is being developed by HLRS in collaboration with ZIH Dresden.

References

1. Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, (1995). http://www.mpi-forum.org
2. Message Passing Interface Forum, *MPI-2: Extensions to the Message Passing Interface*, (1997). http://www.mpi-forum.org
3. mpich Homepage. www.mcs.anl.gov/mpi/mpich
4. W. Gropp, E. L. Lusk, N. E. Doss and A. Skjellum, *A high-performance, portable implementation of the MPI Message Passing Interface standard*. Parallel Computing, **22**, 789–828, (1996).
5. Open MPI Homepage. http://www.open-mpi.org

6. E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham and T. S. Woodall, *Open MPI: goals, concept, and design of a next generation MPI implementation*, in: 11th European PVM/MPI, LNCS **3241**, pp. 97–104, (Springer, 2004).
7. E. Lusk, C. Falzone, A. Chan and W. Gropp, *Collective error detection for MPI collective operations*, in: 12th European PVM/MPI, LNCS **3666**, pp. 138–147, (Springer, 2005).
8. ParMA: Parallel Programming for Multi-core Architectures - ITEA2 Project (06015). http://www.parma-itea2.org/
9. B. Krammer, K. Bidmon, M. S. Müller and M. M. Resch, *MARMOT: An MPI analysis and checking tool.*, in: PARCO 2003, Dresden, Germany, (2003).
10. B. Krammer, M. S. Müller and M. M. Resch, *MPI application development using the analysis tool MARMOT*, in: ICCS 2004, LNCS **3038**, pp. 464–471. (Springer, 2004).
11. B. Krammer, M. S. Mueller and M. M. Resch, *Runtime checking of MPI applications with MARMOT*. in: *ParCo 2005*, Malaga, Spain, (2005).
12. B. Krammer, M. S. Müller and M. M. Resch, *MPI I/O analysis and error detection with MARMOT*, in: Proc. EuroPVM/MPI 2004, Budapest, Hungary, September 19–22, 2004, LNCS vol. **3241**, pp. 242–250, (Springer, 2004).
13. B. Krammer and M. M. Resch, *Correctness checking of MPI one-sided communication using MARMOT*, in: Proc. EuroPVM/MPI 2006, Bonn, Germany, September 17–20, 2006, LNCS vol. **4192**, pp. 105–114, (Springer, 2006).
14. B. Krammer, *Experiences with MPI application development within int.eu.grid: interactive European grid project*, in: Proc. GES2007, Baden-Baden, Germany, May 2–4, (2007).
15. D. Kranzlmüller, *Event graph analysis for debugging massively parallel programs*, PhD thesis, Joh. Kepler University Linz, Austria, (2000).
16. G. Luecke, Y. Zou, J. Coyle, J. Hoekstra and M. Kraeva, *Deadlock detection in MPI programs*, Concurrency and Computation: Practice and Experience, **14**, 911–932, (2002).
17. B. Kuhn, J. DeSouza and B. R. de Supinski, *Automated, scalable debugging of MPI programs with Intel Message Checker*, in: SE-HPCS '05, St. Louis, Missouri, USA, (2005). http://csdl.ics.hawaii.edu/se-hpcs/papers/11.pdf
18. J. L. Träff and J. Worringen, *Verifying collective MPI calls*, in: 11th European PVM/MPI, LNCS vol. **3241**, pp. 18–27, (Springer, 2004).
19. J. S. Vetter and B. R. de Supinski, *Dynamic software testing of MPI applications with Umpire*, in: SC 2000, Dallas, Texas, (ACM/IEEE, 2000). CD-ROM.
20. gdb. The GNU Project Debugger. http://www.gnu.org/manual/gdb.
21. DDT. http://www.allinea.com/index.php?page=48
22. Totalview. http://www.totalviewtech.com/productsTV.htm.

Compiler Support for Efficient Instrumentation

Oscar Hernandez[1], **Haoqiang Jin**[2], **and Barbara Chapman**[1]

[1] Computer Science Department
University of Houston
501 Phillip G. Hoffman, Houston, Texas
E-mail: {oscar, chapman}@cs.uh.edu

[2] NASA Advanced Supercomputing Division, M/S 258-1
NASA Ames Research Center, Moffet Field, CA
E-mail: hjin@nas.nasa.gov

We are developing an integrated environment for application tuning that combines robust, existing, open source software - the OpenUH compiler and performance tools. The goal of this effort is to increase user productivity by providing an automated, scalable performance measurement and optimization system. The software and interfaces that we have created has enabled us to accomplish a scalable strategy for performance analysis, which is essential if performance tuning tools are to address the needs of emerging very large scale systems. We have discovered that one of the benefits of using compiler technology in this context is that it can direct the performance tools to decide which regions of code they should measure, selectively considering both coarse grain and fine grain regions (control flow level) of the code. Using a cost model embedded in the compiler's interprocedural analyzer, we can statically assess the importance of a region via an estimated cost vector that takes its size and the frequency with which it is invoked into account. This approach enables us to set different thresholds that determine whether or not a given region should be instrumented. Our strategy has been shown to significantly reduce overheads for both profiling and tracing to acceptable levels. In this paper, we show how the compiler helps identify performance problems by illustrating its use with the NAS parallel benchmarks and a cloud resolving model code.

1 Introduction

Tracing and profiling play a significant role in supporting attempts to understand the behaviour of an application at different levels of detail and abstraction. Both approaches can be used to measure a variety of different kinds of events, where memory profiling/tracing are the most expensive. Profiling aggregates the results of event tracking for phases in the code or for the entire execution of the program, while tracing can keep track of every single instance of an event at run time, and captures patterns over time. Tools such as TAU[1] support both profiling and tracing. KOJAK[2], on the other hand, uses tracing to find performance patterns. This can enable the tools to detect communication and synchronizations problems for both MPI and OpenMP codes, as well as giving support for hardware counter patterns.

When applications are deployed on systems with thousand of processors, it is critical to provide scalable strategies for gathering performance data. A suitable approach must minimize perturbations and reduce the amount of data gathered, while at the same time providing a significant coverage of the important code regions. Studies have shown that the overheads of profiling and tracing[3] are significant when a shared file system becomes a contention point. Tracing can lead to very large trace files and may degrade overall system performance, negatively affecting other applications sharing the same resources.

The use of compilers to support instrumentation has been limited because of portability issues, lack of standard APIs, selective instrumentation and user control. A compiler that has all the necessary language and target support is desired. PDT[4] is a toolkit that was designed in an attempt to overcome this. It gathers static program information via a parser and represents it in a portable format suitable for use in source code instrumentation. This approach does not permit full exploitation of compiler technology; in particular, it does not provide any insight into the compiler translation, or enable the evaluation of any assumptions the compiler makes when deciding to apply a given transformation (which includes metrics used to evaluate static cost models) that may improve the interpretation of performance data. In this paper we show how an open source compiler can be enhanced to enable it to perform selective instrumentation and, as a result, significantly reduce performance measurement overheads.

2 Related Work

Programs may be instrumented at different levels of a program's representation: at the source code level, at multiple stages during the compiler translation, at object code level, and embedded within the run time libraries. All of these techniques have inherent advantages and disadvantages with respect to their ability to perform the instrumentation efficiently and automatically, the type of regions that can be instrumented, the mapping of information to the source code, and the support for selective instrumentation. Among the many performance tools with instrumentation capabilities are KOJAK[2] and TAU[1], which relies on PDT[4] and OPARI[2]to perform source code instrumentation, and Dyninst[5] which performs object code instrumentation. Open SpeedShop[6] and Paradyn support Dyninst and DPCL but instrumenting at the object code level implies a loss of the semantics of the program like losing loop level information. The EP-Cache project [7] uses the NAG compiler to support instrumentation at procedure and loop levels, but is a closed and source system it has not explored ways to improve instrumentation via compiler analysis. Intel's Thread Checker[8] performs instrumentation to detect semantic problems in an OpenMP application but it lacks scalability as it heavily instruments memory references and synchronization points. Other tools like Perfsuite[9], Sun Analyzer and Vtune[10] rely on sampling to provide profile data. Although sampling is a low overhead approach it requires extra system resources, in some cases needing extra threads/processors to support processor monitoring units, and it focuses on low level data gathering.

3 OpenUH

The OpenUH[11] compiler is a branch of the open source Open64 compiler suite for C, C++, and Fortran 95, supporting the IA-64, IA-32e, and Opteron Linux ABI and standards. OpenUH supports OpenMP 2.5 and provides complete support for OpenMP compilation and its runtime library. The major functional parts of the compiler are the front ends, the inter-language interprocedural analyzer (IPA) and the middle-end/back end, which is further subdivided into the loop nest optimizer, auto-parallelizer (with an OpenMP optimization module), global optimizer, and code generator. OpenUH has five levels of a tree-based intermediate representation (IR) called WHIRL to facilitate the implementation

of different analysis and optimization phases. They are classified as being Very High, High, Mid, Low, and Very Low levels, respectively. Most compiler optimizations are implemented on a specific level of WHIRL, for example interprocedural array region and dependence analysis analysis is implemented in the high level whirl. Our efforts form part of the official source code tree of Open64 (http://www.open64.net), which make results available to the entire community. It is also directly and freely available via the web (http://www.cs.uh.edu/ openuh).

4 Instrumentation

OpenUH provides a complete compile-time instrumentation module covering different compilation phases and different program scopes. Advanced feedback-guided analyses and optimizations are part of the compiler for sequential tuning. We have designed a compiler instrumentation API that can be used to instrument a program. It is language independent to enable it to interact with performance tools such as TAU and KOJAK and support the instrumentation of Fortran, C and C++.

Compile-time instrumentation has several advantages over both source-level and object-level instrumentation. Compiler analysis can be used to detect regions of interest before instrumenting and measuring certain events to support different performance metrics. Also, the instrumentation can be performed at different compilation phases, allowing some optimizations to take place before the instrumentation. These capabilities play a significant role in the reduction of instrumentation points, improve users' ability to deal with program optimizations, and reduce the instrumentation overhead and size of performance trace files.

The instrumentation module in OpenUH can be invoked at six different phases during compilation, which come before and after three major stages in the translation: interprocedural analysis, loop nest optimizations, and SSA/DataFlow optimizations. Figure 1 shows the different places in the compilation process where instrumentation can be performed, along with the corresponding intermediate representation level.

For each phase, the following kinds of user regions can be instrumented: functions, conditional branches, switch statements, loops, callsites, and individual statements. Each user-region type is further divided into subcategories when possible. For instance, a loop may be of type *do loop*, *while loop*. Conditional branches may be of type *if then*, *if then else*, *true branch*, *false branch*, or *select*. MPI operations are instrumented via PMPI so that the compiler does not instrument these callsites. OpenMP constructs are handled via runtime library instrumentation, where it captures the fork and joint events, implicit and explicit barriers[12]. Procedure and control flow instrumentation is essential to relate the MPI and OpenMP-related output to the execution path of the application, or to understand how constructs behave inside these regions.

The compiler instrumentation is performed by first traversing the intermediate representation of an input program to locate different program constructs. The compiler inserts instrumentation calls at the start and exit points of constructs such as procedures, branches and loops. If a region has multiple exit points, they will all be instrumented; for example, goto, stop or return statements may provide alternate exit points. A brief description of the API can be found in Ref.[13].

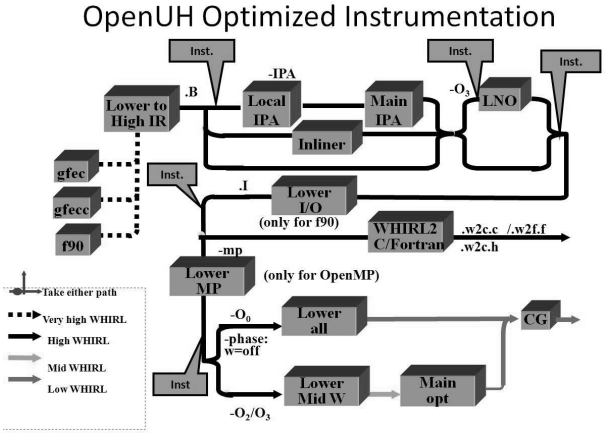


Figure 1. Multiple stages of compiler instrumentation.

5 Selective Instrumentation Analysis

We take advantage of the interprocedural analysis within the compiler to reduce the number of instrumentation points. Here the compiler performs inlining analysis, which attempts to determine where the program will benefit from replacing a procedure call with the actual code of the called procedure's body. As part of this, the compiler must determine if a procedure is invoked frequently and whether the caller and callee meet certain size restrictions in order to avoid code bloat. We adapted this methodology to enable selective instrumentation, for which we have defined a cost model in the form of scores to evaluate the above conditions. We avoid instrumenting any procedure that meets the criteria for inlining. We do instrument procedures that are significant and are infrequently called and have large bodies. We call a procedure significant if it contains many callsites and is well connected in the callgraph.

Our cost model consists of three metrics in the form of instrumentation scores. The first metric computes the *weight* of the procedure using the compiler's control flowgraph, which is defined as $PU_{weight} = (5 * total\ basic\ blocks) + total\ statements + total\ callsites$. As can be seen, this metric puts emphasis on procedures with multiple basic blocks. If runtime information is known, the PU_{weight} formula will use the number of times or *effective* number of basic blocks, statements and callsites invoked at runtime. The other metric we use is the frequency with which a procedure is invoked, taking their position within loop nests into account. The formula used is: $PU_{loop-score} = (100 - loopnest\ level) * 2048$. This formula gives higher scores to procedures invoked with fewer nesting levels. The third metric is a score that quantifies how many calls exist within a procedure. $PU_{callsite-score} = (callsites\ in\ callee) * 2048^2$. This formula gives a small score to procedures invoked as leaf nodes in the callgraph or that have few calling edges. The constants of the formulas were determined empirically based on the inlining algorithm of the compiler which was tunned to avoid under or over inlining. Our assumption here is that important procedures are connected with others, and thus are associated with several edges in the callgraph. It is important to note that we will not count callsites to procedures

that are not going to be instrumented. The overall score used to decide whether we will instrument a procedure is as follows:

$$Instrumentation\ Score = PU_{weight} + PU_{loop-score} + PU_{callsite-score}$$

Our strategy for computing this score means that we will favour procedures with large bodies, invoked few times and with multiple edges connecting them to other procedures in the callgraph. We avoid the instrumentation of small procedures invoked at high loopnest levels and that are leaf nodes in the callgraph.

With this score we then define a threshold that can be changed depending on the size of the application, in order to avoid over or under-instrumentation. Also, we generalize our approach to take into consideration the lowest score that a procedure has from its different callsites. If a score for a procedure is below a pre-defined threshold, the procedure will not be instrumented.

$$Instrument\ Procedure < Threshold < Do\ not\ Instrument$$

Table 1 contains the instrumentation scores for some of the procedures in the BT OpenMP benchmark from the NAS parallel benchmarks. It shows that procedures corresponding to leaf nodes in the callgraph have a low instrumentation score. Heavily connected nodes in the callgraph (those that have several callsites) are among the ones with the highest score, as is to be expected. If we define a threshold to be 204800 (the score of an empty procedure with no callsites being invoked outside a loop), then we will not instrument the following procedures: *matvecsub, binvchrs, matmul_sub, lhsinit, exact_solution* and do instrument the following procedures: *adi, x_solve, y_solve, z_solve, main.*

Table 1. Instrumentation scores for the BT OpenMP benchmark

Proc.	Weight	Loop Score (level)	Callsite Score (sites)	Inst. Score
matvecsub	23	198656(3)	0(0)	198679
binvchrs	240	198656(3)	0(0)	198896
binvrhs	115	198656(3)	0(0)	198771
matmul_sub	27	198656(3)	0(0)	198683
lhsinit	57	198656(3)	0(0)	198713
exact_solution	23	196608(4)	0(0)	196631
adi	45	202752(1)	20971520(5)	21174317
x_solve	278	204800(0)	41943040(10)	42148118
y_solve	278	204800(0)	41943040(10)	42148118
z_solve	278	204800(0)	41943040(10)	42148118
main	459	204800(0)	58720256(14)	58720256

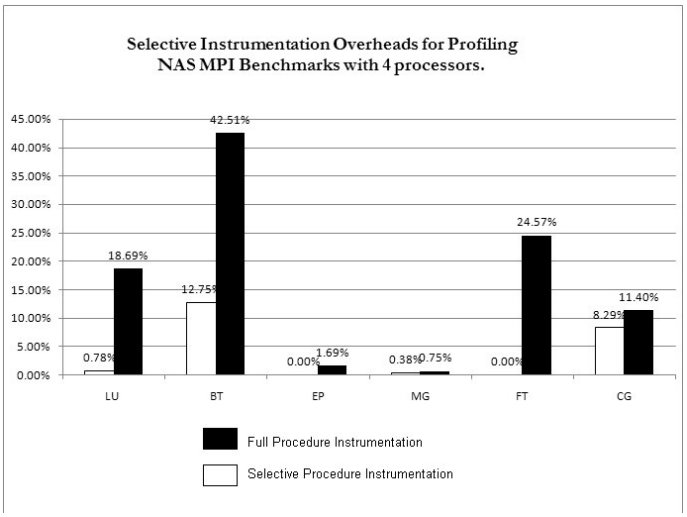


Figure 2. Selective instrumentation in the NAS MPI Benchmarks.

6 Experiments

We applied the the selective instrumentation algorithm to six benchmarks from the NAS parallel benchmarks in both MPI and OpenMP implementations. Our experiments used the class A problem size and were conducted on an SGI Altix 4000 with 16 Itanium 2, 1.6 Ghz processors, and the NUMAlink interconnect. Figures 2 and 3 shows the overheads incurred when performing full procedure instrumentation versus performing selective instrumentation when using the TAU profiling libraries. For the MPI benchmarks we turned off TAU throttle. The purpose of THROTTLE is to disable the instrumentation library at runtime when a procedure reaches a given threshold.

Because of the massive overheads of full instrumentation in the case of OpenMP, we turned on the TAU THROTTLE for NUMCALLS=300 and PERCALL=300000 environment variables. Selective instrumentation reduces the overheads by an average of 90 times in the OpenMP version even when TAU THROTTLE is enabled in full instrumentation. The FT-OMP benchmark has particularly high overheads compared to the other benchmarks.

This is because it invokes the procedures *fttz2, cfftz* a significant number of times. We note that *cfftz* calls *fttz2*. Our selective instrumentation score for *cfftz* is 202817 and *fftz2* is 198772 , which is below our instrumentation threshold. As a result, we also do not instrument the *cfftz* procedure since it only has one callsite that is not being instrumented. In the CG benchmark we do not instrument the procedures *buts, jaci, blts, jacld and exact* which are leaf nodes in the callgraph that are invoked many times and have small weights. The higher overheads in OpenMP instrumentation versus MPI are due to memory contentions and locks on where the performance data is stored and modified. In the Altix this contention becomes a problem due to cache line invalidations and remote memory accesses due to the cc-NUMA architectures.

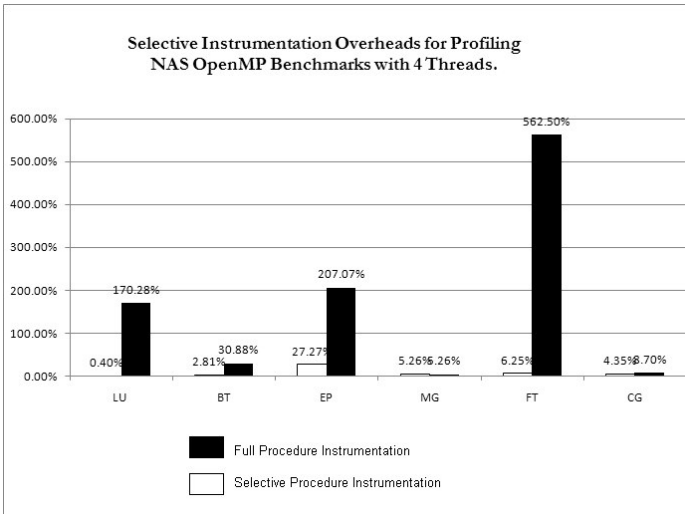


Figure 3. Selective instrumentation in the NAS OpenMP Benchmarks.

We also applied our algorithm to an MPI implementation of a Cloud-Resolving Model code[14] using a grid size of 104x104x42 and 4 MPI processes for our experiment. Our selective algorithm determined that we should not instrument three leaf procedures in the callgraph. We were able to reduce the profiling overhead from 51% to 3% based on this alone.

7 Conclusions and Future Work

In this paper we have presented a selective instrumentation algorithm that can be implemented in a compiler by adapting a typical strategy for performing inlining analysis. In the examples presented here, selective instrumentation based upon this algorithm was able to reduce profiling overheads by 90 times on average in the NAS OpenMP parallel benchmarks and 17 times for the cloud formation code. Our future work will combine feedback-directed optimizations and the inlining analysis to further improve selective instrumentation. We will also continue to explore opportunities to enhance the working of performance tools via direct compiler support.

8 Acknowledgements

We would like to thank NSF for supporting this work[a]. We will also want to thank Bob Hood and Davin Chang from CSC at Nasa Ames for providing us the Altix System to perform the experiments.

[a] This work is supported by the National Science Foundation, under contract CCF-0444468

References

1. Allen D. Malony, Sameer Shende, Robert Bell, Kai Li, Li Li and Nick Trebon, *Advances in the TAU performance system*, Performance analysis and grid computing, pp. 129–144, (2004).
2. B. Mohr and F Wolf, *KOJAK - a tool set for automatic performance analysis of parallel applications*, in: Proc. European Conference on Parallel Computing (EuroPar), pp. 1301–1304, (2003).
3. K. Mohror and K. L. Karavanic, *A study of tracing overhead on a high-performance linux cluster*, in: PPoPP '07: Proc. 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 158–159, (ACM Press, New York, 2007).
4. K. A. Lindlan, J. E. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh and C. Rasmussen, *A tool framework for static and dynamic analysis of object-oriented software with templates*, in: Supercomputing, (2000).
5. B. Buck and J. K. Hollingsworth, *An API for runtime code patching*, in: International Journal of High Performance Computing Applications, **14**, 317–329, (2000).
6. M. Schulz, J. Galarowicz and W. Hachfeld, *Open—SpeedShop: open source performance analysis for Linux clusters*, in: SC '06: Proc. 2006 ACM/IEEE Conference on Supercomputing, p. 14, (ACM Press, NY, 2006).
7. E. Kereku and M. Gerndt, *Selective instrumentation and monitoring,*, in: International Workshop on Compilers for Parallel Computers, CPC'04, (2004).
8. P. Petersen and S. Shah, *OpenMP support in the Intel thread checker*, in: WOMPAT, pp. 1–12, (2003).
9. R. Kufrin, *PerfSuite: an accessible, open source, performance analysis environment for Linux*, in: 6th International Conference on Linux Clusters (LCI-2005), Chapel Hill, NC, (2005).
10. J. H. Wolf, *Programming methods for the Pentium III processor's streaming SIMD extensions using the VTune performance enhancement environment*, Intel Technology Journal, no. Q2, 11, (1999).
11. C. Liao, O. Hernandez, B. Chapman, W. Chen and W. Zheng, *OpenUH: an optimizing, portable OpenMP compiler*, in: 12th Workshop on Compilers for Parallel Computers, (2006).
12. V. Bui, O. Hernandez, B. Chapman, R. Kufrin, D. Tafti and P. Gopalkrishnan, *Towards an implementation of the OpenMP collector API*, in: PARCO, (2007).
13. O. Hernandez, F. Song, B. Chapman, J. Dongarra, B. Mohr, S. Moore and F. Wolf, *Instrumentation and compiler optimizations for MPI/OpenMP applications*, in: International Workshop on OpenMP (IWOMP 2006), (2006).
14. H.-M. H. Juang, W.-K. Tao, X. Zeng, C.-L. Shie, S. Lang and J. Simpson, *Implementation of a message passing interface into a cloud-resolving model for massively parallel computing*, in: Monthly Weather Review., (2004).

Comparing Intel Thread Checker and Sun Thread Analyzer

Christian Terboven

Center for Computing and Communication
RWTH Aachen University, 52074 Aachen, Germany
E-mail: terboven@rz.rwth-aachen.de

Abstract

Multiprocessor compute servers have been available for many years now. It is expected that the number of cores and threads per processor chip will increase in the future. Hence, parallel programming will become more common. Posix-/Win32-Threads and OpenMP are the most wide-spread programming paradigms for shared-memory parallelization.

At the first sight, programming for Posix-Threads or OpenMP may seem to be easily understandable. But for non-trivial applications, reasoning about the correctness of a parallel program is much harder than of a sequential control flow. The typical programming errors of shared-memory parallelization are data races and deadlocks. Data races cause the result of a computation to be non-deterministic and dependent on the timing of other events. In case of a deadlock two or more threads are waiting for each other. Finding those errors with traditional debuggers is hard, if not impossible.

This paper compares two software tools: Intel Thread Checker and Sun Thread Analyzer, that help the programmer in finding these errors. Experiences using both tools on multithreaded applications will be presented together with findings on the strengths and limitations of each product.

1 Introduction

Posix-Threads[1] and OpenMP[2] are the most popular programming paradigms for shared-memory parallelization. Typically it turns out that for non-trivial parallel applications reasoning about the correctness is much harder than for sequential control flow. The programming errors introduced by shared-memory parallelization are data races and deadlocks. Finding those errors with traditional debuggers is hard, as the errors are induced by the program flow, which itself is influenced by the debugger. In order to find problems incurred multithreading in acceptable time, we recommend to use specialized tools.

The first commercial product for detecting threading errors in Posix-Threads and OpenMP programs was KAI Assure for Threads. In 2000, Intel acquired Kuck and Associates, hence the Intel Thread Checker is the descendant of Assure. It is available on Intel and compatible architectures on Linux and Windows, the current version is 3.1.

With the release of Sun Studio 12 in May 2007, Sun released the Sun Thread Analyzer, although it has been available in the Studio Express program earlier. It is available on Intel and compatible and UltraSPARC architectures on Linux and Solaris.

Both programs support Posix-Threads and OpenMP programs, in addition Intel supports WIN32-Threads and Sun supports Solaris-threads.

This paper is organized as follows: In chapter 2 we take a look at how these two tools function and what kind of errors the user can expect to be detected. In chapter 3 we describe and compare our experiences of using these tools on small and larger software projects, with a focus on OpenMP programs. In chapter 4 we draw our conclusions.

2 Functioning and Usage

Both tools aim to detect data races and deadlocks in multithreaded programs. The simplest condition for a data race to exist is when all following requirements can occur concurrently (see[3] for a formal definition): Two or more threads of a single process access the same memory location concurrently, between two synchronization points in an OpenMP program, at least one of the threads modifies that location and the accesses to the location are not protected e.g. by locks or critical regions. Data races typically occur in a non-deterministic fashion, for example in an OpenMP program the sequence of the execution of parallel loop iterations is non-deterministic and may change from run to run. In the case of OpenMP, data races typically are caused by missing private clauses or missing synchronization constructs like barriers and critical regions. If threads wait for an event that will never happen, a deadlock occurs. Both kinds of errors do not happen in sequential programs.

Functioning. The basic functioning is pretty similar: Both tools require some sort of application instrumentation and both tools trace references to the memory, thread management operations (such as thread creation) and synchronization operations during an application run. With this information the definition of a data race (similar to the one given above) is checked for pairs of events on different threads. The results are post-processed and presented to the user via a GUI program. For each error report two stack traces (one for each thread) are shown. The deadlock detection works on the same trace data.

The dynamic analysis model implies the principle limitation of both tools[4]: Only problems in the program parts that have been executed during the analysis can be found. That means the analysis result depends on the actual input dataset. Therefore it is crucial to carefully select datasets for the analysis that cover all relevant code parts.

Usage. In order to discuss the usage of both tools, Fig. 1 shows a C version of the Jacobi solver from the OpenMP website, in which we deliberately introduced two parallelization mistakes. The *parallel region* spans the whole block. This *for*-loop is parallelized using OpenMP's worksharing (line 4), the loop iterations are grouped into chunks and then distributed among the threads. The loop variable j is private by default, that means it's name provides access to a different block of storage for each thread; loop variable i has been declared private in line 1. All other variables are shared by default, that means all threads access the same block of storage. The loop variables have to be private as they have different values for different threads during program runtime. The array U has to be shared, as two different threads will always access distinct parts of it.

The problems are related to $resid$ and $error$, which both are shared, thus data races will occur if the program is executed in parallel. $resid$ has to be private as it just stores a temporal computation result, $error$ has to be made a reduction variable in order to accumulate the results contributed by all threads. In this case, the minimum requirement on a data race detection tool is to report the races on $resid$ and $error$. In addition it would be very valuable if the user is pointed to the fact that $error$ depends on the result of individual

thread contributions and therefore it would not be correct to declare the variable private.

```
1   #pragma omp parallel private(i)
2   {
3      /* compute stencil, residual and update */
4   #pragma omp for
5      for (j=1; j<m-1; j++)
6         for (i=1; i<n-1; i++){
7            resid =(ax * (UOLD(j,i-1) + UOLD(j,i+1))
8                  + ay * (UOLD(j-1,i) + UOLD(j+1,i))
9                  + b * UOLD(j,i) - F(j,i) ) / b;
10           U(j,i) = UOLD(j,i) - omega * resid;
11           error = error + resid*resid;
12        }
13  }
14 /* end of parallel region */
```

Figure 1. Detail of the Jacobi program, erroneous OpenMP parallelization.

3 Comparison

In this chapter we compare the Intel Thread Checker and the Sun Thread Analyzer on different applications and scenarios. In subsections 3.1 and 3.2 we take a look at how the tools support the user in principle and discuss the simple OpenMP program presented above. In 3.3 we present how we used this kind of tools to do the actual parallelization. Subsection 3.4 examines how the tools handle C++ programs. Subsection 3.5 presents the memory consumption and runtime increase for selected applications. In subsection 3.6 we compare the ability to check libraries for thread safety. Subsection 3.7 briefly summarizes additional features offered by the tools.

3.1 Simple Use Case: Intel Thread Checker

The Intel Thread Checker supports two different analysis modes, which also can be combined. *Thread-count independent mode* allows for checking existing applications without the need of recompilation. Source information on reported problems can only be given if debug information is available. It requires the application to be executed under the control of the Thread Checker program. *Thread-count dependent mode* can provide additional symbolic information and it allows the code to be analyzed to be executed outside the control of the Thread Checker program. This can be a requirement for software components executed on demand. This mode requires the program to be compiled with the Intel compilers and the $-tcheck$ switch to be used. It allows additional analysis on OpenMP programs, but it is only applicable if the program flow does not depend on the number of threads used.

3.1.1 Thread-count independent mode

In the Jacobian example program, the most important part with respect to the parallelization has been shown in Fig. 1. The result of the analysis depends on the number of threads

used; when running with only one thread no data races are reported. In total 10 errors for 3 different program locations are reported.

A data race in line 8 is reported, both for unsynchronized write accesses by two threads and unsynchronized read and write access by two threads. Although no variable name is given, two source locations are displayed for each error and it is easy to recognize where the write and read accesses are happening. A data race in line 12 is reported, this report results from the race in variable $resid$, as that variable is read in this line. A data race in line 13 is reported, both for unsynchronized write accesses by two threads and unsynchronized read and write access by two threads, and one additional report depending on $resid$ as well.

Together with a detailed error description several guidelines on how to address the errors found are given via the GUI. For all three items it is proposed to either make the variables $private$ or synchronize the access to them. Of course, adding synchronization constructs does not make sense here, although privatizing the variable $error$ will not lead to the desired result as well. Nevertheless, we followed the Thread Checker's advice and rerun the application, again with binary instrumentation. This time, no error in the program is detected. While it is true that all data races were eliminated by privatizing the variables $resid$ and $error$, privatizing $error$ breaks the serial equivalence. That means, the output of the parallel program differs from the original program's output; in this case the result is wrong. As already mentioned above, the variable $error$ has to be made a reduction variable.

3.1.2 Thread-count dependent mode

This mode provides some advantages for OpenMP programs, if the program flow does not depend on the actual number of threads. In total only 5 errors for 2 different program locations are reported, this time the actual variable names $resid$ and $error$ are given. Again it is proposed to make $resid$ a private variable. For variable $error$ the data race occurred in three different kinds: unsynchronized write accesses by two threads and unsynchronized read and write access by two threads (in two different orders). For two of these three reports it is proposed to privatize the variable, but for one the user should consider declaring the variable as a reduction. The exact variable names are given, not just source code locations, and as the tool has additional knowledge of OpenMP, the advice given to the user respects the context and aids the user in correcting the data races.

If one ignores the advice, declares the variable private and reruns the analysis with source instrumentation, the Thread Checker even tells the user that the variable cannot be private, as this leads to undefined accesses in the parallel region. Unfortunately, then there is no further detailed advice given of how to overcome this situation.

3.2 Simple Use Case: Sun Thread Analyzer

In order to analyze an application with the Sun Thread Analyzer, it has to be recompiled using the Sun compilers using the switch *-xinstrument=datarace*. In addition, debug information generation has to be turned on, otherwise no useful source code locations can be given.

The result of the analysis depends on the number of threads used. When running the instrumented Jacobian solver with just one thread, no data races are reported. In order

to find any data races, the program to be analyzed has to be executed with two or more threads. Running with two threads, in total 6 data races for 2 different program locations are reported, namely for variables $resid$ and $error$. After privatizing these two variables, no data races are reported anymore, but again the program is not correct, as $error$ should be a reduction variable. The Sun Thread Analyzer does not give the user any special advice with respect to OpenMP.

3.3 Guidance in the Parallelization Process

Although these tools were designed to find errors in multithreaded programs as demonstrated above, they can assist during the actual parallelization process as well. After performance-critical hotspots were identified, the user can insert e.g. OpenMP parallelization directives, without reasoning about the correctness. Running an erroneous parallelization in one of the tools will show all code locations were data races occur. Then the user is responsible to correctly eliminate all occurrences, with more or less guidance by the tools. This process is iterated until all data races are resolved.

We successfully applied this technique on a FORTRAN code named Panta, beside others. As this code has been tuned for vector computers, the compute-intensive routines contained up to several hundred different scalar variables. Finding all variables that have to be privatized is a lot of work and also error-prone[5], nevertheless in OpenMP every variable has to be given a scope, either explicitly or implicitly. At that time we used the Assure tool to generate a list of all variables on which data races occurred. While most of these variables have to be made private, some have to be declared as first- or lastprivate or reduction. If the requirements for Intel Thread Checker's source instrumentation are fulfilled, the user is helped with these decisions as well.

A study on frequently made mistakes in OpenMP programs has been presented in[6]. While we agree that providing a checklist is a valid approach for the errors classified as performance-related, we recommend using the tools discussed here as early in the parallelization process as possible. From the mistakes classified as correctness-related, eight out of ten can be identified automatically by using the appropriate tools and compilers.

3.4 Handling of C++ Programs

Many of today's tools still have problems with C++ codes, e.g. profilers sometimes are unable to map measurement results to the source correctly. In order to investigate how good the Intel Thread Checker and Sun Thread Analyzer deal with C++ codes, we examined a CG solver that has been parallelized in OpenMP using the external parallelization approach[7]. That means, the parallel region spans the whole iteration loop, the OpenMP worksharing constructs are hidden inside member functions of the data types representing matrix and vector.

We found that both tools detect the data races we inserted into the C++ code. In addition, both tools reported the races where they occur and the user is able to use the GUI to navigate along the call stack. Nevertheless, the Intel Thread Checker with binary instrumentation and also the Sun Thread Analyzer reported additional data races to occur in the STL code, probably induced by the actual races. We found that, for example, when multiple threads create instances of objects and assign them to a single (shared) pointer,

beside the race in the pointer assignment additional races in the object type's constructor are reported. As the number of additional reports is significantly higher than the number of issues of real interest, the user might get distracted and it might take some time to understand the problem cause. Only the Intel Thread Checker in thread-count dependent mode shows just the occurring races.

3.5 Memory Consumption and Runtime

The memory consumption and the execution time of programs can increase dramatically during the analysis, as shown in Table 1 for selected programs. All experiments with the Intel Thread Checker were carried out on a Dell PowerEdge 1950 with two Intel Xeon CPUs, 8 GB of memory, running Scientific Linux 4.4, using the Intel 10.0 compilers and Threading Tools 3.1 update2. All experiments with the Sun Thread Analyzer were carried out on a Sun Fire V40z with four AMD Opteron CPUs, 8 GB of memory, running Solaris 10, using the Sun Studio 12 compilers. *Jacobi* denotes the Jacobian solver as discussed above. *SMXV* is a sparse Matrix-Vector multiplication written in C++, with a relatively small matrix size. *AIC* is an adaptive integration solver employing Nested OpenMP. In *SMXV* and *AIC* the program flow depends on the number of threads used and therefore the thread-count dependent mode of the Intel Thread Checker is not available. Both product

Table 1. Memory consumption (in MByte) and Performance / Runtime of selected programs.

Program	Jacobi		SMXV		AIC	
	Mem	MFLOP/s	Mem	MFLOP/s	Mem	Time
Original, Intel with 2 threads	5	621	40	929	4	5.0 s
Intel Thread Checker tc. indep., 2thr.	115	0.9	1832	3.5	30	9.5 s
Intel Thread Checker tc. dependent	115	3.1	—	—	—	—
Original, Sun with 2 threads	5	600	50	550	2	8.4 s
Sun Thread Analyzer with 2 threads	125	1.1	2020	0.8	17	8.5 s

documentations advise to use the smallest possible and still meaningful dataset. *Small* means that the memory consumption during normal runtime should be minimal, and the runtime itself should be as short as possible. This can be achieved by decreasing the grid resolution, limiting the number of iterations in a solver method, simulating just a couple of time steps, and so on. But it is important that the critical code paths are still executed. In order to ensure this, we found code coverage tools useful, which are provided by both vendors.

Our experience in practice is that it is impossible to analyze programs with production datasets. Looking at the very small Jacobian solver with a memory footprint of about 5 MB, it increases to significantly more than 100 MB with both tools, this is a factor of

about 25. The increase depends[8] on the parallelization of the program and we found it very hard to predict, but average factors of 10 to 20 forbid the usage of datasets using 10 GB of memory, for example, on most current machines.

The Intel Thread Checker in thread-count dependent mode and the Sun Thread Analyzer both run in parallel during the analysis, but we only found the Sun tool to profit from multiple threads. For example the SMXV program achieved with 2 threads 0.8 MFLOP/s, and 1.4 MFLOP/s with 4 threads.

3.6 Checking Libraries for Thread Safety

If library routines are called from possibly multiple threads at a time, the called routines have to be thread safe. Some advice given in the past in order to improve performance, such as declaring variables static, can lead to severe problems in multithreaded programs. We found many libraries in the public domain and also commercial ones to cause problems with parallel applications, for example techniques as reference-counting via internal static variables are still prevalent. Manually verifying library routines can be a tedious task, if not impossible, e.g. if the source code is not available.

```
1  #pragma omp parallel sections
2  {
3  #pragma omp section
4     routine1(&data1);
5  #pragma omp section
6     routine1(&data2);
7  #pragma omp section
8     routine2(&data3);
9  }
```

Figure 2. Pattern to check libraries for thread safety.

The pattern in Fig. 2 tests two scenarios: The thread safety of $routine1$ when two threads are calling this routine in parallel, and the thread safety of routines $routine1$ and $routine2$ being called concurrently. As the Thread Checker supports binary instrumentation without recompilation, it can be used to verify existing libraries for which no source code is available. Using the Sun Thread Analyzer, the source code of the library has to be present.

3.7 Other Features

Both tools offer the ability to detect deadlocks. If two threads already holding locks are requesting new locks such that a chain is formed, the application may deadlock depending on the thread scheduling. It is possible to detect both occurring deadlocks and potential deadlocks.

Some applications have been found to use explicit memory flushes for synchronization, instead of locks or critical regions. Doing so is not recognized by either of the tools, therefore data races are reported that will never occur during normal program runs. Both tools offer APIs to depict user-written synchronization mechanisms in order to avoid false positives.

4 Conclusion

We state that a user should never put a multithreaded program in production before using one of these tools. Both tools are capable of detecting data races in complex applications and using the provided GUIs, the user is presented with two call stacks to locate the races.

For OpenMP programs, the thread-count dependent mode of the Intel Thread Checker tool can provide a noticeable surplus value, but it is only available for a limited class of applications. The second advantage of the Intel Thread Checker is the ability to analyze existing binaries without the need of recompilation. As the runtime increase during the analysis is significantly, the Sun Thread Analyzer's ability to still offer some scalability is advantageous.

Independent of the tool used, the memory consumption may increase dramatically and finally render the tools unusable. Nevertheless, if suitable datasets are available, both tools can easily be embedded in the software development process.

References

1. IEEE, Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API), IEEE Std 1003.
2. OpenMP Architecture Review Board, *OpenMP Application Program Interface*, Version 2.5., (2005),
3. U. Banerjee, B. Bliss, Z. Ma and P. Petersen, *A theory of data race detection*, in: Proc. 2006 workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2006), (2006).
4. U. Banerjee, B. Bliss, Z. Ma and P. Petersen, *Unraveling data race detection in the Intel Thread Checker*, in: Proc. First Workshop on Software Tools for Multi-Core Systems (STMCS 2006), (2006).
5. Y. Lin, C. Terboven, D. an Mey and N. Copty, *Automatic scoping of variables in parallel regions of an OpenMP program*, in: Workshop on OpenMP Applications and Tools (WOMPAT 2004), Houston, USA, (2004).
6. M. Suess and C. Leopold, *Common mistakes in OpenMP and how to avoid them*, in: Second Internation Workshop on OpenMP (IWOMP 2005), (2005).
7. C. Terboven and D. an Mey, *OpenMP and C++*, in: Second International Workshop on OpenMP (IWOMP 2006), Reims, France, (2006).
8. P. Sack, B. E. Bliss, Z. Ma, P. Petersen and J. Torrellas, *Accurate and efficient filtering for the Intel Thread Checker race detector*, in: Proc. 1st workshop on Architectural and System Support for improving software dependability (ASID 2006), New York, USA, pp. 34–41, (2006).

Continuous Runtime Profiling of OpenMP Applications

Karl Fürlinger and Shirley Moore

Innovative Computing Laboratory
EECS Department
University of Tennessee
Knoxville, Tennessee, USA
E-mail: {*karl, shirley*}*@eecs.utk.edu*

In this paper we investigate the merits of combining tracing and profiling with the goal of limiting data volume and enabling a manual interpretation, while retaining some temporal information about the program execution characteristics. We discuss the general dimensions of performance data and which new kind of performance displays can be derived by adding a temporal dimension to profiling-type data. Among the most useful new displays are *overheads over time* which allows the location of when overheads such as synchronization arise in the target application and *performance counter heatmaps* that show performance counters for each thread over time.

1 Introduction

Profiling and tracing are the two common techniques for performance analysis of parallel applications. Profiling is often preferred over tracing because it generates smaller amounts of data, making a manual interpretation easier. Tracing, on the other hand, allows the full temporal behaviour of the application to be reconstructed at the expense of larger amounts of performance data and an often more intrusive collection process.

In this paper we investigate an approach to combine the advantages of tracing and profiling with the goal of limiting the data volume and enabling manual interpretation, while retaining information about the temporal behaviour of the program. Our starting point is a profiling tool for OpenMP applications called `ompP`[1]. Instead of capturing the profiles only at the end of program execution ("one-shot" profiling), in the new approach profiles are captured at several points of time while the application executes. We call our technique *incremental* or *continuous* profiling and demonstrate its usefulness with a number of examples.

The rest of this paper is organized as follows: Sect. 2 briefly introduces our profiling tool and describes its existing capabilities. Sect. 3 then describes the general dimensions of performance data and the new types of data (often best displayed as graphical views) that become available with continuous profiling. Sect. 4 serves as an evaluation of our idea where we show examples form the SPEC OpenMP benchmark suite[2]. We describe related work in Sect. 5 and conclude and discuss further directions for our work in Sect. 6.

2 Application Profiling with `ompP`

`ompP` is a profiling tool for OpenMP applications designed for Unix-like systems. `ompP` differs from other profiling tools like `gprof` or OProfile[3] in primarily two ways. First, `ompP` is a measurement based profiler and does not use program counter sampling. The

```
R00002 main.c (20-23) (unnamed) CRITICAL
 TID      execT      execC      bodyT     enterT      exitT
   0       1.00          1       1.00       0.00       0.00
   1       3.01          1       1.00       2.00       0.00
   2       2.00          1       1.00       1.00       0.00
   3       4.01          1       1.00       3.01       0.00
 SUM      10.02          4       4.01       6.01       0.00
```

Figure 1. Profiling data delivered by ompP for a critical section for a run with four threads (one line per thread, the last line sums over all threads). All times are durations in seconds.

instrumented application invokes ompP monitoring routines that enable a direct observation of program execution events (like entering or exiting a critical section). An advantage of the direct approach is that its results are not subject to sampling inaccuracy and hence they can also be used for correctness testing in certain contexts.

The second difference is in the way of data collection and representation. While other profilers work on the level of functions, ompP collects and displays performance data in the OpenMP user model of the execution[4]. For example, the data reported for critical section contains not only the execution time but also lists the time to enter and exit the critical construct (enterT and exitT, respectively) as well as the accumulated time each threads spends inside the critical construct (bodyT) and the number of times each thread enters the construct (execC). An example profile of a critical section is given in Fig. 1.

Profiling data in a similar style is delivered for each OpenMP construct, the columns (execution times and counts) depend on the particular construct. Furthermore, ompP supports querying hardware performance counters through PAPI[5] and the measured counter values appear as additional columns in the profiles. In addition to OpenMP constructs that are instrumented automatically using Opari[6], a user can mark arbitrary source code regions such as functions or program phases using a manual instrumentation mechanism.

Profiling data is reported by ompP both as flat profiles as well as callgraph profiles, giving inclusive and exclusive times in the latter case. ompP performs an overhead analysis where four well-defined overhead classes (synchronization, load imbalance, thread management, limited parallelism) are quantitatively evaluated. ompP also tries to detect common inefficiency situations, such as load imbalance in parallel loops, contention for locks and critical sections, etc. The profiling report contains a list of the discovered instances of these – so called – *performance properties*[7] sorted by their severity (negative impact on performance).

3 From Profiling to Continuous Profiling

For both profiling and tracing, the following dimensions of performance data can be distinguished in general:

- Kind of data: describes which type of data is measured or reported to the user. Examples include time stamps or durations, execution counts, performance counter values, and so on.

- Source code location: data can be collected globally (for the entire program) or for specific source code entities such as subroutines, OpenMP constructs, basic blocks, individual statements, etc.

- Thread / process dimension: measured data can either be reported for individual threads or processes or accumulated over groups (by summing or averaging, for example).

- Time dimension: Describes when a particular measurement was made (time-stamp) or for which time duration values have been measured.

A distinguishing and appealing property of profiling data is its low dimensionality, i.e., it can often be comprehended textually (like gprof output) or it can be visualized as 1D or 2D graphs in a straightforward way. Adding a new dimension (time) jeopardizes this advantage and requires more sophisticated performance data management and displays. The following description lists performance data displays from continuous profiles that are based on the (classic) performance data delivered by the `ompP`, extended with a temporal dimension.

- **Performance properties over time:**

  Performance properties[7] are a very compact way to represent performance analysis results and their change over time can thus be visualized easily. There is an extended formalism for specifying properties[7], an informal example for illustration is "Imbalance in parallel region `foo.f (23-42)` with severity of 4.5%". The definition carries all relevant context information with it and the severity value denotes the percentage of total execution time improvement that can be expected if the cause for the inefficiency could be removed. The threads dimension is collapsed in the specification of the property and the source code dimension is encoded as the context of the property (`foo.f (23-42)` in the above example).

  Properties over time data can be visualized as a 1D lineplot, where the x-axis is the time (t) and the y-axis denotes the severity value at time t. That is, the severity value at time t is determined by program behaviour from program start (time 0 until t). Depending on the particular application, valuable information can be deduced from the shapes of the graphs. An example is shown in Fig. 2.

- **Region invocations over time:**

  Depending on the size of the application and the analyst's familiarity with the source code, it can be valuable to know when and how often a particular OpenMP construct, such as a parallel loop, was executed. The region invocation over time displays offers this functionality. This view is most useful when aggregating (e.g., summing) over all threads, the x-axis displays the time and the y-axis counts the region invocations in this case. In certain situations it can also be valuable to see which thread executed a construct at which time. In this case either multiple lineplots (one line per thread) or a surface plot (y-axis representing threads and z-axis counting invocations) can be used for visualization. Another option is colour coding the number of invocations similar to the performance counter heatmap view (discussed below).

- **Region execution time over time:**

  This display is similar to the region invocation over time display but shows the execution time instead of execution count. Again this display allows the developer to see when particular portions of the code actually get executed. In addition, by dividing the execution time by the execution count, a normalized execution time can be determined. This allows a developer to see if the execution time of the region instances changed over time and to derive conclusions from that, e.g., effects like cache pollution can show up in this type of display.

- **Overheads over time:**

  `ompP` evaluates four overhead classes based on the profiling data for individual parallel regions and for the program as a whole. For example, the time required to enter a critical section is attributed as the containing parallel region's synchronization overhead. A detailed discussion and motivation of this classification scheme can be found in[8].

  The overheads over time can be visualized easily as 1D lineplots similar to the properties over time view. The x-axis represents time and the y-axis shows the incurred overhead. It is usually convenient to display the overheads in percentages of execution time lost, i.e., the y-axis ranges from 0 to 100% and for each of the four supported overhead classes (synchronization, imbalance, limited parallelism, thread management), a line indicates the percentage of execution time lost due to that overhead class. Different to the properties over time display, the overheads are plotted as they occur for each time step (Δt) and are not accumulated from program start. An example for this is the graph in Fig. 3.

- **Performance counter heatmaps:**

  The performance counter heatmap display is a tile map where the x-axis corresponds to the time while the y-axis corresponds to the thread ID. The tiles are filled and a colour gradient coding is used to differentiate between higher and lower counter values. A tile is not filled if no data samples are available for that time period. This type of display is supported for both the whole program as well as for individual OpenMP regions.

4 Implementation and Evaluation of Continuous Runtime Profiling

A straightforward way to add a temporal component to profiling-type performance data is to capture profiles at several points during the execution of the target application (and not just at the end) and to analyze how the profiles change between those capture points. Alternatively (and equivalently), the changes between capture points can be recorded incrementally and the overall state at capture time can later be recovered.

Several trigger events for the collection of profiling reports are possible. The trigger can either be based on a fixed-length or adaptive timer, or it can be based on the overflow of a hardware counter. Another possibility is to expose a mechanism to dump profiles to the user. In this paper we investigate the simplest form of incremental profiling: capturing profiles in regular, fixed-length intervals during the entire lifetime of the application. We

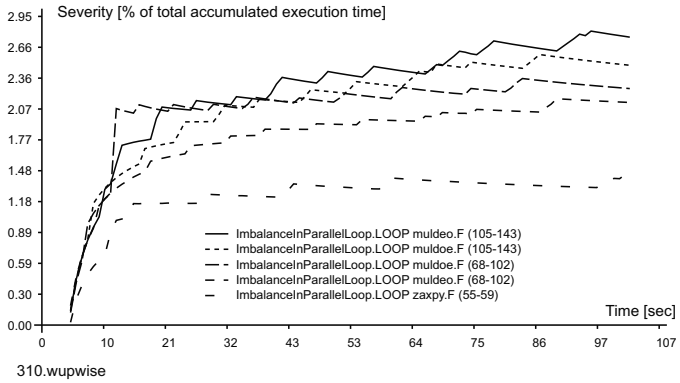


Figure 2. An example for the "performance properties over time" display for the 310.wupwise application. The five most severe performance properties are shown.

have implemented this technique in our profiler `ompP` by registering a timer signal (using `SIGALRM`) that is delivered to the profiler in regular intervals and causes the current state of the profiling data to be stored to a memory buffer. On program termination the buffer is flushed to disk as a set of profiling reports. A collection of Perl scripts that come with `ompP` can then be used to analyze the profiling reports and create the performance displays described below in the form of SVG (scalable vector graphics) and PNG (portable network graphics) images.

We have tested this technique with a dumping interval of 1 second on the applications from the medium size variant of the SPEC OpenMP benchmarks on a 32 CPU SGI Altix machine based on Itanium-2 processors with 1.6 GHz and 6 MB L3 cache used in batch mode. Due to space limitations we can only show results from a very small number of runs.

Fig. 2 shows the properties over time display for the 310.wupwise application. It is evident that the severity of the properties which are all imbalance related appears to be continuously increasing as time proceeds, indicating that the imbalance situations in this code will become increasingly significant with longer runtime (e.g., larger data sets or more iterations). Other applications from the SPEC OpenMP benchmark suite showed other interesting features such as initialization routines that generated high initial overheads which amortized over time (i.e., the severity decreased).

Figure 3 shows the overheads over time display for the 328.fma3d application. The most noticeable overhead is synchronization overhead starting at about 30 seconds of execution and lasting for several seconds. A closer examination of the profiling reports reveals that this overhead is caused by critical section contention. One thread after the other enters the critical section and performs a time-consuming initialization operation. This effectively serializes the execution for more than 10 seconds and shows up as an overhead of $31/32 = 97\%$ in the overheads graph.

The graphs in Fig. 4 show examples of performance counter heatmaps. Depending on the selected hardware counters, this view offers very interesting insight into the behaviour of the applications. Phenomena that we were able to identify with this kind of display

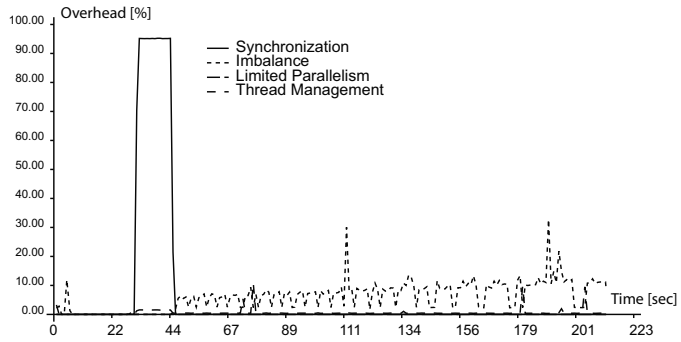


Figure 3. This graph shows overheads over time for the 328.fma3d application.

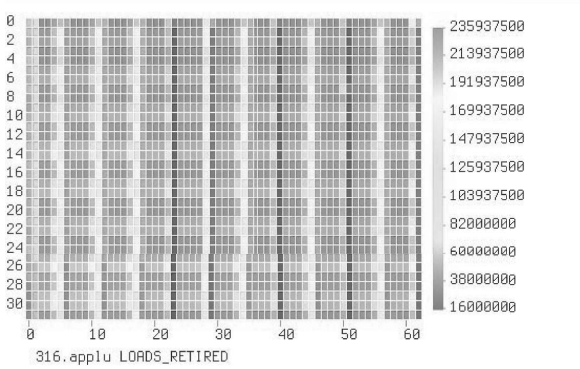
include iterative behaviour (e.g.,Fig. 4(a)), thread grouping and differences in homogeneity or heterogeneity of thread behaviour. E.g., often groups of threads would show markedly different behaviour compared to other threads in a 32 thread run. Possible reasons for this difference in behaviour might be in the application itself (related to the algorithm) but they could also come from the machine organization or system software layer (mapping of threads to processors and their arrangement in the machine and its interconnect). As another example, Fig. 4(b) gives the number of retired floating point operations for the 324.apsi application and this graph shows a marked difference for threads 0 to 14 vs. 15 to 31. We were not able to identify the exact cause for this behaviour yet.

5 Related Work

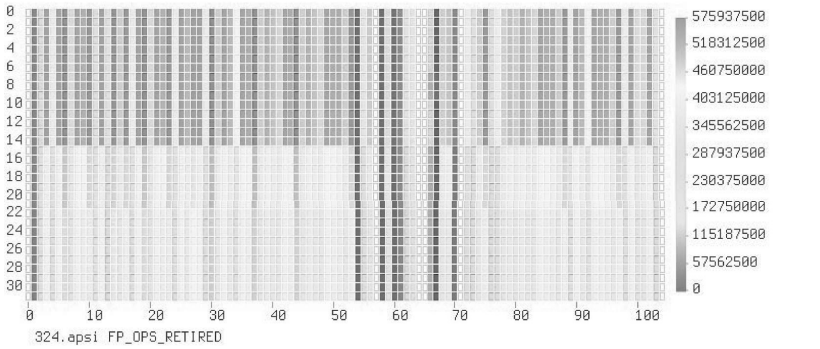
There are several performance analysis tools for OpenMP. Vendor specific tools such as the Intel Thread Profiler and Sun Studio are limited to a single platform but can take greater advantage of internal details of the compiler's OpenMP implementation and the runtime system than more general tools. Both the Intel and the Sun tool are based on sampling and can provide the user with some timeline profile displays. Neither of those tools however has a concept similar to `ompP`'s high-level abstraction of performance properties or the properties over time display.

TAU[9, 10] is also able to profile and trace OpenMP applications by utilizing the Opari instrumenter. Its performance data visualizer Paraprof supports a number of different profile displays and also supports interactive 3D exploration of performance data, but does not currently have a view similar to the performance counter heatmaps. The TAU toolset also contains a utility to convert TAU trace files to profiles which can generate profile series and interval profiles.

OProfile and its predecessor, the Digital Continuous Profiling Infrastructure (DCPI), are system-wide statistical profilers based on hardware counter overflows. Both approaches rely on a profiling daemon running in the background and both support the dumping of profiling reports at any time. Data acquisition in a style similar to our incremental profiling approach would thus be easy to implement. We are, however, not aware of any study



(a) Retired load instructions for the 316.applu application.



(b) Retired floating point operations for the 324.apsi application.

Figure 4. Example performance counter heatmaps. Time is displayed on the horizontal axis (in seconds), the vertical axis lists the threads (32 in this case).

using OProfile or DPCI that investigated continuous profiling for parallel applications. In practice, the necessity of root privileges and the difficulty of relating profiling data back to the user's OpenMP execution model can be a major problem employing these approaches, both are no issues with `ompP` since it is based on source code instrumentation.

6 Outlook and Future Work

We have investigated continuous profiling of parallel applications in the context of an existing profiling tool for OpenMP applications. We have discussed several general approaches to add temporal dimension to performance data and have tested our ideas on applications from the SPEC OpenMP benchmarks suite.

Our results indicate that valuable information about the temporal behaviour of applications can be discovered by incremental profiling and that this technique strikes a good balance between the level of detail offered by tracing and the simplicity and efficiency of profiling. Using continuous profiling we were able to get new insights into the behaviour

of applications which can, due to the lack of temporal data, not be gathered from traditional "one-shot" profiling. The most interesting features are the detection of iterative behaviour, the identification of short-term contention for resources, and the temporal localization of overheads and execution patterns.

We plan continued work in several areas. In a future release of `ompP` we plan to support other triggers for capturing profiles, most importantly user-added and overflow based. Furthermore we intend to test our approach in the context of MPI as well, a planned integrated MPI/OpenMP profiling tool based on `mpiP`[11] and `ompP` is the first step in this direction.

References

1. K. Fuerlinger and M. Gerndt, *ompP: a profiling tool for OpenMP*, in: Proc. 1st International Workshop on OpenMP (IWOMP 2005), Eugene, Oregon, USA, (2005).
2. V. Aslot and R. Eigenmann, *Performance characteristics of the SPEC OMP2001 benchmarks*, SIGARCH Comput. Archit. News, **29**, 31–40, (2001).
3. J. Levon, *OProfile, A system-wide profiler for Linux systems*, Homepage: http://oprofile.sourceforge.net.
4. M. Itzkowitz, O. Mazurov, N. Copty and Y. Lin, *An OpenMP runtime API for profiling*, Accepted by the OpenMP ARB as an official ARB White Paper; available online at http://www.compunity.org/futures/omp-api.html.
5. S. Browne, J. Dongarra, N. Garner, G. Ho and P. J. Mucci, *A portable programming interface for performance evaluation on modern processors*, Int. J. High Perform. Comput. Appl., **14**, 189–204, (2000).
6. B. Mohr, A. D. Malony, S. S. Shende and F. Wolf, *Towards a performance tool interface for OpenMP: an approach based on directive rewriting*, in: Proc. Third Workshop on OpenMP (EWOMP'01), (2001).
7. M. Gerndt and K. Fürlinger, *Specification and detection of performance problems with ASL*, Concurrency and Computation: Practice and Experience, **19**, 1451–1464, (2007).
8. K. Fuerlinger and M. Gerndt, *Analyzing overheads and scalability characteristics of OpenMP applications*, in: Proc. 7th International Meeting on High Performance Computing for Computational Science (VECPAR'06), Rio de Janeiro, Brasil, LNCS vol. 4395, pp. 39–51, (2006).
9. S. S. Shende and A. D. Malony, *The TAU parallel performance system*, International Journal of High Performance Computing Applications, ACTS Collection Special Issue, (2005).
10. A. D. Malony, S. S. Shende, R. Bell, K. Li, L. Li and N. Trebon, *Advances in the TAU performance analysis system*, in: Performance Analysis and Grid Computing, V. Getov, M. Gerndt, A. Hoisie, A. Malony, and B. Miller, (Eds.), pp. 129–144, (Kluwer 2003).
11. J. S. Vetter and F. Mueller, *Communication characteristics of large-scale scientific applications for contemporary cluster architectures*, J. Parallel Distrib. Comput., **63**, 853–865, (2003).

Mini-Symposium

“DEISA: Extreme Computing in an Advanced Supercomputing Environment”

DEISA: Extreme Computing in an Advanced Supercomputing Environment

Hermann Lederer[1], **Gavin J. Pringle**[2], **Denis Girou**[3], **Marc-André Hermanns**[4], **Giovanni Erbacci**[5]

[1] Rechenzentrum Garching der Max-Planck-Gesellschaft, Garching
E-mail: lederer@rzg.mpg.de

[2] EPCC, University of Edinburgh, Edinburgh
E-mail: g.pringle@epcc.ed.ac.uk

[3] CRNS/IDRIS, Paris
E-mail: denis.girou@idris.fr

[4] Central Institute for Applied Mathematics, Research Centre Jülich
E-mail: m.a.hermanns@fz-juelich.de

[5] CINECA, Bologna
E-mail: g.erbacci@cineca.it

DEISA (Distributed European Infrastructure for Supercomputing Applications) is a consortium of leading national supercomputing centres in Europe. Starting in 2004 it deploys and operates a persistent, production quality, distributed infrastructure for supercomputing applications. It connects eleven European supercomputing centres in seven different countries via a dedicated high bandwidth network, enabling seamless and transparent access to a Europe-wide shared global file system. The coordinated operation of this environment is tailored to enable new, ground breaking applications in computational sciences. Scientists from all over Europe have successfully used this platform for extreme computing endeavours with concrete scientific impact, benefiting from the DEISA Extreme Computing Initiative DECI.

This Mini-Symposium highlights the scientific impacts achieved so far with the DEISA infrastructure, while also giving a forum for the extensive enabling work done for applications running in the DECI and encouraging discussion of current limitations of distributed computing as well as future developments.

Mini-Symposium Contributions

For a better overview of the key areas involved, the Mini-Symposium was thematically grouped into three sessions related to the topics infrastructure, applications, and scientific projects.

In the first session, the key note presentation by Victor Alessandrini from IDRIS-CNRS, the coordinator of DEISA, on *Enabling Cooperative Extreme Computing in Europe*, outlines the objectives and design principles of DEISA: To act as a vector of integration of High Performance Computing (HPC) resources at continental scale in Europe, and to enable seamless access to, and high performance cooperative operation of leading supercomputing platforms in Europe. Through the DEISA services, deployed on top of a dedicated high speed network infrastructure connecting computing platforms, and by using selected middleware, capability computing across remote computing platforms and data repositories is enabled. By reviewing the existing services for the DEISA Infrastructure which allow the execution of challenging computational science projects within the

DEISA Extreme Computing Initiative, it is demonstrated how DEISA has been paving the way to the deployment of a coherent HPC environment in Europe. Persistency of the services is regarded mandatory to support and cooperate with new initiatives like PRACE in the area of HPC, to advance towards the efficient operation of a future European HPC ecosystem.

In his presentation about *Effective Methods for Accessing Resources in a Distributed HPC Production System*, Andrea Vanni from CINECA presents the different options for different user needs how to best access the distributed DEISA infrastructure.

Details about the design and specific features of the global file system GPFS, being used at continental scale both by DEISA and by TeraGrid, are given in the talk by Klaus Gottschalk from IBM about *GPFS: a Cluster Filesystem*.

In his contribution *Submission Scripts for Scientific Simulations on DEISA*, Gavin Pringle from EPCC presents how DEISA users can also manage their batch jobs and data via the DEISA Services for the Heterogeneous management Layer, or DESHL, which is both a command line tool and an application programming interface to support workflow simulations, automatic code resubmission, and DEISA usage as a multi-site Task Farm.

In the application related session, Alice Koniges from Lawrence Livermore National Laboratory reports about *Development Strategies for Modern Predictive Simulation Codes*. The work describes the process of designing modern simulation codes, based on a set of development tools, libraries and frameworks suitable to reduce the time to solution, assuring, at the same time, portability and modularity.

Hermann Lederer from RZG et al. presents in the talk *Application Enabling in DEISA: Hyperscaling of Plasma Turbulence Codes* an example of application enabling for plasma turbulence simulation codes with high relevance for the European fusion community. It is shown how, in a joint effort of application specialists from DEISA and theoretical plasma physicists two important European simulation codes have been adapted for extreme scalability and for portable usage within the heterogeneous DEISA infrastructure.

In the third session related to scientific projects, Frank Jenko et al from IPP report in *First Principles Simulations of Plasma Turbulence within DEISA* about the DECI project GYROKINETICS. Magnetic confinement fusion, aiming at providing an energy resource free of CO_2 emissions, depends on a large degree on the value of the so-called energy confinement time. Two of the most advanced tools describing the underlying physical processes, codes ORB5 and GENE, have been used in DEISA to address some of the outstanding issues in fusion research.

Alessandra S. Lanotte et al, from CNR in Italy, in her talk *Heavy Particle Transport in Turbulent Flows* shows the results of the DECI Project HEAVY: State of the art Direct Numerical Simulation (DNS) of heavy particles in an homogeneous and isotropic stationary forced turbulence flow at Reynolds number $Re_\lambda \simeq 400$ The simulation was the most accurately resolved numerical simulation of Lagrangian turbulence done worldwide.

Marc Baaden from CNRS presents *Membranes Under Tension: Atomistic Modelling of the Membrane-Embedded Synaptic Fusion Complex*, studied in DECI project SNARE. Simulations allow to better understand the guiding principles of membrane fusion, a target for studying several pathologies such as botulism and tetanus.

The mini-symposium and the selected presentations document the transition from an initial test environment to a now mature production-oriented infrastructure, and thus the role and global importance of DEISA for a future European HPC ecosystem.

DEISA: Enabling Cooperative Extreme Computing in Europe

Hermann Lederer[1] **and Victor Alessandrini**[2]

[1] Rechenzentrum Garching der Max-Planck-Gesellschaft
Boltzmannstr. 2, D-85748 Garching, Germany
E-mail: Lederer@rzg.mpg.de

[2] CNRS/IDRIS, Bâtiment 506, BP 167
91403 Orsay cedex, France
E-mail: va@idris.fr

The DEISA European Research Infrastructure has been designed to act as a vector of integration of High Performance Computing (HPC) resources at the continental scale. Its services have been tailored to enable seamless access to, and high performance cooperative operation of, a distributed park of leading supercomputing platforms in Europe. The DEISA services are deployed on top of a dedicated high speed network infrastructure connecting computing platforms, using selected middleware. Their primordial objective is enabling capability computing across remote computing platforms and data repositories. Workflows across different platforms, transparent remote I/O, and large file transfers are starting to operate without inducing performance bottlenecks that would invalidate high performance computing. These services will bloom as the number of scientific users accessing different computing platforms in Europe will grow. After reviewing the existing services and the DEISA research Infrastructure based today on the DEISA Extreme Computing Initiative, we discuss how DEISA has been paving the way to the deployment of a coherent HPC environment in Europe, and why their persistency is mandatory to support and cooperate with new initiatives like PRACE in the area of HPC. Some comments are advanced about the relevance of the DEISA environment for the efficient operation of future European supercomputers, and the current vision about the overall role of DEISA in the new emerging European HPC ecosystem is discussed.

1 DEISA — The Origin

In early 2002 the idea was born to overcome the fragmentation of supercomputing resources in Europe both in terms of system availability and in the necessary skills for efficient supercomputing support. After an Expression of Interest to the EU in Spring 2002 proposing to establish a distributed European supercomputing infrastructure, the DEISA project was started in May 2004 as a EU FP6 Integrated Infrastructure Initiative by eight leading European supercomputing centres and expanded in 2006 by three additional leading centres. DEISA — the Distributed European Infrastructure for Supercomputing Applications[1], is now in its fourth year since starting the deployment of the infrastructure, has reached production quality to support leading edge capability computing for the European scientific community.

2 The Consortium

The DEISA Consortium is constituted from eleven partners from seven European countries: BSC, Barcelona (Spain); CINECA, Bologna (Italy); CSC, Espoo (Finland); EPC-

C/HPCx, Edinburgh and Daresbury (UK); ECMWF, Reading (UK); FZJ, Jülich (Germany); HLRS, Stuttgart (Germany); IDRIS-CNRS, Orsay (France); LRZ, Munich/Garching (Germany); RZG, Garching (Germany); and SARA, Amsterdam (The Netherlands).

3 Key Features of DEISA

The guiding strategy of the DEISA Consortium has been the integration of national supercomputing systems using selected Grid technologies, to add value to the existing environments and to provide an extra layer of European HPC services on top of them. The DEISA Consortium has used Grid technologies to enhance HPC in Europe and take the first steps towards the creation of a European HPC ecosystem. Outstanding features of the integrated infrastructure are:

- High speed end-to-end network
- Common global high performance file system at continental scale to greatly facilitate data management across Europe
- Uniform infrastructure access through the UNICORE middleware system
- Harmonization of the manifold heterogeneous software environments through the DEISA Common Production Environment DCPE
- Portals and internet interfaces for transparent access to complex supercomputing environments
- Job re-routing across Europe
- Joint research activities in major scientific disciplines
- Enabling of cooperative Extreme Computing in Europe

3.1 High speed end-to-end network

National supercomputers have been tightly interconnected by a dedicated high performance network infrastructure provided by GÉANT2 and the National Research Network providers (NREN). As illustrated in Fig. 1, the network backbone is based today on 10 Gbit/s technology.

3.2 Common global high performance file system at continental scale

High performance wide-area global file systems such as GPFS from IBM open up totally new modes of operation within grid infrastructures, especially in supercomputing grids with a fairly limited number of participating sites. A common data repository with fast access, transparently accessible both by applications running anywhere in the grid, and by scientists working at any partner site as entry point to the grid, greatly facilitates cooperative scientific work at the continually increasing geographically distributed scientific communities. During the Supercomputing Conference 2005, a supercomputing hyper-grid

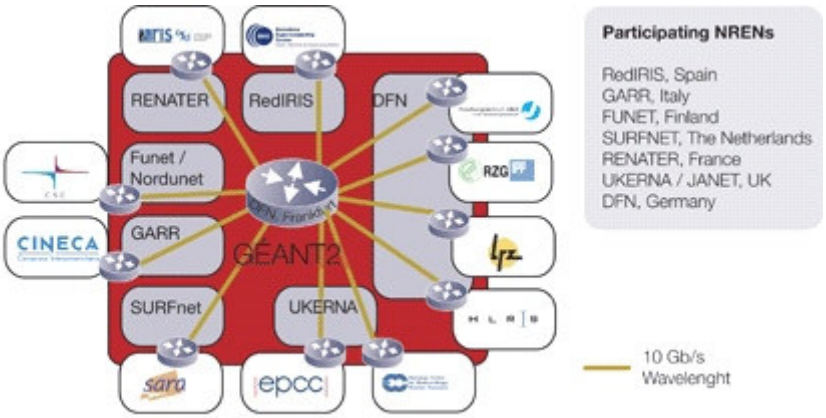


Figure 1. DEISA dedicated network based on 10 Gbit/s technology provided by GÉANT2 and the NRENs

was created to move a step towards interoperability of leading grids. A dedicated network connection was established between DEISA and TeraGrid[2], and the DEISA and TeraGrid global file systems were interconnected to build a huge, global file system spanning two continents [3] [4] [5].

3.3 Uniform infrastructure access through the UNICORE middleware system

The UNICORE middleware[6] is *the* main gateway to DEISA. It facilitates end-user instrumentation, provides secure access to distributed compute resources, enables single sign-on and provides a graphical user interface to batch-subsystems and to file systems. And UNICORE greatly facilitates the set-up of complex workflows[7] [8]. A command line tool is available via the DEISA Services for the Heterogeneous management Layer, or DESHL, which permits the user to manage their DEISA jobs directly from their workstation.

3.4 Harmonization of the manifold heterogeneous software environments

The DEISA supercomputing infrastructure is characterized by a large diversity of HPC architectures with heterogeneous software environments. Architectures include today systems based on Power4, Power4+, and Power5 processors from IBM in various node sizes, running the operating system AIX, Power5+ and PowerPC from IBM running Linux, SGI Altix 4700 from SGI with Intel IA64 Itanium2/Montecito processors running Linux, Cray XT4 from Cray with AMD Opteron processors running Linux, NEC SX-8 vector system running Super UX.

Application and user support specialists have therefore developed the DEISA Common Production Environment (DCPE), to provide a uniform DEISA environment for the users, with the same appearance and functionality, whatever the underlying architecture: homogeneous access to the heterogeneous supercomputing cluster. Details are given in the DEISA Primer[9].

3.5 Science gateways

Infrastructure access for the Life Science and Materials Science communities has additionally been facilitated by specific science gateways, completely hiding the complexity of the distributed infrastructure and providing application-specific enhancements[10] [11].

3.6 Job re-routing across Europe

Job re-routing across Europe has been put into place among five platforms with the same operating system to a) free resources for a big job requiring close to all resources at one site, and b) to facilitate simultaneous usage of many platforms for independent subtasks of a big project, to accelerate project turn-around in a complementary approach to using UNICORE.

3.7 Joint Research Activities in major scientific disciplines

Joint Research Activities (JRAs) have been carried out to closely work with early adopters of the infrastructure from different scientific communities to address specific needs, e.g. from Materials Science, Cosmology, Plasma Physics, Life Sciences, and Engineering & Industry. As an example, the JRA in plasma physics has supported EFDA[12] in adhoc and expert groups for HPC and has addressed the enabling and optimization of the majority of the European turbulence simulation codes considered by EFDA as highly relevant for ITER[13].

3.8 Extreme Computing and Applications Enabling in Europe

Of key importance for an adequate usability of state-of-the-art and next generation supercomputers is applications enabling. A team of leading experts in high performance and Grid computing, the so-called Applications Task Force, provides application support in all areas of science and technology. In 2005 the DEISA Extreme Computing Initiative (DECI) was launched for the support of challenging computational science projects. One of the key tasks is hyperscaling, a unique service not offered by any other European grid computing project. Hyperscaling aims at application enabling towards efficient use of thousands or tens of thousands of processors for the same problem, a prerequisite for applications to benefit from forthcoming Petaflop scale supercomputers.

Application and user support specialists have enabled and optimized applications for usage in DEISA. Two European plasma physics simulation codes, GENE and ORB5, have been scaled up to 8000 processor-cores and beyond[14]. Examples of application enabling in addition to hyperscaling include: design of coupled applications; determination of best suited architecture(s) in DEISA; adaptation of applications to the DEISA infrastructure and architecture dependent optimizations.

Since late 2005, DEISA has been used for conducting the most challenging European projects in computational sciences requiring the most advanced supercomputing resources available through DEISA. Over 50 projects from the 2005 and 2006 DECI calls have already benefited from the Extreme Computing Initiative. Close to thirty papers have already been published upon DECI projects. Examples of very successful projects include:

Figure 2. Cover page of Nature (May 24, 2007) referring to the results of the DECI project POLYRES[15]

POLYRES (German/British group, Principal Investigator (P.I.) Kurt Kremer): Water & Salt, Fluid Membranes. For almost two decades, physicists have been on the track of membrane mediated interactions. Simulations in DEISA have now revealed that curvy membranes make proteins attractive, reported as cover story of NATURE[15] (see Fig. 2).

BET (Italian/Swiss/British consortium, P.I. Paolo Carloni): First-Principles and Mixed quantum classical QM/MM Simulations of Biological Electron Transfer. The study showed that full ab initio calculations on bio-molecules at physiological conditions are now possible by exploiting modern quantum chemistry protocols and today's large-scale supercomputing facilities, like those made available through the DEISA initiative[16 17].

CAMP (Swiss/Hungarian group, P.I. Michele Parrinello): Catalysis by Ab-initio Metadynamics in Parallel. Pyrite has been suggested as the key catalyst in the synthesis of prebiotic molecules under anoxic conditions in the early history of the Earth. A better understanding of the pyrite surface and its chemistry was achieved by means of ab-initio calculations[18 19].

GIMIC: The international VIRGO Consortium (http://www.virgo.dur.ac.uk, P.I.'s Simon White and Carlos Frenk) have performed the Galaxies-Intergalactic Medium Interaction Calculation. The results of the cosmological simulations are illustrated in Fig. 3

In the third successful year of the DECI, over 60 proposals were received from the 2007 call, requesting a total of more than 70 million processor hours. Since multi-national proposals are especially encouraged by DEISA, scientists from 15 different European countries have so far been involved in DECI, some with partners from other continents: namely North and South America, Asia and Australia.

4 Conclusions

The DEISA Consortium has contributed to raising awareness of the need for a European HPC ecosystem. Since its formation in 2002, the consortium has directed its efforts towards the integration of leading national supercomputing resources on a continental scale. Its fundamental objective has been the deployment and operation of a distributed Euro-

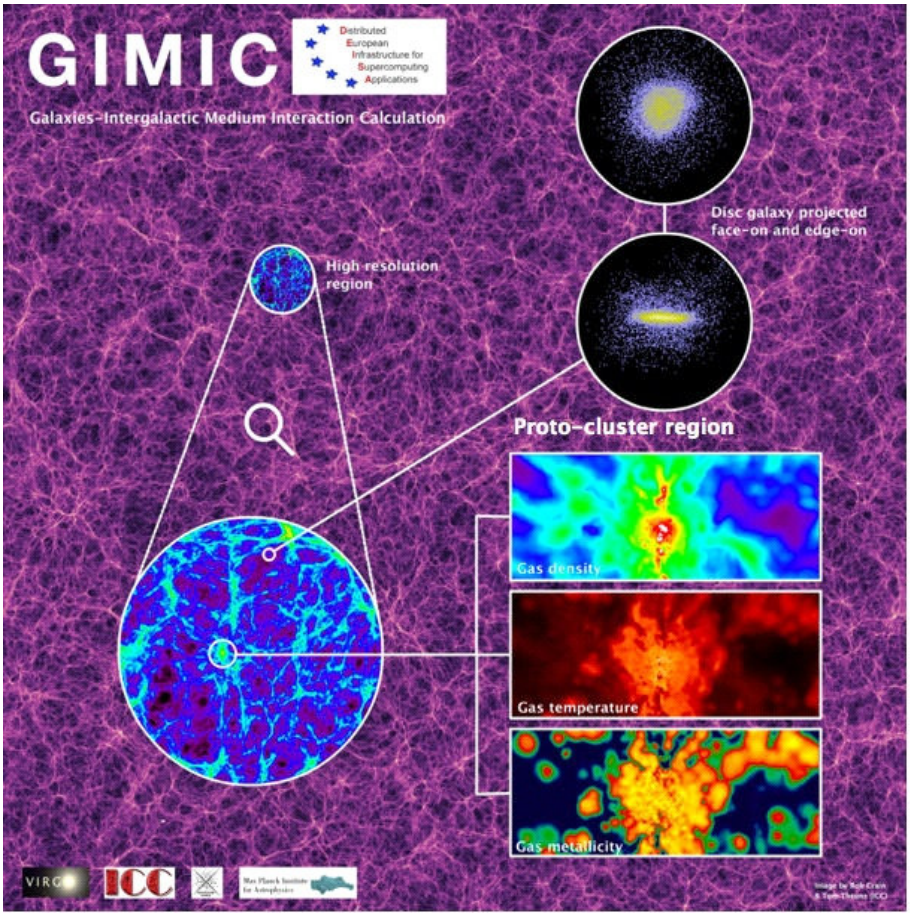


Figure 3. Illustration of the results of the GIMIC DECI project of the international VIRGO Consortium, simulating Galaxies-Intergalactic Medium Interaction Calculation

pean HPC environment capable of enabling the high performance cooperative operation of a number of leading national supercomputing systems in Europe, thereby drawing international attention to Europe's rising international competitiveness in the HPC area. The importance of a European HPC ecosystem was recognised in the ESFRI. DEISA has paved the way to provide the operational infrastructure for the forthcoming shared European supercomputing platforms that are the target of new initiatives in Europe (PRACE) today. DEISA is advancing computational sciences in Europe and has increased Europe's visibility in the supercomputing domain.

Acknowledgements

The DEISA Consortium thanks the European Commission for support through contracts FP6-508830 and FP6-031513.

References

1. Distributed European Infrastructure for Supercomputing Applications (DEISA), http://www.deisa.org
2. TeraGrid http://www.teragrid.org
3. http://cordis.europa.eu/ictresults/index.cfm/section/news/tpl/article/BrowsingType/-Features/ID/80183
4. P. Andrews, C. Jordan and H. Lederer, *Design, implementation, and production experiences of a global storage grid*, in: *Proc. 14th NASA Goddard, 23rd IEEE Conference on Mass Storage Systems and Technologies*, (2006).
5. P. Andrews, M. Buechli, R. Harkness, R. Hatzky, C. Jordan, H. Lederer, R. Niederberger, A. Rimovsky, A. Schott, T. Soddemann and V. Springel, *Exploring the hypergrid idea with grand challenge applications: The DEISA- TeraGrid interoperability demonstration*, in: *Challenges of Large Applications in Distributed Environments, 2006 IEEE*, pp. 43–52, (2006).
6. UNICORE http://www.unicore.org
7. L. Clementi, M. Rambadt, R. Menday and J. Reetz, *UNICORE Deployment within the DEISA Supercomputing Grid Infrastructure, Proc. 2nd UNICORE Summit 2006 in conjunction with EuroPar 2006 Dresden, Germany*, LNCS **4375**, pp. 264–273, (2006).
8. M. Rambadt, R. Breu, L. Clementi, Th. Fieseler, A. Giesler, W. Gürich, P. Malfetti, R. Menday, J. Reetz and A. Streit, *DEISA and D-Grid: using UNICORE in production Grid infrastructures, Proc. German e-Science Conference 2007, Baden-Baden*, Max Planck Digital Library, ID 316544.0, (2007).
9. DEISA Primer http://www.deisa.org/userscorner/primer
10. T. Soddemann, *Science gateways to DEISA*, in: Concurrency Computat. Pract. Exper., DOI 10.1002/cpe, (2006).
11. T. Soddemann, *Job management enterprise application*, in: W. Lehner et al. (eds): Euro-Par 2006 Workshops, Lecture Notes in Computer Science **4375**, pp. 254–263, (2007)
12. European Fusion Development Agreement EDFA. http://www.efda.org
13. ITER http://www.iter.org
14. H. Lederer, R. Hatzky, R. Tisma and F. Jenko, *Hyperscaling of plasma turbulence simulations in DEISA*, *Proc. 5th IEEE workshop on Challenges of large applications in distributed environments*, p. 19, (ACM Press, New York, 2007).
15. B.J. Reynwar, G. Illya, V.A. Harmandaris, M.M. Müller, K. Kremer and M. Deserno, *Aggregation and vesiculation of membrane proteins, by curvature mediated interactions*, Nature, **447**, doi:10.1038/nature05840, (2007).
16. M. Cascella, A. Magistrato, I. Tavernelli, P. Carloni and U. Rothlisberger, *Role of protein frame and solvent for the redox properties of azurin from Pseudomonas aeruginosa Cascella, Proc. National Academy of Sciences of the United States of America*, **103**, pp. 19641–19646, (2006).
17. M. Sulpizi, S. Raugei, J. VandeVondele, M. Sprik and P. Carloni, *Calculation of redox properties: understanding short and long range effects in rubredoxin*, J. Phys. Chem. B, **111**, 3969–3976, (2007).
18. A. Stirling, M. Bernasconi and M. Parrinello, *Defective pyrite (100) surface: An ab*

initio study, Phys. Rev. B, **75**, 165406, (2007).
19. T. D. Kuehne, M. Krack, F. R. Mohamed and M. Parrinello, *Efficient and accurate Car-Parrinello-like approach to Born-Oppenheimer molecular dynamics*, Phys. Rev. Lett., **98**, 066401, (2007).

Development Strategies for Modern Predictive Simulation Codes

Alice. E. Koniges, Brian T.N. Gunney, Robert W. Anderson, Aaron C. Fisher, Nathan D. Masters

Lawrence Livermore National Laboratory
Livermore, CA 94550, U.S.A.
E-mail: koniges@llnl.gov

Modern simulation codes often use a combination of languages and libraries for a variety of reasons including reducing time to solution, automatic parallelism when possible, portability, and modularity. We describe the process of designing a new multiscale simulation code, which takes advantage of these principles. One application of our code is high-powered laser systems, where hundreds of laser beams are concentrated on centimeter-sized targets to enable the demonstration of controlled fusion and exciting new experiments in astrophysics and high-energy-density science. Each target must be carefully analyzed so that debris and shrapnel from the target will be acceptable to optics and diagnostics, creating new simulation regimes. These simulations rely on a predictive capability for determining the debris and shrapnel effects. Our new three-dimensional parallel code uses adaptive mesh refinement (AMR) combined with more standard methods based on Arbitrary Lagrangian Eulerian (ALE) hydrodynamics to perform advanced modelling of each different target design. The AMR method provides a true multiscale simulation that allows for different physical models on different scales.

We discuss our code development strategies. The code is built on top of the SAMRAI library (structured adaptive mesh refinement application interface) that provides scalable automatic parallelism. During the development phase of this code we have instituted testing procedures, code writing styles, and team coordination applicable to a rapidly changing source code, several dependent libraries, and a relatively small team including university collaborators with their own source and platforms. We use modern coding techniques and open source tools when possible for code management and testing including CppUnit, Subversion (and previously GNU ARCH), TiddlyWiki and group chat rooms. Additionally, we are conducting experiments aimed at providing a data set for validation of the fragmentation models.

1 Introduction

The high-performance computing environment continues to provide new opportunities for simulations of physical phenomena. Processes that a few years ago were too complex for accurate computer models are now amenable to computer solutions. Yet the simulation environment contains new challenges because of the diversity of tools and methodology that accompany these new simulations. The code developers must know not only the physics, engineering, and mathematics of their simulation, but also some expertise in a variety of programming languages, parallel libraries or languages, and tools for code development is required. We describe our experiences in the design of a new code based on an ALE-AMR method. The code is currently being used to model target fragmentation properties for high-powered laser systems. We start this paper with a discussion of the application environment and the new challenges for this particular area of simulation including the ALE-AMR approach. The next section, Section 3, overviews some of the basic principles of our code development approach including team development methods, verification and

other issues. Automatic parallelism is obtained by calling parallel libraries and building the code on a parallel framework. We discuss this framework in Section 4. Finally, in Section 5 we give examples of some simulations.

2 Application Environment

The environment of a high-powered laser chamber contains a good deal of open space from the vacuum that surrounds the small fusion target, often a hohlraum, located at the chamber centre. One goal of our simulation code is to model how the target dismantles after being hit by either laser beams or by x-rays that result from the lasers interacting with other target components. For the purposes of our simulation the "target" includes not only the hohlraum where the laser is focused, but also ancillary target elements such as cooling rings, shields, or appendages that improve diagnostic capabilities. Pieces of the target that are closest to target centre where the laser is focused will be vaporized and thus are relatively benign. However target components that are further from the main laser focus point are subject to lower levels of energy and therefore may be fragmented. It is important to determine the size of these fragmented pieces and their velocity vectors after the laser shot so that optics and diagnostics that line the chamber will be protected from damage. Dedicated experiments as well as experience from recent high-powered laser shots provide more information on this environment and the usefulness of mitigation procedures to direct fragments in benign directions.[1,2]

This environment creates a natural basis for the combination of an ALE method with AMR. An Arbitrary Lagrangian Eulerian (ALE) Method is a standard technique whereby the evolution of the continuum equations is performed by an initial Lagrangian step which allows the mesh to deform according to the fluid motion. After this step, the mesh quality is evaluated and an optional remesh step is performed to recover either the original mesh (fully Eulerian) or more likely an intermediate or "arbitrary" mesh that is optimized for the simulation. Combining this method with Adaptive Mesh Refinement (AMR) allows one to not only gain advantages from the ALE technique but also to concentrate the calculation on areas of the domain where the fragments are being formed.

The ALE-AMR method was pioneered by Anderson and Pember in a series of works for hydrodynamics resulting in the basic code that we are using and updating for solution of problems in solid mechanics.[3] Features of our ALE-AMR code that are relevant to the solid mechanics implementation include multimaterial interface reconstruction[4] and material model specific features.[5]

3 Team Development and Tools

There are a variety of free or open source tools that make code development much easier than previously. We briefly discuss a few of these that our team has found useful. We also discuss our verification ideas.

Subversion[6] is a version control system designed for the open source community, released under an Apache/BSD-style opens source license. Our group found the version control system particularly easy to use, and given that some of our developers had previous experience with systems like CVS, using this system was quite straight forward. On

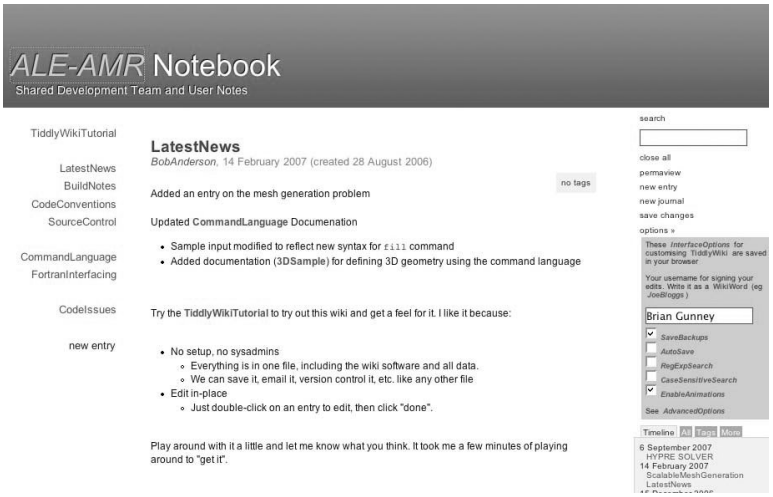


Figure 1. The ALE-AMR notebook demonstrates the use of a wiki for fast code documentation.

top of the subversion system we have scripts that control nightly builds and other checks as part of code verification and maintenance.

The term Wiki, as used in computer science is defined as a collaborative website, in which participants can directly edit the text of the "living" document. Perhaps the most famous implementation of a computer science Wiki is Wikipedia, the online free encyclopedia that is written collaboratively by people throughout the world. Wiki in computer science is derived from the Hawaiian word *wiki* that means to hurry. The notion of "hurry" or fast, is particularly important for us, since it encourages people to enter their documentation directly and quickly without being bound by a particular format. While the wiki itself is not a replacement for a traditional "user's manual," it serves as a convenient place to store daily or weekly information that is easily and quickly accessible. The particular wiki format we have chosen is TiddlyWiki. [7] A picture of our TiddlyWiki is given in Fig. 1. For reference, this figure shows the help message from the basic Wiki available online.

Another resource that our code team has used to great advantage is a "chat room." Use of the chat room allows developers who reside in physically different offices to discuss code changes, machine changes, strategies, etc. For the purposes of chat room security, the following policies are implemented: 1) Off-site chat does not work unless you set up an SSH tunnel to the server, 2) Not all rooms are "public"; a user must be invited before they can successfully join a room, 3) There may be more rooms in existence than what you see: rooms may be defined as invisible to outsiders. Our team has found that this strategy eliminates slow email question and answer periods and also encourages answers from developers who might not be queried on a particular issue. We have noticed a marked increase in the productivity of new team members when their chat room configuration was implemented.

A large portion of our code is based on libraries. The major library/framework that encompasses all of the parallelism is described in the next section. It is a patch-based Adaptive Mesh Refinement (AMR) library/framework. We also use the visualization package

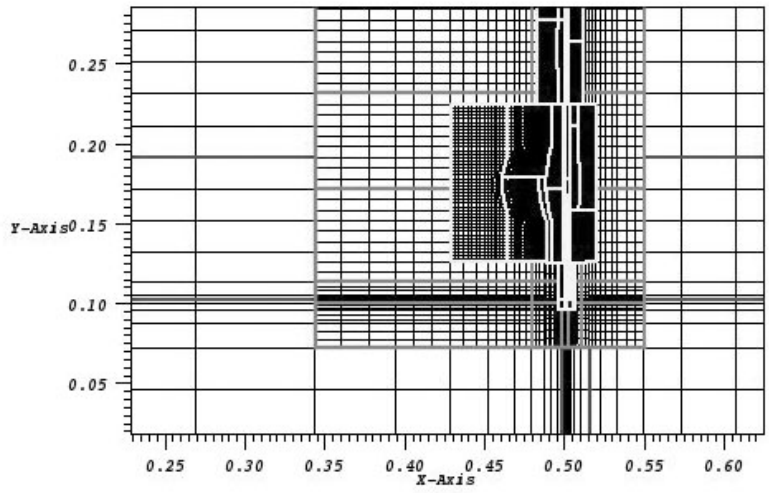


Figure 2. The figure shows the mesh as well as the patch boundaries. VisIt allows one to display mesh only, patch boundaries only, or both. Note that grid lines are intentionally not equally spaced, as is allowed in our ALE-AMR method. This yields better shock-capturing physics in the simulation.

called VisIt .[8] VisIt, which can be downloaded from the web, is particularly well suited to moving hydrodynamic simulations. Enhanced capabilities for parallelism and AMR grids are also included. Figure 2 shows example output from VisIt. Controls in VisIt allow one to selectively look at patches and/or mesh pieces for a better understanding of the simulation. Here we show the combination of both mesh and patches. In some areas concentration of mesh fills in as black. Zoom features allow one to study these regions. Automatic movie making commands are also included. VisIt also automatically handles groups of parallel files that are generated by the parallel code. Finally the visualization engine may be run remotely and/or in parallel. This visualization toolkit also helps in finding code errors and depicting how the automatic refinement works in the AMR scheme. This capability is useful in both finding bugs and also in determining the appropriate trigger for mesh refinement and de-refinement (coarsening).

4 The SAMRAI Framework

SAMRAI is an object-oriented structured AMR (SAMR) library supporting multiscale multiphysics applications and serving as a platform for SAMR algorithms research at LLNL.

SAMRAI factors out most of the common components and tasks in SAMR applications, allowing the application writers to focus on the specifics of their application. Among the most prominent tasks are mesh generation, mesh adaptation, data management, data transfer, high-level math operations, plotting interfaces, restart capabilities and parallelism. Less prominent time-saving services include computing mesh dependencies and mesh boundaries.

SAMRAI's interface design[9] is flexible and expandable while supporting powerful features. Typical mesh data (cell-centred, node-centred and face-centred) are provided. One can seamlessly integrate arbitrary user-defined data without having to modify SAMRAI. For example, arbitrary data residing in sparse cells has been used to represent surfaces cutting through the mesh. Application writers implement a small set of operations whose interfaces are defined by SAMRAI, and SAMRAI integrates the new data as if it were natively supported data. Data interpolation and copying is similarly integrated. SAMRAI natively supports Cartesian meshes, though its interface allows user-defined meshes such as deforming meshes such as that used by our ALE-AMR code.

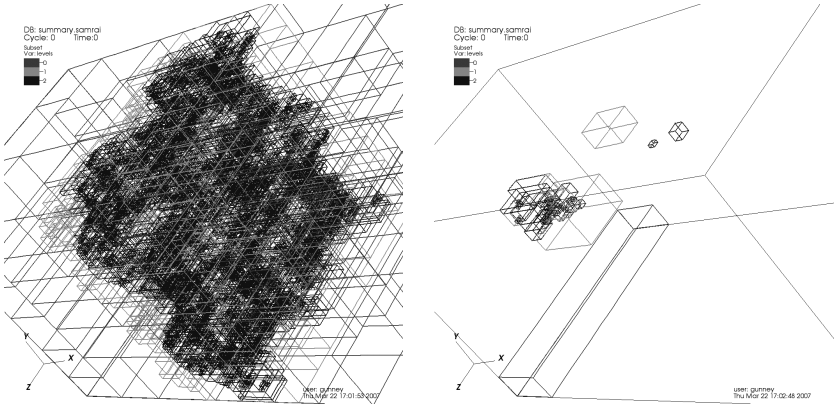
Parallelism is largely hidden, allowing application developers to focus on serial code development and still build a parallel application. All mesh data communications occur outside the interface seen by the application developers. User codes required for supporting user-defined data are all serial. Techniques such as combining messages are employed to enhance communication efficiency.

SAMRAI also serves as a research platform in new SAMR algorithms, resulting in significant recent advances such as box search algorithms,[10] clustering algorithms,[11] load balancing algorithms[12] and a new approach for managing SAMR meshes on distributed memory computers.[12] Figure 3 shows the differences between traditional and new approaches and their associated weak scaling characteristics. The baseline results, from current widely used SAMR algorithms show that although the serial physics engine scales well, other costs increase rapidly and quickly overwhelm the simulation. The new results give much more ideal scaling behaviour.

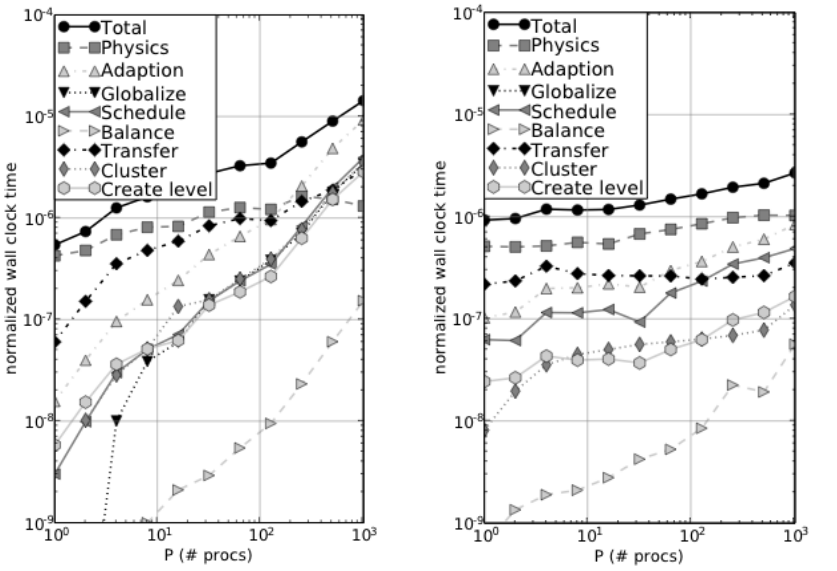
5 Code Applications

The code is being used to model the process of fragmentation and dismantling of high-powered laser facility targets. Figure 4 shows spall off the back of a thin target element that has energy deposition at the front surface of the thin plate in a localized region. This simulation is similar to Fig. 2 that shows the ALE-AMR grid and VisIt options. Here, the energy deposition first causes a plasma blow-off at the front surface of the plate. The top half of the figure is colour-coded to show density contours and the bottom half of the figure is colour-coded to show temperature contours. The plasma blow-off boundary is reconstructed via the multimaterial scheme described in Masters, et al.[4] The interface between the metal and the gas is shown in black. AMR patch boundaries are given in light grey. If this shield were part of a target configuration, we would use the code to determine the direction and quantity of plasma blow-off to the left, and the size and velocity of particle spall off the back. Spall off the back comes from the pressure wave that propagates horizontally through the material following the energy deposition at the front face.

Another simulation of interest to high-powered laser facilities is what happens to material that surrounds a can-shaped hohlraum target. For instance, for the target to be cryogenically cooled, it is often surrounded by metal washer-shaped cooling rings. Figure 5(a) shows a simulation of the failure and fragmentation of a typical Al cooling ring subject to an impulsive loading. Annular shaped spall planes break off from the main cooling ring. In the right view, Fig. 5(b) shows a modification of a cooling ring simulation as described also in Koniges, et al.[2] Here the ring was not a complete annulus, as in full ring, but instead was notched for experimental purposes. Simulations of this configuration showed how



Baseline mesh view | New mesh view



Baseline performance | New performance

Figure 3. The top figures show the mesh as seen by the baseline approach and the new approach. The baseline approach stores and manages the global mesh. The new approach distributes the mesh information to reduce the amount of work per processor. The bottom figures shows the weak scaling characteristics for common tasks in an SAMR applications. For ideal weak scaling, the time for a given task does not vary with the number of processors. The new approach achieves more ideal scaling.

potentially damaging pieces could form when the stress in the ring were not distributed uniformly in the annulus. Such simulations and experiments aid in the validation process.

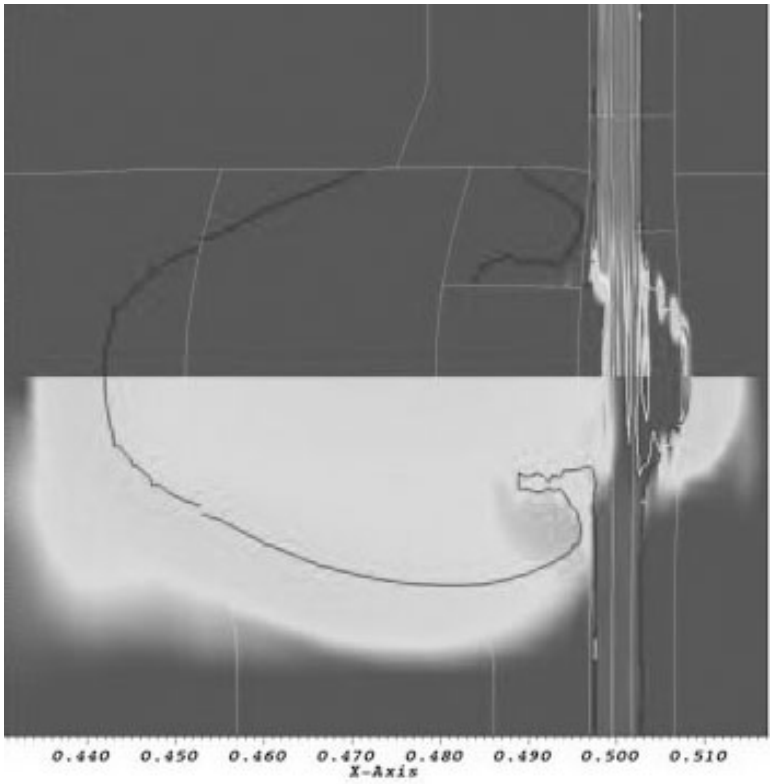


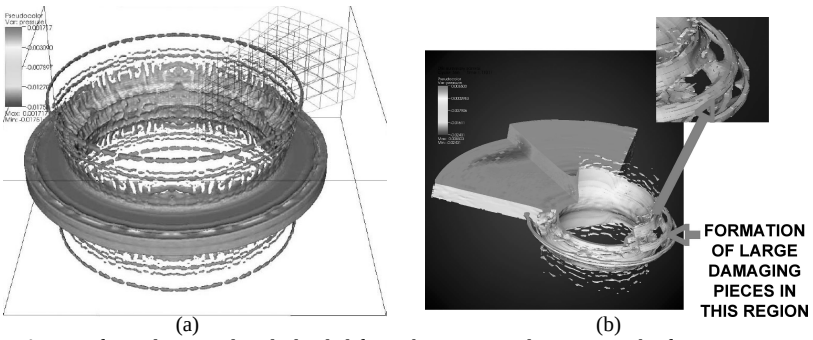
Figure 4. Formation of spall planes off the back of a thin target is shown. Additionally, contours of density (top) and temperature (bottom) are given. Interface reconstruction between the thin target metal and the gas surrounding the target is shown in black. AMR patch boundaries are given in light grey.

6 Summary

In this paper we have described the development of a new simulation code by a team of roughly 4 - 6 people. The code uses a variety of languages and some of the modern interaction techniques such as a wiki and a chat room are described. The code is scalable and parallel, based on the use of an advanced library framework known as SAMRAI. The use of SAMRAI isolates most of the parallel code from the physics code leading to faster development and better debugging opportunity. Additionally, parallel scalability improvements to the SAMRAI library can be seamlessly incorporated into the main code. The code is providing new results for an upcoming area of physics/engineering, namely the protection of high-powered laser facility target chambers.

Acknowledgements

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.



(a) (b)

Figure 5. A ring of metal is impulsively loaded from the centre to demonstrate the fragmentation pattern are target cooling rings. In the left view, the ring is symmetric and loaded from the interior. In the right view, an alternative design has a notched ring. This design shows the formation of potentially damaging pieces during its fragmentation process.

References

1. A. E. Koniges, R. W. Anderson, P. Wang, B. T. N. Gunney and R. Becker, *Modeling NIF experimental designs with adaptive mesh refinement and Lagrangian hydrodynamics* J. Physics IV, **133**, 587–593, (2006)
2. A. E. Koniges, et al., *Experiments for the validation of debris and shrapnel calculations*, J. Physics, (submitted 2007).
3. R. W. Anderson, N. S. Elliott and R. B. Pember, *An arbitrary Lagrangian–Eulerian method with adaptive mesh refinement for the solution of the Euler equations*, J. Comp. Phys., **199**, 598–617, (2004).
4. N. Masters, et al., *Interface reconstruction in two- and three-dimensional arbitrary Lagrangian-Eulerian adaptive mesh refinement simulations* J. Physics, (submitted 2007).
5. A. Fisher, et al., *Hierarchical material models for fragmentation modeling in NIF-ALE-AMR*, J. Physics, (submitted 2007).
6. http://subversion.tigris.org/
7. http://www.tiddlywiki.org/
8. http://www.llnl.gov/visit/doc.html
9. R. D. Hornung and S. R. Kohn, *Managing application complexity in the SAMRAI object-oriented framework*, Concurrency and Computation: Practice and Experience, **14**, 347–368, (2002).
10. A. M. Wissink, D. Hysom and R. D. Hornung, *Enhancing scalability of parallel structured AMR calculations*, in: Proc. 17th ACM International Conference on Supercomputing (ICS03), pp. 336–347, San Francisco, (2003).
11. B. T. N. Gunney, A. M. Wissink and D. A. Hysom. *Parallel clustering algorithms for structured AMR*. Journal of Parallel and Distributed Computing, **66**, 1419–1430, (2006).
12. B. T. N. Gunney. *Scalable metadata and algorithms for structured AMR*, SIAM Computational Science and Engineering Conference (2007), UCRL-PRES-228081.

Submission Scripts for Scientific Simulations on DEISA

Gavin J. Pringle, Terence M. Sloan, Elena Breitmoser, Odysseas Bournas, and Arthur S. Trew

EPCC, University of Edinburgh
Mayfield Road, Edinburgh, EH9 3JZ, UK
E-mail: {g.pringle, t.sloan, e.breitmoser, o.bournas, a.trew}@epcc.ed.ac.uk

The DEISA Services for the Heterogeneous management Layer, better known as the DESHL, allows users and their applications to manage batch jobs and data across a computing Grid in a uniform manner regardless of the underlying hardware and software on the various nodes. The DESHL provides a means for coordinating and integrating services for distributed resource management in a heterogeneous supercomputing environment. It provides users and their applications with a command line tool and application programming interfaces for job and data management in DEISA's UNICORE-based grid.

The DESHL employs the SAGA (Simple API for Grid Applications) and JSDL (Job Submission Description Language) emerging grid standards as well as the Open Group Batch Environment Services specification. Reliance on Grid standards will allow interoperability with other Grid environments.

Three DESHL-based bash scripts have been created which demonstrate how DEISA can be exploited directly from the user's workstation. These scripts are for Task Farms, Workflows, and for Code Migration/Resubmission.

1 Introduction

DEISA[2] - the Distributed European Infrastructure for Supercomputing Applications - is currently a Grid of eleven supercomputers spread across eight different European countries, where homogeneous access is given to this Grid of heterogeneous supercomputers.

There are several user interfaces to the DEISA infrastructure, including UNICORE[3] and the DESHL[1,4], or DEISA Services for the Heterogeneous management Layer.

A UNICORE GUI provides a seamless interface for preparing and submitting jobs to a wide variety of computing resources. An alternative command-line interface is provided by the DESHL.

The creation of the DESHL was driven by the fact that a large number of users prefer to use the command line, rather than a GUI, when managing their supercomputing simulations. These users are typically more accustomed to accessing HPC resource via the command line, however, some users simply cannot use the mouse for physical reasons and require command line access.

A light weight DESHL client is run on the users workstation, Linux/Solaris or Windows/DOS, where a user can transfer files to and from their workstation and any of the DEISA sites, and also submit and monitor DEISA batch jobs. The client sits on top of the existing UNICORE infrastructure, but is not dependent on the UNICORE graphical client.

Thus, the DESHL provides simplified and seamless access to DEISA resources, without needing to log into any individual site. Indeed, under the DEISA access model, direct login to any site other than the users home site is not permitted. Authentication/Authorisation is by the existing UNICORE mechanisms: any valid X.509 certificates for UNICORE are also valid certificates for DESHL.

A set of DESHL-based bash scripts have been created to permit users to launch supercomputing applications on DEISA from their workstation, without the need for multiple usernames and passwords, nor the need to learn the various flavours of compilers, batch systems or performance characteristics of each of the various platforms.

There are four bash script suites available, which can be employed for Code Resubmission, Code Migration, Simulation Workflows and Task Farms.

This paper presents the basics of the DESHL frameworks, and then contains a detailed discussion of each of the four suites.

2 The DESHL

The DESHL allows users and their applications to manage batch jobs and data across a computing grid in a uniform manner regardless of the underlying hardware and software on the various nodes of a grid. The DESHL employs emerging grid standards so that a user and their applications are not affected by the differences or changes in the hardware and software configuration on the grid nodes.

Through its use of UNICORE, the DESHL shields users from the underlying platform and batch scheduler heterogeneity of the DEISA infrastructure. It also even further insulates the users from the UNICORE middleware itself. The use of standards at the command line and API levels allows users to work in a grid-independent manner at the application, command line and batch script level.

The DESHL software has been developed by DEISA's seventh Joint Research Activity (JRA7). EPCC and ECMWF from the UK, FZJ from Germany and CINECA from Italy are the participants in this research activity. The DEISA JRA7 activity is using modern Grid standards to develop a means for coordinating and integrating services for distributed resource management in a heterogeneous supercomputing environment.

The DESHL employs the SAGA[5] (Simple API for Grid Applications) and JSDL[6] (Job Submission Description Language) emerging grid standards as well as the Open Group Batch Environment Services specification[7]. Reliance on Grid standards will allow interoperability with other Grid environments.

The DESHL command line tool job management and file transfer capabilities include: determining the DEISA sites to which a user can submit a batch job to; submitting a batch job to a DEISA site; terminating a batch job at a DEISA site; viewing the status of a batch job on a DEISA site; uploading a file to a DEISA site; down-loading a file from a DEISA site; deleting a file from a DEISA site; determining if a file exists on a DEISA site; listing the contents of a directory on a DEISA site; renaming a file on a DEISA site; and, copying/moving a file between DEISA sites.

The DESHL supports the command line tool operations through a SAGA-based API, the DESHL Client library, which simplifies access to heterogeneous computing and data resources. This SAGA API interfaces with the more powerful and complex Grid Access library (also known as Roctupus) that in turn interacts with the low-level UNICORE specific library, ARCON[8]. Any of these APIs can be used by Java applications developers as appropriate.

The DESHL also includes a GUI-based installer, an installation and user manual, a design description, and API documentation for the DESHL Client and Grid Access libraries.

The DESHL Client library was the first publicly-available Java-based implementation

of file and job management using the proposed SAGA grid standard. More importantly, it was the first SAGA implementation to be actively deployed, tested against and used with a production-level, continental grid infrastructure and thus represents an important contribution to global Grid computing.

2.1 UNICORE

The DESHL sits on top of the UNICORE-based DEISA Grid, where UNICORE has a three-tier design: the client GUI, the Gateway and the Target System Interface. A UNICORE client GUI is used for the preparation, submission, monitoring, and administration of jobs. The Gateway is a site's point of contact for all UNICORE connections and is typically the user's so-called Home Site. It also checks if the user's certificate is signed by a trusted CA. Execution Site specific information on the computing resources, including the availability of applications, is provided by a Network Job Scheduler (NJS). This server dispatches the jobs to a set of dedicated target machines or clusters, known as the Execution Sites, and handles dependencies and data transfers for complex workflows. It transfers the results of executed jobs from the target machine. A Target System Interface (TSI) is a server that runs on each of the Execution Sites, interfacing to each specific batch scheduler.

3 DESHL-Based Bash Scripts

Four suites of DESHL-based bash scripts have been developed to demonstrate the DESHL's ability to perform Code Resubmission, Code Migration, Workflow Simulations and Task Farms. All four suites are freely available from the authors.

Code Resubmission is where an application is launched and, if the queue time limit expires before the simulation has completed, the script repeatedly resubmits the job.

Code Migration is where an application is launched on one platform and completes on another. This is not to be confused with Job Migration, where a job is submitted to 'local' platform's batch system, and is migrated to another batch queue on a 'remote' platform where the job runs on this 'remote' platform.

Workflow Simulations are multi-step simulations which may also have many executables, which run in a given sequence, where the output of one forms the input to the next. This is a common scenario for UNICORE users.

Task Farms are a collection of independent parallel jobs, where the master coordinates groups of processors to concurrently execute each job.

3.1 Code Resubmission

A Resubmission Suite is one where the application is launched and, if the queue time limit expires before the simulation has ended, the Resubmission Suite repeatedly resubmits the job until the simulation is complete. Very large simulations may require several days to complete and, by default, no DEISA site currently permits jobs longer than 24 hours.

We have developed a DESHL-based bash script which the user can employ to submit such a large job. The job is submitted only once. The suite performs the work normally carried out by the user in such circumstances, namely job monitoring and simulation restarts.

The process is as follows: the batch queue time limit is passed to the application upon submission. The application itself then regularly checks how much time remains. If time is short, then the application dumps the restart file and exists cleanly. During this time, the Resubmission Suite has been monitoring the simulation and, once the application is found to have exited, then the simulation is submitted again. The process repeats until the simulation is complete.

A Simulation Code should have the following two characteristics to be able to migrate:

Restart capability: the application code must be able to create a restart file, where this file contains sufficient information for the same simulation to continue running from the point at which the restart file was created. These files are sometimes referred to as checkpoint files. Typically, these files can be created at fixed intervals of: wall-clock seconds or at a fixed number of simulation time steps. The application automatically employs the most recent restart file as its 'initial conditions'.

Batch awareness: to be fully automated, the application code must have a measure of how much time remains within its associated batch queue. This can be achieved quite simply: the application code includes an internal timer which measures how long the simulation has been running. Then, by passing the associated wall clock limit to the application, as a runtime parameter, the application may check at, say, every time step to measure how much wall clock time remains. The batch queue time limit is passed to the application upon submission.

3.2 Code Migration

Code Migration is an extension of the Code Resubmission, where multiple sites are employed rather than just a single site. It is an important feature for simulations that require more resources than any DEISA site offers, such as disk space or computer cycles. We refer to this process as Code Migration. This is not to be confused with Job Migration, where a job is submitted to one platform, and is migrated to another batch queue on another, separate platform.

As with Code Resubmission, the associated batch queues maximum time limit is passed to the application, and the application dumps the restart file automatically before the time limit is reached. However, in for Code Migration, the application is then automatically resubmitted to the *next* platform to be employed within the DEISA infrastructure. The process repeats until the simulation is complete.

As for Code Resubmission, the application code itself must have certain capabilities.

Restart capability: à la Code Resubmission.

Batch awareness: à la Code Resubmission.

Portable Data files: due to the heterogeneous nature of DEISA, the application must utilise a portable (binary) data format, for example, HDF5 or NetCDF. This will ensure that the restart file does not restrict the choice of platform.

Pre-ported application: the application is installed, by DEISA staff, on each of the participating platforms. This ensures each invocation will utilise a tuned application. Employing staged executables restricts the user to homogeneous platforms only and does not ensure that each executable is optimised nor hyper-scaled. (In this context, to stage a file is to place it on a remote platform immediately prior to its use, and to hyper-scale an application code is to ensure that the code scales to thousands of processors.)

3.3 Simulation Workflows

A Simulation Workflow is a simulation which has many executables, run in a prescribed order, where the output of one forms the input to the next. This is a common scenario for UNICORE users. A Simulation Workflow is a more simple form of the Code Migration Suite where different application codes run on each site, rather than the same code. The DESHL can be employed to implement all of the UNICORE workflow schema, such as 'do-while', 'repeat-until', 'if-then', etc.

If resubmission is not required, then the only restrictions are that portable binary data files must be employed and that executables should be pre-installed rather than staged.

3.4 Task Farms

A DESHL-based Task Farm Suite, for multiple independent parallel jobs, has been created. Here, the user's workstation assumes the role of the master and the each group of worker processors are individual DEISA platforms.

As before, employing portable (binary) data input files is required, so that we are not limited by our choice of platforms. Further, we also assume that the executable has been installed, by DEISA staff, on each of the participating DEISA platforms.

The process is as follows: the master submits a number of parallel jobs to each of the participating platforms and then monitors each job; once a job has completed, the output is retrieved to the user's workstation; and the next job sent to that particular platform.

Note that DEISA platforms limit the number of jobs that can be submitted at any given time, where this limit can range from one to tens of jobs, depending on the each site's local configuration. This site-dependent variation is hidden from the user by the DESHL script and the user simply sees the Task Farm running to completion.

Currently, the Task Farm Suite assumes that each job runs within a single site, however, this restriction can be lifted if the job is actually two applications which are loosely-coupled, as DEISA supports co-scheduling.

It is worth noting that Task Farms are currently not possible with UNICORE GUI.

4 Pseudo-Code of the Code Migration Suite

This section contains a step-by-step discussion of how the Code Migration Suite proceeds. The pseudo-code of the other Suites are not presented as the other Suites are straightforward extensions of this Code Migration Suite.

We assume that the user runs the DESHL client on his local workstation, where this client manages the job submissions, job monitoring and staging of data. Further, we assume that only two platforms are required for a two part job. Lastly, all input and output files are stored locally, on the user's workstation.

Stage the startup files. The input files are sent from the user's workstation to the first platform's `USPACE`, using the `deshl copy` command. Here `USPACE`, or User SPACE, is a UNICORE construct and is a automatically created, hidden directory reserved for use by UNICORE and, hence, the DESHL, and is used as a temporary scratch directory. When employing the `deshl copy` command, wildcards can only be used when staging data into the `USPACE`. At present, wildcards cannot be employed when copying data out of

USPACE. Therefore, we stage files via a bash script, which creates a tar file of all the files to be staged, which is then moved to the target machine by DESHL. Then, on the target machine, another script unpacks the tar file upon arrival. Indeed, this method may be faster than employing wildcards for a large number of large files.

Run first part of the job. The executable is then submitted to the first platform, at the local workstation, using the `deshl submit` command.

Monitor Job Status. The job's status is then determined by repeatedly calling the `deshl status` command at a fixed interval. This is currently set to one minute. A `sed` operation is performed on the output to isolate the 'state', where the state can be `Running` (which actually means running or pending), `Done` or `Failed`.

Once Complete, Retrieve Restart File. Once the application has closed successfully, the restart file is copied back to the workstation using the `deshl fetch` command.

Stage the startup files The restart files required for the second half of this job are then staged to the second platform's USPACE using the tar script and the `deshl copy` command. NB: the restart files can be staged from the USPACE of the first platform directly to the USPACE of the second platform, however, in this example, the user wishes to examine the state of the intermediate restart file.

Run last part of the job The executable is then submitted to run at the second platform using the `deshl submit` command.

Monitor Job Status Again, the status is monitored by polling the second platform's batch queues with repeated calls to the `deshl status` command.

Once Complete, Retrieve Results Once the simulation has completed successfully, the output files are taken from the second platform's USPACE back to the local workstation using the `deshl fetch` command.

5 Brokering

If a broker is introduced, then Code Migration could be employed to repeatedly submit a job to the *next available* platform, rather than a particular platform. This would significantly reduce the user's time-to-solution, as no time is spent waiting in batch.

Further, if the requested batch queues are small, i.e. a small number of processors and/or a short length of time, then the Task Farm Suite could be used to soak up any spare cycles being offered as national services drain their queues when switching from different modes of operation, i.e. switching between day and night modes, where a day mode typically has many smaller queues, whilst a night mode might disable all interactive queues. There are also modes when a whole machine must be drained when preparing for huge simulations which require the whole machine, i.e. preparing for Capability Runs.

Soaking up these spare cycles is, effectively, cycle scavenging, and would contribute to higher utilisation figures of all participating sites.

6 Conclusion

In this paper, we have described the basic form of the DESHL, or the DEISA Services for the Heterogeneous management Layer.

We then described four Suites, formed from DESHL-based bash scripts, namely Code Resubmission, Code Migration, Workflow Simulations and Task Farm, along with the necessary alterations required to the application code(s). All four suites are freely available from the authors.

Thus, users can now submit multiple, very long/large simulations, with complex workflows directly from their workstation, which will run on any number of remote platforms with DEISA's heterogeneous supercomputing Grid. What makes this more remarkable is that the user does not need knowledge of the local vagaries of any of the platforms, only requires a single DEISA Certificate, and does not need to monitor nor nurse the simulation to completion.

We thank the DEISA Consortium, co-funded by the EU, FP6 projects 508830/031513.

References

1. T. M. Sloan, R. Menday, T. Seed, M. Illingworth and A. S. Trew, *DESHL - standards-based access to a heterogeneous European supercomputing infrastructure*, in: Proc. 2nd IEEE International Conference on e-Science and Grid Computing - eScience 2006, 4th–6th December, Amsterdam, (2006).
2. DEISA: Distributed European Infrastructure for Supercomputing Applications, http://www.deisa.eu
3. UNICORE, http://www.unicore.eu
4. DESHL, http://deisa-jra7.forge.nesc.ac.uk
5. OGF (Open Grid Forum) SAGA (Simple API for Grid Applications) Research Group. https://forge.gridforum.org/projects/saga-rg
6. OGF (Open Grid Forum) JSDL (Job Submission Description Language) Working Group. http://forge.gridforum.org/projects/jsdl-wg
7. Open Group specification for Batch Environment Services. http://www.opengroup.org/onlinepubs/009695399/utilities/xcu_chap03.html
8. UNICORE 5 ARCON Client Library documentation. http://www.unicore.eu/documentation/api/unicore5/arcon

Application Enabling in DEISA: Petascaling of Plasma Turbulence Codes

Hermann Lederer[1], Reinhard Tisma[1], Roman Hatzky[1], Alberto Bottino[2], and Frank Jenko[2]

[1] Rechenzentrum Garching der Max-Planck-Gesellschaft
Boltzmannstr. 2, D-85748 Garching, Germany
E-mail: {lederer, tisma, hatzky}@rzg.mpg.de

[2] Max Planck Institut für Plasmaphysik
Boltzmannstr. 2, D-85748 Garching, Germany
E-mail: {jenko, bottino}@ipp.mpg.de

The ITER (International Thermonuclear Experimental Reactor) experiment must be accompanied by advanced plasma turbulence simulations. Due to the high demands for compute power and memory, simulations must be capable of using thousands or tens of thousands of processor-cores simultaneously. Highly scalable applications are mandatory.

Through a joint effort of application specialists from DEISA (Distributed European Infrastructure for Supercomputing Applications) and scientists engaged in the theory support for ITER, two important European simulation codes for core turbulence, ORB5 and GENE, have been adapted for portable usage within the heterogeneous DEISA infrastructure.

Moreover, the codes were thoroughly analyzed, bottlenecks were identified and removed, and, most importantly, the scalability of the codes could be significantly enhanced. Through application of the domain cloning concept, the PIC code ORB5 was enabled for high scalability. Efficient usage of ORB5 code could be demonstrated up to 8k processors, both on a Cray XT3 and on an IBM BlueGene/L system. GENE was parallelized through domain decomposition of the five-dimensional problem grid to such a high degree that close to loss-free efficiency on up to 32k processors of an IBM BlueGene/L machine was achieved. Results combined from both strong and weak scaling measurements indicate an even higher scalability potential for GENE. Extrapolations suggest an efficient usage on up to the order of 1M processor-cores of a similarly scalable future HPC architecture is possible, representing a milestone on the way towards realistic core turbulence simulations of future fusion devices.

1 Introduction

ITER[1] is a joint international research and development project that aims to demonstrate the scientific and technical feasibility of fusion power as a viable future energy option offering long-term, safe, and environmentally benign energy to meet the needs of a growing and developing world population. The partners in the project — the ITER Parties — are the European Union (represented by EURATOM), Japan, the People's Republic of China, India, the Republic of Korea, the Russian Federation and the USA. ITER will be constructed in Europe, at Cadarache in the South of France.

EFDA, the European Fusion Development Agreement, is an agreement between European fusion research institutions and the European Commission to strengthen their coordination and collaboration, and to participate in collective activities. EFDA, in supporting ITER, is also favouring a strong theory support that will play an essential role. Large scale simulations require most powerful supercomputers. The DEISA, an EU FP6 project, has

developed and put into operation a grid of the most powerful supercomputing platforms in Europe, but DEISA also provides advanced application support. Application enabling towards petaflops computing of two important European plasma core turbulence codes, ORB5 and GENE, is described here in detail.

2 DEISA

Major European supercomputing centres have combined their competences and resources to jointly deploy and operate the Distributed European Infrastructure for Supercomputing Applications, DEISA[2], to support leading edge capability computing for the European scientific community.

2.1 Extreme Computing and Applications Enabling in Europe

Application enabling is of key importance for adequate usability of state-of-the-art and next generation supercomputers. A team of leading experts in high performance and Grid computing, the Applications Task Force, provides application support in all areas of science and technology. In 2005 the DEISA Extreme Computing Initiative (DECI) was launched for the support of challenging computational science projects. One of the key tasks is hyperscaling, a unique service not offered by any other European Grid computing project. Hyperscaling aims at application enabling towards efficient use of thousands or tens of thousands of processors for the same problem, a prerequisite for applications to benefit from forthcoming Petaflop scale supercomputers.

2.2 DEISA and the European Plasma Physics Community

DEISA also maintains Joint Research Activities in the major fields of computational sciences. The DEISA Joint Research Activity in Plasma Physics is advised through Principal Investigators like Karl Lackner (former EFDA leader) from Max Planck Institute for Plasma Physics (IPP), Garching, and Laurent Villard from Centre de Recherches en Physique des Plasmas (CRPP), Lausanne. The objective is the enabling and optimization of important European plasma physics simulation codes in DEISA. So far the simulation codes TORB, ORB5, GENE, EUTERPE, and GEM have been enabled and optimized. The enabling of ORB5 and GENE is reported here.

3 Enabling of ORB5 simulation code

The ORB code family uses a particle-in-cell (PIC), time evolution approach, and takes advantage of all the recent techniques of noise reduction and control in PIC simulations. In particular, it uses a statistical optimization technique that increases the accuracy by orders of magnitude. Initiated at CRPP, Lausanne, ORB has been substantially upgraded at IPP, Garching. The ongoing code development is made under a close collaborative effort.

The ORB5 code[3][4][5] is able to simulate plasmas of higher complexity which can be used e.g., to simulate effects such as long-living zonal flow structures analogous to those seen in the Jovian atmosphere, or so-called Geodesic Acoustic Modes (GAM). Of course,

a higher complexity of the simulation model has its impact on the complexity of the programming model. Since ORB5 has high relevance for ITER, special effort was given to the ORB5 code to enable it to run with high scalability on existing DEISA and further relevant systems.

3.1 Single Processor Optimization and Portability in DEISA

The code was instrumented to detect bottlenecks and the most CPU time consuming routines were identified. Two bottlenecks were identified: the handling of Monte Carlo particles and the implementation of the FFT (Fast Fourier Transform).

The particle handling was improved by a cache sort algorithm that sorts the Monte Carlo particles relative to their position in the grid cells of the electrostatic potential. Hence, a very high cache reuse of the electrostatic field data could be achieved, significantly improving the Mflop rates of two important routines. The overhead introduced by the sort routine itself was minimized. In addition, a switch was implemented to optionally enlarge a work array for the sorting process: this can speed up the sorting routine by a factor of 3. Usually the resident memory size of the simulation is small enough to take advantage of this new feature.

The bottleneck related to the FFT was due to the inclusion of the source of an own FFT implementation with very poor performance. Therefore a module was written with interfaces to important optimized FFT libraries: FFTW v3.1, IBM ESSL, and Intel MKL. Performance improvements up to a factor of 8 could be measured. All three named FFT library routines have the advantage that they are no longer restricted to vector lengths of powers of two. This results in a much higher degree of flexibility when choosing the grid resolution of the electrostatic potential. With the new FFT module, all DEISA architectures besides the NEC vector system are supported.

3.2 Scaling of ORB5

The domain cloning concept was implemented in the ORB5 code to optimize scaling and decouple the selectable grid resolution from the number of processors used for the simulation. The relatively new parallelization concept is a combination of the two techniques domain decomposition and particle decomposition. This strategy[6] was applied on ORB simulations[7][8]. With the new ORB5, a simulation with 5×10^8 particles was first tested on the IBM Power4 system at RZG, achieving a speedup of 1.9 from 256 to 512 processors. Next ORB5 was tested on the Cray XT3 system (Jaguar) at ORNL in strong scaling mode up to 8k processor-cores. The Ion Temperature Gradient (ITG) driven simulation was based on a grid of $256 \times 256 \times 256$ cubic B-splines and 512M particles. On 8k processors a parallel efficiency of 70% (relative to 1k processors) was demonstrated for a fixed problem size requiring about 400 GB of main memory. ORB5 was then ported to and tested on IBM BlueGene/L. An Electron Temperature Gradient (ETG) driven simulation with a grid of $256 \times 256 \times 256$ quadratic B-splines and 800M particles was done up to 8k processors on the BlueGene/L system at IBM Watson Research Center. In strong scaling mode, a high parallel efficiency of 88% was achieved on 8k processors (relative to 1k processors, in so-called co-processor mode when the co-processor is used as an off-load engine to help with communication but not with calculations). Tests on higher processor numbers were

not possible due to the principle memory limitations on the BlueGene/L system (0.5 GB per node). This situation will improve with BlueGene/P systems expected soon at DEISA sites. In addition, weak scaling measurements were done for the same ETG simulation (using about 0.8M particles per processor), fully exploiting the available memory by always executing with the largest problem size possible per processor number. After a slight super-linear behaviour on 2k and 4k processors, due to increasing cache reuse, the parallel efficiency approached 1 again on 8k processors. This proves that a 4 TB problem case can be efficiently treated without degradation on 8k BlueGene/L nodes in co-processor mode.

4 Enabling of GENE Simulation Code

GENE[9][10][11][12], is a so-called continuum (or Vlasov) code. All differential operators in phase space are discretized via a combination of spectral and higher-order finite difference methods. For maximum efficiency, GENE uses a coordinate system which is aligned to the equilibrium magnetic field and a reduced (flux-tube) simulation domain. This reduces the computational effort by 2–3 orders of magnitude. Moreover, it can deal with arbitrary toroidal geometry (tokamaks or stellarators) and retains full ion/electron dynamics as well as magnetic field fluctuations. At present, GENE is the only plasma turbulence code in Europe with such capabilities.

GENE version 9 (GENE v9) employed a mixed parallelization model with OpenMP for intra-node communication in an SMP node, and MPI for inter-node communication across SMP nodes. Architectural characteristics of large SMP-based systems were fully exploited. However, for the number of possible MPI tasks, a hard limit of 64 was detected. On large SMP-based systems as IBM p690 with 32 processor-cores per SMP, this was not a limiting factor, since theoretically up to 2048 ($= 32 \times 64$) processor-cores could have been used.

Scalability tests with scientifically relevant problem cases revealed scalability bottlenecks already starting at 256 processors, with a relative speedup of 1.5 from 256 to 512 processors on IBM p690 with the IBM High Performance Switch interconnect. And for small-node based systems with two processor-cores per node, the technical upper limit was usage of 128 processor-cores, with 64 MPI tasks and 2 OpenMP threads per task.

4.1 Improving the scalability of GENE

As described in detail elsewhere[8], the structure and the parallelization scheme of the code was further analyzed. The GENE code has a total of 6 dimensions with the potential for parallelization. The spatial coordinates x and y, originally treated serially, contain significant potential for domain decomposition. A large number of 2-dimensional FFTs are done on the xy planes. If the xy plane is distributed, it must be transposed in order to perform the FFT in the x and y directions. The transposition, however, is communication intensive since it requires an all-to-all communication. A major change to the overall data structure with consequences for many parts of the code was done by the code authors. The new code version GENE v10 was tested on RZG's IBM p690 system. The speedup from 256 to 512 processors is significantly improved from 1.5 to 1.94 with GENE v9. It was assumed that with the described measures, the scalability of the code was pushed towards efficient usability on up to several thousands of processors.

4.2 Porting GENE to the Major Supercomputing Architectures in DEISA

The GENE v10 code originally used Bessel functions from the NAG library. Three new subroutines were written implementing these Bessel functions for GENE. Further library routines used are FFTs. Interfaces were provided or added for the FFT-routines from the optimized libraries IBM ESSL, public domain FFTW, the Math Kernel Library (MKL) from INTEL, and ACML from AMD.

4.3 Petascaling of GENE

Tests of GENE v10 on higher processor numbers were first done on the Cray XT3 system (Jaguar) at ORNL. Up to 4k (single core) processors, a linear speedup was achieved for strong scaling of a fixed grid size, requiring about 300–500 GB of main memory. For tests on higher processor numbers, the BlueGene/L system at IBM Watson Research Center was used. Here a functionally and algorithmically improved code version, GENE v11+, was used, with the same parallelization scheme as in GENE v11. Strong scaling measurements showed close to linear speedup up to 4k processors; on 8k processors, a parallel efficiency of 73% was achieved, revealing some degradation. For further scalability improvements well beyond 8k processors, the second velocity dimension was parallelized in addition. Measurements with the new code version GENE v11+ on BlueGene/L demonstrate the significant improvements achieved, with excellent scaling behaviour well into the range between of 10^4 and 10^5 processor-cores. In strong scaling mode from 1k to 16k processors, a speedup of 14.2 was achieved, corresponding to an efficiency of 89% (Fig. 1). Here, the problem size was $64 \times 32 \times 128$ grid points in the radial, binormal, and parallel direction, 64×32 grid points in $(v_\parallel, \mu)$ velocity space, and two species, corresponding, e.g., to the physical situation described in a recent publication[12]. Complementary strong scaling measurements in so-called virtual node mode, using both processors per node for computation, reveal a comparable good scalability, with only a small performance degradation of approx. 15% (Fig. 2). For weak scaling runs (using a fixed problem size per processor of about 200 MB) an excellent parallel efficiency of 99% from 2k up to 32k processors is demonstrated (Fig. 3). Here, the grid in the 2k case corresponds to the one described above, while for the other cases, the resolution in the parallel direction and in velocity space has been adapted accordingly.

In this context, we would like to note that various processor grid layouts have been measured for each number of processors, and the best performing processor grid layout with the shortest execution time for one time step was selected. The results from the two different machines can, after normalization with the known factor of 1.022, be combined into one extremely long scaling curve, with 15 measuring points covering four orders of magnitude (from 2 to 32k). The result is shown in Fig. 4, after normalization of the absolute values (time per time step) on the result for 2 processors. A measurement for only one processor was not possible, since at least the two species ion and electron had to be considered and these must be treated in parallel.

The results of the strong and of the weak scaling measurements can now be combined to extrapolate the scalability of the code beyond the actually measured maximum number of processor-cores. For strong scaling (Fig. 1) excellent scalability was proven up to 16 times the number of processors used for the base run (1k). The problem size used there ($\approx$0.5 TB), corresponding to the 2k processor measurement in Fig. 4, can now be

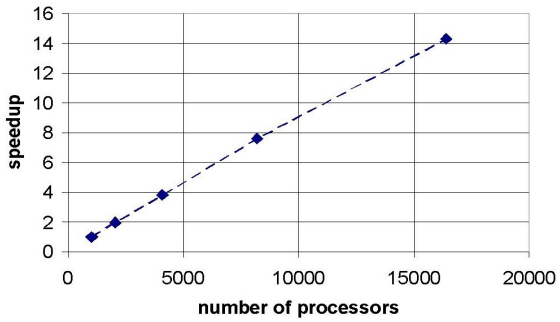


Figure 1. Strong scaling of GENE v11+ on BlueGene/L, normalized to 1k processor result (problem size of ≈300–500 GB; measurements in co-processor mode at IBM Watson Research Center)

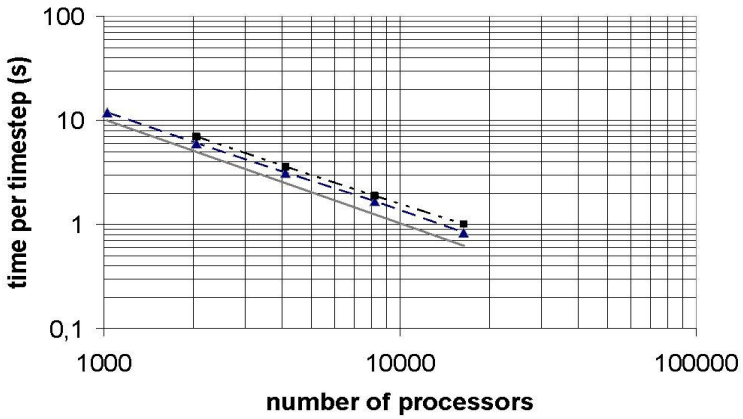


Figure 2. Strong scaling of GENE v11+ on BlueGene/L at IBM Watson Research Center (problem size of ≈300–500 GB; triangles with dashed line: co-processor mode; squares with pointed dashed line: virtual node mode (upper curve); linear scaling: straight line)

stepwise increased (doubled), and the strong scaling curve can be shifted along the weak scaling curve to the right up to the maximum processor number of the weak scaling curve. Through this extrapolation (by increasing the problem size from 0.5 TB to 8 TB), strong scaling of an 8 TB problem size on a scalable architecture is expected to result in good scalability up to $32\text{k} \times 16 = 512\text{k}$ processors.

However, for the BlueGene/L measurements presented so far, only processor grid layouts with y dimension equal to 1 were used, the parallelization of the y dimension has not yet been exploited, since it is most communication intensive. Increasing values of y from 1 to 4 in the grid layout increases the number of MPI threads from 0.5M to 2M for an 8 TB problem size. Scalability tests of y revealed excellent scaling behaviour within an IBM shared memory p690 node (up to 32 processor-cores). Therefore multi-core based archi-

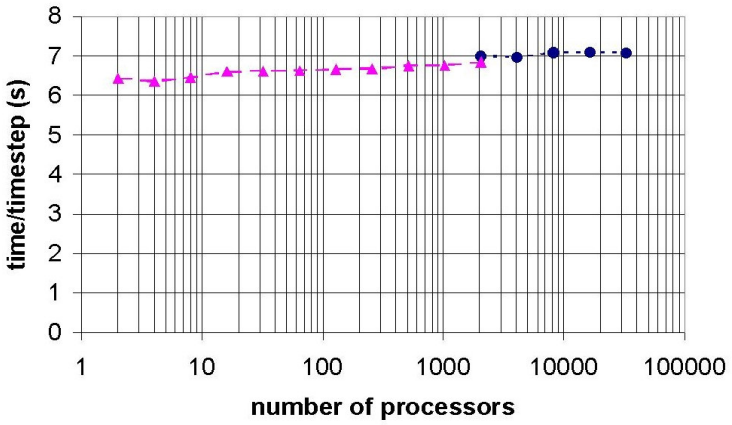


Figure 3. Weak scaling of GENE v11+ on BlueGene/L in virtual node mode (problem size per processor ≈200 MB); circles: measurements at IBM Watson Research Center from 2k to 32k processors (efficiency of 99% with 32k processors when normalizing on the 2k result); triangles: measurements at IBM Rochester Center from 2 to 2k processors; machine speeds differ by 2.2% for the 2k processor results)

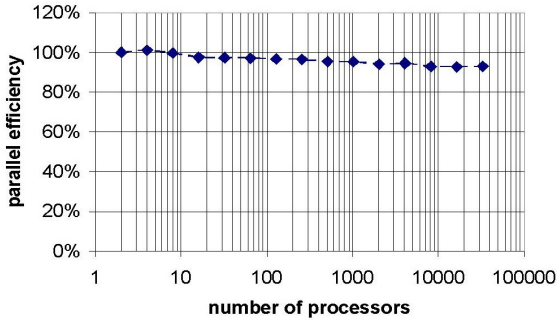


Figure 4. Weak scaling of GENE v11+ on BlueGene/L from 2 to 32k processors over four orders of magnitude (in virtual node mode, problem size per processor ≈200 MB), through combination of the two curves from Fig. 3 and normalization on the 2 processor result. Parallel efficiency on 32k processors, based on the 2 processor run, is 93%

tectures appear especially interesting for the additional exploitation of the y parallelism of GENE.

5 Conclusions

Two important European plasma core turbulence simulation codes, GENE and ORB5, were ported to the major supercomputing architectures in DEISA and beyond, and were optimized and adapted to very high scalability, as a milestone on the way towards realistic core turbulence simulations of future fusion devices. In addition, GENE can be considered a

real physics code able to diagnose the scalability of Petaflops architectures into the range of millions of processor-cores.

Acknowledgements

The authors thank the European Commission for support through contracts FP6-508830 and FP6-031513. We thank IBM for access to the BlueGene/L systems at Watson Research Center and at Rochester Center, and J. Pichlmeier for support on using the systems. We thank Cray Inc. for scalability runs on the Cray XT3 system at ORNL.

References

1. ITER http://www.iter.org
2. Distributed European Infrastructure for Supercomputing Applications (DEISA), http://www.deisa.org
3. T. M. Tran, K. Appert, M. Fivaz, G. Jost, J. Vaclavik and L. Villard, *Energy conservation for a nonlinear simulation code for ion-temperature-gradient-driven (ITG) modes for the theta-pinch*, in: Theory of fusion plasmas: Proc. Joint Varenna-Lausanne International Workshop, 1998, edited by J. W. Connor, E. Sindoni and J. Vaclavik, p. 45, Società Italiana di Fisica, Bologna, (1999).
4. A. Bottino, A. G. Peeters, R. Hatzky, S. Jolliet, B. F. McMillan, R. M. Tran and L. Villard, *Nonlinear low noise particle-in-cell simulations of ETG driven turbulence*, Physics of Plasmas, **14**, Art. No. 010701, (2007).
5. S. Jolliet, A. Bottino, P. Angelino, R. Hatzky, T. M. Tran, B. F. McMillan, O. Sauter, K. Appert, Y. Idomura and L. Villard, *A global collisionless PIC code in magnetic coordinates*, Computer Phys. Comm., **177**, 409–425, (2007).
6. C. C. Kim and S. E. Parker, *Massively parallel three-dimensional toroidal gyrokinetic flux-tube turbulence simulation*, J. Comp. Phys., **161**, 589–604, (2000).
7. R. Hatzky, *Domain cloning for a Particle-in-Cell (PIC) code on a cluster of symmetric-multiprocessor (SMP) computers*, Parallel Comp., **32**, 325–330, (2006).
8. H. Lederer, R. Hatzky, R. Tisma., A. Bottino, and F. Jenko, *Hyperscaling of plasma turbulence simulations in DEISA*, in: Proc. 5th IEEE workshop on Challenges of Large Applications in Distributed Environments (CLADE) 2007, Monterey Bay, pp. 19–26, (ACM Press, New York, 2007).
9. F. Jenko, et al., *Electron temperature gradient driven turbulence*, Physics of Plasmas, **7**, 1904, (2000).
10. F. Jenko and W. Dorland, *Prediction of significant tokamak turbulence at electron gyroradius scales*, Phys. Rev. Lett., **89**, Art. No. 225001, (2002).
11. T. Dannert and F. Jenko, *Gyrokinetic simulation of collisionless trapped electron mode turbulence*. Physics of Plasmas, **12**, Art. No. 072309, (2005).
12. P. Xanthopoulos, F. Merz, T. Görler and F. Jenko, *Nonlinear gyrokinetic simulations of ion-temperature-gradient turbulence for the optimized Wendelstein 7-X stellarator*, Phys. Rev. Lett., **99**, Art. No. 035002, (2007).

HEAVY: A High Resolution Numerical Experiment in Lagrangian Turbulence

Alessandra S. Lanotte[1] **and Federico Toschi**[2]

[1] Istituto per le Scienze dell'Atmosfera e del Clima, CNR, Str. Prov. Lecce-Monteroni 73100 Lecce, Italy and INFN, Sezione di Lecce, Italy
E-mail: a.lanotte@isac.cnr.it

[2] Istituto per le Applicazioni del Calcolo CNR, Viale del Policlinico 137, 00161 Roma, Italy and INFN, Sezione di Ferrara, Via G. Saragat 1, I-44100 Ferrara, Italy
E-mail: toschi@iac.cnr.it

In the context of the DEISA Extreme Computing Initiative (DECI), we performed a state-of-the-art Direct Numerical Simulation (DNS) of heavy particles in an homogeneous and isotropic stationary forced three-dimensional turbulent flow at Reynolds number $Re_\lambda \simeq 400$. We report some details about the physical problem, computational code, data organization and preliminary results, putting them in the context of the present international research on Lagrangian Turbulence.

1 Introduction

Water droplets in clouds, air bubbles in water, pollutants or dust particles in the atmosphere, coffee stirred in a milk cup, diesel fuel spray into the combustion chamber are all examples of one single phenomenon: the transport of particles by incompressible turbulent flows. Particles can be neutrally buoyant, if their density matches the one of the carrying fluid, or they can have a mismatch in density. This causes light particles to be trapped in high vortical regions while heavy particles are ejected form vortex cores and concentrate in high strain regions. Recently, significant progress has been made in the limit of very heavy, pointwise particles[1], and some work has been done comparing experiments and numerical simulations[2]. Also, an International collaboration has been established at the purpose: the International Collaboration for Turbulence Research (ICTR)[3].

However, many questions are still unanswered: from very *simple* ones, e.g. concerning the correct equations for the motion of finite-size, finite density particles moving in a turbulent flow, to very complex ones, e.g. about the spatial correlations of particles with flow structures, or the inhomogeneous spatial distribution of inertial particles. This last is one of the most intriguing feature of these suspensions, i.e. the presence of particle clusters, in specific regions of the flow. It is easy to understand the relevance of such behaviour, since it can strongly modify the probability to find particles close to each other and thus influence their chemical, biological or mechanical interactions.

Here we present the results of high-resolution DNS of incompressible turbulent flows, seeded with millions of passive point-wise particles much heavier than the carrier fluid. The complete system -flow and particles-, can be characterized in terms of two dimensionless numbers, the Reynolds number Re and the Stokes number St. The former measures the turbulent status of the flow, while the latter measures the particle inertia. Our goal is to study aspects of the particle dynamics and statistics at varying both the flow turbulence, Re, and the particle inertia, St.

2 Physical Model and Computational Details

The numerical code, which solves the Navier-Stokes equations (2.1) for an incompressible three dimensional fluid, uses standard Pseudo-Spectral Methods[4]. The code relies on the use of Fast Fourier Transforms (FFT) for an efficient evaluation of the non-linear term of equations like (2.1) in Fourier space. We used the open source FFTW 2.1.5 libraries[5].
The Navier-Stokes equations describe the temporal evolution of a $3D$ viscous fluid, of viscosity ν, subject to the external force $\boldsymbol{f}$:

$$\frac{\partial \boldsymbol{u}(\boldsymbol{x},t)}{\partial t} + \boldsymbol{u}(\boldsymbol{x},t)\cdot\boldsymbol{\nabla}\boldsymbol{u}(\boldsymbol{x},t) = -\boldsymbol{\nabla}p(\boldsymbol{x},t) + \nu\Delta\boldsymbol{u}(\boldsymbol{x},t) + \boldsymbol{f}(\boldsymbol{x},t), \tag{2.1}$$

$$\boldsymbol{\nabla}\cdot\boldsymbol{u}(\boldsymbol{x},t) = 0, \tag{2.2}$$

where the latter equation is the mass continuity for an incompressible flow. The force $\boldsymbol{f}(\boldsymbol{x},t)$, acting only at the largest scales of motion, keeps the system in a stationary state and is chosen to obtain a statistically homogeneous and isotropic flow.

The numerical code integrates the evolution for the vector potential field $\boldsymbol{b}(\boldsymbol{k},t)$, from which the velocity field can be obtained taking the curl ($\boldsymbol{u} = \boldsymbol{\nabla}\times\boldsymbol{b}$ or in Fourier space $\hat{\boldsymbol{u}} = \boldsymbol{k}\times\hat{\boldsymbol{b}}$). Thanks to this setting, the velocity field is automatically divergence free and in the evolution equation for $\boldsymbol{b}$ no pressure contribution is present. The code uses an explicit second order Adams-Bashforth (AB) scheme for the integration of the vector potential (the equation for $\hat{\boldsymbol{b}}$ can be derived from Eq.(2.1) with the viscous term exactly integrated).

The simplest form of the Newton equations for the heavy particle dissipative dynamics reads as follows:

$$\frac{d\boldsymbol{v}(\boldsymbol{X}(t),t)}{dt} = -\frac{1}{\tau_p}\left[\boldsymbol{v}(\boldsymbol{X}(t),t) - \boldsymbol{u}(\boldsymbol{X}(t),t)\right]. \tag{2.3}$$

These also are integrated with a second-order AB scheme.

The flow $\boldsymbol{u}(\boldsymbol{x},t)$ is integrated over a cubic domain, with periodic boundary conditions, discretized with a regular grid of spacing $\delta x = L/N$, where L is the size of the cubic box and N is the number of collocation points per spatial direction (see Table 1). Particle velocities $\boldsymbol{v}(\boldsymbol{X}(t),t)$ and positions $\boldsymbol{X}(t)$ are calculated by means of a trilinear interpolation.

3 Run Details and Performance

Within the DEISA project, we performed a DNS of Eqs.(2.1) at the resolution of $2048\times2048\times2048$ grid points, seeding the flow with millions of heavy particles. The flow and particle dynamics are fully parallel; a differential dumping frequency for particles is applied in order to have both highly resolved trajectories, and a sufficiently large statistical dataset (see details in the sequel). We used 512 processors, as the best compromise between having an acceptable number of processors (with respect to the global machine configuration), and memory limitations (our code requires about 500 Gbytes). Here we underline that, as the parallelization is performed in slices, we would have been anyhow limited to a number of processors less than or equal to 2048.
Regarding the monitoring of the numerical simulations, we divided the total run (about 400k cpu hours), in batch jobs ~ 5 hours long, for a total of about 155 jobs. At the end and beginning of each job a consistency check on the input/output files was performed. Indeed

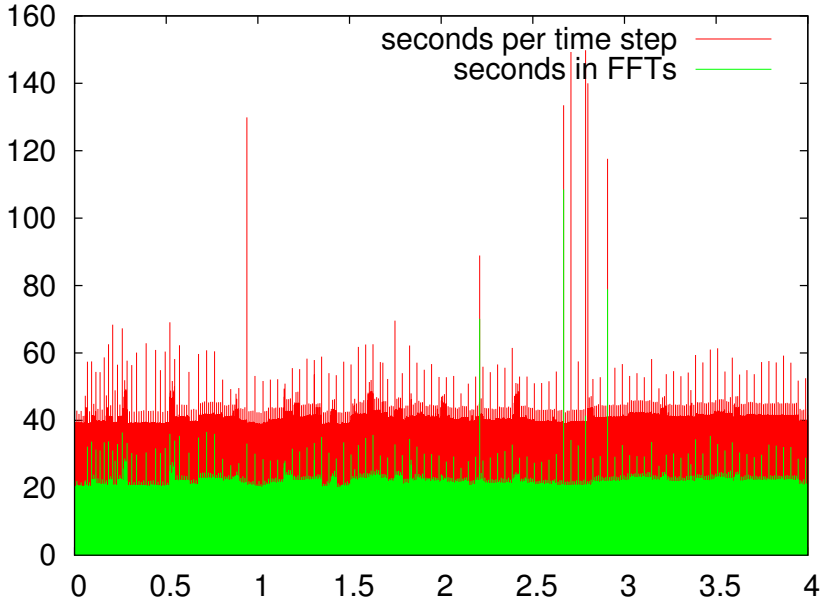


Figure 1. The computational performance of the DNS has been constantly monitored during the run (time of simulation in the horizontal axis is given in arbitrary units). The figure shows the total wall-clock time in seconds per time step, and the time spent (in seconds, per time step) for the FFTs. The relative weight of FFTs with respect to the whole loop in the time step is very close to 50%.

not only the program output was controlled for consistency, together with some overall physical indicators recorded very frequently, but also the sustained run performances were constantly monitored in order to understand the state of the machine during the runs. In Fig. 1 we show that performances were quite uniform during the run, with the presence of rare slowing down events which could take the wall-clock seconds per time step from the average 40 s to about 140 s i.e. a slow down of a factor $3 \div 4$. Obviously this phenomenon is related to hardware or networking issues.

An important observation which can be derived from Fig. 1 is the relative time spent in FFTs, with respect to the total wall-clock time step duration. As it can be seen, despite the relatively large load with $N_p = 2.1 \cdot 10^9$ particles (a load of 25%, i.e. one particle every four Eulerian grid points), almost 50% of the time is spent in Fourier transform and only the remaining 50% is spent in the temporal evolution of the Eulerian fields and of the Lagrangian particles. Probably this result can be understood as a consequence of the efficient and massively parallel way we use to integrate the particles: our conclusion is that for particle loads $\leq 25\%$ of the number of Eulerian collocation points, the computational cost for particle integration can be considered a fraction of the Eulerian cost.

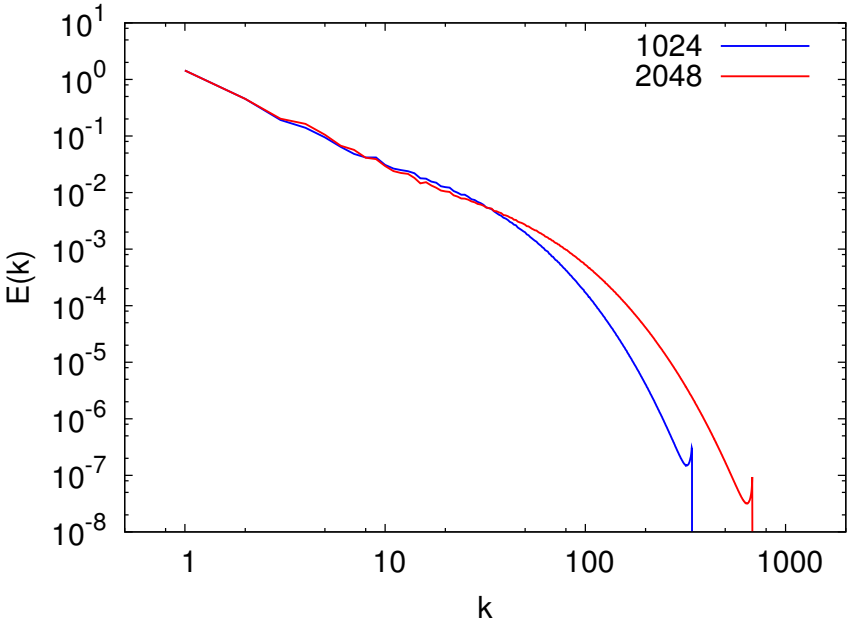


Figure 2. Log-log plot of the kinetic energy spectra $E(k)$ versus wavenumber k. The blue (lower) line is the spectrum of the starting configuration (coming from a thermalized simulation with $1024 \times 1024 \times 1024$ collocation points), while the red (upper) line is the thermalized spectrum for the current simulation with $2048 \times 2048 \times 2048$ collocation points. Other parameters, as the size of the system L and the force amplitude, are kept the same in the runs at the two resolutions.

3.1 Production Runs

Since the force injecting energy in the system acts only at the largest scales of motion, the time necessary for the turbulent flow to reach a statistically stationary state - with energy at all scales-, can be quite long. To reduce such thermalization time, the run was started from a stationary configuration obtained from a smaller resolution run with $1024 \times 1024 \times 1024$ grid points, performed during year 2004 at CINECA on 64 processors of the IBM SP4 machine[6,7]. The configuration at the lower resolution, expressed in Fourier space, was completed with zeroes at the largest wavenumbers and the run was started on the grid with $2048 \times 2048 \times 2048$ collocation points. Figure 2 shows the energy spectra, which measure the density of kinetic energy $E(k)$ per wavenumber $k = |\boldsymbol{k}|$, at the beginning and at the end of the thermalization stage. Once the Eulerian turbulent flow is stationary, we injected particles at initial random positions, with a uniform spatial distribution. We integrated the dynamics of a large number of particles with 21 different Stokes numbers. It would have been impossible to store the temporal evolution, with high temporal frequency- which means at about $\tau_\eta/10$-, for such a large number of particles. Hence we opted for a compromise. We dumped very seldom (every 4000 time steps, for a total of 13 dumps during the whole run duration), the information for all particles (see Table 1), together with the full Eulerian velocity field. This large amount of data can be very useful if one wants to study local particle concentrations -where large statistics is necessary-, or correlations

between particle properties/positions and Eulerian flow statistics.
Then, in order to investigate the temporal correlations along particle trajectories, we stored the particle information very frequently (typically every 10 time steps, i.e. roughly $\tau_\eta/10$). These data were stored only for a small subset of the total number of the particles (roughly $4 \cdot 10^6$ of the total number of particles, $N_p = 2.1 \cdot 10^9$). This mixed approach produced two distinct particle datasets, named **slow dumps** and **fast dumps**, and an Eulerian flow dataset, for a total disk occupation of 6.3 Tbytes.

At each particle dump, be it a slow or a fast one, we recorded all know information about the particle, i.e. name (a unique number useful in order to reconstruct particle trajectories), position, velocity, the fluid velocity at the particle position, the spatial gradients of the fluid at the particle position. These constitute a very rich amount of information about the statistical properties of heavy particles in turbulence. For some of the quantities we could measure, experimental Lagrangian measurements are extremely difficult because of limitations in the particles concentration, camera spatial resolution, etc.. In particular, to our knowledge, there are no measurements yet of the gradients of the fluid velocity at particle positions. So our large database will represent a very important test ground for theory and modelling.

As for particle inertia, we considered a wide range of values for the Stokes number St. Particles with very small inertia - e.g. microdroplets in clouds-, are very interesting to understand the statistics and dynamics of inertial particle in the limit of vanishing inertia, a limit which is thought to be singular. Particles with large Stokes number are also interesting since there are theoretical predictions[8], that still lack a direct assessment either by numerical or experimental measurements. Finally values around $St = 0.5$ are those for which preferential concentration is maximal[9], and this also needs further investigation. The physical parameters of our typical production runs are given in Table. 1.

Grid points	$N^3 = 2048^3$
Size of the system	$L = 2\pi$
Taylor Reynolds number	$Re_\lambda = 400$
Root-mean-square velocity	$u_{rms} = 1.4$
Kolmogorov time scale	$\tau_\eta = 0.02$
Fluid viscosity	$\nu = 4 \times 10^{-4}$
Time step	$dt = 1.1 \times 10^{-4}$
Kolmogorov scale	$\eta = 0.002$
Total number of particles	$N_p = 2.1 \times 10^9$
Stokes number	$St = 0.14, 0.54, 0.9, 1.81$
	$2.72, 4.54, 9.09, 18.18$
	$27.27, 36.36, 45.45, 63.63$

Table 1. Turbulent flow seeded with heavy particles: run parameters

4 Postprocessing and Pre-Analysis

4.1 Data Postprocessing

In order to attain the maximal performances during the numerical integration and I/O tasks, our program had the numerical integration of the particle dynamics fully parallelized among processors. This choice allowed us for a large particle load, but it implied that every processor had to dump its own particles at each time step. In order to reconstruct the trajectories of particles, from time to time, from one processor to another, we had to sort them during a postprocessing stage. We stored particles and Eulerian fields in the HDF5 dataformat[10]. For the Eulerian fields, the rationale of our choice was to store data in such a way that they could be easily analysed (for example in small cubes or in slabs) on small memory machines.

4.2 Pre-Analysis

We are currently running preliminary analysis, in particular we started by focusing on the Lagrangian Structure Functions, namely on moments of velocity increments at different time lags τ,

$$S_p(\tau) = \langle [v(t+\tau) - v(t)]^p \rangle = \langle (\delta_\tau v)^p \rangle. \tag{4.1}$$

In the previous expression, $v(t)$ is any of the three components of the particle velocity field, and the average is defined over the ensemble of particle trajectories evolving in the flow. As stationarity is assumed, moments of velocity increments only depend on the time lag τ. Morover, as our flow is statistically isotropic, different velocity components should be equivalent.
The presence of long spatial and temporal correlations, typical of any turbulent flow, suggests the existence of scaling laws in the inertial range of time $\tau_\eta \ll \tau \ll T_L$:

$$S_p(\tau) \sim \tau^{\xi(p)} \,. \tag{4.2}$$

In Fig. 3, we plot the second order Lagrangian structure function $S_2(\tau)$ for particles with three different values of inertia, $St = 0,\ 1.8,\ 63.6$. It is easy to see that while for the first two there are no detectable differences in the scaling behaviour, in the case of high inertia the shape of the curve dramatically changes. We remember that particles with very large inertia are very poorly sensible to the velocity fluctuations of the fluid velocity field, and their motion is almost ballistic. In other words, because the large value of their response time τ_p, for these particles fluid velocity fluctuations at time scales smaller than τ_p are filtered out: particle response time acts thus as a low pass filter for frequencies higher than $\sim 1/\tau_p$.

5 Concluding Remarks

In conclusion, within the DEISA Extreme Computing Initiative (DECI), we performed the most accurately resolved numerical simulation of Lagrangian turbulence worldwide. The

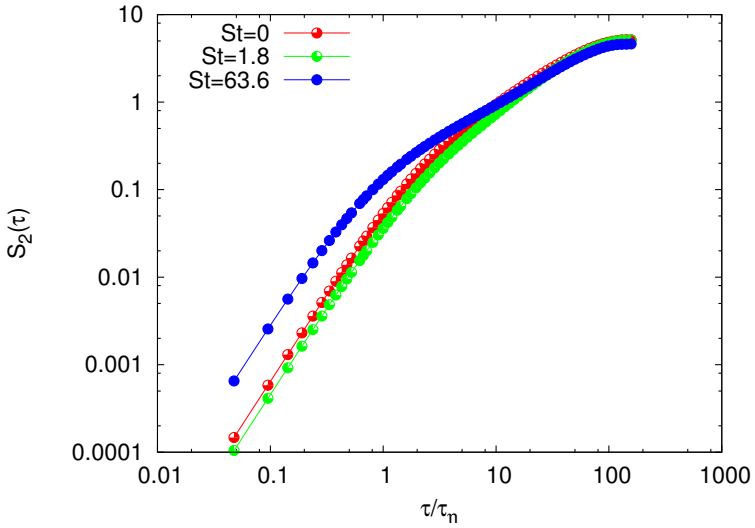


Figure 3. Log-log plot of the second order Lagrangian Structure Function $S_2(\tau)$ computed along the x component of the velocity field, versus time lags normalized with the Kolmogorov scale τ/τ_η. Here we have the curves for particles with three different values of the inertia: from below, the first curve is for tracer particles, $St = 0$; the central curve is for particles with $St = 1.8$, and the third curve is for particles with very large inertia, $St = 63.6$.

statistics of particles, and the space/time accuracy of the dynamics superseded those of all former studies.

Together with the handling of the output from this simulation, we faced the issue of establishing a standardized format for Lagrangian and Eulerian data. We relied upon HDF5 libraries and we implemented the dataformats in a way to ease as much as possible, from the coding and computational costs point of view, future data analysis.

The data from this study, after a preliminary period, will be made available to the whole scientific community at the International CFD database, iCFD database, accessible at the web site http://cfd.cineca.it and kindly hosted by CINECA[11].

Acknowledgements

We are indebted to Leibniz-Rechenzentrum for the extremely professional support, for the help provided during all the phases of the run: deployment, production and data postprocessing. In particular we would like to warmly thank Dr. Reinhold Bader.

We wish to thank Giovanni Erbacci, Chiara Marchetto and particularly Ivan Girotto from CINECA, for the continuos support and for the help during the delicate phase of data transfer and data storage. We thank Roberto Ammendola and Nazario Tantalo for the help with temporary data storage support at Tor Vergata University. We thank Luca Biferale, Massimo Cencini, Enrico Calzavarini and Jérémie Bec for constant support during the run preparation, testing and data analysis. We thank Claudio Gheller and Ugo Piomelli for many interesting discussions about data formats and, particularly, the HDF5 format. We

thank CINECA for the hosting of the datasets produced from this run and former runs at the iCFDdatabase, http://cfd.cineca.it

Finally we thank the DEISA Consortium (co-funded by the EU, FP6 project 508830) for support within the DEISA Extreme Computing Initiative (www.deisa.org).

References

1. J. Bec, L. Biferale, M. Cencini, A. Lanotte, S. Musacchio and F. Toschi, *Phys. Rev. Lett.*, **98**, 084502, (2007).
2. L. Biferale, E. Bodenschatz, M. Cencini, A. S. Lanotte, N. T. Ouellette, F. Toschi and H. Xu. Lagrangian structure functions in turbulence: A quantitative comparison between experiment and direct numerical simulation (2007). http://it.arxiv.org/abs/0708.0311
3. http://www.ictr.eu
4. D. Gottlieb, S. A. Orszag, *Numerical analysis of spectral methods: Theory and applications* , SIAM (1977).
5. http://www.fftw.org
6. CINECA Keyproject for year 2004.
7. L. Biferale, G. Boffetta, A. Celani, B. Devenish, A. Lanotte and F. Toschi, *Physical Review Letters*, **93**, 064502, (2004).
8. J. Bec, M. Cencini and R. Hillerbrand, *Physica D* **226**, 11, (2007).
9. J. Bec, L. Biferale, G. Boffetta, A. Celani, M. Cencini, A. Lanotte, S. Musacchio, and F. Toschi, *J. Fluid Mech.*, **550**, 349, (2006).
10. http://hdf.ncsa.uiuc.edu/products/hdf5
11. http://cfd.cineca.it

Atomistic Modeling of the Membrane-Embedded Synaptic Fusion Complex: a Grand Challenge Project on the DEISA HPC Infrastructure

Elmar Krieger[1], Laurent Leger[2], Marie-Pierre Durrieu[3], Nada Taib[4], Peter Bond[5], Michel Laguerre[4], Richard Lavery[6], Mark S. P. Sansom[5], and Marc Baaden[3]

[1] Center for Molecular and Biomolecular Informatics, Radboud University Nijmegen
Toernooiveld 1, 6525ED Nijmegen, The Netherlands

[2] IDRIS, Bâtiment 506, F-91403 Orsay cedex, France

[3] Laboratoire de Biochimie Théorique, Institut de Biologie Physico-Chimique, CNRS UPR 9080
13, rue Pierre et Marie Curie, F-75005 Paris, France, *E-mail: baaden@smplinux.de*

[4] UMR 5144 CNRS Mobios, Institut Européen de Chimie et Biologie, Université de Bordeaux 1
2 Av. Robert Escarpit, F-33607 Pessac, France

[5] Department of Biochemistry, University of Oxford
South Parks Road, Oxford 1 3QU, United Kingdom

[6] Institut de Biologie et Chimie des Protéines, Département des Biostructures Moléculaires
CNRS UMR 5086 / Université de Lyon, 7, passage du Vercors, F-69367 Lyon, France

The SNARE protein complex is central to membrane fusion, a ubiquitous process in biology. Modelling this system in order to better understand its guiding principles is a challenging task. This is mainly due to the complexity of the environment: two adjacent membranes and a central bundle of four helices formed by vesicular and plasma membrane proteins. Not only the size of the actual system, but also the computing time required to equilibrate it render this a demanding task requiring exceptional computing resources. Within the DEISA Extreme Computing Initiative (DECI), we have performed 40 ns of atomistic molecular dynamics simulations with an average performance of 81.5 GFlops on 96 processors using 218 000 CPU hours. Here we describe the setup of the simulation system and the computational performance characteristics.

1 Introduction

Exocytosis involves the transport of molecules stored within lipid vesicles from the inside of a cell to its environment. The final step of this process requires fusion of the vesicles with the plasma membrane and is mediated via SNARE (soluble N-ethylmaleimide sensitive factor attachment protein receptor) fusion proteins. Figure 1A shows an atomic model of the SNARE complex, a bundle of four protein helices, that supposedly brings and holds the two biological membranes together. This model was built upon previous studies and used as starting point for molecular dynamics simulations. The SNARE complex is a target for studying several pathologies such as botulism and tetanus. The purpose of our simulations is to obtain a detailed atomic picture of the structure, conformational dynamics and interactions in this system in order to improve our understanding of membrane fusion and related molecular processes like the diseases mentioned before. For further details and biological background information, see Ref.[1] and the references cited therein, as these

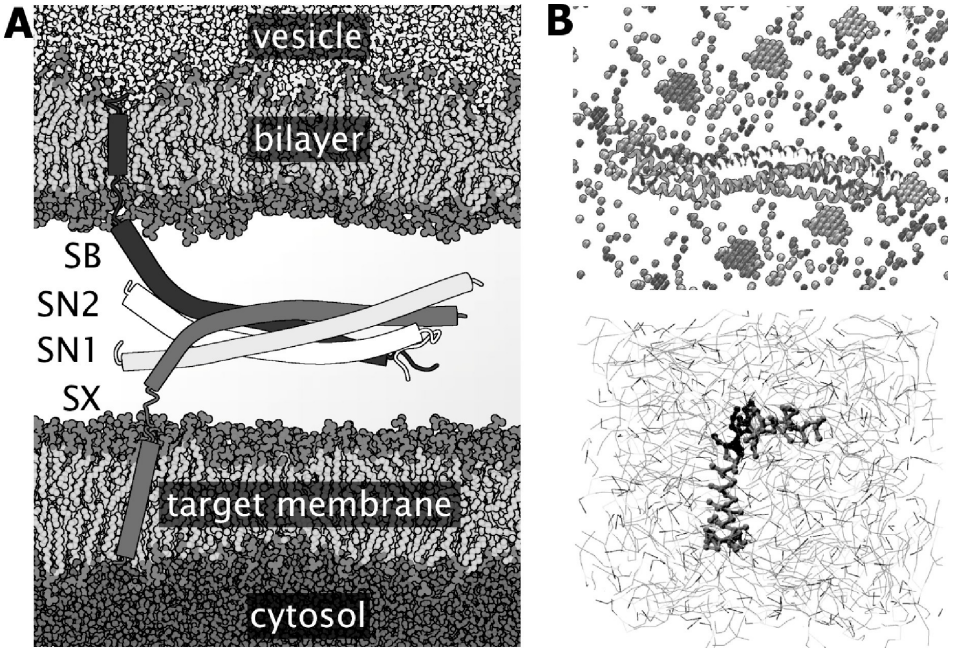


Figure 1. A: Illustration of the simulation system. The atomistic model comprises 339 792 atoms consisting of four proteins (346 amino acids), two charged POPC:POPS lipid bilayers (1008:123 lipids), 296 Na^+ ions, 166 Cl^- ions and 92 217 water molecules. The simulation production runs were carried out with the Gromacs software[3]. B: Other simulation approaches[2a,b]: all-atom molecular dynamics simulation of the cytosolic domain (top) and coarse-grained simulation of the transmembrane domain of Synaptobrevin (bottom).

aspects will not be discussed in the present paper. Here we will focus on the technical and computational aspects of the computer simulations that were carried out.

Several challenges exist from a computational point of view. First of all, no single simulation will suffice to fully understand such a complex biological system as exists around the SNARE complex. It is necessary to perform several types of simulations, each addressing a specific aspect of the whole system. Thus we are also pursuing simulations on the cytosolic soluble domain of the fusion complex and coarse-grained studies of the transmembrane domains (Fig. 1B)[2a,b]. In the present paper we will discuss a particularly ambitious simulation of a single SNARE complex embedded in a double lipid bilayer. This challenging simulation was enabled via the DEISA Extreme Computing Initiative. Biological systems - and membranes in particular - are 'soft matter' with a delicate balance of forces and interactions. The size of the model is important and long simulation times are necessary. Memory and disk space requirements further add to the complexity and can impose changes to existing software for efficient processing of the data.

A brief overview of the overall project organisation is given in Section 2. The setup of this simulation is outlined in Section 3. Computational performance and benchmark results are discussed in Section 4. We conclude with an outlook on possible improvements and extensions in Section 5.

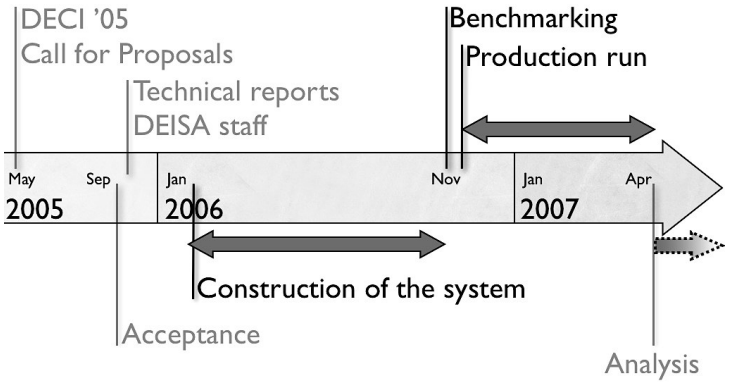


Figure 2. Timeline and project overview.

2 Timeline / Project Overview

The project started 24 months ago with a substantial grant of computing time (see Fig. 2). In collaboration with the DEISA staff, we first assessed the technical requirements of the simulation by analyzing the potential runtime characteristics. Next, we started with the construction of the atomistic model. This part, which required a little more than 12 months, will be described in more detail in the next section. Benchmarking was then achieved within 1-2 weeks, after which the production runs started for a duration of four months. Analysis of the simulation is currently under way and we expect it to become the longest part of the project. It should be pointed out that the initial lead time needed for setting up the simulation system cannot be neglected. It took longer than the actual production run and is an integral part of the challenges that one has to face when modelling complex 'soft matter' systems.

3 Simulation Setup

As mentioned in Section 2, the initial construction of the simulation system was one of the longest tasks of the project. In this section we describe the overall procedure that was adopted, then we provide details on one important sub-step.

We set up a full atomic model in explicit solvent, inserted into two fully hydrated mixed lipid bilayers consisting of 1-palmitoyl-2-oleoyl-phosphatidylcholine (POPC) and 1-palmitoyl-2-oleoyl phosphatidylserine (POPS). The construction process is depicted in Fig. 3. From left to right: 1/ We started from pre-existing all-atom simulations for the soluble cytosolic domain and the two transmembrane domains Synaptobrevin (Sb) and Syntaxin (Sx)[2a,c]. The system construction was guided by data from AFM experiments[4], aiming at an approximate 50 Å separation of the two membranes, a rather short distance compared to the size of vesicles and cells. The fragments were brought into a suitable arrangement. 2/ In order to connect the cytosolic and transmembrane fragments, it was necessary to stretch the core complex. We thus carried out adaptive biasing force runs[7] combined with subsequent interactive molecular dynamics[8]. This will be described in more

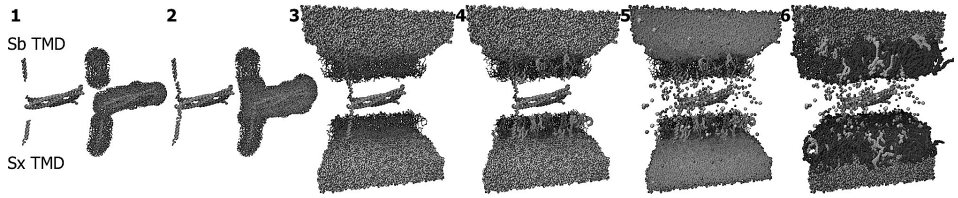


Figure 3. Stepwise construction of the starting system.

detail in the next subsection. 3/ We then created two larger hydrated POPC lipid bilayers spanning the whole simulation system. 4/ Subsequently these bilayers were mutated into mixed charged bilayers by replacing one in eight POPC lipids with POPS. 5/ Neutralizing counter ions were introduced by mutating water molecules into ions. 6/ The final structure with a mixed bilayer of 11% POPS and a 0.1 mol/l NaCl solution was equilibrated during several nanoseconds of molecular dynamics. The GROMOS-87 forcefield[5] with additional POPS parameters[6] was used. Simulations were carried out during 40 ns using full electrostatics with the Particle Mesh Ewald (PME) method. Such a timescale is short with respect to biological processes, but currently limited by the necessary computing time. Compared to other simulations of biological systems of this size, it is one of the longest simulations reported so far.

3.1 SNARE Separation via Adaptive Biasing Force Simulations

The Adaptive Biasing Force approach[7] was used to determine the potential of mean force (PMF) for the separation of the four-helical SNARE bundle and to drive the C-termini apart so that they connect to the transmembrane domains. We chose the distance between the C-termini of Sb and Sx as reaction coordinate. The PMF was refined in three successive runs with a total sampling time of three nanoseconds. Given the elongated shape of the complex, the fully solvated system requires a large solvation box and amounts to 140 000 atoms. It was simulated with the NAMD software[9] using the CHARMM forcefield[10]. The high observed pulling speed of 0.1 Å/ps in this exploratory simulation contributes to the observed loss of secondary structure (Fig. 4B) at the termini. Nevertheless the overall structure of the complex is little perturbed and only the connecting parts moved. Visual analysis shows an unzipping of the C-termini for about $1/4$ of the length of the complex with the destruction of the knobs-into-holes packing of the helices. As illustrated by the PMF, the complex shows elastic behaviour under the simulation conditions. The contact interface between Sb and Sx shown on Fig. 4C is remarkably unaffected by the C-terminal perturbation.

4 Computational Performance

4.1 Benchmarks

Benchmarks were carried out on an IBM SP Power 4 system at the national DEISA site IDRIS. A short 0.2 ps test run was performed for up to 128 processors. The latest stable and

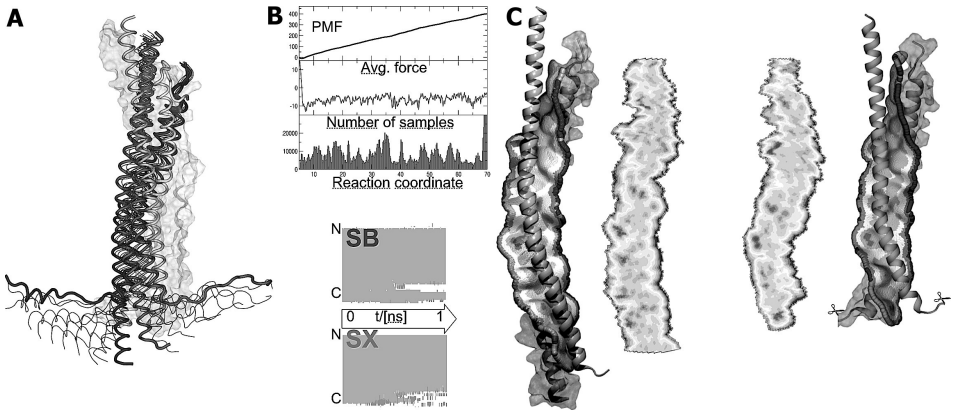


Figure 4. A: Cumulated snapshots during SNARE separation with initial and final structures highlighted. B (top): ABF estimated PMF, average biasing force and number of samples as a function of the reaction coordinate. B (bottom): secondary structure of Sb and Sx as function of time. C: Sb/Sx contact interface at the beginning and end of the ABF simulation. On the right the N-termini are omitted for clarity.

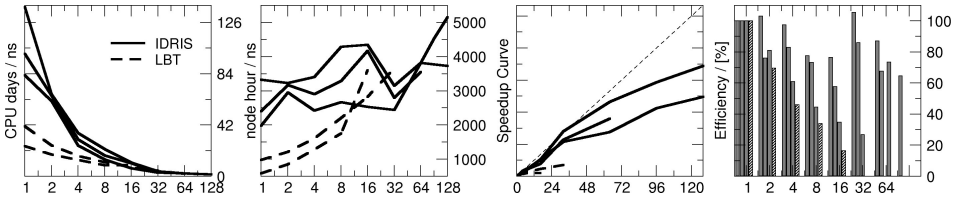


Figure 5. Benchmarks carried out at the IDRIS supercomputer centre and locally at the LBT laboratory. From left to right: CPU days per nanosecond (ns), node hours per ns, speedup curve and computational efficiency.

development versions of the Gromacs software[3] were used. Figure 5 shows that the CPU days per nanosecond ratio decreased rapidly up to 16 processors, then the gain levelled off. A comparison of the calculation cost in node hours per nanosecond showed that up to eight processors, small in-house clusters are significantly cheaper than running on a supercomputer. Above 32 processors, it is preferable to run at a dedicated computing infrastructure. The speedup curve revealed a degradation in scaling beyond 64 processors. The efficiency for a 96-processor run was a little above 60 percent. With such a setup, a 50 ns simulation would take 98.4 days and consume 227 000 CPU hours.

4.2 Production Runs

The very first production runs were carried out locally at the LBT laboratory. The main CPU time resources for this DECI project were allocated at the Rechenzentrum Garching, where subsequent runs were submitted. There we achieved a stable simulation throughput as shown in Fig. 6 over the whole four months period. An improved version of the Gromacs software[3] was used, which allowed acceptable scaling up to 96 processors. The simulated timescale was 40 ns corresponding to 20 million iterations. This equals a real timescale

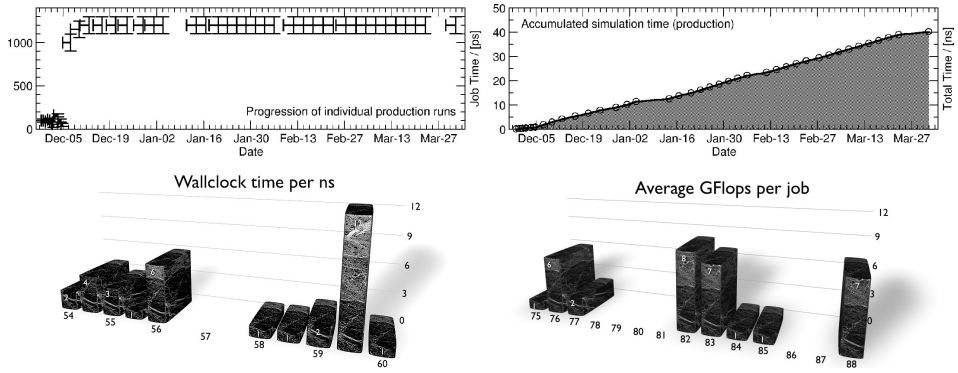


Figure 6. Top: History of job execution (left) and status of advancement (right) are shown in these graphs. The length of the line on the left plot corresponds to the execution time of a given job. The error bar on the ordinate is proportional to the number of processors used. Bottom: observed wall clock times for the production runs (left) and floating point performance (right).

of 99 days and a total of 218 000 CPU hours. The overall performance was 30% slower than on the machine used for benchmarking. The average rate was 2.4 days (57 h) per ns, generating 75 GigaByte of compressed trajectory data.

Although we had access to a dedicated part of the supercomputer, the performance in a production environment was not completely stable, but varied by about 10%. A similar spread was observed for the floating point performance, with an average of 81.5 GigaFlops (min.: 74.9%; max.: 87.6%). Although we did not carry out detailed analysis to identify the source of the variability, it seems likely that network related issues are at its origin. The jobs were run on three P690 32-CPU nodes, with MPI communications between the three nodes sharing the same network as other jobs running on other nodes. I/O access through the Global File System also used the same network and influenced job performance. Both factors can lead to variations in performance depending on all other jobs running on the whole infrastructure at the same time.

5 Outlook

5.1 Calibrating Coarse-Grained Simulations

Atomistic simulations may serve as reference for calibrating lighter, faster and more approximate coarse-grained simulations[15]. The computational gain is huge! The current 340 000 atom simulation of 40 ns duration remains a *tour de force* given the size of the system and the long simulation time. It consumed five months of computing time on 96 processors, whereas a corresponding coarse-grained calculation with 37 000 particles would run for one week on one processor and produce about 200 ns of trajectory.

5.2 Anchoring the SNARE Complex Deeper in the Membrane

A molecular dynamics simulation of a double membrane with periodic boundaries is difficult, because water molecules cannot travel between the two resulting compartments. The

distance between the two membranes is thus more or less fixed and needs to be chosen carefully. We started the simulation presented in this paper with a membrane distance of 100 Å (centre to centre) based on an estimation using Ref.[4]. After 40 nanoseconds, we found that one of the anchor helices had been partly pulled out of the membrane, indicating that the initial distance might have been slightly too large. We used YASARA[11] to move the membranes closer together in small steps of 0.5 Å, each one followed by a short steepest descent energy minimization and 200 steps of simulated annealing with the Yamber force field[12]. Force field parameters for the phospholipid molecules were derived automatically using AutoSMILES[13] in the framework of the GAFF force field[14]. The hydrogen bonds within alpha-helices were constrained to keep the SNARE protein fully intact. At a membrane distance of 86 Å, the procedure was stopped because the critical linker residues Trp 89 and Trp 90 had been buried again. This choice proved sensible, since so far the anchors stay firmly attached to the membrane (new production run; work in progress[2b]).

5.3 Analysis

The analysis of our simulations will focus on several key aspects such as the structural integrity of the four-helical SNARE bundle under tension, the membrane insertion of Sb and Sx, and the perturbation of the membranes by the fusion complex.

6 Concluding Remarks

We have described the setup, benchmarking and production calculations for a model of the synaptic fusion complex inserted into two adjacent fully hydrated lipid bilayers. It was observed that the initial setup and system construction represents an unpredictably long and delicate step. Benchmarking and production characteristics indicated machine-dependent differences and important fluctuations. The most time-consuming part in such a project is often the post-production and analysis phase. DEISA as a vast HPC infrastructure provided a stable environment for production runs and proved valuable for enabling large, challenging simulations.

Acknowledgements

We are indebted to the DEISA staff for support throughout the planning and production phase of this project. We thank the French supercomputer centres IDRIS and CINES for computing time (Projects No 041714, 051714 and LBT2411). The most CPU-intensive computations were performed with a grant of computer time provided by the DEISA Extreme Computing Initiative 2006 and carried out at the Rechenzentrum Garching. MPD acknowledges support from the French Ministry of Research.

References

1. R. Jahn, T. Lang and TC Südhof, *Membrane fusion*, Cell, **112**, 519–533, (2003); Y. A. Chen and R. H. Scheller, *Snare-mediated membrane fusion*, Nat. Rev. Mol. Cell Biol., **2**, 98–106, (2001).

2. Unpublished results. (a) MP Durrieu, R Lavery, and M Baaden. (b) MP Durrieu, P Bond, R Lavery, MSP Sansom, and M Baaden. (c) N Taib, and M Laguerre.
3. http://www.gromacs.org
4. A. Yersin, H. Hirling, P. Steiner, S. Magnin, R. Regazzi, B. Hüni, P. Huguenot, P. De los Rios, G. Dietler, S. Catsicas and S. Kasas, *Interactions between synaptic vesicle fusion proteins explored by atomic force microscopy*, Proc. Natl. Acad. Sci. U.S.A., **100**, 8736–8741, (2003).
5. W. F. van Gunsteren and H. J. C. Berendsen, *Gromos-87 Manual*, (Biomos BV, Groningen, 1987).
6. P. Mukhopadhyay, L. Monticelli and D. P. Tieleman, *Molecular dynamics simulation of a palmitoyl-oleoyl phosphatidylserine bilayer with Na+ counterions and NaCl*, Biophys. J., **86**, 1601–1609, (2004).
7. C. Chipot and J. Hénin, *Exploring the free-energy landscape of a short peptide using an average force*, Chem. Phys., **123**, 244906, (2005).
8. J. Stone, J. Gullingsrud, P. Grayson and K. Schulten, *A system for interactive molecular dynamics simulation*, in: ACM Symposium on Interactive 3D Graphics, J. F. Hughes and C. H. Squin, eds., pp. 191–194, ACM SIGGRAPH, New York, (2001).
9. J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé and K. Schulten, *Scalable molecular dynamics with NAMD*, J. Comput. Chem., **26**, 1781–1802, (2005).
10. A. D. Mackerell, D. Bashford, M. Bellott, R. L. Dunbrack, J. D. Evanseck, M. J. Field, S. Fischer, J. Gao, H. Guo, S. Ha, D. Joseph-Mccarthy, L. Kuchnir, K. Kuczera, F. T. K. Lau, C. Mattos, S. Michnick, T. Ngo, D. T. Nguyen, B. Prodhom, W. E. Reiher, B. Roux, M. Schlenkrich, J. C. Smith, R. Stote, J. Straub, M. Watanabe, J. Wiorkiewicz-Kuczera, D. Yin and M. Karplus, *All-atom empirical potential for molecular modeling and dynamics studies of proteins*, J. Phys. Chem. B, **102**, 3586–3616, (1998).
11. E. Krieger, G. Koraimann and G. Vriend, *Increasing the precision of comparative models with yasara nova–a self-parameterizing force field*, Proteins, **47**, 393–402, (2002).
12. E. Krieger, T. Darden, S. B. Nabuurs, A. Finkelstein and G. Vriend, *Making optimal use of empirical energy functions: force-field parameterization in crystal space*, Proteins, **57**, 678–683, (2004).
13. http://www.yasara.org/autosmiles
14. J. Wang, R. M. Wolf, J. W. Caldwell, P. A. Kollman and D. A. Case, *Development and testing of a general AMBER force field*, J. Comput. Chem., **25**, 1157–1174, (2004).
15. M. Baaden and R. Lavery, *There's plenty of room in the middle: multi-scale modelling of biological systems*, in: Recent Adv. In Structural Bioinformatics, Research Signpost India, A. G. de Brevern, ed., (2007).
16. http://www.baaden.ibpc.fr/projects/snaredeci/

Mini-Symposium

“Parallel Computing with FPGAs”

Parallel Computing with FPGAs - Concepts and Applications

Erik H. D'Hollander[1], **Dirk Stroobandt**[1], **and Abdellah Touhafi**[2]

[1] Ghent University
Electronics and Information Systems Department
Parallel Information Systems, B-9000 Ghent, Belgium
E-mail: {Erik.DHollander, Dirk.Stroobandt}@UGent.be

[2] Brussels University Association, Department IWT
B-1070 Brussels, Belgium
E-mail: atouhafi@info.vub.ac.be

The Mini-Symposium "Parallel computing with FPGAs" aimed at exploring the many ways in which field programmable gate arrays can be arranged into high-performance computing blocks. Examples include high-speed operations obtained by sheer parallelism, numerical algorithms mapped into hardware, co-processing time critical sections and the development of powerful programming environments for hardware software co-design.

Introduction

The idea to launch a mini-symposium on parallel computing with FPGAs, was inspired by the need to explore the huge performance potential which can be tapped from the tiny computing blocks called field programmable gate arrays. Their features are so flexible and reconfigurable that they are capable of massively parallel operations, explicitly tailored to the problem at hand. That said, there have been a lot of paradigms to put FPGAs at work in a high performance computing environment. We are all beginning to see the new and exciting possibilities of reconfigurable computing. Because a new idea is as good as its successful application, we found that it is in the tradition of the ParCo parallel computing conferences to focus on application oriented solutions. This has led to seven interesting papers, which have been presented in this symposium.

Contributions

The paper *Parallel Computing with Low-Cost FPGAs - A Framework for COPACOBANA* by Tim Güneysu, Christoph Paar, Jan Pelzl, Gerd Pfeiffer, Manfred Schimmler and Christian Schleiffer, describes a novel extensible framework of clusters of FPGAs, geared at high-performance computing. A communication library is used to configure up to 120 FPGAs simultaneously and the system is operated from a host computer. Applications in the area of cryptanalysis show the potential of the system when compared to high-cost alternatives.

The following paper *Accelerating the Cube Cut Problem with an FPGA-augmented Compute Cluster* by Tobias Schumacher, Enno Lübbers, Paul Kaufmann and Marco Platzner, employ FPGAs to speed up the bit-operations of a compute intensive exhaustive

search problem. Using parallel compute nodes each equipped with FPGAs, the authors obtain speedups of respectively 27 and 105 on 1 and 4 processors, compared to the same computations done on 1 and 4 processors without FPGAs.

A Cache Subsystem Supporting Run-time Reconfigurability by Fabian Nowak, Rainer Buchty and Wolfgang Karl, addresses performance from a cache point of view. Instead of optimizing code and data locality for a particular type of cache, a reconfigurable system is presented which optimizes the cache for a particular type of locality. During the execution the cache can be adapted to the characteristics of the different program phases. The idea is to change the associativity, the number of lines and the replacement strategy, without flushing the cache.

In *A Brain Derived Vision System Accelerated by FPGAs* by Jeff Furlong, Andrew Felch, Jayram Moorkanikara Nageswaran, Nikil Dutt, Alex Nicolau, Alex Veidenbaum, Ashok Chandrashekar and Richard Granger, a highly parallel neural model is used to overcome the limited parallelism in most programs due to sequential code. Using a winner-take-all competition between competing neurons, a massively parallel FPGA system is put in place which is able to outperform a general-purpose CPU by more than an order of magnitude.

Accelerating digital signal processing by FPGAs is studied in *Programmable Architectures for Realtime Music Decompression* by Martin Botteck, Holger Blume, Jörg von Livonius, Martin Neuenhahn and Tobias G. Noll. The paper analyzes the efficiency gained by using FPGAs for decoding MP3 streams. A pipelined implementation of the decoder gives good results, but at the same time it is shown that the power consumption of the solution is too heavy for portable devices.

High-level programming of FPGAs is a subject of *The HARWEST High Level Synthesis Flow to Design an FPGA-Based Special-Purpose Architecture to Simulate the 3D Ising Model* by Alessandro Marongiu and Paolo Palazzari. ANSI type C programs are analyzed and converted into a control and data flow graph, which are further converted into a data path and a control finite state machine. This approach is applied to the 3-D Ising spin glass model.

Web crawlers have the huge task to correlate and rank web pages. This application is dominated by a sparse matrix times vector multiplication. In the paper *Towards an FPGA Solver for the PageRank Eigenvector Problem* by Séamas McGettrick, Dermot Geraghty and Ciarán McElroy , it is shown that after pipelining and parallelizing, the computations can be mapped onto an FPGA accelerator. Reordering the data structure of the matrix allows the accelerator to outperform the PC, even when the FPGA clock is about 10 times slower.

Conclusion

FPGAs offer a number of paradigms to speed up calculations in a hardware software co-design environment. Creativity and innovation is needed to exploit all avenues and select promising and effective solutions. Trade-offs are necessary between competing goals such as portability, power consumption, performance and communication. This mini-symposium has shown that in these various areas successful ideas and implementations are obtainable. However, much work remains to be done to integrate these efforts into a framework unifying FPGAs with high-performance parallel computing.

Parallel Computing with Low-Cost FPGAs: A Framework for COPACOBANA

Tim Güneysu[1], Christof Paar[1], Jan Pelzl[3], Gerd Pfeiffer[2], Manfred Schimmler[2], and Christian Schleiffer[3]

[1] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany
E-mail: {gueneysu, cpaar}@crypto.rub.de

[2] Institute of Computer Science and Applied Mathematics, Faculty of Engineering
Christian-Albrechts-University of Kiel, Germany
E-mail: {gp, masch}@informatik.uni-kiel.de

[3] escrypt GmbH - Embedded Security, Bochum, Germany
E-mail: {jpelzl, cschleiffer}@escrypt.com

In many disciplines such as applied sciences or computer science, computationally challenging problems demand for extraordinary computing power, mostly provided by super computers or clusters of conventional desktop CPUs. During the last decades, several flavours of super computers have evolved, most of which are suitable for a specific type of problem. In general, dedicated clusters and super computers suffer from their extremely high cost per computation and are, due to the lack of cheap alternatives, currently the only possible solution to computational hard problems. More recently, emerging low-cost FPGAs tend to be a very cost-effective alternative to conventional CPUs for solving at least some of the computational hard problems such as those appearing in cryptanalysis and bio-informatics.

In cryptanalysis, breaking symmetric or asymmetric ciphers is computationally extremely demanding. Since the security parameters (in particular the key length) of almost all practical crypto algorithms are chosen such that attacks with conventional computers are computationally infeasible, the only promising way to tackle existing ciphers (assuming no mathematical breakthrough) is to build special-purpose hardware. Dedicating those machines to the task of cryptanalysis holds the promise of a dramatically improved cost-performance ratio so that breaking of commercial ciphers comes within reach.

This contribution presents the realization of a very generic framework for the COPACOBANA (Cost-Optimized Parallel Code Breaker) machine[1]. COPACOBANA consists of up to 120 low-cost FPGAs and can be realized for US$ 10,000 while being able to outperform conventional computers by several orders in magnitude, depending on the application. The presented framework allows for a simple distribution of parallelizable tasks and corresponding control mechanisms. With the framework at hand, the overhead and, as a consequence, the time to deploy custom made applications on the COPACOBANA platform is minimized. Exemplarily, we show how cryptanalytical applications can be based on top of the framework, resulting in a very low cost per computation.

1 Introduction

Although modern computers provide high performance at low cost, the cost-performance ratio of special purpose hardware can be better by several orders of magnitude, dependent on the problem. The main drawback of special purpose hardware, the high development costs and the lack of flexibility is overcome by the advent of modern reconfigurable hardware, i.e. Field Programmable Gate Arrays (FPGA). FPGAs are not able to reach the

performance of Application Specific Integrated Circuits (ASIC) but they provide the possibility of changing the hardware design easily while outpacing software implementations on general purpose processors.

A common approach to increase the performance of a single unit, i.e., a personal computer, is to build a cluster of low-cost off-the-shelf machines. One weakness of this solution is the low communication bandwidth, since most clusters are loosely coupled based on slow external peripherals introducing a high communication overhead. Another drawback of such a general-purpose architecture is its cost: solving a specific problem usually does not involve all features of the underlying hardware, i.e., parts of the hardware are running idle since they are not used at all. Another approach is to construct a parallel computer where several processors share the same system resources. This allows for very fast data throughput but requires system design constraints to remain scalable. In general, the same principle works for special purpose hardware as well.

In contrast to software development, it is a tedious task to develop hardware. This has changed with powerful software tools for design and simulation combined with the high performance of today's computers. The design of a single chip circuit becomes sufficiently affordable and reliable for small groups of engineers. The demands on design complexity are exponentially increasing when a system of several chips has to be set up since the communication between the chips gets more sophisticated due to asynchronous interfaces.

This paper presents a low-cost approach of a highly parallel FPGA cluster design with intended but not limited use in cryptography. COPACOBANA (Cost-Optimized Parallel Code Breaker) utilizes up to 120 Xilinx Spartan-3 FPGAs connected through a parallel backplane and interfaces the outside world through a dedicated controller FPGA with an Ethernet interface and a MicroBlaze soft-processor core running uClinux. The final production cost of the system was planned not to exceed material costs of US$10,000.

In the following we will give a brief overview about the hardware structure of the system, then we will provide details on the framework and programming of the system followed by currently implemented applications, applications under development and speed estimations for the system.

2 Architectural Overview

COPACOBANA consists of three basic blocks: one controller module, up to twenty FPGA modules and a backplane providing the connectivity between the controller and the FPGA modules, see Fig. 1.

The decision to pick a contemporary low-cost FPGA for the design, the Xilinx Spartan3-1000 FPGA (XC3S1000, speed grade -4, FT256 packaging) was derived by an evaluation of size and cost over several FPGA series and types. This Spartan3 FPGA comes with 1 million system gates, 17280 equivalent logic cells, 1920 Configurable Logic Blocks (CLBs) equivalent to 7680 slices, 120 Kbit Distributed RAM (DRAM), 432 Kbit Block RAM (BRAM), and 4 digital clock managers (DCMs)[2].

A step towards an extendable and simple architecture has been accomplished by the design of small pluggable FPGA modules. We decided to settle with small modules in the standard DIMM format, comprising 6 Xilinx XC3S1000 FPGAs. The FPGAs are directly connected to a common 64-bit data bus on board of the FPGA module which is interfaced to the backplane data bus via transceivers with 3-state outputs. While disconnected from

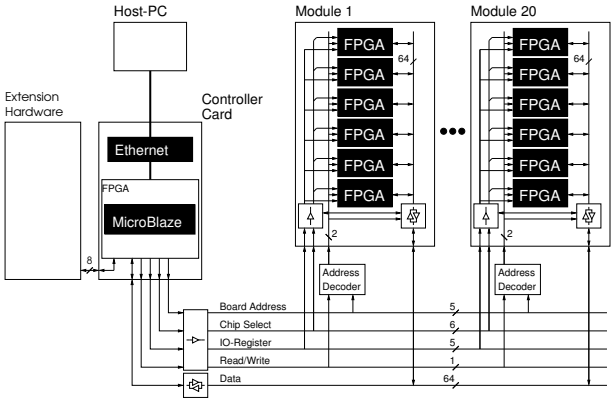


Figure 1. Architecture of COPACOBANA

the bus, the FPGAs can communicate locally via the internal 64-bit bus on the DIMM module. The DIMM format allows for a very compact component layout, which is important to closely connect the modules by a bus. From the experience with current implementations on the same FPGA type, we dispense with active cooling of the FPGAs using 80 mm fans. Depending on the heat dissipation of future applications, additional heat sinks and head spreaders might be options for an upgrade.

For the controller board we decided to use a commercially available solution to reduce the costs. In terms of flexibility we went for an FPGA board being large enough to host a CPU core and some additional logic for the bus access to move time critical and recurring bandwidth intensive tasks into hardware. We finally settled for a board with another Spartan-3S1000 FPGA as well as RAM, Flash, two serial ports and a 100 MBit/s Ethernet interface. We synthesized a Xilinx MicroBlaze system-on-chip running uClinux to which the backplane and communication logic is connected via a dedicated coprocessor interface.

The software stack includes uClinux because of its multitasking and networking abilities, i.e. access to the COPACOBANA machine is granted via a proprietary text-based protocol over TCP/IP networks. The application suite for the controller implements all necessary commands for configuring the slave FPGAs, communicating with the slave FPGAs and even for detecting how many FPGAs are actually present in the machine.

The backplane hosts all FPGA-modules and the controller card. All modules are connected by a 64-bit data bus and a 16-bit address bus. This single master bus is easy to control because no arbiter is required. Interrupt handling is totally avoided in order to keep the design as simple as possible. If the communication scheduling of an application is unknown in advance, the bus master will need to poll the FPGAs.

Moreover, the power supply is routed to every FPGA module and the controller interface. The backplane distributes two clock signals from the controller card to the slots. Every FPGA module is assigned a unique hardware address, which is accomplished by Generic Array Logic (GAL) attached to every DIMM socket. Hence, all FPGA cores can have the same configuration and all FPGA modules can have the same layout. They can easily be replaced in case of a defect. Figure 3 (Appendix) shows a complete COPACOBANA system. The backplane bus is calculated to run at a maximum frequency of

70 MHz, however currently it is working at 25 MHz due to power dissipation.

The authors are currently not aware of any similar project, realizing a FPGA cluster using low-cost components. However, there are several super computing architectures like the BEE2[3] or the commercial Computing Node Platform by Syntective Labs[4], both focusing on high computational power as well as on sophisticated node interconnections, introducing high costs per chip and computation.

3 Framework Structure

Along with the construction of the hardware, we developed a framework[5] for the ease of developing applications. It includes libraries for the software of the host computer, the hardware and software set for the controller, and a small stub residing in every slave FPGA. The framework provides a transparent communication from the host computer to the slave FPGAs.

The communication endpoint in every slave FPGA is a memory block, accessible from the FPGA implementation and the host computer as well as a status register providing several information about the current state of operation of the FPGA, i.e. a progress indicator and several status bits. Finally, the framework stub of the slave FPGAs includes arbitration and bus access mechanisms for the backplane as well as clock and reset generation for the actual computing application.

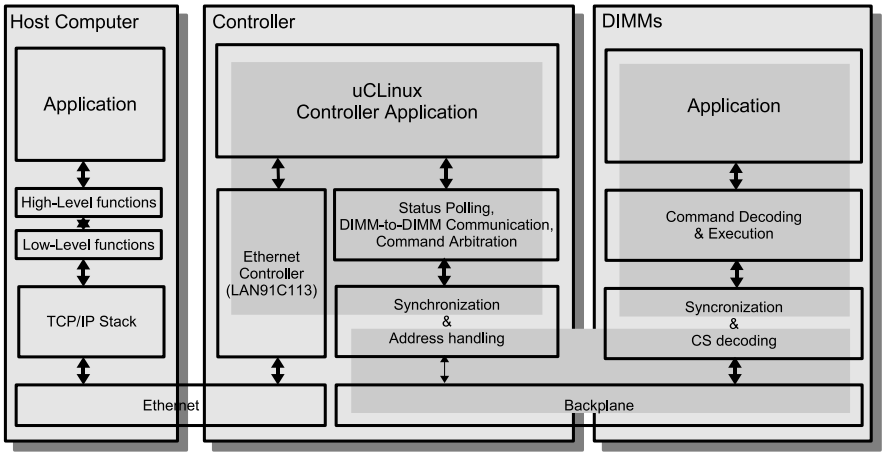


Figure 2. Framework structure; independent clock domains are marked by shadows

The controller implements the bus mastering and address generation on the lowest bus-synchronous layer. All higher layers are synchronous to the MicroBlaze CPU but do not have to be synchronous to the backplane. The hardware is able to perform read and write operations on the shared memories of the slave FPGAs by a single software command. Furthermore, it uses time slots without commands from the CPU to continuously poll the status registers of the slave FPGAs caching them in the controller for faster access. An additional benefit is the automatic detection of FPGAs that lock up and halt. Finally, the

polling system can emulate interrupts, i.e., it notifies the CPU upon detecting a service-request flag in the status register, and it is prepared to emulate a multi-master bus by moving data from one slave to another upon detection of a communication-request flag in the status register. The highest operational layer of the controller is a software component providing a textual command interpreter through a network socket.

3.1 COPACOBANA Host-PC Library

On the host computer, a single library was created to operate the COPACOBANA system. It implements a class establishing and monitoring a connection to the hardware system through a network connection to the command interpreter mentioned before. The network connection is handled transparently for the user, i.e., the library can even auto-detect COPACOBANA systems within the broadcast domain of the current network. The library is implemented in GNU-C++ and therefore is portable to a large number of operating systems and hardware platforms.

Besides the Ethernet port, the controller provides two serial ports, where one of them transmits a serial console from the uClinux operating system, which is not required in normal operation but can be useful for debugging purposes. The second serial port can be connected to an optional graphic display board, to show progress data and system information. The display board can be accessed through the COPACOBANA host libraries.

3.2 Configuration

To lower the system complexity, the same addressing and data infrastructure is used both for communication and for configuration of the FPGAs, i.e. the lowest eight bits of the data bus are connected to the slave parallel configuration ports of all slave FPGAs. The address decoding logic is used for selecting the FPGAs to be configured. With this powerful system it is possible to configure a single FPGA, a set of FPGAs or all of the 120 FPGAs at the same speed.

To allow for this approach we did not connect the feedback lines of the configuration ports to the controller, i.e., the configuration speed must stay below the maximum allowed speed of the slave FPGAs (50 MHz in this case). The configuration concept was described in detail by Pfeiffer et al.[6]

4 Applications

Though the design was intended for use in cryptanalytic applications, it is not necessarily restricted to it. In the following we will present a few applications already implemented and planned to be run on COPACOBANA as well as a brief overview about the special properties of applications to unveil the power of COPACOBANA since the compromise of low-cost hardware implies several limitations.

4.1 Cryptanalytical Applications

The first realized proof-of-concept application running on COPACOBANA was an exhaustive key search of the Data Encryption Standard (DES)[1]. The algorithm has a key size of

56 bit and a block size of 64 bit. The current implementation is capable of doing a key recovery in approximately 6.4 days at a total key test rate of 65.28 billion keys per second. Each FPGA is running four fully pipelined DES engines running at 136 MHz.
The latest presentation of a special purpose system for a key search called Deep Crack was presented by the Electronic Frontier Foundation in 1998 and consisted of 1856 ASICs that were able to do a key search in 56 hours. The overall system cost was US$250,000. Since then, no other DES cracker became publicly known.

The COPACOBANA is not limited to do only a stand-alone exhaustive key search. In case that only partial information of a plain-ciphertext for the exhaustive key search is available, the COPACOBANA can return potential key candidates to the host computer so that only a small subset of remaining keys needs to be checked in a second software based step. For example, encryption schemes relying on an effective key length of 48 bits due to export regulations are still in use. Having only 28 bits of known plaintext, a key search application running at 100 MHz can return a key candidate after 2^{28} encryptions on average. In a second processing step, these remaining 2^{20} key candidates can easily be tested in software using a conventional PC in reasonable time.

Besides the straightforward brute force attacks, another field of application of the COPACOBANA is to support cryptanalysis for public-key algorithms. The most efficient generic attack on Elliptic Curve Cryptosystems (ECC) is a parallel implementation of the Pollard-Rho algorithm capable to solve the Elliptic Curve Discrete Logarithm Problem (ECDLP) in approx. $\sqrt{\pi b/2}/\left|\mathcal{P}\right|$ steps where b is the bitlength of the underlying prime field and $\mathcal{P}$ the number of processors used. Hence, the ECDLP for small bitlengths, e.g., $b = 80$ and $b = 112$ bits standardized by the Standards for Efficient Cryptography Group (SECG)[7,8] can be solved using a set of COPACOBANAs within two weeks and several months, respectively[9].

Other applications under development are co-factorization algorithms to support the sieving step of large integer factorization based on the General Number Field Sieve (GNFS), e.g., to tackle the security of RSA-768[10]. For this purpose the Elliptic Curve Method factoring mid-sized integers can be implemented on a COPACOBANA allowing for high-performance sieving.

4.2 Further Applications

As mentioned before, COPACOBANA is not necessarily restricted to cryptanalytical applications. Basically every application with low memory demands and low to medium communication requirements can be run on the system with high performance, provided that the FPGA implementation is efficient. One candidate of an algorithm to be evaluated soon is the Smith-Waterman[11,12] algorithm for local sequence alignment, e.g. for determining similar regions between protein sequences.

5 Speed Estimations

The current version of the framework is still in early beta stage. However, with the current design we can make reasonable assumptions on throughput and latency. The layered design behaves like a pipeline, i.e. the latency for single operations is a little higher but bulk data transfers can be done at a reasonable speed. Writing a single data block of 64 bits from the

controller CPU to a slave takes approximately eight clock cycles whereas a read operation takes twelve clock cycles. Assuming that every data transfer on the bus can be done in 4 clock cycles (a conservative value since especially block-write transfers are much faster), the raw data bandwidth of the bus is approximately 6.25 GBit/s at 25 MHz.
During configuration of the slave FPGAs, a more moderate timing is used to guarantee reliability, i.e. an FPGA configuration takes about 1.5 s. Both, the controller and the 100 MBit/s Ethernet interface are a major bottleneck in the communication to the host computer, i.e. the payload data rate is limited to about 30 MBit/s.

6 Conclusion and Future Work

The work at hand presents the design and first implementation of a generic framework for the cost-optimized parallel FPGA cluster COPACOBANA. COPACOBANA can be built for less than US$ 10,000 and hosts 120 low-cost FPGAs which can be adopted to any suitable task which is parallelizable and has low requirements for communication and memory.

We implemented and described a complete framework using only freely available development tools such as uClinux, allowing for the rapid development of applications without deep knowledge and understanding of the architecture of the system. The use of the framework reduces overhead and realization time of custom made applications for COPACOBANA to a minimum, making COPACOBANA an extremely interesting alternative to expensive super computers or clusters of conventional PCs.

COPACOBANA is the first and currently only available cost-efficient design to solve cryptanalytical challenges. COPACOBANA was intended to, but is not necessarily restricted to solving problems related to cryptanalysis. Almost certainly, there will exist more interesting problems apart from cryptology which can be solved efficiently with the design at hand. In an ongoing project, we plan to apply the Smith-Waterman algorithm for scanning sequences of DNA or RNA against databases.

Currently, we are also evaluating an upgrade of the controller interface to Gigabit Ethernet and to change the processor or the FPGA type at the same time, to enhance data throughput.

References

1. S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer, and M. Schimmler, *Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker*, (2006). http://www.copacobana.org/paper/copacobana_CHES2006.pdf
2. Xilinx, Spartan-3 FPGA Family: Complete Data Sheet, DS099, (2005). http://www.xilinx.com
3. C. Chang, J. Wawrzynek and R. W. Brodersen, *BEE2: A high-end reconfigurable computing system*, IEEE Design & Test of Computers, pp. 114–125, (2005).
4. Syntective Labs, *Pure computational power – the computing node platform – CNP*, (2007). http://www.synective.com/synective_products.pdf
5. C. Schleiffer, *Hardware and software implementation of a communication layer for a parallel FPGA cluster with application in cryptography*, Master's thesis, Horst Görtz Institute, Ruhr University of Bochum, (2007).

6. S. Kumar, C. Paar, J. Pelzl, G. Pfeiffer and M. Schimmler, *A configuration concept for a massively parallel FPGA architecture*, (2006).
7. Certicom research, *Standards for efficient cryptography — SEC 1: elliptic curve cryptography*, Version 1.0, (2000).
Available at http://www.secg.org/secg_docs.htm .
8. Certicom research, *Standards for efficient cryptography — SEC 1: recommended elliptic curve domain parameters*, Version 1.0 (2000).
Available at http://www.secg.org/secg_docs.htm .
9. T. E. Güneysu, *Efficient hardware architectures for solving the discrete logarithm problem on elliptic curves*, Master's thesis, Horst Görtz Institute, Ruhr University of Bochum, (2006).
10. M. Šimka, J. Pelzl, T. Kleinjung, J. Franke, C. Priplata, C. Stahlke, M. Drutarovský, V. Fischer and C. Paar, *Hardware factorization based on elliptic curve method*, in: IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM 2005, Napa, California, USA, J. Arnold and K. L. Pocek, Eds., pp. 107–116, (2005).
11. C. W. Yu, K. H. Kwong, K. H. Lee and P. H. W. Leong, *A Smith-Waterman systolic cell*, in: Proc. 13th International Workshop on Field Programmable Logic and Applications — FPL 2003, pp. 375–384, (Springer. 2003).
12. G. Pfeiffer, H. Kreft and M. Schimmler, *Hardware enhanced biosequence alignment*, in: International Conference on METMBS, pp. 11–17, (CSREA Press, 2005).

Appendix

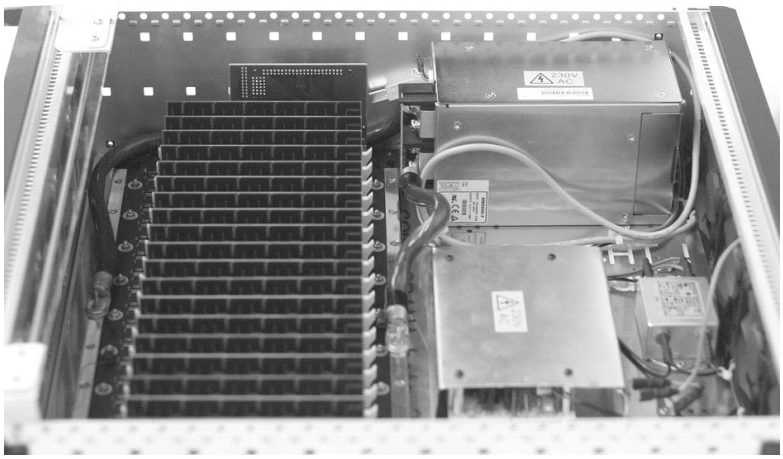


Figure 3. Complete COPACOBANA system with 20 DIMM modules

Accelerating the Cube Cut Problem with an FPGA-Augmented Compute Cluster

Tobias Schumacher, Enno Lübbers, Paul Kaufmann, and Marco Platzner

Paderborn Center for Parallel Computing (PC^2)
University of Paderborn
E-mail: {tobe, luebbers, paulk, platzner}@uni-paderborn.de

The cube cut problem refers to determining the minimum number of hyperplanes that slice all edges of the d-dimensional hypercube. While these numbers are known for $d \leq 6$, the exact numbers for $d > 6$ are as yet undetermined. The cube cut algorithm is able to compute these numbers, but is computationally extremely demanding. We accelerate the most time-consuming part of the cube cut algorithm using a modern compute cluster equipped with FPGAs. One hybrid CPU/FPGA node achieves a speedup of 27 over a CPU-only node; a 4-node hybrid cluster accelerates the problem by a factor of 105 over a single CPU. Moreover, our accelerator is developed using a structured, model-based approach for dimensioning the hardware implementation and analyzing different design points.

1 Introduction

The d-dimensional hypercube consists of 2^d nodes connected by $d \times 2^{d-1}$ edges. A prominent and still unsolved problem in Geometry is to determine $C(d)$, the minimal number of hyperplanes that slice all edges of the d-dimensional hypercube. An upper bound for $C(d)$ is given by d. This is achieved, for example, by the d hyperplanes through the origin and normal to the unit vectors. For $d \leq 4$, it has been known that at minimum these d hyperplanes are required. For $d = 5$, it was shown only a few years ago that actually five hyperplanes are required[1]. Surprisingly, several sets of only 5 hyperplanes have been found that slice all edges of the 6-dimensional hypercube.

The cube cut problem relates to the question of linear separability of vertex sets in a d-hypercube. Linear separability plays also a central role in the areas of threshold logic[2], integer linear programming[3] and perceptron learning[4]. In threshold logic, a threshold function f of d binary variables is defined as $f(x_1, x_2, \ldots, x_d) = \{1 : \text{iff } \sum_{i=1}^{d} w_i x_i \geq T; 0 : \text{iff } \sum_{i=1}^{d} w_i x_i < T\}$, with $w_i, T \in \mathbb{R}$. The input variables span an d-hypercube and f separates input variables (vertices) mapped to 0 from vertices mapped to 1 by a linear cut through the d-hypercube. Here, for example, it has been shown that $C(d)$ is the lower bound for the size of a threshold circuit for the parity function[2].

In the past, much work was done for finding $C(d)$[5,6,7]. On one hand, scientists approach this problem analytically and try to find a mathematical proof for this number. While an analytical solution would be extremely desirable, such a proof seems to be very hard to obtain. On the other hand, the problem can be approached computationally by evaluating *all possible slices* of the d-dimensional hypercube using *exhaustive search*. While the computational method lacks the mathematical strength of a formal proof, it has indeed brought remarkable success during the last years[1,8].

The complexity of computing all possible slices for higher dimensions leads to very long runtimes. The main contribution of this paper is the acceleration of the most runtime-

intense part of the cube cut algorithm by a high-performance compute cluster utilizing FPGAs. A second contribution is the use of an architecture and application model that helps us to structure the implementation of the FPGA accelerator for maximum efficiency. Section 2 gives an overview of the cube cut problem and the algorithm used to solve it. In Section 3, we introduce the basics of the FPGA accelerator modelling. The implementation decisions we have taken based on information gathered from the model are detailed step-by-step in Section 4. This section also presents performance data and compares them to a software-only implementation on the same compute cluster. Finally, Section 5 concludes the paper and presents an outlook into future work.

2 The Cube Cut Algorithm

The main idea of the computational solution to the cube cut problem is to generate all possible hyperplanes that slice some of the edges of the d-dimensional hypercube. Because the d-dimensional hypercube contains $n = d \times 2^{d-1}$ edges, such a cut can be described by a string of n bits. Each bit position in that string corresponds to a specific edge and is set to 0 when the edge is not sliced by a cut and to 1 when the edge is sliced, respectively. For example, for $d = 2$ a cut is described by a string of 4 bits where $(\{1010\}, \{0101\})$ forms a minimal set of hyperplanes cutting all edges.

The cube cut algorithm consists of three phases. First, all possible cuts for a specific d are generated. This results in a huge list of bit strings. Second, this list is reduced by a technique described below. Third, the remaining set of bit strings is searched for a subset of bit strings that represents a set of cuts slicing all edges of the d-dimensional hypercube. This is achieved by identifying a minimum set of bit strings $b_0, b_1, \ldots, b_{C(d)-1}$ where each of the n bits is set in at least one of the strings. Since the number of possible cuts grows rapidly with d, searching the full set of bit strings for a minimum number of cuts is practically impossible. Hence the reduction of the list of bit strings in phase two is of utmost importance to reduce the complexity for phase three. The reduction technique is based on the property of dominance.

Consider a cut c_1 that slices the edges $E = \{e_0, e_1, \ldots, e_{k-1}\}$ and another cut c_2 that slices all edges in E plus some additional ones. Obviously, cut c_1 can be discarded since we know that another cut exists that already slices all edges in E. We say that c_2 *dominates* c_1. Formally:

$$b \text{ dominates } a \Leftrightarrow \forall i : (\neg a_i \vee b_i) = 1; i = 0, ..., n-1 \tag{2.1}$$

From the complete set of bit strings we can typically discard a large number of cuts that are being dominated. Rather than checking all pairs of bit strings for dominance, we employ the following algorithm: First, we generate two separate lists A and B from the initial list of possible cuts. B is initialized with all bit strings that slice a maximal number of edges. Note that these bit strings can not be dominated. Assume that every element of B contains exactly k_{max} ones. A is initialized with all bit strings containing exactly $k_{max} - 1$ ones. Every element in A is then compared to the elements in B. The elements of A that are dominated by an element of B are discarded. After this step, the remaining elements of A are known to be non-dominated by any of the remaining elements of the initial list as the latter contain fewer ones and therefore slice fewer edges. The remaining elements of A are thus added to the set B. Then, a new set A is created containing all elements with

$k_{max} - 2$ ones and checked against B, and so on. The algorithm runs until all elements of the original list have been checked for dominance. The result of this algorithm is a reduced list of bit strings where all irrelevant cuts have been discarded.

Although phase two of the cube cut algorithm consists of rather simple bit operations on bit strings, it shows a very high runtime in software. This is mainly due to the fact that the required bit operations and the lengths of the bit strings do not match the instructions and operand widths of commodity CPUs. FPGAs on the other hand can be configured for directly operating on a complete bit string at once and hence exploit the high potential of fine-grained parallelism. Moreover, provided sufficient hardware area is available, many of these operations can be done in parallel with rather small overhead. An FPGA implementation can therefore also leverage both the fine- and coarse-grained parallelisms inherent in phase two of the cube cut algorithm. We have implemented that phase of the cube cut algorithm in hardware and mapped it to an FPGA (see Section 4).

3 Modelling FPGA Accelerators

Many common modelling and analysis techniques for parallel algorithms, e.g., PRAM, describe algorithms by simple operations and express the execution times by counting the number of such operations performed. While these models are of great importance from a theoretical point of view, their usefulness is often also rather limited when it comes to implementations on real parallel machines. The major problem with these models is that they consider only the time spent for performing calculations and do not regard the time spent for data accesses, i.e., memory and I/O.

To overcome this limitation, the LDA (latency-of-data-accesses) model[9] was created previously. The LDA technique actually subsumes an architecture model, an execution model, and an execution time analysis. The architecture model defines the various properties of the target architecture, especially the available memory hierarchies together with their latency and bandwidth parameters. The execution model describes algorithms in terms of LDA operations. LDA operations are classes of original machine instructions that can be characterized by common latencies. For memory accesses, the model differentiates between accesses to L1-/L2-cache, main memory, or several levels of remote memory. *Tasks* are the basic units of computation. A task consists of an *input* phase (synchronize with and receive inputs from preceding tasks), an *execution* phase (computation and memory access), and an *output* phase (synchronize with and send output to subsequent tasks). The three phases are executed in sequence as shown in Fig. 1. Control flow is modeled by connecting several tasks. The LDA model is not cycle-accurate but aims at providing a means to analyze different application mappings on parallel architectures and to estimate their performance either analytically or by simulation. Simulators have been developed atop the LDA model and used successfully to investigate applications on SMP machines[10].

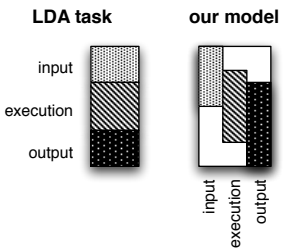


Figure 1. Modification of the LDA model

While the LDA model takes communication into account, it is not well-suited for modelling FPGA accelerators. This is for a number of reasons. First, FPGA cores can utilize the

unique capabilities of programmable hardware in many different ways. Besides exploiting huge amounts of fine-grained parallelism at the bit-level, custom cores can also employ SIMD style parallelism, pipelining and systolic execution modes. The execution phase of such a core is not at all sequential, as the LDA model assumes. Especially when the execution phase is interleaved with load and store phases, the LDA model with its sequential input–execute–output semantics is not applicable. However, one feature of the LDA model that transfers well to FPGA accelerators is its support of data accesses with different latencies. This is relevant to FPGA designs as data accesses can target anything from distributed RAM, BlockRAM and off-chip SRAM and DRAM to memories connected via I/O busses.

We therefore work on an LDA-based modelling technique for FPGA accelerators. While we retain LDA's three-phase structure, we allow for overlapping phases and consider the fine-grained parallelism of hardware. Figure 1 shows the main difference between our model and LDA. In our model, the input, execution, and output phases may occur in parallel with arbitrary time offsets. All phases are characterized by their *bandwidth*, while the architecture nodes implemented on the FPGA also possess an *area* parameter. While the bandwidth of a communication phase is dependent on architecture parameters like bus width or clock frequency, the bandwidth of an execution phase depends on a larger set of design parameters. For example, the designer can vary pipeline depth, pipeline width, and the delay of one pipeline stage, all of which have direct impact on the area requirements, the attainable clock frequency and the required I/O bandwidth.

In the work presented in this paper, we use the LDA-based model to describe and reason about different design options for an FPGA core accelerating the cube cut algorithm.

4 Implementation and Results

4.1 Target Platform

The implementation of our cube cut algorithm is targeting the Arminius compute cluster provided by the Paderborn Center for Parallel Computing[11]. This cluster consists of 200 compute nodes, each one equipped with two Xeon processors running at $3.2\ \mathrm{GHz}$ and using $4\ \mathrm{GB}$ of main memory. The nodes are connected by a $1\ \mathrm{Gbit/s}$ Ethernet and an additional $10\ \mathrm{Gbit/s}$ Infiniband network for high speed communication.

Additionally, four nodes of the cluster are equipped with an AlphaData ADM-XP FPGA board[12]. The user FPGA of this board, a Xilinx XC2VP70-5, is connected to the host system by a PCI 64/66 bridge implemented in an additional Virtex-II FPGA. This bridge provides a rather simple 64bit wide multiplexed address/databus to the user FPGA, called `localbus`. The user FPGA can be clocked from 33 to $80\ \mathrm{MHz}$, and data can be transfered using direct slave cycles or by DMA transfers. Two DMA channels are available which can operate in different modes. For our application, the most useful DMA mode is the demand mode which is optimized for connecting to a FIFO. In this mode, the DMA channels block the host application running on the CPU until the FIFOs are ready for data transfers. By experimenting with micro benchmarks we were able to determine a bidirectional bandwidth of $230\ \mathrm{MB/s}$ using demand mode DMA.

A sketch of the architecture model for our node setup is shown in Fig. 2. The algorithm kernel that is to be accelerated on the FPGA is logically located between the two FIFOs (one for input, one for output). These FIFOs in turn connect to the DMA controller and

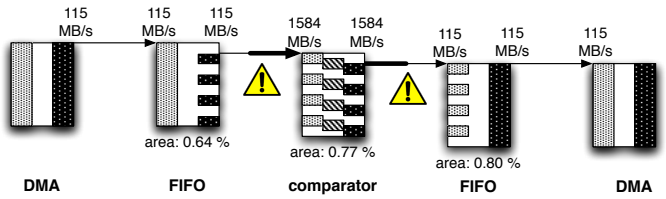


Figure 2. Architecture model

are also used to convert between bit widths and clock frequencies that might be different on the PCI bus and the user FPGA. Figure 2 shows an example where the DMA channels transmitting data over the PCI bus form a bottleneck. In our model, the DMA channels and the FIFOs are responsible for data transfers and do not have any internal execution phase.

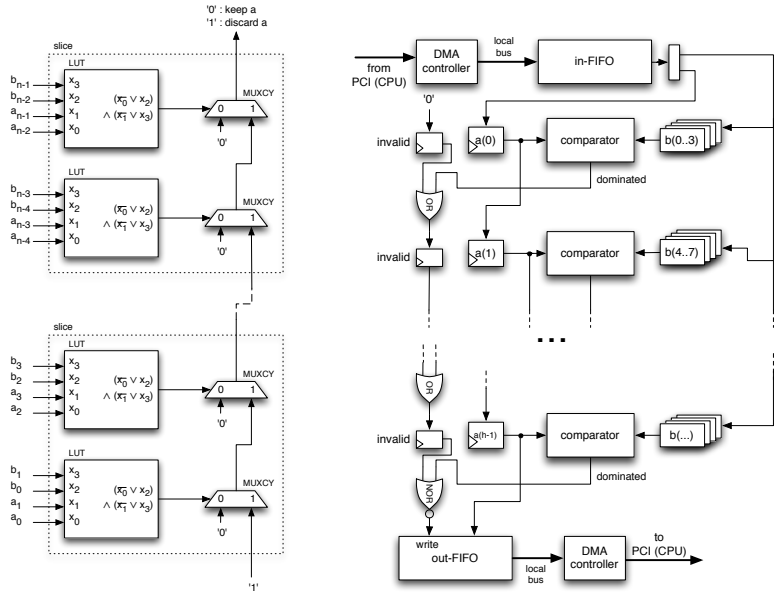
4.2 FPGA Core for Checking Dominance

The central element of the algorithm outlined in Section 2 is the comparison of two bit strings for dominance. A dominance comparator takes as inputs two n-bit strings a and b from lists A and B, respectively. It checks every bit of a against the corresponding bit of b to detect a possible dominance, according to Equation 2.1. We rely on an optimized design that makes efficient use of the LUTs and the carry logic offered by FPGAs. The design mapped to Xilinx Virtex-II Pro technology in shown in Fig. 3(a). Every 4-input LUT checks two bits of the input strings a and b for dominance, e.g., $dom = (\neg a_0 \vee b_0) \wedge (\neg a_1 \vee b_1)$. As a result, dom is one when this part of b dominates the appropriate part of a. All the LUTs required for a comparator of given bit string length n are connected to a n-input OR gate formed by the slice's carry logic, driven by a multiplexer (MUXCY). The output of this multiplexer c_{out} is tied to constant zero when $dom = 0$, and to the value of c_{in} else. The first c_{in} of this chain is tied to a constant one; the last c_{out} is the result of the complete dominance check. This approach follows the design pattern for wide logic gates[13] and modifies it to include the dominance check.

In the following, we focus on an FPGA accelerator for $d = 6$ which results in bit strings of length $n = d \times 2^{d-1} = 192$ bit. On our target device, an XC2VP70-5, a comparator for this length consumes 0.77 % of the FPGA's logic resources and can be clocked at a maximum of 66 MHz. Using the design of Fig. 3(a), one pair of bit strings can be checked for dominance in just one clock cycle. Under the assumption that an element from list B has already been loaded into the FPGA, the comparator design needs to read one element from list A per cycle. In the worst case no $a \in A$ is dominated by any of the elements of B. Then, the comparator will also output one element per cycle. The required combined input and output bandwidth of the single comparator for $n = 192$ is therefore:

$$bandwidth_{req} = 192 \text{ bit} \times 66 \text{ MHz} \times 2 = 3168 \text{ MB/s}$$

However, the available bandwidth for the input and output channels of the DMA controller is only 230 MB/s – roughly $\frac{1}{14}$ of the combined data bandwidth requested by the comparator design. Such an FPGA accelerator would be perform poorly as the DMA bandwidth over the PCI bus forms a severe performance bottleneck. This situation is reflected



(a) Efficient comparator implementation

(b) Comparator pipeline

Figure 3. Dominance-check core architecture

by the model shown in Fig. 2. Moreover, the single comparator design utilizes only 0.77 % of the available FPGA area which is rather unsatisfying.

In order to reduce the comparator's bandwidth requirements until it matches the DMA channels' bandwidth, we modified the design such that m elements of the list B are stored on the FPGA. When the element a is not dominated by any of the stored elements b, the comparator will need a new input value and write a new output value only every m-th cycle. To increase the FPGA utilization, we have two options. First, we could instantiate l comparator circuits onto the FPGA that operate in parallel. This would, however, again increase the necessary bandwidth by the factor l. The second option is to arrange p comparators in a pipelining fashion. Each stage of the pipeline stores m elements of the list B and compares them with an element a. Then, the element a proceeds to the next pipeline stage where it is compared to the next bunch of m elements of B. As soon as an element a is found to be dominated, it is invalidated. While such an element continues to run through the pipeline as a "bubble" for the reason of keeping all pipeline stages synchronized, the output FIFO will not accept this element. The resulting design for $m = 4$ is shown in Fig. 3(b).

We easily conclude that given the 0.77 % area utilization of one comparator, we could fit 130 comparators on the FPGA. Considering that we have to assign $m = 16$ elements of B to each comparator stage to reduce the bandwidth to $\frac{192\,\text{bit}\times 66\,\text{MHz}\times 2}{16} = 198\,\frac{\text{MB}}{\text{s}}$, we can still map 100 comparator stages to the target device. As a result, every element a sent to the FPGA is compared to at most 1600 elements b. At a clock rate of 66 MHz and assuming that the pipeline is completely filled with valid elements a, we achieve a performance of roughly 10^{11} bit string comparisons per second.

4.3 The Host Application

The software part of the cube cut algorithm, phase two, consists of feeding the FPGA with proper lists A and B and waiting for results to be read back. The host application consists of two threads, a sender and a receiver, that perform these tasks. The sender loads the b's into the comparators' memories and streams the a's through the pipeline, while the receiver simply waits for bit strings that were not dominated and writes the values back into a buffer. The buffers for a's and the results are then swapped, new b's are loaded, and the threads proceed as before.

We have also parallelized the host application by distributing the list A over different compute nodes such that only a fraction of the overall comparisons needs to be performed on one node. Since the Arminius cluster consists of homogenous compute nodes, we can expect every FPGA and CPU to perform a nearly equal number of comparisons per second. Consequently, a straight-forward static load balancing is sufficient to map the application onto the cluster. Portability was achieved by using MPI for the parallelized version.

4.4 Results

To verify the design presented in Section 4 and determine its performance, we conducted tests with real data generated by the cube cut algorithm. The dataset used consisted of $409'685$ bit strings containing $148-160$ ones (list B) and $5'339'385$ strings containing 147 ones (list A). Figure 4 shows the resulting runtime. Using one CPU, the software solution finishes after 2736 seconds, while the FPGA takes only 99 seconds. Parallelizing the computations shows the expected nearly linear speedup. A four-node CPU configuration takes 710 seconds, while the same configuration equipped with FPGAs finishes after 26 seconds. To summarize, one hybrid CPU/FPGA node achieves a speedup of 27 over a CPU-only node; a 4-node hybrid cluster is faster by a factor of 105 over a single CPU.

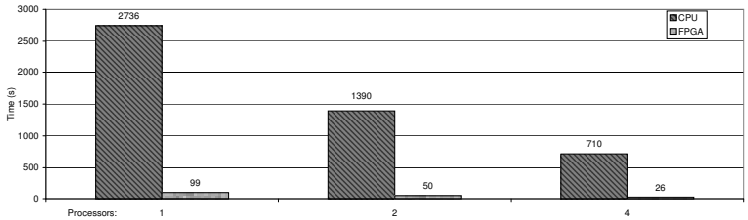


Figure 4. Computation time using CPU vs. FPGA

5 Conclusion and Future Work

In this paper we presented an FPGA accelerator that achieves impressive speedups for the most runtime-intense part of the cube cut algorithm. We supported the design process by modelling the architecture and the application. Although the considered application led to a straight-forward accelerator design, the model proved to be very useful for analyzing performance bottlenecks and comparing design alternatives at an early design phase. Further, the model will be invaluable for optimizing the accelerator to different target platforms.

We would like to know, for example, what performance we can expect by using a larger FPGA, a faster FPGA, improved DMA controllers, a faster I/O bus, etc.

We synthesized our accelerator design to one of the largest Xilinx FPGAs currently available, the Virtex5 LX330. On that device, we would be able to enlarge the comparator chain to about 400 comparators running at about 116 MHz. This would increase our speedup by a factor of eight in the best case. Basically, our accelerator is described by the parameters (n, l, p, m), the length of the bit string, the number of parallel pipelines, the numbers of stages per pipeline, and the number of comparisons per pipeline stage. Depending on the speed of the design and the bandwidth of the DMA controllers and the FIFOs, our model allows us to determine the best settings for these parameters.

We believe that creating models of the architecture and the application can greatly facilitate performance estimation and optimization. We have to investigate, however, to what extent the experience with the model for the cube cut problem can be generalized.

Future work on accelerating the cube cut algorithm will focus on the algorithm's third phase. The final composition of the remaining cuts to a minimal set of cuts slicing all edges might be another interesting candidate for FPGA acceleration. Further, we will evaluate the scalability of our design by porting it to different FPGA accelerator boards providing denser FPGAs and an improved transfer bandwidth.

References

1. C. Sohler and M. Ziegler, *Computing cut numbers*, in: Proc. 12th Annual Canadian Conf. on Comput. Geometry (CCCG 2000), pp. 73–79, (2000).
2. A. Hajnal, W. Maass, P. Pudlák, G. Turán and M. Szegedy, *Threshold circuits of bounded depth*, J. Comput. Syst. Sci., **46**, 129–154, (1993).
3. E. Balas and R. Jeroslow, *Canonical cuts on the unit hypercube*, SIAM Journal on Applied Mathematics, **23**, 61–69, (1972).
4. Novikoff, *On convergence proofs on perceptrons*, in: Proc. Symp. Math. Theory of Automata, vol. 12, pp. 615–622, Polytechnic Institute of Brooklyn. 1962.
5. M. R. Emamy-Khansary, *On the cuts and cut number of the 4-cube*, in: J. of Combinatorial Theory Series A 41, pp. 211–227, 1986.
6. P. O'Neil, *Hyperplane cuts of an n-cube*, in: Discrete Mathematics **1**, 193, (1971).
7. M. E. Saks, *Slicing the hypercube*, in: Surveys in Combinatorics, Keith Walker, (Ed.), pp. 211–255, (Cambridge University Press, 1993).
8. http://wwwcs.uni-paderborn.de/cs/ag-madh/WWW/CUBECUTS
9. J. Simon and J.-M. Wierum, *The latency-of-data-access model for analysing parallel Ccomputation*, in: Information Processing Letters - Special Issue on Models of Computation, vol. 66/5, pp. 255–261, (1998).
10. F. Schintke, J. Simon and A. Reinefeld, *A cache simulator for shared memory systems*, in: Comp. Sci. (ICCS), LNCS 2071, vol. **2074**, pp. 569–578, (Springer, 2001).
11. http://wwwcs.uni-paderborn.de/pc2/services/systems/arminius.html
12. http://www.alpha-data.com/adm-xp.html
13. R. Krueger and B. Przybus, *Virtex variable-input LUT architecture*, White paper, Xilinx, (2004).

A Run-time Reconfigurable Cache Architecture

Fabian Nowak, Rainer Buchty, and Wolfgang Karl

Institut für Technische Informatik (ITEC)
Universität Karlsruhe (TH)
76128 Karlsruhe, Germany
E-mail: {nowak, buchty, karl}@ira.uka.de

Cache parameters for CPU architectures are typically defined for a best overall match regarding the targeted field of applications. However, such may hinder high-performance execution of single applications and also does not account for cache access phases as occurring in long-running applications, e.g. from field of high-performance computing. It has been shown that matching the cache parameters to a running application results in both application speed-up and increased energy efficiency. The aim of the presented work is to create a versatile, reconfigurable cache hardware infrastructure for cache performance analysis. Such an infrastructure enables real-time monitoring of running applications and therefore is able to better trace a running application's behaviour compared to off-line analysis of a more or less reduced trace. In this paper, we will address the problems and side-effects of a run-time reconfigurable cache architecture, to which we present appropriate solutions. We will also give an outline of the upcoming hardware prototype.

1 Introduction and Motivation

In traditional hardware architectures, hardware parameters are defined with respect to given application scenarios. For general purpose architectures, this results in a compromise-based architecture serving best exactly this application set, but possibly unsuitable for any other application set.

This is especially true for the cache subsystem and may hinder the execution of high-performance computing applications which reportedly show distinct phases of certain cache access behaviour[1]. Such applications would greatly benefit from a reconfigurable cache infrastructure which can be reconfigured on-the-fly in order to match the current requirements. Using dedicated cache architectures can also improve energy efficiency[2,3,4].

The presented work aims at providing a new means for on-line cache performance analysis by creating an adjustable, versatile hardware infrastructure. In contrast to off-line analysis, such an infrastructure will enable real-time monitoring of running applications instead of off-line analysis of a more or less reduced trace.

A substantial technology for exploiting on-line cache analysis is reconfiguration of the cache infrastructure, which introduces certain problems that we will address in this paper. After identifying possible cache reconfiguration points, we will prove the feasibility of such a system and explain how to overcome the individual problems. Finally, we will outline the upcoming hardware prototype implementation of our approach providing insight into our ongoing research.

Such an architecture is well suited for cluster architectures with heterogeneous, FPGA-based nodes.

2 Related Work

Much effort has already been put into the analysis of applications' cache behaviour[5,6] and the development of a configurable cache infrastructure[2,3,7,8,9]. In their approach, applications are measured with respect to their individual behaviour and the cache infrastructure is configured with respect to a single running application. During run-time of an application, the cache configuration remains unchanged. As a result of that, no problematic data loss (cache flush) resulting from cache reconfigurations occurs because reconfiguration never takes place while an application is under execution.

The introduction of a switched L1-/L2-/L3-hierarchy that distributes on-chip memory to the different cache levels based on the CPI value of a program phase[4] suffers from the big drawback that measuring the CPI value requires changes in the internal architecture of the executing hardware, and it is even shown[10] that more sophisticated metrics are needed for phase detection.

On a further side note, benchmarking of applications in such environments is typically done off-line; with changed or new applications the benchmarking has to be re-done offline in order to determine the corresponding suitable cache configuration, which can then be used to initially configure the cache for the following application.

In contrast, our work focuses on run-time reconfiguration possibility without inducting the need for changes in the processing unit. Hence, much effort was put into the development of an efficient reconfigurable cache architecture, which allows on-the-fly reconfiguration of typical cache parameters while avoiding unnecessary cache flushes.

3 Cache Parameters and Reconfiguration

Caches[11] are defined by their overall size, associativity, line size, replacement, write-back and allocation strategy.

It can be observed that cache line sizes bigger than a few bytes are more efficient in terms of memory resource usage being able to hold more complex data structures than only basic data types. But they also are more complex in construction resulting in increased access latencies and area requirements because of their need to repeatedly initiate memory transfers until the line be completely filled. On the other hand, cache memory resource usage can also be improved by increasing the so-called associativity, which in term raises the overall cache complexity.

Replacement and allocation strategies define how the cache memory resources are (re-)used whereas the write-back strategy targets memory access "beyond" the current cache level. The replacement strategy contributes to the overall efficiency of memory use to a bigger extent than the other caching strategies. Frequently, write-allocate is used in conjunction with write-back, while write-through performs better with no-write-allocate[12].

3.1 Reconfiguration Effects

It is vital that run-time reconfiguration leave the cache infrastructure in a sane state. This, of course, can be achieved by a forced invalidation (preceded by a write-back of dirty lines), but for obvious reasons unnecessary invalidation has to be avoided. We will therefore first elaborate on the types of reconfiguration and their implications for the already cached data.

If not explained otherwise, in the following discussion increasing or decreasing means doubling or halving the respective number.

3.2 Changing Associativity with Constant Number of Lines

We first show that it is possible to increase associativity at the cost of a decreased number of sets while keeping a correct cache state. This can be achieved by means of reordering, as constructively proven below.

Let $n = 2^m$ be the cache associativity for $m \in \mathbb{N}_0$, let s be the number of sets, a the address size in bits, b the line size in bytes, and t the tag vector in bits.

For a one-way associative, i.e. direct-mapped, cache, we therefore have $a = t + \mathrm{ld}(s) + \mathrm{ld}(b)$. Using $p = \mathrm{ld}(s)$ and $q = \mathrm{ld}(b)$, this can be written as $a = t + p + q$ or $a - q = t + p$. Setting $d = a - q$ leads to $d = t + p$ —the tag length and the set address.

Going from an n-way ($m > 0$) associative cache to a $2n$-way associative, the necessary doubling of the associativity at constant line size results in the set address size being decreased by exactly one bit, as the number of sets was implicitly halved. Correspondingly, a tag size increase of one bit occurs, hence $d = t + 1 + p - 1$.

For a given line with address d' let $d' \equiv k \pmod{s}$ and w.l.o.g. $0 < k < \frac{s}{2}$ and $k \equiv 0 \pmod{2}$, and another line $d'' \equiv l \pmod{s}$ with $l \neq k$ and $l \neq k + \frac{s}{2}$, we therefore get $d' \equiv k \pmod{\frac{s}{2}}$ and $d'' \equiv l \pmod{\frac{s}{2}}$ or $d'' \equiv l - \frac{s}{2} \pmod{\frac{s}{2}}$. Consequently, different lines are stored in different sets.

For checking the second half of cache lines starting with line $\frac{s}{2}$, the restriction of $l \neq k + \frac{s}{2}$ does no longer apply. Therefore, for a third cache line $d''' \equiv k \pmod{\frac{s}{2}}$, $t_{d'} = t_{d'''}$ is possible and valid.

Accordingly, $d''' \equiv k + \frac{s}{2} \pmod{s}$ applies and we get the first bit of the set address $p_{d'}(0) = 0 \neq p_{d'''}(0) = 1$, resulting in a new tag address $t^{new}_{d'} = t_{d'} \;\&\; p_{d'}(0) \neq t^{new}_{d'''} = t_{d'''} \;\&\; p_{d'''}(0)$, i.e. the new line can be stored in the very same set ("&" indicates concatenation of the bits).

Looking at one set, in the general case of $n = 2^m, \text{ with } m \in \mathbb{N}, \forall\, i \in \{1, \ldots, n\}$, we have all addresses $d^i \equiv k \pmod{s}$, thus $t_{d^i} \neq t_{d^j} \;\forall\, i \neq j$. Likewise, addresses $\equiv l \pmod{s}$, $l \neq k \;\wedge l \neq k + \frac{s}{2}$ will not be mapped to the same set.

For addresses $e^{(1)}, \cdots, e^{(n)}$ of a different set having $e^i \equiv k + \frac{s}{2} \pmod{s}$ and $e^i \equiv k \pmod{\frac{s}{2}}$, we therefore get $t_{e^i} \neq t_{e^j}$ for $i \neq j$. So, for the first bit of the set address, $p_{d^i}(0) = 0 \neq p_{e^j}(0) = 1$ applies for any given i, j. Adding this bit to the former tag address, the new tag addresses therefore become $t^{new}_{d^i} = t_{d^i} \;\&\; p_{d^i}(0) \neq t^{new}_{e^j} = t_{e^j} \;\&\; p_{e^j}(0)$ meaning that they all can be stored safely in the new, double-sized set.

It must be noted that this reconfiguration cannot be performed sequentially without requiring an additional half-sized memory. So, the designer has the choice on whether to first back the rear half into the additional memory, then rearrange the front half and finally add the backed rear half, or to first flush and write-back the rear half and then only rearrange the front half. The second possibility clearly reveals that a trade-off between minimum chip area (used for reconfiguration logic) and data preservation has to be made.

Decreasing the associativity with a fixed number of lines is only possible if each set contains the same amount of lines with odd and even tag. Even then, the problem remains that successive reordering is not possible. Therefore, the same scheme of flushing (or backing) the rear cache half and—starting with the first line proceeding to the end—transferring

just one half of a set's lines from the front half into the rear half is proposed as trade-off.

3.3 Changing Number of Lines with Constant Number of Sets

Changing the number of lines while keeping the number of sets constant directly affects associativity: increasing the number of lines results in double set size and therefore double associativity. The cache needs to be reordered by successively moving the blocks to their new position.

Decreasing the number of lines inevitably leads to forced displacement of half a set's lines. It is sensible to apply an LRU strategy here so that the most recently used cache lines remain stored within the cache.

3.4 Changing Number of Lines with Constant Associativity

Keeping associativity while changing the number of lines results in a change of the number of sets and is equivalent to adding empty cache lines. If addition takes place in powers of two, the redistribution of cache lines is easy, i.e. odd lines (their tags end with '1', they belong to the new rear half) are moved from the original "front" lines to the newly added "rear" lines and the old line is invalidated. Because associativity remains the same and due to the fact that a potentially added set can store the same number of lines as the "old" sets, the reconfiguration is also possible with sets containing entirely odd or even tags. Of course, lines with even tags are not moved as they already are where they belong to. In any case, the tag has to be shortened due to the larger amount of sets.

When decreasing the number of lines, again in powers of two, each removed line from the rear part of the cache might have a corresponding "antagonist" in the front half, one of them both having to be displaced. Ideally, an unmodified cache line would be flushed in order to save effort. In addition, the tag is prolonged by one bit to distinguish between former front or rear position just as explained above. Again, a good trade-off is to flush the rear half and only modify the remaining front part.

3.5 Changing the Replacement Strategy

Change of replacement strategy has no direct effect on the stored data, but only affects the quality of data caching. However, it must be taken into account that switching the replacement strategy might introduce some information loss regarding the reuse statistics of cache lines.

Replacement strategies form an implicit hierarchy: on lowest level, we find random replacement, which does not keep track of previous line accesses. With pseudo-random replacement, a global counter register is introduced. More information is given by the FIFO strategy, which knows the storing order of each line per set. On top, we find pseudo-LRU and LRU strategies, providing knowledge of cache-line-individual usage history.

It is trivial to change from a more sophisticated to a simpler strategy; for this way, only minor housekeeping has to be done such as setting the counter register to a random value for pseudo-LRU, or entering the least recently used line into the FIFO register.

Going the other way suffers from a lack of information quality. The simpler strategy cannot provide replacement information as detailed as would have been possible with the newly chosen strategy.

For the first series of measurements, however, we did not reuse any previous information at all leading to only slightly worse results concerning cache line replacement and significantly decreasing the size of the reconfiguration controller.

3.6 Changing the Write Strategy

When switching off write-allocation, subsequent read or write accesses to the cache need to take care of eventually previously modified contents. In this case, the old information still has to be written back into memory.

However, information is not lost, because for any stored and modified data, the "Modified" bit is set accordingly and therefore upon later displacement of the modified cache line, the information will be written back.

When changing from write-back to write-through, it might also occur that dirty lines have not been written back to memory yet.

Again, checking the "Modified" bit is done in any case before accessing a cache line in order to care for regular write-after-read accesses, so no additional logic has to be implemented for guaranteeing data consistency after any parameter change concerning write strategy. Hence, offering the possibility to change write-access strategies does not impose any side effects.

3.7 Resource Usage and Time Consumption

The reconfiguration controller currently takes 420 Slice Flip Flops and 12,068 LUTs. It can run at a maximum frequency of 10.05 MHz. These numbers reflect the current state of the implementation, which has not yet been technology mapped. Though, we are confident that an optimized version will lead to significant speed-up.

Reconfiguration of parameters such as write-strategies only requires to set a value in a dedicated register, which consumes four cycles. When changing the replacement strategy, for each set the replacement information is updated. This requires a basic setup time of two cycles, then one cycle per set to clear the information and one final cycle to signal the successful end. Thus, for a cache with 1024 sets, 1027 cycles are needed.

Increasing the associativity also needs a basic setup time of two cycles. For each line of the "rear" half, the "Modified" bit is checked and, when modified, the line is written back. Additionally, the line is totally cleared. All in all, the processing time per line is two cycles in the best case, and twelve cycles in the worst case. After having cleared the "rear" half, one synchronization and setup cycle is inserted. Each line of the "front" part is now moved to its new location within three cycles, and two final cycles end the reconfiguration process. Thus, for 1024 lines, 2+512*2+1+512*3+2 = 2565 cycles are needed in the best case, while the worst case requires 2+512*12+1+1512*3+2 = 7685 cycles.

Few additional cycles are added by protocol overhead. See Section 4.5 for details.

4 RCA: Reconfigurable Cache Architecture

The proposed reconfigurable cache architecture (RCA) enables run-time changes of several cache parameters as outlined in this section. However, due to hardware implications, the maximum value of certain design parameters—such as e.g. overall cache size or the maximum amount of associativity—is already set before synthesis.

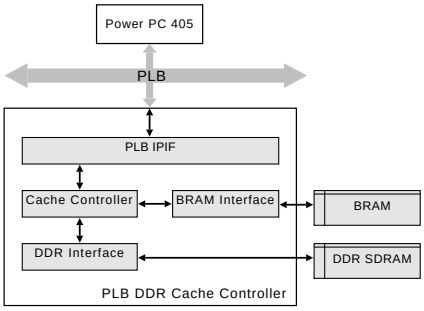


Figure 1. Basic Cache Controller Setup

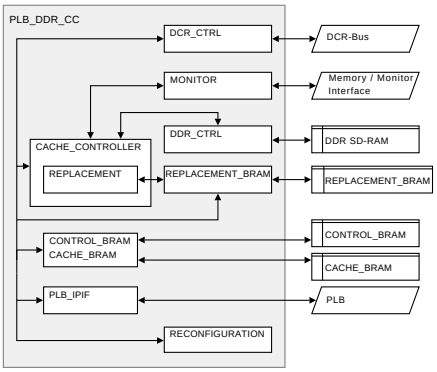


Figure 2. Reconfigurable Controller: Architecture

4.1 Cache Controller: Basic Architecture

For our Reconfigurable Cache Architecture, we developed a cache controller from scratch, merging it with the proven, working designs provided by the Xilinx FPGA development software. Our basic setup consists of a Power PC processor embedded into the FPGA, the CoreConnect[13] bus interface technology (Processor Local Bus, PLB), and a DDR SDRAM controller (PLD DDR Controller V1.11a)[14] as shown in Fig. 1. CPU memory accesses appear on the PLB and therefore are processed by the cache controller, which will then correspondingly access the cache and external SDRAM used as main memory.

Our targeted hardware platform[15] supplies only two 64M*16 DDR SDRAM controllers, i.e. with every clock cycle 2*32 bits can be written to or read from memory. Thus, we use the standard PLB data width of 64 bits resulting in a preferred cache line width of 64*n bits, where n is usually set to 1 in order to avoid additional latencies caused by memory transfers.

The FPGA's BlockRAM (BRAM) provides fast cache memory and is additionally used to store housekeeping data.

Figure 2 shows the internal architecture of the cache controller. We decided to split the BlockRAM resources into individual, physically separated chunks—holding cache data, control and replacement information—instead of using one monolithic memory area. These chunks are kept separate with respect to reconfiguration, higher throughput, and clock rates although that leads to multiple instances of BlockRAM interfaces.

Reconfiguration is achieved by means of device control registers (DCR), which are accessed via the DCR bus, part of the CoreConnect specification. A rather complex state machine controls the overall functionality of the cache controller. In addition, monitoring functionality is embedded with respect to run-time analysis of memory and cache accesses.

In the following sections, we provide a more detailed description of the individual building blocks of the Reconfigurable Cache Architecture.

4.2 State Machine

The state machine consists of four basic parts for handling read requests, write requests, replacement information, and, finally, reconfiguration. The request handling parts have sev-

eral states for initiating the request processing, handling write-back, and data transfer. The reconfiguration part, which manages cache reconfiguration and ensures cache coherency, is only entered when no cache requests are pending. More specifically it is enforced that all pending write accesses from cache to memory be finished prior to reconfiguration.

4.3 Replacement Strategy

In our design, we provide various replacement strategies ranging from the trivial random and FIFO strategies to the rather complex LRU strategies.

In the FIFO case, only a single register is required per set in order to keep track of the last written line number. Prior to writing a line, this register is increased by one modulo the cache associativity.

LRU and Pseudo LRU, in term, require use of additional BlockRAM area to store the per-set access bits, for instance 28 bits for one line of an 8-way set-associative cache when using LRU, or 10 bits per set when using pLRU. Due to this enormous chip area usage, we narrowed the configuration space to a maximum associativity of eight when using LRU or pseudo-LRU.

4.4 Monitoring

Both of two available modes track memory access characteristics by writing the collected information in a consecutive stream to the monitor output register, which in turn can be stored in a dedicated BlockRAM area by a custom monitoring information handler. Once that area would be filled, an interrupt might be risen so that the monitored data could be transferred to external memory and further processed.

The monitor output register decodes the affected cache line number within a set, information regarding transfer mode, whether the transfer was a hit or a miss, four bits indicating the byte in the cache line access, and whether the collected information can be regarded as valid. In the second mode, 30 bits of the corresponding memory address are additionally tracked.

4.5 Cache Reconfiguration

Reconfiguration is possible through a simple register interface accessible via the DCR bus. This way, the introduction of user-defined instructions and use of the APU controller is avoided. Figure 3 depicts how the processor, the controller and the reconfiguration mechanism communicate.

The interface basically consists of a single control/status register, and seven dedicated configuration registers responsible for the individual configurable values.

Configuration takes place by writing the appropriate value into the corresponding configuration register. The end of a reconfiguration process is signalled by sending a “Done” signal to the control/status register.

4.6 Simulation Results

Our architecture was evaluated running a test program. The hardware parameters for the tests are 4096 cache sets, a data width of 64 bits, and, accordingly, a line size of 8 bytes.

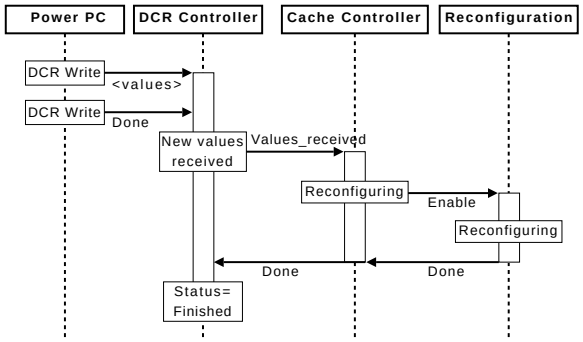


Figure 3. Components involved in reconfiguration process and their communication behaviour.

Associativity	Write-Allocate	Write-Back	Replacement	Run Time
1	√	WT	–	36,525 ns
1	×	WT	–	36,600 ns
1	√	WB	–	36,750 ns
1	×	WB	–	36,850 ns
2	√	WB	pLRU	37,000 ns
4	√	WB	pLRU	36,525 ns
4	√	WB	FIFO	36,575 ns
4	√	WB	LRU	36,825 ns
4	√	WB	pRandom	38,425 ns
8	√	WB	pLRU	38,325 ns
16	√	WB	FIFO	40,475 ns
16	√	WB	pRandom	40,250 ns

WB=Write-Back, WT=Write-Through

Table 1. Simulation Run Results

This pseudo-code was manually translated into PPC assembly language in order to ensure that no compiler code optimization or rearrangement take place. With this test program, we simulate initial writes of multiple cache lines, overwriting of cache lines with and without replacement, cache hits on stored and overwritten lines, and cache read and write misses on lines with wrong tag or wrong byte-enable bits.

The results of our simulations are depicted in Table 1, listing the run-time for our test program for various cache configurations. It must be noted that the aim of this test program was to mainly validate proper functionality and perform debugging, where required.

With our simple test routine, we were able to validate our hardware design and to successfully demonstrate changing cache hardware parameters such as cache associativity, write allocation, as well as write-back and replacement strategy. Note that the raising times with higher associativities are due to non-parallel implementation of the different ways, which makes the reconfiguration process a lot easier.

5 Toolchain Integration

Configuration of the architecture takes place within boundaries defined before the synthesis process. These boundaries are specified by a set of generics, which determine the most complex case for later run-time reconfiguration; for instance, setting the maximum replacement strategy to “pseudo-random” for synthesis will effectively disable later use of “superior” strategies like LRU because the required resources won’t be synthesized.

In order to ease configuration editing, we developed a configuration tool, which tightly integrates into the Xilinx XPS work flow. The user enters the desired maximum configuration and desired timing behaviour, based on which the project data files will be altered accordingly.

6 Conclusion and Outlook

The described cache architecture has been implemented and thoroughly simulated. We thereby proved theoretically and practically that a reconfigurable cache architecture is feasible and that it is indeed possible to achieve run-time reconfiguration while keeping the overall cache in a sane state without a mandatory need to entirely flush and reinitialize the cache memory and controller.

The design is currently synthesized into hardware to match the Xilinx ML310 and ML403 evaluation boards in order to get numbers regarding additional hardware and timing costs compared to a standard DDR SDRAM controller.

In a next step, the hardware infrastructure will be tightly integrated into our existing monitoring and visualization tool chain for data locality and cache use optimization. Furthermore, we are investigating on further possible run-time reconfiguration changes like cache line size, dynamically providing more BRAMs and exploiting them by unifying them with the already existent cache memory. Besides, we plan on creating a run-time library for use by both operating systems and applications themselves allowing them to individually fine-tune their caching system.

Once fully implemented, the architecture is supposed to speed up long-running applications by means of cache reconfiguration with the cache parameters being determined at run-time by a monitoring environment both in hardware and at operating system level. Of course, excluding the monitoring components and having the program reconfigure the cache itself, is also possible.

In a side-project, cache partitioning is explored. By partitioning the cache, serving memory-hungry applications better than short-term applications should be possible for multitasking systems.

References

1. J. Tao and W. Karl, *Optimization-oriented visualization of cache access behavior*, in: Proc. 2005 International Conference on Computational Behavior, LNCS vol. **3515**, pp. 174–181, (Springer, 2005).
2. A. Gordon-Ross, C. Zhang, F. Vahid and N. Dutt, *Tuning caches to applications for low-energy embedded systems*, in: Ultra Low-Power Electronics and Design, Enrico Macii, (Ed.), (Kluwer Academic Publishing, 2004).

3. A. Gordon-Ross and F. Vahid, *Dynamic optimization of highly configurable caches for reduced energy consumption*, Riverside ECE Faculty Candidate Colloquium, Invited Talk, (2007).
4. R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu and S. Dwarkadas, *Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures*, in: MICRO 33: Proc. 33rd annual ACM/IEEE international symposium on Microarchitecture, pp. 245–257, (ACM Press, New York, 2000).
5. A. Gordon-Ross, F. Vahid and N. Dutt, *Automatic tuning of two-level caches to embedded applications*, in: DATE '04: Proc. Conference on Design, Automation and Test in Europe, p. 10208, (IEEE Comp. Soc., Washington, DC, 2004).
6. C. Zhang and F. Vahid, *Cache configuration exploration on prototyping platforms*, in: IEEE International Workshop on Rapid System Prototyping, p. 164, (IEEE Computer Society, 2003).
7. A. Gordon-Ross, F. Vahid and N. Dutt, *Fast configurable-cache tuning with a unified second-level cache*, in: ISLPED '05: Proc. 2005 International Symposium on Low Power Electronics and Design, pp. 323–326, (ACM Press, New York, 2005). http://portal.acm.org/citation.cfm?id=1077681
8. A. Gonzàlez, C. Aliagas and M. Valero, *A data cache with multiple caching strategies tuned to different types of locality*, in: ICS '95: Proc. 9th International Conference on Supercomputing, pp. 338–347, (ACM Press, New York, 1995).
9. C. Zhang, F. Vahid and W. Najjar, *A highly configurable cache architecture for embedded systems*, in: ISCA '03: Proc. 30th Annual International Symposium on Computer Architecture, pp. 136–146, (ACM Press, New York, 2003).
10. T. Sherwood, E. Perelman, G. Hamerly, S. Sair and B. Calder, *Discovering and exploiting program phases*, IEEE Micro, **23**, 84–93, (2003).
11. A. J. Smith, *Cache memories*, ACM Computing Surveys (CSUR), **14**, pp. 473–530, (1982).
12. S. M. Müller and D. Kröning, *The impact of write-back on the cache performance*, in: Proc. IASTED International Conference on Applied Informatics, Innsbruck (AI 2000), pp. 213–217, (ACTA Press, 2000).
13. IBM, *Coreconnect architecture*, (2007). http://www-03.ibm.com/chips/products/coreconnect
14. Xilinx, Inc., *Plb double data rate (ddr) synchronous dram (sdram) controller – product specification*, (2005). http://japan.xilinx.com/bvdocs/ipcenter/data_sheet/plb_ddr.pdf
15. Xilinx, Inc., *Xilinx development boards: Virtex-4 ML403 embedded platform*, (2007). http://www.xilinx.com/xlnx/xebiz/designResources/ip_product_details.jsp?key=HW-V4-ML403-USA.

Novel Brain-Derived Algorithms Scale Linearly with Number of Processing Elements

Jeff Furlong[1], Andrew Felch[2], Jayram Moorkanikara Nageswaran[1], Nikil Dutt[1], Alex Nicolau[1], Alex Veidenbaum[1], Ashok Chandrashekar[2], and Richard Granger[2]

[1] University of California, Irvine
Irvine, CA 92697, USA
E-mail: {jfurlong, jmoorkan, dutt, nicolau, alexv}@ics.uci.edu

[2] Dartmouth College
Hanover, NH 03755, USA
E-mail: {andrew.felch, ashok.chandrashekar, richard.granger}@dartmouth.edu

Algorithms are often sought whose speed increases as processing elements are added, yet attempts at such parallelization typically result in little speedup, due to serial dependencies intrinsic to many algorithms. A novel class of algorithms have been developed that exhibit intrinsic parallelism, so that when processing elements are added to increase their speed, little or no diminishing returns are produced, enabling linear scaling under appropriate conditions, such as when flexible or custom hardware is added. The algorithms are derived from the brain circuitry of visual processing[10,17,8,9,7]. Given the brain's ability to outperform computers on a range of visual and auditory tasks, these algorithms have been studied in attempts to imitate the successes of real brain circuits. These algorithms are slow on serial architectures, but as might be expected of algorithms derived from highly parallel brain architectures, their lack of internal serial dependencies makes them highly suitable for efficient implementation across multiple processing elements. Here, we describe a specific instance of an algorithm derived from brain circuitry, and its implementation in FPGAs. We show that the use of FPGAs instead of general-purpose processing elements enables significant improvements in speed and power. A single high end Xilinx Virtex 4 FPGA using parallel resources attains more than a 62x performance improvement and 2500x performance-per-watt improvement. Multiple FPGAs in parallel achieve significantly higher performance improvements, and in particular these improvements exhibit desirable scaling properties, increasing linearly with the number of processing elements added. Since linear scaling renders these solutions applicable to arbitrarily large applications, the findings may provide a new class of novel approaches for many domains, such as embedded computing and robotics, that require compact, low-power, fast processing elements.

1 Introduction

Brain architecture and computer architecture have adapted to two very different worlds of computation and costs. While computers use connected central processors with central memory storage, the mammalian brain evolved low-precision processing units (neurons) with local memory (synapses) and very sparse connections. From a typical engineering view, brains have a demonstrably inferior computing fabric, yet they still outperform computers in a broad range of tasks including visual and auditory recognition. We propose that these brain circuit components are designed and organized into specific brain circuit architectures, that perform atypical but quite understandable algorithms, conferring unexpectedly powerful functions to the resulting composed circuits.

In this paper we first present the components of a visual brain circuit architecture (VBCA), and an overview of visual object recognition. We then show a parallel algo-

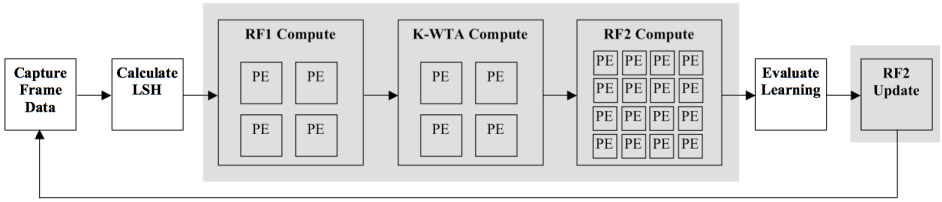


Figure 1. The components of our system are shown. Shaded blocks are targeted for parallelization on FPGAs.

rithm derived for the VBCA, and we demonstrate their application to a particular visual recognition benchmark of known difficulty (the "Wiry Object Recognition Database"[5] from CMU). We characterize the inefficiencies of mapping this intrinsically parallel algorithm onto general purpose CPUs, and then describe our implementation in FPGAs, and we analyze the resulting findings.

1.1 Background

Neurons in the eye are activated by light, and send information electrically to the thalamo-cortical system, which is the focus of our work. Thalamo-cortical circuits, constituting more than 70% of the human brain, are primarily responsible for all sensory processing as well as higher perceptual and cognitive processing.

It has long been noted that the brain operates hierarchically[20,21]: downstream regions receive input from upstream regions and in turn send feedback, forming extensive cortico-cortical loops. Proceeding downstream, neural responsiveness gains increasing complexity in types of shapes and constructs recognized, as well as becoming independent of the exact location or size of the object (translation and scale invariance)[3]. Early visual components have been shown to respond to simple constructs such as spots, lines, and corners,[13]; simulations of these capabilities have been well-studied in machine vision. Further downstream regions are selectively activated in response to assemblies of multiple features, organizing the overall system into a hierarchy[12,14,22]. Our work shows simulations of presumed intermediate stages in these cortical hierarchies, selectively responding to particular feature assemblies composed of multiple line segments. In particular we present processors of three segments ("line triples"). This highly simplified architecture is shown to be very effective on difficult visual applications such as the CMU database. It is hoped that implementations of further downstream areas will extend the work to more abstract perceptual and cognitive processing[17,9]

Key components in the implemented system are briefly described here. A given set of features activates particular regions that we refer to loosely as receptive fields (RF). Any two shapes are as similar as the number of activated neurons shared in their activation patterns. As a result of sparse population codes, most neurons are inactive; this concept is represented in a highly simplified form as sparse bit-vectors. The intrinsic random connectivity tends to select or recruit some areas to respond to some input features (RFs). Neurons train via increments to their synaptic connections, solidifying their connection-based preferences. After a simulated "developmental" phase, synapses are either present or absent and each neuron's level of activation can be represented as the bit vector.

Neurons activate local inhibitory cells which in turn de-activate their neighbors; the resulting competition among neurons is often modeled as the K best (most activated) "winners" take all or K-WTA[6,17,8,9], which our model incorporates.

These K winners activate a next set of neurons, termed RF2. As objects are viewed, these RF2 neurons are synaptically trained, becoming "recognizers" of classes of objects. RF2 activation can in turn be used in "top-down" or feedback processing, to affect the RF1 detectors based on what the RF2 cells "think" is being recognized.

The model described here uses 8192 neurons to represent the first set of input feature detectors (RF1) and 1024 neurons for RF2; other configurations exhibit comparable behaviour. Intuitively, increasing the number of neurons can be used to increase the number of classes of objects that can be recognized. The architecture modeled is depicted schematically in Fig. 1. The shaded regions are those that are targets for FPGA implementation.

1.2 Application

The Wiry Object Recognition Database (WORD)[5] from CMU was used to provide a difficult dataset dependent on shape-based recognition. The database contains a series of videos, in which a barstool is placed in several different office environments. The goal is to determine where in the videos the barstool is located. Successful identification requires a bounding box to enclose the location of the stool in the video frame, within 25% of the stool's actual area.

In top-down processing, thresholds are used to convert the activations of the second set of neurons, those for classes of objects, from data on expected shapes to data on actual shapes. This conversion builds recognition confidence, and confidence above another threshold indicates an actual guess of a recognized visual object. The guesses create bounding boxes that correspond to a particular area of the field of view.

It is important to note that the VBCA system *learns*, i.e., improves and generalizes its performance with experience. Instead of merely recognizing objects based on predetermined criteria, our system acquires information about expected and actual shape locations. The system's ability to recognize many different objects of the same type, such as variants of stools, increases as it views and learns more objects.

1.3 Related Work

Several models of the neocortex have been proposed previously. Some models contain no hierarchical structure, but are accelerated by hardware, even FPGAs[23,19]. Others do contain hierarchy, but not hardware acceleration[16,15]. Some have simulated hierarchical neocortical structures, and suggested that implementations of their models in FPGA fabrics would achieve better results, but have not actually done so and do not report object recognition results[18]. A previous implementation of the K-WTA network on a FPGA has yielded speeds similar to those of a desktop computer[11]. No similar work has shown hierarchical cortical models that operate in real-time.

2 VBCA Algorithm

The entire VBCA algorithm is currently executed on two systems. We use a general purpose CPU to capture inputs and perform high-level computations, while the predominantly

```
Initialize:
   load RF1 with 8192 entries of size 120 bits (ROM)
   load RF2 with 1024 entries of size 8192 bits
(RAM)
Capture Video Frame:
   extract 400 to 600 line segments (32 bits each)
   group 20,000 to 100,000 line triples (30 bits each)
depending on scene complexity
Calculate LSH:
   let lshVector be a locality sensitive hash (LSH)
based on frame and line data
RF1 Compute:
//Computes the dot product of RF1 neurons with a
//120 bit vector based on input frame data;
//Additionally computes a threshold so that about 512
//RF1 neurons are active
   popRAM[] = array of 8192 elements of 7 bits each
   sums[] = array of 121 elements of 13 bits each
   for i=1…8192
      for j=1…20
         popCount = popCount + Max(0, 8 -
Abs(lshVector - RF1[i]))
         lshVector = lshVector >> 6
         RF1[i] = RF1[i] >> 6
      popRAM[i] = popCount
      sums[popCount] = sums[popCount] + 1
K-WTA Compute:
//Computes a vector indicating what RF1 neurons are
//receptive
   i=120
   while totalSum < 512 || i != 0
      totalSum = totalSum + sums[i--]
   threshold = i
   threshold2 = popCount / 4
   for i=1…8192
      if popRAM[i] > threshold
         midVector[i] = 1
      else
         midVector[k] = 0
RF2 Compute:
//Computes dot products between the resultant
//midVector and all RF2 neurons;  Outputs the RF2
//index if the population count of the dot prodcut is
//over threshold2
   popCountROM[] = array of 2^8192 elements of 13
bits each
   //(ROM can be distributed to reduce size)
   for i=1…1024
      popCount = popCountROM[midVector &
RF2[i]]
      if popCount > currentMax
         currentMax = popCount
      if popCount >= threshold2
         output index i
   output threshold2
   output currentMax
Evaluate Learning:
   input results from RF2 Compute
   analyze highly receptive neurons to determine if
they should learn
   modify RF2 neurons (to be similar to frame data)
based on learning
RF2 Update:
   if applicable, send updates to RF2 as a result of
learning
```

Figure 2. Detailed pseudocode for the components of the Visual Brain Circuit Architecture (VBCA) algorithm.

bottom-up or feed-forward computations, exhibiting highly parallel operation, can be sent to FGPAs. Final post-processing is also performed on the general purpose CPU. The optimized VBCA algorithm is shown in Fig. 2, which represents in more detail all of the components of Fig. 1.

As can be seen in the visual brain circuit algorithm, some sections contain a massive amount of potential parallelism, while others sections may contain sequential code or offer very little parallelism. For example, the process of capturing frame data is very well suited for general purpose CPUs. Line segment extraction has been researched extensively and well developed algorithms already exist, so we have not attempted to perform this calculation on FPGAs. While we have implemented the *Calculate LSH* code on general purpose CPUs and FPGAs, we have found it more efficient on CPUs because the computations are small, but with notable memory requirements. With optimizations to software code, it may be possible to execute these setup tasks in real-time on a general purpose CPU.

The sections termed *RF1 Compute*, *K-WTA Compute*, and *RF2 Compute* offer high degrees of parallelism, just like the human brain, and we target these operations on FPGAs. These three sections are the components of bottom-up or feed-forward computations. To run these calculations in real-time with a reaction time of one second, we need to demon-

strate a speedup of 185x over that of optimized CPU code, if we analyze 100,000 line segment triples per frame.

After processing line triples in the *RF2 Compute* section, the results can be streamed back to the CPU, where top-down learning occurs in the section *Evaluate Learning*. For simplicity, we also call this section post-processing.

Though we have a software implementation of how we believe this post processing component works, it is likely to be optimized in the future. These optimizations, some of which may include mapping the computations to more FPGAs, may offer speedups to satisfy overall real-time constraints. However, an evaluation of these post-processing calculations is only practical when the previous computations run in real-time.

3 FPGA Implementation

Here we detail the parallel feed-forward computations and show that the their FPGA implementation is more computationally efficient than that on general purpose CPUs.

Consider the *RF1 Compute* section of the algorithm. This code could contain 8192 parallel processing elements, each with 20 parallel subcomponents. These subcomponents can be small, six bit functional units. However, a classic data convergence problem exists, because each of the 20 partial sums must be added together.

By building small FPGA logic elements that can be replicated, we have introduced just four parallel processing elements, each with 20 subcomponents. These small calculations are wasteful on a general purpose CPU because not all 64 bits of its datapath can be utilized.

The *K-WTA Compute* loop is a very simple comparison loop iterated 8192 times. Again, 8192 processing elements could be used in the optimal case. These elements can be small blocks of logic because the necessary comparison is only on seven bits of data. After each iteration, only one bit of output is generated. A fast 64 bit processor is unlikely to show great utilization because the datapath is simply too big. In our implementation, we have again built four small parallel functional units.

The most complex part of the algorithm is that of the *RF2 Compute* logic. The 8192 bit dot product is performed by ANDing a data value (midVector) with one of 1024 elements in the RF2 array. The number of binary ones in this resultant dot product must be counted (termed "population count"), and compared to a variable threshold. We have found that the most efficient method for the population count is to store population count values in a lookup table (ROM). Simply, the data to be counted is used as an address to the ROM, and the data stored at that address is the actual population count. Because it is not feasible to store all 2^{8192} elements in one ROM, we add parallelism by distributing the data.

In the optimal case, 1024 processing elements could compute all of the RF2 calculations in approximately one cycle. However, this is not realistic because the required logic and routing resources is far too great.

On a general purpose CPU, the simple dot product operation can take a significant amount of cycles, because the data must be partitioned to manageable 64 bit widths. However, on FPGAs, we can utilize 256 bit processing elements, and partition our population count ROM into blocks of eight bits each. More specifically, we utilize 32 parallel lookup ROMs and replicate them 16 times each. By doing so, we are able to compute 4096 bits of dot product per cycle, instead of just 64 bits on a CPU cycle. The use of parallelism on the loop and within each functional unit in that loop reduces the number of cycles from

millions on a general purpose CPU, to just thousands on an FPGA.

We have carefully chosen the level of parallelism to be used on each of the three sections of the algorithm. By doing so in the method presented, each section requires about the same number of cycles to finish its computation. Hence, we can pipeline each section, to produce a data streaming system, increasing the speed by about a factor of three.

Our implementation has been verified in post place and route simulation models. Because of the extremely limited number of Xilinx Virtex 4 FX140 devices at the time of this writing, we have been unable to implement our design in a physical FPGA. Our final performance enhancements can be seen in Table 1, which also compares the results with that of a general purpose CPU. We have demonstrated a 62.7x speedup on a single FPGA, while proving a speedup per $1000 factor that is significantly better than that of CPUs. In addition, the speedup per watt on FPGAs is a 2500x improvement over that of general-purpose CPUs.

Much of our circuit design is dedicated to the *RF2 Compute section*. The amount of logic and memory required is greatest for this part of the algorithm. Overall, our model requires approximately 9.6 Mbit of on-chip memory, limiting our design to only a few specific FPGAs that contain that much BRAM. Table 2 shows our FPGA utilization for the entire design, including a basic I/O interface.

Device	Cycles	Freq. (MHz)	Time (us)	Speedup	Cost ($)	Speedup / $1K	Power (W)	Speedup / W
Intel Core2 Duo	5423430	2930	1851	1.00x	2000	0.500	65.0	0.0154
XC4VFX140	2242	76.00	29.50	62.7x	10000	6.27	1.63	38.5

Table 1. The Xilinx Virtex 4 FPGA has a much longer cycle time, but ultimately requires less computational time. It also has better speedup per $1000 and speedup per watt ratios.

Number of RAMB16s	548	out of	552	99%
Number of Slices	38896	out of	63168	61%
Number of DSP48s	0	out of	192	0%
Number of PowerPCs	0	out of	2	0%

Table 2. The amount of BRAM required for our design is very large, while logic resources to support the BRAM are also considerable.

3.1 Parallel FPGAs

If we consider all optimal cases of the three VBCA components, where logic resources are unlimited, we could complete all required computations in as little as four clock cycles. Of these cycles, two are required for internal pipelining, and two are required for data computation. The required clock period for such massively parallel small processing elements would be great. However, by dividing our ideal unlimited logic resources between parallel FPGAs, we can address the intangibility problem and the long clock period problem.

Utilizing parallelism across multiple FPGAs, instead of purely within a single FPGA, is possible because we have built a data streaming design where our FPGA calculations

do not share data between each other. The only data that must be replicated among all FPGAs are the RF1 elements, the RF2 elements, and the ROM based lookup tables for RF2. Hence, we can create a client/server model, where the general purpose CPU sends lshVectors to each FPGA. which can then return results to the CPU for post processing.

Under ideal assumptions, by simply extrapolating our results with a single FPGA and ignoring I/O demands, we find that using 2048 parallel FPGAs is optimum. This scenario exploits all potential feed-forward parallelism, and requires just four pipelined clock cycles for a feed-forward calculation. The associated computational time is merely 52.63 ns, a speedup of over 35000x under perfect conditions.

However, creating such a client/server system requires a detailed analysis of the bandwidth requirements of multiple FGPAs, which can limit overall system feasibility.

The input bandwidth for the FPGA is quite small. Our pipelined calculations start with a 120 bit lshVector, which must be sent every 29.50 us to sustain the pipeline. That data corresponds to a rate of 4.07 Mbps.

The output bandwidth depends upon how many RF2 activations occur. Because the K-WTA algorithm has a K that is dependent upon the RF1 activations, the number of activations can be between 0 and 1024. In the case where 1024 activations occur, the FPGA must send 10 bits per activation, over a 29.50 us period of time. This data yields a rate of 347.1 Mbps.

Our targeted Xilinx Virtex 4 FX140 FPGA is, at the time of this writing, available on development boards with three distinct I/O interfaces. First, the PCI-X 133 interface is supported, which supports a maximum shared bandwidth of about 8.5 Gbps[1]. Because we must connect these PCI-X 133 interfaces indirectly to our CPU, we cannot utilize all of this bandwidth. Today's CPU motherboards only support about three PCI-X 133 interfaces, limiting the amount of FPGAs in our parallel system. Even with just three parallel FPGAs, our optimal speedup improves from 62.7x to 188.1x, just above our 185x speedup needed.

The second interface supported by the development boards is PCIe x1/x2/x4/x8[2]. The PCIe interfaces support rates of 2.0, 4.0, 8.0, and 16.0 Gbps for each direction, respectively for x1, x2, x4, and x8 lanes. Again, however, current CPU motherboards only support about four PCIe slots, limiting parallelism to four FPGAs. With this parallelism, our optimal speedup increases to 250.8x, well above our 185x requirement.

Finally, the third supported interface is Fibre Channel[2]. This interface supports up to about 2.0 Gbps on the FPGA side. However, these connections can be linked to a 4.0 Gbps Fibre Channel switch, which can connect to our server CPU via a 4.0 Gbps Fibre Channel interface. Hence, the number of interfaces is no longer our limiting factor, but the amount of bandwidth needed. Because our output bandwidth only requires about 347 Mbps, we can use up to 11 parallel FPGAs on 4.0 Gbps of system bandwidth. At this rate, our optimal speedup jumps to 689.7x, which would be fast enough to process additional frames of data and/or decrease the system's reaction time.

Because development boards with multiple Xilinx Virtex 4 FX140 FPGAs do not currently exist, we cannot utilize more than one FPGA per interface. However, in the future, it may be possible to link multiple FPGAs to one interface, allowing for more performance enhancements, and in the case of PCI interfaces, allow for greater bandwidth utilization.

We have considered the optimal speedups for parallel FPGAs. However, some efficiency loss should be expected, due to bus contention periods, data transfer bursts, etc. Despite this small loss, we do not expect the final speedup to be less than 185x, our re-

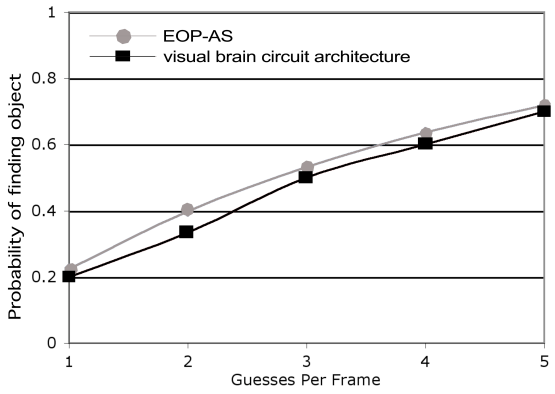


Figure 3. The results of our neocortical VBCA model compares very similarly to that of the EOP-AS model, showing accuracy between 20% and 70%. EOP-AS was estimated from the reported accuracy on "other room" tests of 22%[4].

quired speedup for real-time.

Our general purpose CPU driving the inputs to the FPGAs and collecting the results has been independently tested at a data rate of 1.1 Gbps. This rate is similar to that which would be required for three to four parallel FPGAs. The general purpose CPU may actually allow for far greater parallel FPGAs, but we have not yet tested the I/O beyond this limit.

3.2 Visual Object Recognition Accuracy

Accelerating the targeted computations is only valuable if the entire algorithm performs well in visual object recognition. At the time of this writing, the only other published results for accuracy on the Wiry Object Recognition Database, as mentioned in Section 1, is that of Ref. 4. That work uses an aggregation sensitive version of the cascade edge operator probes (EOP-AS), and differs significantly from our algorithm.

Fig. 3 compares our results with the aforementioned. As can be seen, the VBCA model achieves similar recognition accuracy to that of EOP-AS. As the systems increase guesses, from one to five, the probability of finding the sitting stool in the image increases from 20% to 70%, respectively. Both models produce very good results.

However, our model shows massive parallelism is possible; and, in our work, we have implemented part of that parallelism to produce results significantly faster. In the work of Ref. 4, they "assume access to large amounts of memory and CPU time," noting that all component computations on one 1.67 GHz Athlon CPU require "approximately one day"[4].

4 Conclusion

We have presented a visual processing algorithm (VBCA) derived from brain circuitry, shown that its accuracy is comparable to the best reported algorithms on a standard com-

puter vision benchmark task, and shown that the algorithm can improve linearly with the addition of parallel processing elements. We have found our algorithm works most efficiently by using many small computational units coupled with flexible or custom hardware, which can be implemented in FPGAs. A single Xilinx Virtex 4 FX140 FPGA provided about a 62x performance improvement over general purpose CPUs in the feed-forward computations, four FPGAs could provide enough speedup to perform computations in real-time, and 11 FPGAs could allow more frames to be processed per second for a shorter reaction time. The speedup per dollar and speedup per watt ratios are orders of magnitude better on FPGAs compared to CPUs.

4.1 Future Work

The superior performance of brain circuits on many tasks may be in part due to their unusual operation, using highly parallel algorithms to achieve unusually powerful computational results. These brain circuit methods have been especially difficult to study empirically due to their slow operation on standard processors, yet they perform excellently when implemented on appropriate parallel systems. It is hoped that the methods presented here will enable further brain circuit modelling to test larger and more complex brain systems.

The work described here has been limited by several factors. While we have met all of our very high memory requirements, additional on-chip FPGA memory (BRAM) would allow us to study the effects of increased RF2 size. Also, increased logic would allow us to add even more parallelism to our design, to better mimic the massive parallelism found in the brain, which can improve overall performance. Lastly, improved interfaces, such as additional PCIe slots or 8.0 Gbps Fibre Channel ports, could double our current speedups.

With new Xilinx Virtex 5 LX330 FPGAs, we can hope to reimplement our design and achieve further improvements. That FPGA includes more on-chip memory, faster logic resources, and overall more logic resources. Such a solution addresses several of our current limitations. Speedups beyond what we have already demonstrated could allow for even shorter reaction times or additional frames per second of analyzed data.

Many systems including embedded computations, especially robotics, would benefit from extremely compact, low-power, fast processing, which has been elusive despite efforts in these fields. Further exploration of novel intrinsically parallel algorithms and their low-power parallel implementation may confer substantial benefits to these areas of research.

Acknowledgements

The research described herein was supported in part by the Office of Naval Research and the Defense Advanced Research Projects Agency.

References

1. *ADM-XRC-4FX Datasheet*. http://www.alpha-data.com/adm-xrc-4fx.html
2. *ADPe-XRC-4 Datasheet*. http://www.alpha-data.com/adpe-xrc-4.html
3. C. Bruce, R. Desimone and C. Gross, *Visual properties of neurons in a polysensory area in the superior temporal sulcus of the macaque*, Neurophysiol, **46**, 369–384, (1981).

4. O. Carmichael, *Discriminative techniques for the recognition of complex-shaped objects*, PhD Thesis, The Robotics Institute, Carnegie Mellon University, Technical Report CMU-RI-TR-03-34, (2003).
5. O. Carmichael and M. Hebert, *WORD: Wiry Object Recognition Database*, Carnegie Mellon University, (2006). http://www.cs.cmu.edu/~owenc/word.htm
6. R. Coultrip, R. Granger and G. Lynch, *A cortical model of winner-take-all competition via lateral inhibition*, Neural Networks, **5**, 47–54, (1992).
7. A. Felch and R. Granger, *The hypergeometric connectivity hypothesis: Divergent performance of brain circuits with different synaptic connectivity distributions*, Brain Research, (2007, in press).
8. R. Granger, *Brain circuit implementation: High-precision computation from low-precision components*, in: Replacement Parts for the Brain, T. Berger, D. Glanzman, Eds., pp. 277–294, (MIT Press, 2005).
9. R. Granger, *Engines of the brain: The computational instruction set of human cognition*, AI Magazine, **27**, 15–32, (2006).
10. R. Granger, *Neural computation: Olfactory cortex as a model for telencephalic processing*, Learning & Memory, J. Byrne, Ed., pp. 445–450, (2003).
11. C. Gao, and D. Hammerstrom, *Platform performance comparison of PALM network on Pentium 4 and FPGA*, in: International Joint Conf. on Neural Networks, (2003).
12. J. Haxby, M. I. Gobbini, M. Furey, A. Ishai, J. Schouten and P. Pietrini, *Distributed and overlapping representations of faces and objects in ventral temporal cortex*, Science, **293**, 2425–2430, (2001).
13. D. Hubel, and T. Wiesel, *Receptive fields and functional architecture in two nonstriate visual areas (18 and 19) of the cat*, J. Neurophysiol, **28**, 229–289, (1965).
14. Y. Kamitani and F. Tong, *Decoding the visual and subjective contents of the human brain*, Nat.Neurosci, **8**, 679–685, (2005).
15. D. Mumford, *On the computational architecture of the neocortex*, Biological Cybernetics, **65**, 135–145, (1991).
16. R. Rao and D. Ballard, *Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects*, Nat Neurosci, **2**, 79–87, (1999).
17. A. Rodriguez, J. Whitson and R. Granger, *Derivation and analysis of basic computational operations of thalamocortical circuits.*, J. Cog. Neurosci., **16**, 856–877, (2004).
18. B. Resko, et al., *Visual Cortex Inspired Intelligent Contouring*, in: Intelligent Engineering Systems Proceedings, September 16-19, (2005).
19. J. Starzyk, Z. Zhu and T. Liu, *Self Organizing Learning Array*, IEEE Trans. on Neural Networks, **16**, 355-363, (2005).
20. J. Szentagothai, *The module concept in cerebral cortex architecture*, Brain Research, **95**, 475–496, (1975).
21. F. Valverde, *Structure of the cerebral cortex. Intrinsic organization and comparative analysis of the neocortex*, Rev. Neurol., **34**, 758–780, (2002).
22. G. Wallis and E. Rolls, *A model of invariant object recognition in the visual system*, Prog. Neurobiol, **51**, 167–194, (1997).
23. R. K. Weinstein and R. H. Lee, *Architecture for high-performance FPGA implementations of neural models*, Journal of Neural Engineering, **3**, 21–34, (2006).

Programmable Architectures for Realtime Music Decompression

Martin Botteck[1], Holger Blume[2], Jörg von Livonius[2], Martin Neuenhahn[2], and Tobias G. Noll[2]

[1] Nokia Research Center
Nokia GmbH, Meesmannstraße 103, 44807 Bochum
E-mail: martin.botteck@nokia.com

[2] Chair for Electrical Engineering and Computer Systems
RWTH Aachen University, Schinkelstrasse 2, 52062 Aachen
E-mail: {blume, livonius, neuenhahn, tgn}@eecs.rwth-aachen.de

Field Programmable Gate Array Architectures (FPGA) are known to facilitate efficient implementations of signal processing algorithms. Their computational efficiency for arithmetic datapaths in terms of power and performance is typically ranked better than that of general purpose processors. This paper analyses the efficiency gain for MP3 decoding on an FPGA. The results of an exemplary FPGA implementation of a complete MP3 decoder featuring arithmetic datapaths as well as control overhead are compared concerning performance and power consumption to further implementation alternatives. Furthermore, the results are compared to requirements for a deployment in portable environments.

1 Introduction

Music playback has become a distinguishing and indispensable feature in modern mobile communication devices. Mobile phones with several Gigabytes of built in storage capacity have been released to the market. Consequently, achievable playback times are no longer determined by the amount of available tracks or storage but by power consumption of the playback function. In a previous study[1] the power consumption of general purpose processors such as the ARM 940T processor core which is frequently used in embedded applications has been modeled. As an application example typical signal processing applications such as audio decoding algorithms like MP3 and AAC were regarded. Although such embedded processors yield attractive low power consumption the absolute values are still too high to support radical improvements for future mobile devices. Therefore, further implementation alternatives have to be studied retaining the flexibility of such platforms in order to be able to e.g. adapt to changes in coding standards etc. Generally, modern digital signal processing applications show a rapidly growing demand for flexibility, high throughput and low power dissipation. These contradicting demands however can not be addressed by a single, discrete architecture (e.g. general purpose processor). A typical approach to solve this problem combines different architecture blocks to a heterogeneous System-on-a-Chip (SoC). Figure 1 shows the design space for various architecture blocks like general purpose processors, field programmable gate arrays (FPGAs), physically optimised macros etc. in terms of power-efficiency ($mW/MOPS$) and area-efficiency ($MOPS/mm^2$) for some exemplary digital signal processing tasks[2]. While programmable processor architectures provide less power- and area-efficiency than other architecture blocks they have a

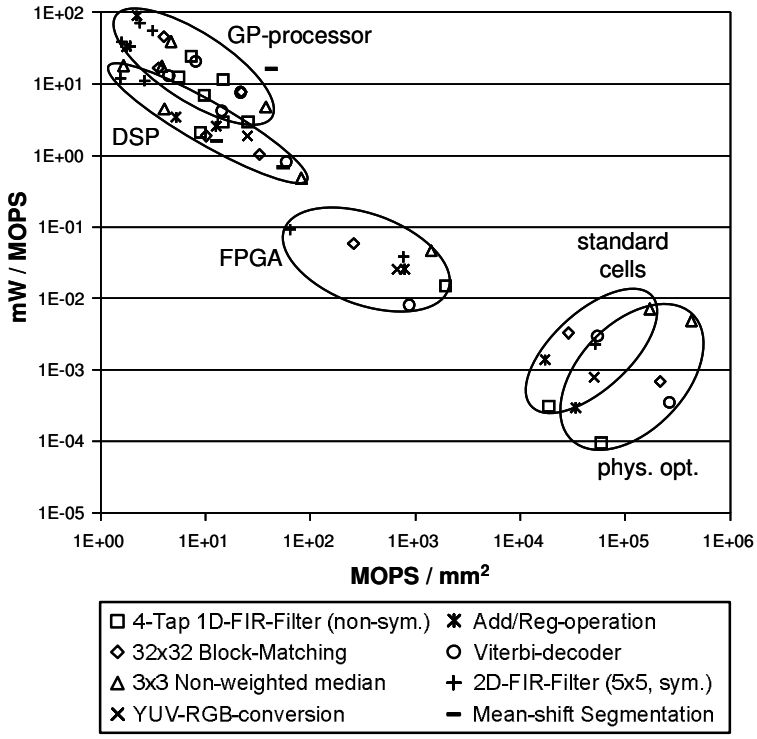


Figure 1. Design space for different architecture blocks[2]

high flexibility (not depicted in Fig. 1). The best power- and area-efficiency is achieved by physically optimised macros. However, they provide no flexibility.

FPGAs present an attractive compromise between these two extremes as they allow for highly parallelised implementations while preserving in-system reconfigurability at moderate implementation cost. The optimum partitioning of a system according to the available architecture blocks on a SoC is a difficult task that requires good knowledge about the design space. Based on these results it is important to study the resulting relations for applications that consist of number crunching components such as typical arithmetic datapaths and which also feature significant control overhead being implemented on processors as well as on FPGAs. MP3 decoding is such an example which features datapaths as well as control parts. Furthermore, it is a key application for commercial mobile devices requiring lowest power consumption and highest area efficiency. This paper is organised as follows: Section 2 describes the implementation of a MP3 decoder on a state-of-the-art FPGA. Section 3 outlines results from simulations and measurements. Section 4 attempts to show opportunities for further improvements by tailoring the FPGA structure for such a processing task. A conclusion is given in Section 5.

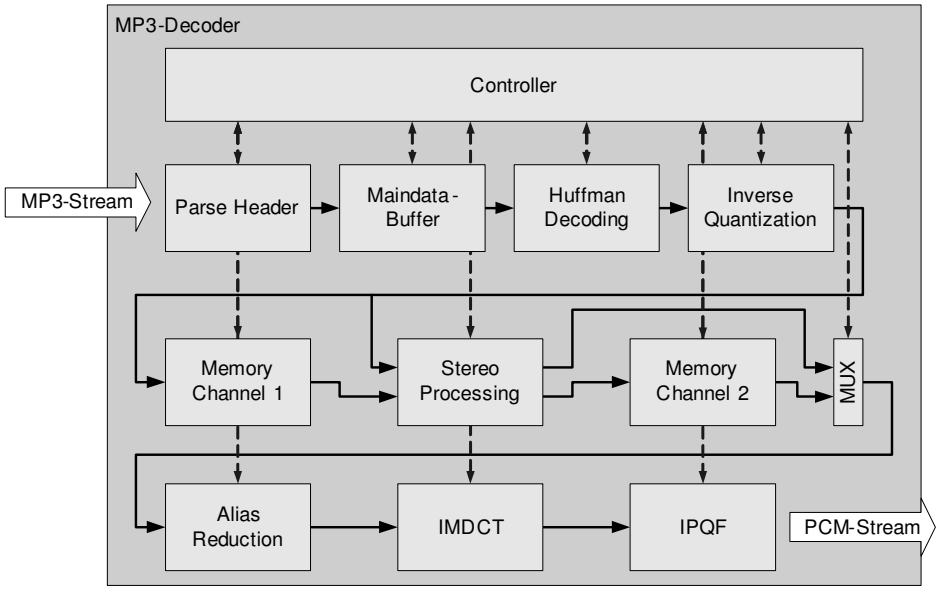


Figure 2. MP3 decoder processing blocks

2 MP3 Decoder Implementation Overview

As being mentioned in Section 1, the decoding of MP3 encoded music data is a key task for mobile devices. Here, the possible implementation results in terms of Audio data. The bitstream is composed of frames containing the compressed audio information[3]. The MPEG-2 standard[3] further describes a set of processing steps for decoding. These include e.g. Huffman decoding, inverse Discrete Cosine Transform, Quadrature Filters and some more.

The described implementation (see Fig. 2) realises each of these processing steps as a separate functional block. All these blocks work in parallel as part of a processing pipeline. Furthermore, separate processing of the left and right stereo channels is done in parallel as well. Especially in case the music is not encoded using the ”joint stereo” mode processing in both channels is completely independent from each other. Because of data dependencies it rendered rather difficult to identify further opportunities for parallelisation inside the processing blocks themselves. Instead, synthesis of the algorithm was optimised for maximum usage of those blocks inside the FPGA that are specialised for multiply/add/accumulate. In fact, 126 of these DSP blocks[5] work in parallel to execute the decoding algorithm. Consequently, the processing pipeline will operate at a rather low clock frequency ($<$5 MHz).

Special attention was paid to those blocks being most complicated to implement: the inverse DCT (IMDCT) and the inverse pseudo QMF (IPQF). These modules involve ROM tables for coefficient mapping (IMDCT) and a rather large ring buffer (IPQF). Implementation of stereo processing covers the basic parts only (dual mono, dual channel stereo). Joint Stereo decoding would substantially increase implementation effort and also power consumption; its absence does not adversely affect the observations described in Section 4.

3 Simulation and Implementation Results

The MP3 decoder has been synthesized for an Altera Stratix II FPGA (EP2S30F484C3, speed grade 3)[5] using the Altera Quartus II V5.0[6] software environment. This FPGA provides programmable logic blocks (ALUTs) and dedicated DSP blocks to realise functionality typical for signal processing algorithms. Further, there are several types of memory available which can be associated to specific processing stages.

Resource-type	Used / Achieved	Utilization
ALUT	18,701	68 %
Memory-Bits	334,592	24 %
DSP-Blocks	126	98 %
f_{max}	4.7 MHz	–

Table 1. FPGA resource usage for MP3 decoder

Table 1 shows the amount of resources from this FPGA component that were needed to perform the desired functionality. The implemented VHDL-Code has been functionally simulated with Mentor Graphics ModelSim (Version SE PLUS 6.1a). Results for processing a MP3 file with 8 seconds length are summarized in Table 2: Obviously, the IPQF block needs the highest amount of computation power; next in line followed by the IMDCT both of them consuming a magnitude more cycles than each of the other components. A similar effort balance can be seen from the resources used per component.

Functional block	Cycles used	Time $[ms]$
Parse Header	81,969	1.74
Maindata Buffer	1,798,298	38.26
Huffman Decoding	1,770,248	37.66
Inverse Quantization	2,013,002	42.83
Stereo Processing	352,512	7.50
Alias Reduction	738,090	15.70
IMDCT	16,845,606	358.42
IPQF	31,848,272	677.62

Table 2. Processing times for decoding 8 seconds of MP3

Power consumption figures have been determined using the power estimation feature of the Quartus II software[6]. Generally, there are two main effects that contribute to the total power consumption of the chip:

- base power dissipation in the clock network, circuit leakage current, I/O pads, internal regulators,
- dynamic power consumption determined by the actual amount of processing elements and memory in use.

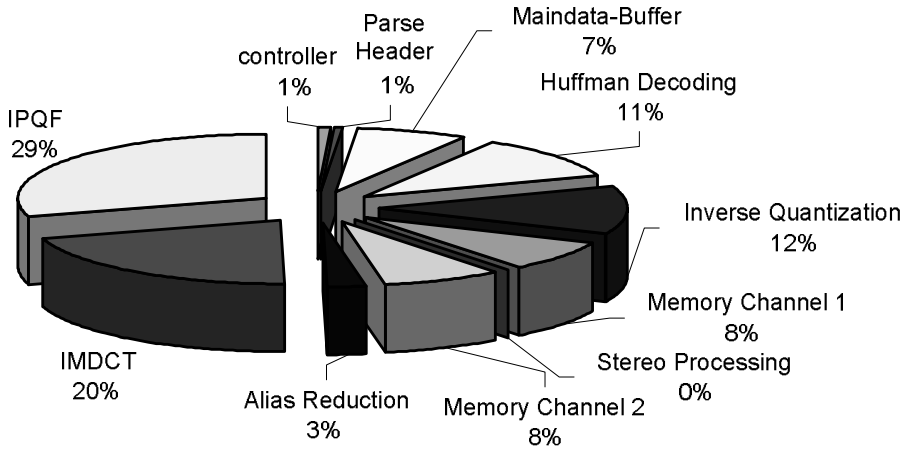


Figure 3. Dynamic power consumption per processing block

As can be seen from Table 3 total power consumption is largely determined by the base power dissipation of the chip even at a rather low clock frequency. In order to achieve these results we tried to minimize the number of registers and combinational logic. Therefore, we reduced the number of pipeline registers to a minimum, since the timing requirements can be fulfilled with a low clock frequency. Another effort to reduce the implementation costs was to optimize the tool settings of the development software, e. g. optimizing for area and using register retiming.

It is possible to further reduce the clock frequency by parallelization of the MP3 decoding on channel and frame level. This would lead to an increased controller complexity and the duplication of most of the MP3 decoder processing blocks (see Fig. 2). The positive impact on the power consumption of the clock frequency reduction interferes with the highly increased area demands. Here, the influence of the increased area on the power consumption is dominating, since the clock frequency is already low. Therefore, we did not apply parallelization of the MP3 decoding on channel and frame level.

Power consumption $[mW]$	
Dynamic P_{dyn}	27
Static P_{stat}	401
Total P_{total}	428

Table 3. Power consumption for the given implementation

Figure 3 confirms that the IPQF and IMDCT are by far the most power consuming processing blocks in the implementation.

4 Evaluation of Results

Power consumption of this implementation contains a fair amount of contributions from unused components in the chip. In order to evaluate suitability for integrating such a component into a mobile device some scaling might be applied. In the deployed FPGA (Altera Stratix II FPGA (EP2S30F484C3, speed grade 3)[5] 32% of the available ALUTs remain unused. Assuming a (hypothetical) FPGA providing (nearly) the number of ALUTs needed for the design, static power consumption P_{stat} given in Table 3 will be reduced by about 32% to

$$P_{stat} \approx (1 - 0,32) \cdot 400\ mW = 272\ mW \tag{4.1}$$

As the clock-frequency has a major influence on power consumption an exact (dynamic) adjustment of the clock frequency would also have a positive effect on static power consumption. In the example simulated in Section 3 eight seconds of MP3-data are decoded in 7.17 seconds. Assuming a timing safety margin of about 5 % for decoding the eight seconds MP3-data will require a processing time of

$$7.17sec \cdot 1.05 \approx 7.5sec \tag{4.2}$$

While meeting these timing requirements, the clock frequency can be reduced to

$$f_{max} = \frac{7.5\ sec}{8\ sec} \cdot 4.7\ \text{MHz} \approx 4.4\ \text{MHz} \tag{4.3}$$

With this reduced clock frequency static power consumption P_{stat} dominated by the clock network will also be reduced.

$$P_{stat}^{'} \approx \frac{4.4\ \text{MHz}}{4.7\ \text{MHz}} \cdot 272\ mW \approx 254\ mW \tag{4.4}$$

The dynamic power consumption will not be affected here as clock-gating is already applied meaning that the clock is switched off after the MP3-data has been decoded. Thus, total power consumption is obtained by

$$P_{total}^{'} = P_{stat}^{'} + P_{dyn} \approx 254\ mW + 27\ mW = 281\ mW \tag{4.5}$$

Especially for the largest processing blocks (IPQF, IMDCT) further design optimisation might lead to a further reduction in clock frequency; improvement estimates at this stage remain rather speculative though. A comparison of several alternative implementations given in Table 4 shows that its power consumption makes an FPGA-based MP3 decoder no attractive alternative to embedded low power processors like the ARM 940T[1,7]. The table also lists a reference ASIC implementation from literature. It's worth mentioning that the reported power consumption value refers to only the IPQF stage being the most power consuming processing block[8].

[a] Power consumption scaled to 0.09 μm technology.
[b] Optimized power consumption including e. g. FPGA size adaptation (see Equation 5).
[c] MP3-decoding @44kHz, only subband-synthesis (IPQF).

Architecture	Chip	Technology [μm]	Power consumption [mW]	Scaled power consumption [mW][a]
RISC[1]	ARM940T	0.18	145	18
FPGA	Stratix II	0.09	428 (281[b])	428 (281[b])
ASIC[8]	—	0.35	3 [c]	—
ASIP[9]	Xtensa HiFi 2 Audio Engine	0.09	30	30
DSP[10]	VS1011e	—	40	—
DSP[11]	—	0.35	165	3
ASIP[12]	AT83SND2CMP3	—	111	—

Table 4. Comparison of MP3 decoder implementations on various architectures

5 Conclusions

An FPGA implementation of a MPEG-1/2 Layer III standard decoder has been presented. The resulting power consumption however is far inferior to solutions on embedded processors optimized for low power consumption and well away from the requirements for integration into mobile devices. Computational requirements of the MP3 decoding algorithm are not high enough to fully utilise the computational capacity of this FPGA. The MP3 decoding algorithm does not provide enough inherent parallelism in order to achieve attractive implementation figures on a FPGA. For such low computational requirements serialized implementations on programmable processors are better suited. FPGA based implementations remain attractive for computational intensive applications involving highly regular datapaths with a high potential for parallelism. Apparently, the MPEG-2 standard prescribes algorithms for decoding that do not have such properties.

In order to evaluate suitability of such an implementation for a mobile device the total power consumption is the most important figure to consider. Modern mobile devices typically are equipped with 1000 mAh/3.6 V batteries. Such a battery would be completely drained by the FPGA implementation alone in less than 8 hrs. This rough calculus does not include additional contributions to the total consumption for ”listening to music” by voltage regulators, A/D conversion, power amplification and user interface operation. Further, innovating implementations shall radically increase listening times acknowledging the large music libraries which can be found on today’s music products. Typical listening times today are in the range of 10 to 30 hrs (e.g. iPod, Sansa) with storage capacities up to 30 or 60 GByte. Consequently, the total power consumption for listening to music should be well below 30 mW in order to achieve satisfactory listening times. This leaves considerably less than 10 mW for the decoding signal processor taking account of regulators and headphone amplification etc. Embedded processors are likely to achieve this target, FPGA implementations are far away from it.

References

1. H. Blume, D. Becker, M. Botteck, J. Brakensiek and T. G. Noll, *Hybrid functional and instruction level power modeling for embedded processor architectures*, in: Proc. Samos 2006 Workshop, Samos, Greece, 17–20 July, LNCS **4017**, pp. 216–226, (Springer, 2006).
2. T. G. Noll, *Application domain specific embedded FPGAs for SoC platforms*, invited survey lecture at the Irish Signals and Systems Conference 2004 (ISSC'04), (2004).
3. ISO/IEC 11172-3, Information Technology, *Coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s, Part3: Audio*, (1993).
4. M. Bosi and R. E. Goldberg, *Introduction to Digital Audio Coding and Standards*, (Kluwer Academic Publishers, 2003).
5. Altera Corporation, *Stratix II Device Handbook*, (2006). http://www.altera.com
6. Altera Corporation, *Quartus II Development Software Handbook v6.0*, (2006). http://www.altera.com
7. ARM Limited, *ARM940T Technical Reference Manual*, (2000). http://www.arm.com
8. T.-H. Tsai, Y.-C. Yang, *Low power and cost effective VLSI design for an MP3 audio decoder using an optimized synthesis-subband approach*, in: IEE Proc. Comput. Digit. Tech., vol. **151**, (2004).
9. Tensilica Inc., *Audio for Xtensa HiFi 2 audio engine and Diamone 330 HiFi*, (2007). http://www.tensilica.com
10. VLSI Solution Oy, *VS1011e - MPEG AIDIO CODEC*, (2005). Datasheet: http://www.vlsi.fi, 2005
11. S. Hong, D. Kim and M. Song, *A low power full accuracy MEPG1 audio layer III (MP3) decoder with on-chip data converters*, in: IEEE Transactions on Consumer Electronics, vol. **46**, (2000).
12. Atmel Corporation, *Single-chip MP3 decoder with full audio interface*, (2006). Datasheet: http://www.atmel.com

The HARWEST High Level Synthesis Flow to Design a Special-Purpose Architecture to Simulate the 3D Ising Model

Alessandro Marongiu[1,2] **and Paolo Palazzari**[1,2]

[1] Ylichron Srl
Via Anguillarese, 301
00123 S. Maria di Galeria (Rome), Italy
E-mail: {a.marongiu, p.palazzari}@ylichron.it

[2] ENEA - Computing and Modelling Unit
Via Anguillarese, 301
00123 S. Maria di Galeria (Rome), Italy

The 3D-Ising model is a mathematical model adopted in statistical mechanics, used to model basic phenomena as the paramagnetic-ferromagnetic transition. Due to its numerical intractability, many Monte Carlo methods have been proposed to solve the 3D Ising model. In this work we refer to the Heat-Bath algorithm and we show how it can be efficiently implemented onto an FPGA-based dedicated architecture through the adoption of the HARWEST High Level Synthesis (HLS) design flow, developed by Ylichron Srl.
Before going into the details of the specific implementation, the HARWEST HLS flow is reviewed, recalling the main modules and functionalities that are part of the flow.
The results achieved in 3D-Ising Model test case evidentiate speedup factors ranging from 1 to 3 orders of magnitudo with respect to optimized implementations on some current high-end processors.

1 Introduction

The 3D-Ising model[1] is a mathematical model adopted in statistical mechanics, used to model basic phenomena as the paramagnetic-ferromagnetic transition. While the 2D Ising model admits an analytical solution, the more interesting 3D case can be solved only with computationally intensive numerical methods. Due to its numerical intractability[2], many Monte Carlo methods have been proposed to solve the 3D Ising model, as the Demon[3] and the Heat-Bath[4] algorithms. In this work we refer to the Heat-Bath algorithm and we show how it can be efficiently implemented on an FPGA-based dedicated architecture through the adoption of an High Level Synthesis design flow.
In the next Section 2 the HARWEST High Level Synthesis design flow is shortly reviewed; the Section 3 recalls the 3D-Ising Model and the Section 4 shortly reports about the Heat-Bath algorithm. Tha last Section 5 reports the results obtained when the HARWEST design flow is used to produce a dedicated architecture which implements the Heat-Bath algorithm on a Virtex4 Xilinx FPGA.

2 The HARWEST High Level Synthesis Design Flow

The HARWEST High Level Synthesis (HLS) methodology, developed by Ylichron Srl, is one of the first outcomes of the HARWEST research project, aimed at creating a fully

automated HW/SW codesign environment. The HLS flow for the HW part is based on the flow reported in Fig. 1.
The system specifics are given by means of an ANSI C program (actually a subset of the ANSI C has been implemented, being pointers and recursion disallowed). At this very high level of abstraction, the correctness of the specifics is checked by running and debugging, on a conventional system, the C program (of course, we do not afford the task of proofing the correctness of a C program: we consider a program to be correct once it works correctly on some given sets of input data). Once the specifics have been fixed, they are translated into two different models of computation, namely the Control and Data Flow Graph (CDFG)[10] used to represent irregular, control dominated computations and the System of Affine Recurrence Equations (SARE)[7], used to represent regular, iterative computations characterized by affine dependencies on the loop nest indices. Given a set of resources (number of logic gates, number and sizes of memory banks, dimension of the I/O, ...), the two computational models are then mapped, with two distinct procedures, into a Data Path and a Control Finite State Machine (FSM) which enforce the behaviour expressed by the starting C program. Such an architecture, represented by the Data Path and the Control FSM, is further optimized (some redundand logic is removed and connections are implemented and optimized) and translated into an equivalent synthsizable VHDL code. Finally, the VHDL program is processed through the standard proprietary design tools (translation and map into the target technology, place and route) to produce the configuration bitstream.
It is worth to be underlined that the debugging phase is executed only at very high level of abstraction (on the C program), while all the other steps are fully automated and result to be correct by construction: this fact ensures that, once we have a working C program, the final architecture will show the same program behaviour.

2.1 Derivation of the CDFG and the SARE Computational Models

As reported in Fig. 1, the first step of the design flow has to transform the C program into an equivalent algorithm expressed by means of an instrinsically parallel computational model. In order to accomplish to this task, the HARWEST design flow uses two different modules to extract the CDFG and the SARE representation of the original C program. The sections of the C code that have to be transformed into the SARE formalism are explicitly marked by a pragma in the source code; if not otherwise specified, the C program is translated into the CDFG model.
When implementing the HARWEST flow, we decided to resort to a CDFG model with blocking semantics (i.e. a computing node does not terminate untill all its outputs have been read by the consumer nodes) because it does not require the insertion of buffers along the communication edges. As a consequence of the blocking semantics and of the precence of cycles on the CDFG, deadlocks could arise. In[13] we individuated a set of transformations into CDFGs of the basic C constructs (sequential statements, iteration, conditional) and of their compositions; these rules ensure deadlock never to arise. Thanks to these elementary transformations, each C program is transformed into an equivalent CDFG which does not deadlock.
The SARE model is used to deal with those portions of code

- constituted by the nesting of N iterative loops whose bounds are either constants or

affine functions of outer indices and

- having the statements in the loops that access m-dimensional arrays ($m \leq N$) through affine functions of the loop indices.

In order to extract the SARE representation of a C program which satisfies the previous two constraints, we have implemented the technique developed by P. Feautrier and described in[12].
An alternative way to express iterative affine computations is to use the SIMPLE language[8] which directly represents the SARE semantics since it allows to define

- the dimensionality of the computing space,
- the set of transformation equations with their affine dependencies and their definition domains,
- the set of input and output equations and their definition domains.

The HARWEST design flow accepts the definition of iterative affine computations through both C and SIMPLE programs.

2.2 Allocating and Scheduling CDFG and SARE

Resorting to the design flow in Fig. 1, after having derived the CDFG/SARE representation of the original algorithm, we are faced with the problem of allocating and scheduling such computations onto a set of predefined computing resources which represent a subset of the input constraints (further constraints could be involved by the need of a real-time behaviour or by throughput requirements).
In order to perform the allocation and scheduling operations, within the HARWEST design flow we created a library of computing modules which are the building blocks to set-up the final architecture. Each module is constituted by
• a collection of static informations regarding
▷ the area requirements (basic blocks needed for the implementation of the module onto a certain technology - LUT, memory modules, multipliers, ...),
▷ the module behaviour (combinational, pipelined, multicycle),
▷ the response time (propagation delay, setup time, latency, ...)
• a collection of files (EDIF format) to implement the module onto different target technologies (for instance, FPGAs from different vendors).
Allocating and scheduling a computation requires a time instant and a computing resource (i.e. a module) to be assigned to each operation of the computation. In order to do this, the HARWEST design flow implements a list scheduling heuristic, derived from the algorithm presented in[11], to deal with CDFG. It is worth to be underlined that the algorithm in[11] has been deeply modified to allow the sharing of the same HW module by different SW nodes.
When the original C program is structured with different functions, the program can be flattened through the inline expansion of the functions, thus generating a unique CDFG and a unique FSM. In order to reduce the complexity of the scheduling operation, as a simplifying option, each function - starting from the deepest ones - can be translated onto a different CDFG. The HARWEST flow initially schedules the CDFGs corresponding to code which

has not inner functions; successively such FSMs are scheduled as asynchronous modules when the CDFG of the calling function is scheduled; these asynchronous modules are synchronized with the FSM of the calling CDFG through a basic start/stop protocol.

Regarding the SARE computations, once the iterative N-dimensional algorithm has been expressed in the SARE formalism, it is allocated and scheduled onto a virtual parallel architecture - with a given spatial dimensionality $p < N$ - by means of an affine space-time transformation which projects the original algorithm, represented through some affine dependencies on the N-dimensional integer lattice, into the p-dimensional processor space[9,7]; the remaining $N - p$ dimensions are allocated to the temporal coordinate and are used to fix the schedule. According to the input constraints on the architecture size, a clustering step is eventually performed to allow the final architecture to fit into the available resources.

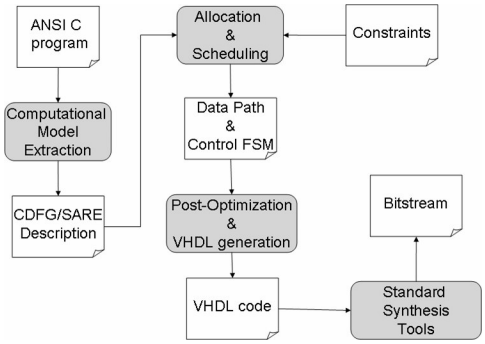


Figure 1. The HARWEST High Level Synthesis design flow

2.3 Post-Optimization and VHDL Generation

Once obtained the allocation and the scheduling for the CDFG and/or the SARE, the final architecture has still to be defined in the part concerning the interconnection/control network. In fact, during the scheduling step, all the details about the connections and the structures to support HW sharing are neglected. In this final phase all the connections are implemented, introducing and defining the multiplexing logic necessary to implement the communication lines; furthermore, a set of registers and multiplexers on the control lines are inserted, in order to allow a coherent management of iterations in presence of HW sharing.
After previous two steps have been executed, all the informations regarding the final architecture are fixed and the synthesizable VHDL code can be generated.

3 The 3D Ising Model

Let us refer to the 3D Edwards-Anderson spin-glass model[5], where a variable σ_i ($i = \pm 1$) is present in each point of a L sized cubic lattice. A variable in site i is coupled to a variable in a neighbouring site j through the (static) random coupling factor $F_{ij} = \{\pm 1\}$. The system energy is given by

$$U = \sum_{i \in Lattice} \left(\sum_{j \in Neigh(i)} F_{ij} \sigma_i \sigma_j \right) \tag{3.1}$$

being the first summation running over all the lattice sites and the second spanning the neighbourhood of each lattice site. In order to study the system properties, a lot of Monte Carlo runs have to be executed, averaging over different choices of the (random and static) coupling factors. This problem is such a computational challenge that Italian and Spanish physicists are running a joint project to set-up a dedicated FPGA-based parallel computer (Ianus) to purposely address the aforementioned computation[6]. In our opinion, the main disadvantage of a Ianus-like approach to special purpose computing is that the architecture has to be manually developed, through the slow and error prone classical hardware design flow. On the base of our experience, we think that FPGA based computing architectures should be coupled with the - now emerging - HLS methodologies.

4 The Heat Bath Algorithm

The kernel to implement the Heat-Bath algorithm, expressed in a C-like language, is shown through the following portion of code.

```
int spin[L][L][L];
int Jx[L][L][L];
int Jy[L][L][L];
int Jz[L][L][L];
for (k =0; k <L; k++ )
  for (j =0; j <L; j++ )
   for (i=0; i <L; i++ ) {
    nbs = spin[(i+1)modL][j][k]*Jx[(i+1)modL][j][k]+
     spin[(i-1) mod L][j][k]*Jx[(i-1)modL][j][k]+
     spin[i][(j+1)modL][k]*Jy[i][(j+1)modL][k]+
     spin[i][(j-1)modL][k]*Jy[i][(j-1)modL][k]+
     spin[i][j][(k+1)modL]*Jz[i][j][(k+1)modL]+
     spin[i][j][(k-1)modL]*Jz[i][j][(k-1)modL];
    if ( rand () < HBT( nbs) )
     spin[i][j][k] = +1;
    else
     spin[i][j][k] = -1;
   }
```

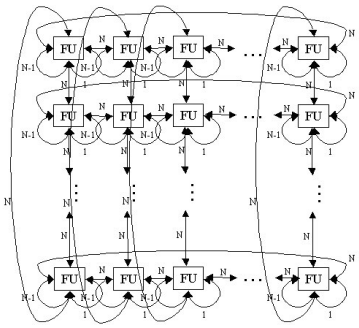


Figure 2. Structure of the parallel architecture produced by the HARWEST Design flow applied to the SARE expression of the Heat Bat algorithm

Thanks to its regularity, its transformation in the SARE formalism is straightforward. For this reasons, and because the efficiency of manually written SARE is usually higher with respect to the automatically generated ones, we decided to directly implement the Heat-Bath algorithm through the SIMPLE language.
In the previous code, the 3D array spin[L][L][L] contains the status (i.e. spin up or spin $down$) of all the lattice sites. The Jx, Jy and Jz arrays are used to store the (static) random coupling factors in the three coordinate directions (they represents the F_{ij} factors in Eq. 3.1). The three nested for cycles perform the spin update in all the lattice sites. The 3D system spin dynamic is simulated by iterating over time the previous three loops.

5 Results

After applying the design flow depicted in Fig. 1 to the SARE algorithm derived from the Heat-Bat code sketched above, we obtained the parallel architecture - constituted by $N \times \frac{N}{2}$ Functional Units - reported in Fig. 2 (N=L=24). Edges represent communication paths and the labels are the time delay (in clock cycles) associated to each edge. The picture of the architecture is reported to show how the regularity of the initial algorithm is maintained in the final architecture.

Each FU is constituted by

- One Random Number Generator, implemented through a 32-bit shift register feedback algorithm;
- One LUT (HBT[7] in the Heat-Bath code (Section 4)) containing the probability values to decide wether a spin value should be set or reset, according to a comparison with the locally generated random value;
- One computing engine which, starting from the local spin value, the neighbouring spin values and the coupling coefficients, computes the ”local energy” value to address HBT

- One update engine which, on the basis of HBT(local energy) and the local random value, determines the new spin value.

In our tests we set L=24 and, on the Xilinx Virtex4 FPGA XC4VLX160, we obtained the following synthesis results:

- Number of Slices: 61918
- Number of Slice Flip Flops: 95906
- Number of 4 input LUTs: 57678
- Number of FIFO16/RAMB16s: 168
- Number used as RAMB16s: 168
- f_{ck} = 91.5 MHz

According to the scheduling enforced by the HLS flow, 2×24 clock cycles are necessary to update the whole cubic lattice, corresponding - at the synthesis clock frequency of 91.5 MHz - to 520 ns. Resorting to the time necessary to update a single spin site (a commonly used reference figure), the synthesized architecture is able to update a spin location every 38 ps. To give an idea of the quality of such a result, we report in Table 1 the time extimations, derived in[6], for the time necessary to update a spin using optimized SW implementations on a Blue Gene/L node, on a Pentium4 processor, on a Cell processor and on the ClearSpeed CSX600 processor:

Processor	Spin Update Time (ps)	SlowDown vs FPGA
Blue Gene/L	45000	1184
Pentium 4	24000	632
Cell	1250	33
CSX600	1330	35

Table 1. comparative results for the implementation of the Heat Bath algorithm on different processor architectures

6 Conclusions

In this work we addressed the feasibility of an High Level Synthesis (HLS) approach to design a FPGA-based architecture dedicated to the simulation of the 3D Ising model. To this aim, we used the HARWEST HLS design flow developed by Ylichron S.r.l., achieving - in about two week - a working prototype which offers up to three order of magnitudo speed-up with respect to optimized SW implementations on current high-end processing nodes. These results clearly demonstrate how the automatic exploitation of low level parallelism within FPGA devices characterized by very high internal memory bandwidth could be a very efficient answer to some classes of computationally challenging problems. Furthermore, we think that the very good results achieved (in terms of performances and of

developing time), make reasonable the adoption of FPGA based computing architectures which can be reliably, quickly and efficiently programmed through the HLS developing environments such as the one presented in this work.

Acknowledgements

We are indebted to Raffaele Tripiccione and Filippo Mantovani which introduced us to the Ising 3D model.

References

1. S. G. Brush, *History of the Lenz-Ising model*, Reviews of Modern Physics, **39**, 883–893, (1967).
2. B. A. Cipra, *The Ising model is NP-complete*, SIAM News, **33**, 6.
3. J. J. Ruiz-Lorenzo and C. L. Ullod, *Study of a microcanonical algorithm on the J spin-glass model in d=3*, Comp. Physics Communications, **105**, 210–220, (2000).
4. M. Creutz, *Quantum Fields on the Computer*, (World Scientific, 1992).
5. M. Mezard, G. Parisi and M. A. Virasoro, *Spin Glass Theory and Beyond*, (World Scientific, 1997).
6. F. Belletti et al., *Ianus: an adaptive FPGA computer*, in: IEEE Computing in Science and Engineering, **1**, 41–49, (2006).
7. A. Marongiu and P. Palazzari, *Automatic implementation of affine iterative algorithms: design flow and communication synthesis*, Comp. Physics Commun., **139**, 109–131, (2001).
8. A. Marongiu, et al., *High level software synthesis of affine iterative algorithms onto parallel architectures*, in: Proc. 8th International Conference on High Performance Computing and Networking Europe (HPCN Europe 2000), May 8–10, Amsterdam, (2000).
9. A. Marongiu and P. Palazzari, *Automatic mapping of system of N-dimensional affine recurrence equations (SARE) onto distributed memory parallel systems*, IEEE Trans. Software Engineering, **26**, 262–275, (2000).
10. G. G. De Jong, *Data Flow Graphs: System specification with the most unrestricted semantics*, in: Proc. EDAC, Amsterdam, (1991).
11. G. Lakshminarayana et al., *Wavesched: a novel scheduling technique for control flow intensive designs*, IEEE Trans on CAD, **18**, 5, (1999).
12. P. Feautrier, *Data flow analysis of array and scalar references*, Int. J. of Parallel Programming, **20**, 23–52, (1991).
13. G. Brusco, *Generazione di control data flow graph a partire da linguaggi imperativi e loro schedulazione*, Master Thesis in Electronic Engineering, University ”La Sapienza” (Rome), (2004, in Italian).

Towards an FPGA Solver for the PageRank Eigenvector Problem

Séamas McGettrick, Dermot Geraghty, and Ciarán McElroy

Dept of Mechanical and Manufacturing Engineering
Trinity College Dublin, Ireland
E-mail: {*mcgettrs, tgerghty, ciaran.mcelroy*}*@tcd.ie*

Calculating the Google PageRank eigenvector is a massive computational problem dominated by Sparse Matrix by Vector Multiplication (SMVM) where the matrix is very sparse, unsymmetrical and unstructured. The computation presents a serious challenge to general-purpose processors (GPP) and the result is a very lengthy computation. Other scientific applications have performance problems with SMVM and FPGA solutions have been proposed. However, the performance of SMVM architectures is highly dependent on the matrix structure. Architectures for finite element analysis, for example, may be highly unsuitable for use with Internet link matrices. An FPGA architecture can bring advantages such as custom numeric units, increased memory bandwidth and parallel processing to bear on a problem and can be deployed as a coprocessor to a GPP.

In this paper, we investigate the SMVM performance on GPPs for Internet matrices. We evaluate the performance of two FPGA based SMVM architectures originally designed for finite element problems on a Virtex 5 FPGA. We also show the effect of matrix reordering on these matrices. We then investigate the possibility of outperforming the GPP using parallelization of processing units, showing that FPGA based SMVM can perform better than SMVM on the GPP.

1 Introduction

The Internet is the world's largest document collection. As it contains over 25 billion pages[1] to choose from, finding one desired page can seem a daunting task. Search engines have been designed to wade through the vastness of the Internet and retrieve the most useful documents for any given query. A search engine has to do much work before it can return any results. Long before a user submits a query, a search engine crawls the Internet and gathers information about all the pages that it will later search. These pages are then indexed, so that a list of related pages can be retrieved when a query is submitted. The next step is ranking the pages returned for a query. This can be done either at or before query time. In both cases, this process is handled by a ranking algorithm. Documents with relevant content get a high rank and irrelevant documents receive low ranking scores. In response to a query the search engine displays the results of a search in order of rank.

Before the advent of Google, spammers had found ways to manipulate the existing ranking algorithms. Thus, one often had to search through pages of useless results (Spam) before finding a page with useful information. Google tended to be immune to spam for every query. This improved search engine was made possible by a new ranking algorithm called PageRank. PageRank remains at the heart of the Google search engine to this day[2].

PageRank achieved a higher quality result by taking advantage of the hyperlinked structure of the Internet[2]. Each hyperlink to a page is counted as a vote for the content of the page to which the hyperlink leads. The more votes a page gets, the better its content is

assumed to be, and thus the higher its ranking. The PageRank calculation has been dubbed the largest matrix calculation in the world[3]. The matrix at the centre of the calculation is of the order of 25 billion rows and columns. It is constantly growing as new pages are added to the Google index.

In Section 2, the PageRank Algorithm is discussed and the algorithm is explained. We show how the algorithm is essentially a problem in linear algebra. In Section 3, some benchmark data for PCs and custom FPGA architectures, which are used in other scientific applications, is presented. These tests show the effects of reordering on Internet matrices. These results are then extrapolated to show the performance achievable using multiple processing units in parallel on a Virtex 5 FPGA. Finally, a reflection on the current work to implement the PageRank algorithm on FPGA and discussion of future progress is presented.

2 PageRank Algorithm

The PageRank algorithm was developed by Sergey Brin and Lawrence Page in 1998 at Stanford University[4]. They later created Google, which is currently the world's largest and most used search engine[2]. PageRank is query independent, which means that page rankings are calculated prior to query time. This system allows Google to return query results very quickly. PageRank does not rank in order of relevance to the query, but instead it ranks by general importance of the page. By definition the rank of a page, $r(P_i)$, is calculated from the sum of the ranks of those pages, $r(P_j)$, which point to it, moderated by the number of out-links from each of these pages. Thus,

$$r(P_i) = \sum_{P_j \epsilon B_{P_i}} \frac{r(P_j)}{|P_j|} \tag{2.1}$$

where B_{P_i} is the set of pages which point to page P_i and $|P_j|$ is the number of out-links on page P_j. To begin with all pages are assigned an initial PageRank of 1/n where n is the total number of pages in the network or the indexed part of the network. The equation above can be written in iterative form, as follows:

$$r_{k+1}(P_i) = \sum_{P_j \epsilon B_{P_i}} \frac{r_k(P_j)}{|P_j|} \tag{2.2}$$

where $r_k(P_i)$ is the PageRank of P_i after k iterations. The equation is such that that after a series of iterations of this equation, the PageRank will converge to a stationary value. The PageRank equation for a full system can be more conveniently represented in terms of vectors and matrices. Thus, the graph of the web is represented by an Internet link matrix, H, which defines the interconnects between the pages. Each row in H shows the out-links from a page and each column shows the in-links to a page. When the moderating factor, $|P_j|$ is applied to a row, a probability vector is obtained. i.e. the elements of a row sum to one.

Now the PageRank vector, which is a vector, can be defined, where each element is the rank of the corresponding page, as follows:

$$\pi^{(k+1)T}(P_i) = \pi^{(k)T} H \tag{2.3}$$

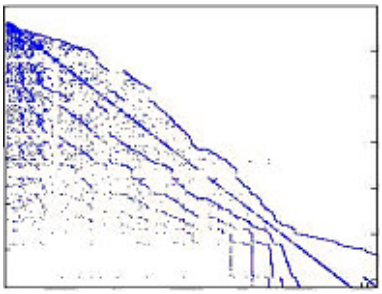


Figure 1. Example Internet Matrix (from a crawl of tcd.ie)

This equation is of the form $\lambda x = Ax$ with $\lambda = 1$. Therefore, it is an eigenvector problem. One of the most straightforward ways of solving this system is the power method. In fact, the above iterative equation is a statement of the power method. The equation clearly shows that the predominant operation needed to solve this problem is Sparse Matrix by Vector Multiplication (SMVM) between the Internet link matrix (H) and the PageRank vector.

3 Performance

The matrix used in Google's PageRank calculations is a very large matrix. It has 25 billion rows and columns; this number is constantly growing as Google includes more pages in its search database[5]. The matrix is also unsymmetrical and very sparse; it has, on average, about 10 entries per column[6]. Fig. 1 shows a sample web matrix. Currently PageRank is calculated on general-purpose processors like the Intel and AMD processors. These processors often perform poorly with large SMVM problems due to the lack of structure of the matrix[7]. An FPGA solution would have a number of advantages over the traditional processor approach. An FPGA can have more than one memory channel and so can run the calculation in parallel across multiple processing units. The hardware of the FPGA can be specially optimized to compute ranking algorithms efficiently. To the best of our knowledge, no FPGA solution for search algorithms exists. To better understand how the PageRank algorithm could benefit from being implemented on an FPGA, two FPGA based architectures designed for use in Finite Element problems were benchmarked against a GPP. The field of finite element also deals with large sparse matrices. However, Internet link matrices differ from the matrices used finite element analysis as they are not symmetric or tightly banded[8]. The first of the two FPGA-based Finite Element solvers used was the column based SMVM as described by Taylor et al[9]. The second FPGA-based solver is a tile solver. This architecture is patent pending so no details will be given at this time. Some technical details on the architectures used for these benchmarks are given in Table 1. The models used to calculate the performance of the two FPGA based architectures were verified in RTL on a Virtex II device and extrapolated to Virtex 5[10].

Table 2 details the matrices used. All of these matrices come from the Stanford WebBase project[11] or the WebGraph datasets[12] with the exception of web_trinity, which was generated from a crawl of the Trinity College Website. The matrices vary in size from 2.4

	CPU/FPGA	CLK	PEAK
PC	Pent. Xeon Woodcrest	3 GHz, 1333 MHz FSB	6 GFLOPs
Column Solver	Virtex 5	222 MHz	444 MFLOPs
Tile Solver	Virtex 5	444 MHz (2x222 MHz)	888 MFLOPs

Table 1. Benchmark architecture details

Num.	Name	Nodes	No. of Links
1	arabic-2005_5K	500000	9877485
2	cnr-2000_5K	325557	3216152
3	eu-2005_5K	500000	11615380
4	in-2004_5K	500000	5422294
5	indochina-2004_5K	500000	8147699
6	it-2004_5K	500000	9387335
7	sk-2005_4K	400000	13391888
8	uk-2002_5K	500000	6998368
9	uk-2005_5K	500000	11192060
10	web_matrix_1-5M	1500000	12392081
11	web_matrix_1M	1000000	8686242
12	web_standford	281903	2312497
13	web_trinity	608358	2424439
14	webbase-2001_5K	500000	4214705

Table 2. Benchmark matrices details

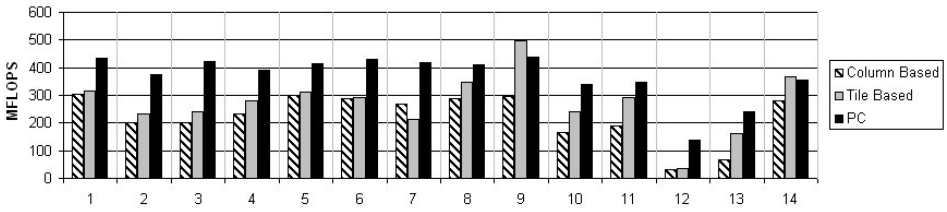


Figure 2. SMVM benchmarks for Internet Link Matrices

million non-zeros to 13.4 million non-zeros. They are much smaller than full size Internet link matrices but are large enough to test the efficiency of an architecture. The PC benchmark was a dual threaded C program. The results of the benchmarking are shown in Fig. 2.

The benchmark tests in Fig. 2 show how a single Processing Element (PE) on the FPGA-based architectures compare to the 3 GHz Intel Xeon (Woodcrest). PageRank matrices have massive storage requirements. In order to achieve a relatively cheap and scalable memory DRAM can be used. The two FPGA solvers used for benchmarking can connect to DDR DRAM. Xilinx have shown that Virtex 5 FPGAs can be connected to DDR2-667[13].

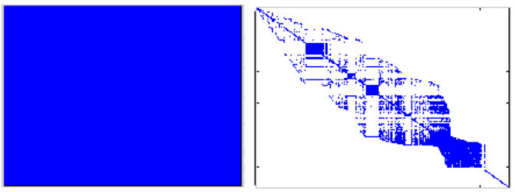


Figure 3. Web_stanford before and after RCM reordering

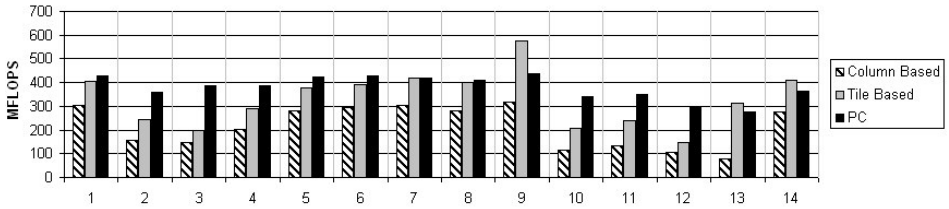


Figure 4. SMVM Benchmark Performance with RCM reordering

Using DDR2-667 RAM a 64 bit word can be read from memory every 667 MHz clock cycle. Each of the two FPGA solvers used for benchmarking requires a 96 bit word every clock cycle (64 bit non-zero entry and a 32 bit address). To maximise the memory bandwidth being used the FPGA solvers would need to run at 444 MHz, which, is unachievable with current FPGA technology. The column based architecture therefore uses a more realistic 222 MHz clock, thus, using only half the available memory bandwidth. The tile solver approaches this problem slightly differently. It internally has two MAC units running at 222 MHz in parallel which allows it to effectively run at 444 MHz. A similar system could be implemented for the column based system. However, to be consistent with Taylor's architecture it was decided to use a single MAC unit.

It is interesting to see in Fig. 2 that the FPGA-based tile solver actually outperforms the PC on two of the matrices (9,14), even though the FPGA clock is running almost 12 times slower. The GPP only achieves between 3% and 6% of its theoretical peak. Matrices 12 and 13 perform poorly on all architectures. These matrices give poor performance for two reasons. They have very few entries per row and their entries are very scattered. The left side of Fig. 3 shows a pictorial view of matrix 12. The matrix is so scattered that the picture of the whole matrix appears dense.

It was decided that some investigation should be done into how a well-known reordering scheme like Reverse Cuthill McKee (RCM)[14] would affect this performance. RCM reordering is used to reduce the bandwidth of link graphs by renumbering the nodes. Fig. 3 shows the difference in non-zero distribution before and after RCM reordering of test matrix 12. The diagram shows that simply renumbering the nodes decreased the bandwidth of this matrix dramatically. The matrices were reordered and benchmarked on the three architectures. The results of the reordering benchmarks are shown in Fig. 4. Fig. 5 shows the change in performance with reordering. Reordering the matrices caused all 3 archi-

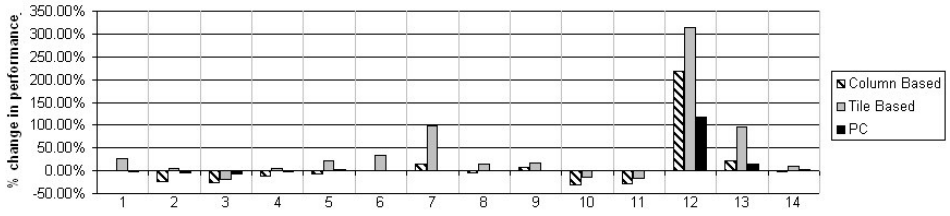


Figure 5. % change in performance with RCM reordering

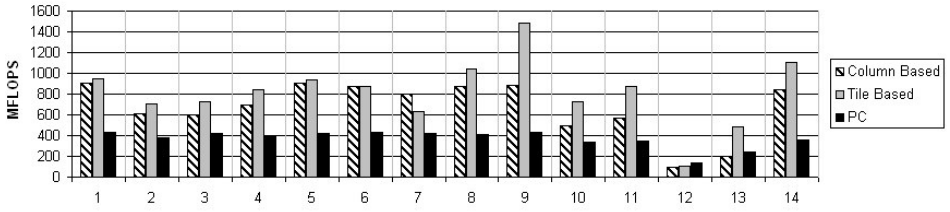


Figure 6. SMVM Benchmark Performance of 3 x PE connected via 3 DDR interfaces

tectures to improve performance on the two poorly performing matrices. Reordering also caused an increase in performance for a number of the other matrices. It also caused slight degradation in performance of others. Matrices 12 and 13 were the only matrices that were not ordered by the spiders that generated them, and thus showed the largest increase in performance when RCM was applied. This highlights the importance of reordering. The reordering could be implemented by the spider and so avoiding an extra reordering stage. The column based FPGA architecture showed a less of an increase in performance with reordering than the tile solver, which can be attributed to RAW hazards which stalls processing[15]. Since these tests use an adder pipeline 14 cycles long[10] and the average column only contains 10 entries, a large number of RAW hazards occur. There are many different types of reordering schemes. RCM is expensive to implement and is probably not the best algorithm for Internet link matrices but it does highlight that reordering is necessary.

4 Parallelisation

In the previous section we saw that a single PE could outperform a 3 GHz Pentium Woodcrest processor when calculating SMVM for some, though not all, Internet link matrices. Further performance improvements are possible through the use of parallel PEs and by increasing memory bandwidth by implementing multiple parallel memory ports. In this section we investigate the performance of a system with three parallel PEs and three memory ports. As before the tile solver's PE has a pair of double precision floating point MAC units operating at 222 MHz for an effective rate of 444 MHz. Both solvers show an increase in performance.

Fig. 6 shows that the FPGA-based solutions have the ability to outperform the GPP

by as much as 3 times. This is due to their ability to take advantage of parallelism in the computations. Both FPGA-based solvers outperform the 3 GHz Pentium Woodcrest Xeon for all our test matrices except 12, even though their clock rate is over 13 times slower. The FPGA-based tile solver is the architecture that shows the best performance with a peak performance of almost 1450 MFLOPS. The FPGA-based column ordered solver performs well, but its performance is still badly hampered by RAW hazards as discussed earlier.

5 Conclusions

The PageRank eigenvector calculation is a large and computationally intensive matrix problem. General-Purpose processors only achieve a fraction of their peak performance when calculating large Sparse Matrix by Vector products[16]. The GPP has a single pipe to memory and so cannot exploit parallelisms in its calculations. The large number of IO pins on modern FPGAs makes it possible to use them to achieve high memory bandwidth by implementing multiple memory ports and exploiting parallelisms in algorithms. Little performance data for Internet link matrices exists in the public domain and no co-processor architectures specialized for Internet link matrices exist to the best of the authors knowledge. Finite Element Analysis uses large sparse matrices and a number of FPGA based architectures exist for solving these problems. Some of these architectures were benchmarked against a GPP. The benchmark results show a number of interesting results. The FE FPGA based solvers were more efficient than the GPP. They achieved approximately 50% FPU utilisation when computing Internet style matrices. The GPP (3 GHz Intel Xeon (Woodcrest)) only achieved a 3-8% FPU utilisation. This increased efficiency meant that a single FPGA-based SMVM processing element could out-perform the GPP for SMVM of some internet link matrix even though the FPGA's clock rate was over 12 times slower.

A number of the matrices showed very poor performance across the three test architectures. The NZ elements in these matrices were very scattered. RCM , which is a well known reordering scheme, was applied to all matrices. RCM improved the performance of the matrices that performed very poorly by a much as 3 times , but it did not give an across the board increase in performance. RCM is not the right reordering scheme for Internet link matrices. These results highlight the need to further investigate reordering. A suitable reordering scheme has the potential to further increase the performance of an FPGA solver for the PageRank algorithm.

The large number of pins on the FPGA makes it possible implement multiple, parallel memory ports. This allows 3 PE to work in parallel and increases the performance of the FPGA solutions so that they both outperform the GPP for all our test matrices. In one case the tile solver calculated the solution 3 times quicker than the GPP despite having a clock rate 13 times slower. The tile solvers peak performance was measured at 1450 MFLOPS. This superior performance is due to the FPGAs ability to exploit parallelism in the calculation.

The solvers used in these experiments were designed for use with Finite Element matrices. FE calculations are usually done using IEEE floating-point doubles. It is thought that Internet link matrix calculations do not need to be calculated using this precision[6]. Lower precision number formats would allow for greater parallelism as well as a reduction in adder and multiplier latencies. Investigating this possibility is the next step towards designing an architecture for the PageRank eigenvector problem on FPGA. This architec-

ture will need access to large banks of memory and will rely greatly on parallelisation to achieve maximum performance for Internet link matrices.

In this paper we have shown how even FPGA based hardware not optimised for Internet link matrix SMVM can outperform the GPP ,by up to 3 times using slow but cheap and scalable DDR memory, and by using parallel processing units.

Acknowledgements

This work has been funded by Enterprise Ireland's Proof of Concept fund.

References

1. D. Austin, *How Google finds your needle in the web's haystack*, http://www.ams.org/featurecolumn/archive/pagerank.html
2. http://www.google.com/technology/
3. C. Moler, *The World's largest computation*, Matlab news and notes, **Oct.**, 12–13, (2002).
4. S. Brin, L. Page and M. Coudreuse, *The anatomy of a large-scale hypertexual web search engine*, Computer Networks and ISDN systems., **33**, 107–117, (1998).
5. A. N. Langville and C. D. Meyer, *A survey of eigenvector methods for web information retrievel*, The SIAM Review, **27**, 135–161, (2005).
6. A. N. Langville and C. D. Meyer, *Google's PageRank and Beyond, The Science of Search Engine Rankings*, (Princeton University Press, 2006).
7. W. D. Gropp, D. K. Kasushik, D. E. Keyes and B. F. Smith, *Towards realistic bounds for implicit CFD codes*, Proceedings of Parallel Computational Fluid Dynamics, , 241–248, (1999).
8. S. McGettrick, D. Geraghty and C. McElroy, *Searching the Web with an FPGA-based search engine*, ARC-2007, LNCS, **4419**, 350–357, (2007).
9. V. E. Taylor, *Application specific architectures for large finite element applications*, PhD Thesis, Berkeley California, (1991).
10. Xilinx LogiCore, *Floating Point Operators V.3*, (2005). http://www.xilinx.com
11. http://dbpubs.stanford.edu:8091/~testbed/doc2/WebBase/
12. P. Boldi and S. Vigna, *The WebGraph framework 1: Compression techniques*, in: WWW-2004, pp. 595–601, (ACM Press, 2004).
13. K. Palanisamy and M. George, *High-Performance DDR2 SDRAM Interface in Virtex-5 Devices*, Virtex 5 Application notes, XAPP858 (v1.1), (2007).
14. http://people.scs.fsu.edu/~burkardt/f_src/rcm/rcm.html
15. C. Mc Sweeney, D. Gregg, C. McElroy, F. O'Connor, S.McGettrick, D. Moloney and D. Geraghty, *FPGA based Sparse Matrix Vector Multiplication using commodity memory*, FPL, (2007).
16. C. Mc Sweeney, *An FPGA accelerator for the iterative solution of sparse linear systems*, MSc Thesis, Trinity College Dublin, Ireland, (2006).

Author Index

A

Ahn, Dong H. 621
Aldinucci, Marco 235, 355
Alessandrini, Victor 689
Aliaga, José I. 287
Alvarez, José Antonio 165
an Mey, Dieter 571, 619
Anderson, Robert W. 697
Anshus, Otto J. 71
Arai, Yusuke 459
Arnold, Dorian C. 621
Arnold, Guido 61
Arnold, Lukas 467
Atienza, David 509

B

Bücker, H. Martin 303, 451
Baaden, Marc 729
Badía, José M. 255
Bader, Michael 175
Badia, Rosa M. 129
Barker, Kevin J. 89
Barrault, Maxime 29
Becker, Daniel 619
Beetz, Christoph 467
Behr, Marek 19
Belletti, Francesco 553
Bencteux, Guy 29
Benner, Peter 255
Bernreuther, Martin 53
Berrendorf, Rudolf 425
Berthold, Jost 121
Birkeland, Tore 433
Bischof, Christian H. 303
Bjørndalen, John Markus 71
Blanco, Vicente 99
Blume, Holger 777
Boeck, Thomas 483
Boldarev, Alexei 475
Boldyrev, Sergei 475
Bollhöfer, Matthias 287
Bonacic, Carolina 389
Bond, Peter 729
Botteck, Martin 777
Bottino, Alberto 713
Boullón, Marcos 99
Bounanos, Stylianos 397
Bournas, Odysseas 705
Breitmoser, Elena 705
Brunst, Holger 637
Buchholz, Martin 53
Buchty, Rainer 757
Bui, Van 573
Bungartz, Hans-Joachim 53
Buzzard, Gregery T. 609

C

Cabaleiro, José Carlos 99
Cacitti, Manuel 355
Cancès, Eric 29
Casas, Marc 129
Castillo, Maribel 255
Chandrashekar, Ashok 767
Chapman, Barbara . . 3, 571, 573, 661
Clematis, Andrea 139, 147, 441
Coleman, Thomas F. 295
Corbalan, Julita 501
Cotallo, Maria 553
Crngarov, Ace 425
Cruz, Andres 553

D

D'Agostino, Daniele . . . 139, 147, 441
D'yachenko, Sergei 475
Danelutto, Marco 235, 355
Davis, Kei 89
de Supinski, Bronis R. 621
Desplat, Jean Christophe 593
Dolfen, Andreas 601
Drakenberg, N. Peter 329
Dreher, Jürgen 467
Durrieu, Marie-Pierre 729
Dutt, Nikil 767
Dümmler, Jörg 321

E

Eberhard, Peter 37

Eicker, Norbert 381
Erbacci, Giovanni 687

F

Falcou, Joel 243
Fan, Shiqing 517
Faßbender, Heike 255
Felch, Andrew 767
Fernández, José Jesús 165
Fernández, Luis Antonio 553
Fisher, Aaron C. 697
Fleissner, Florian 37
Fleury, Martin 397
Fox, Jeffrey J. 609
Frings, Wolfgang 583
Furlong, Jeff 767
Fürlinger, Karl 677

G

Güneysu, Tim 741
Gómez, Antonio 347
Galizia, Antonella 139, 441
García, Carlos 279
García, José M. 347
Gasilov, Vladimir 475
Geimer, Markus 645
Geraghty, Dermot 793
Gerndt, Michael 113
Gervaso, Alessandro 355
Girou, Denis 687
González, Alberto 263
Gopal, Srinivasa M. 527
Gopalkrishnan, Pradeep 573
Gordillo, Antonio 553
Granger, Richard 767
Grauer, Rainer 467
Gropp, William D. 583
Guim, Francesc 501
Gunney, Brian T.N. 697

H

Ha, Phuong Hoai 71
Hager, William W. 29
Hanigk, Sebastian 175
Hatzky, Roman 713
Hermanns, Marc-André 583
Hermanns, Marc-André 425, 687
Hernandez, Oscar 573, 661
Himmler, Valentin 653
Hofmann, Michael 405
Homann, Holger 467
Honecker, Andreas 271
Huang, Lei 3
Huck, Kevin A. 629
Huckle, Thomas 175

J

Janda, Rick 373
Janoschek, Florian 45
Jedlicka, Ed 583
Jenko, Frank 713
Jin, Haoqiang 661
Jordan, Kirk E. 583
Jurenz, Matthias 637

K

Karl, Wolfgang 757
Kartasheva, Elena 475
Kaufmann, Paul 749
Keller, Rainer 517
Kerbyson, Darren J. 89
Kessler, Christoph W. 227
Kilpatrick, Peter 235
Klenin, Konstantin V. 527
Knüpfer, Andreas 637
Knaepen, Bernard 483
Knafla, Bjoern 219
Koch, Erik 601
Koniges, Alice E. 697
Krammer, Bettina 619, 653
Krasnov, Dmitry 483
Krieg, Stefan 543
Krieger, Elmar 729
Krishnan, Manoj 339
Krusche, Peter 193
Kufrin, Rick 573
Kuhlen, Torsten 79
Kuhlmann, Björn 645
Kunis, Raphael 321

L

Löwe, Welf 227
Lübbers, Enno 749
Labarta, Jesús 129, 501

Laguerre, Michel 729
Lanotte, Alessandra S. 721
Lavery, Richard 729
Le Bris, Claude 29
Lecomber, David 653
Lederer, Hermann 687, 689, 713
Lee, Gregory L. 621
Lee, Kyu H. 527
Leger, Laurent 729
Leopold, Claudia 203, 211, 219
Lieber, Matthias 637
Lippert, Thomas 61, 381
Loogen, Rita 121
Luo, Yuan Lung 601

M

Müller, Matthias S. 373, 619, 637
Maiorano, Andrea 553
Malony, Allen D. 629
Mantovani, Filippo 553
Marin, Mauricio 389
Marinari, Enzo 553
Marongiu, Alessandro 785
Martín-Mayor, Victor 553
Martínez, Diego R. 99
Martínez-Zaldívar, Francisco-Jose 263
Martín, Alberto F. 287
Maruyama, Tsutomu 459
Masters, Nathan D. 697
Mayo, Rafael 255
McElroy, Ciarán 793
McGettrick, Séamas 793
Merelli, Ivan 139, 147
Milanesi, Luciano 139, 147
Miller, Barton P. 621
Miller, Robert 609
Mintzer, Fred 583
Mix, Hartmut 637
Moore, Shirley 619, 677
Morris, Alan 629
Muñoz-Sudupe, Antonio 553

N

Nagel, Wolfgang E. 365, 373, 637
Nageswaran, Jayram M. 767
Naumann, Uwe 311
Navarro, Denis 553
Neuenhahn, Martin 777
Nicolai, Mike 19
Nicolau, Alex 767
Nieplocha, Jarek 339
Noll, Tobias G. 777
Nowak, Fabian 757
Numrich, Robert W. 107

O

Oh, Jung S. 527
Olcoz, Katzalin 509
Olkhovskaya, Olga 475
Orro, Alessandro 147
Orth, Boris 583

P

Pütz, Matthias 585
Pérez-Gaviro, Sergio 553
Paar, Christof 741
Palazzari, Paolo 785
Palmer, Bruce 339
Pelzl, Jan 741
Pena, Tomás F. 99
Petrini, Fabrizio 339
Pfeiffer, Gerd 741
Pflüger, Stefan 365
Piera, Javier Roca 165
Platzner, Marco 749
Polzella, Francesco 355
Prasanna, Viktor K. 185, 561
Prieto, Manuel 279
Pringle, Gavin J. 687, 705
Probst, Markus 19
Pulatova, Farzona 645

Q

Quintana-Ortí, Enrique S. 255, 287
Quintana-Ortí, Gregorio 255

R

Rasch, Arno 303
Rath, Volker 451
Resch, Michael 517
Reshetnyak, Maxim 491
Richter, Marcus 61
Rodero, Ivan 501
Rossi, Mauro 553

Ruiz-Lorenzo, Juan Jesus 553
Rünger, Gudula 321, 405

S

Sérot, Jocelyn 243
Salama, Rafik A. 413
Sameh, Ahmed 413
Sansom, Mark S. P. 729
Sawai, Ryo 459
Schöne, Robert 365
Schüle, Josef 271
Schifano, Sebastiano Fabio 553
Schimmler, Manfred 741
Schirski, Marc 79
Schleiffer, Christian 741
Schug, Alexander 527
Schulz, Martin 621
Schumacher, Jörg 585
Schumacher, Tobias 749
Schwarz, Christian 467
Sciretti, Daniele 553
Seidel, Jan 425
Sevilla, Diego 347
Shah, N. Jon 155
Shende, Sameer 629
Siso-Nadal, Fernando 609
Sloan, Terence M. 705
Spinatelli, Gianmarco 355
Stöcker, Tony 155
Stüben, Hinnerk 535
Steffen, Bernhard 491
Stratford, Kevin 593
Streuer, Thomas 535
Strohhäcker, Sebastian 113
Stødle, Daniel 71
Suess, Michael 203, 211
Sutmann, Godehard 45
Sørevik, Tor 433

T

Tafti, Danesh 573
Taib, Nada 729
Tarancón, Alfonso 553
Terboven, Christian 669
Tipparaju, Vinod 339
Tirado, Francisco 279, 509
Tiskin, Alexander 193
Tisma, Reinhard 713
Torquati, Massimo 355
Toschi, Federico 721
Trautmann, Sven 329
Trenkler, Bernd 373
Trew, Arthur S. 705
Trieu, Binh 61
Tripiccione, Raffaele 553

V

Vahedipour, Kaveh 155
Vanneschi, Marco 355
Varnik, Ebadollah 311
Veidenbaum, Alex 767
Velasco, Jose Luis 553
Velasco, Jose M. 509
Verma, Abhinav 527
Vidal-Maciá, Antonio. M. 263
Viré, Axelle 483
Viti, Federica 139
von Livonius, Jörg 777

W

Wenzel, Wolfgang 527
Wolf, Andreas 451
Wolf, Felix 619, 645
Wolter, Marc 79
Wylie, Brian J. N. 645

X

Xia, Yinglong 185
Xu, Wei 295

Y

Yamaguchi, Yoshiki 459
Yasunaga, Moritoshi 459

Z

Zhuo, Ling 561
Zuccato, Pierfrancesco 355

Already published:

Modern Methods and Algorithms of Quantum Chemistry - Proceedings
Johannes Grotendorst (Editor)
Winter School, 21 - 25 February 2000, Forschungszentrum Jülich
NIC Series Volume 1
ISBN 3-00-005618-1, February 2000, 562 pages
out of print

Modern Methods and Algorithms of Quantum Chemistry - Poster Presentations
Johannes Grotendorst (Editor)
Winter School, 21 - 25 February 2000, Forschungszentrum Jülich
NIC Series Volume 2
ISBN 3-00-005746-3, February 2000, 77 pages
out of print

Modern Methods and Algorithms of Quantum Chemistry - Proceedings, Second Edition
Johannes Grotendorst (Editor)
Winter School, 21 - 25 February 2000, Forschungszentrum Jülich
NIC Series Volume 3
ISBN 3-00-005834-6, December 2000, 638 pages
out of print

Nichtlineare Analyse raum-zeitlicher Aspekte der hirnelektrischen Aktivität von Epilepsiepatienten
Jochen Arnold
NIC Series Volume 4
ISBN 3-00-006221-1, September 2000, 120 pages

Elektron-Elektron-Wechselwirkung in Halbleitern: Von hochkorrelierten kohärenten Anfangszuständen zu inkohärentem Transport
Reinhold Lövenich
NIC Series Volume 5
ISBN 3-00-006329-3, August 2000, 146 pages

Erkennung von Nichtlinearitäten und wechselseitigen Abhängigkeiten in Zeitreihen
Andreas Schmitz
NIC Series Volume 6
ISBN 3-00-007871-1, May 2001, 142 pages

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Kei Davis, Yannis Smaragdakis, Jörg Striegnitz (Editors)
Workshop MPOOL, 18 May 2001, Budapest
NIC Series Volume 7
ISBN 3-00-007968-8, June 2001, 160 pages

Europhysics Conference on Computational Physics - Book of Abstracts
Friedel Hossfeld, Kurt Binder (Editors)
Conference, 5 - 8 September 2001, Aachen
NIC Series Volume 8
ISBN 3-00-008236-0, September 2001, 500 pages

NIC Symposium 2001 - Proceedings
Horst Rollnik, Dietrich Wolf (Editors)
Symposium, 5 - 6 December 2001, Forschungszentrum Jülich
NIC Series Volume 9
ISBN 3-00-009055-X, May 2002, 514 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms - Lecture Notes
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
NIC Series Volume 10
ISBN 3-00-009057-6, February 2002, 548 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms- Poster Presentations
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
NIC Series Volume 11
ISBN 3-00-009058-4, February 2002, 194 pages

Strongly Disordered Quantum Spin Systems in Low Dimensions: Numerical Study of Spin Chains, Spin Ladders and Two-Dimensional Systems
Yu-cheng Lin
NIC Series Volume 12
ISBN 3-00-009056-8, May 2002, 146 pages

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Jörg Striegnitz, Kei Davis, Yannis Smaragdakis (Editors)
Workshop MPOOL 2002, 11 June 2002, Malaga
NIC Series Volume 13
ISBN 3-00-009099-1, June 2002, 132 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms - Audio-Visual Lecture Notes
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
NIC Series Volume 14
ISBN 3-00-010000-8, November 2002, DVD

Numerical Methods for Limit and Shakedown Analysis
Manfred Staat, Michael Heitzer (Eds.)
NIC Series Volume 15
ISBN 3-00-010001-6, February 2003, 306 pages

Design and Evaluation of a Bandwidth Broker that Provides Network Quality of Service for Grid Applications
Volker Sander
NIC Series Volume 16
ISBN 3-00-010002-4, February 2003, 208 pages

Automatic Performance Analysis on Parallel Computers with SMP Nodes
Felix Wolf
NIC Series Volume 17
ISBN 3-00-010003-2, February 2003, 168 pages

Haptisches Rendern zum Einpassen von hochaufgelösten Molekülstrukturdaten in niedrigaufgelöste Elektronenmikroskopie-Dichteverteilungen
Stefan Birmanns
NIC Series Volume 18
ISBN 3-00-010004-0, September 2003, 178 pages

Auswirkungen der Virtualisierung auf den IT-Betrieb
Wolfgang Gürich (Editor)
GI Conference, 4 - 5 November 2003, Forschungszentrum Jülich
NIC Series Volume 19
ISBN 3-00-009100-9, October 2003, 126 pages

NIC Symposium 2004
Dietrich Wolf, Gernot Münster, Manfred Kremer (Editors)
Symposium, 17 - 18 February 2004, Forschungszentrum Jülich
NIC Series Volume 20
ISBN 3-00-012372-5, February 2004, 482 pages

Measuring Synchronization in Model Systems and Electroencephalographic Time Series from Epilepsy Patients
Thomas Kreutz
NIC Series Volume 21
ISBN 3-00-012373-3, February 2004, 138 pages

Computational Soft Matter: From Synthetic Polymers to Proteins - Poster Abstracts
Norbert Attig, Kurt Binder, Helmut Grubmüller, Kurt Kremer (Editors)
Winter School, 29 February - 6 March 2004, Gustav-Stresemann-Institut Bonn
NIC Series Volume 22
ISBN 3-00-012374-1, February 2004, 120 pages

Computational Soft Matter: From Synthetic Polymers to Proteins - Lecture Notes
Norbert Attig, Kurt Binder, Helmut Grubmüller, Kurt Kremer (Editors)
Winter School, 29 February - 6 March 2004, Gustav-Stresemann-Institut Bonn
NIC Series Volume 23
ISBN 3-00-012641-4, February 2004, 440 pages

Synchronization and Interdependence Measures and their Applications to the Electroencephalogram of Epilepsy Patients and Clustering of Data
Alexander Kraskov
NIC Series Volume 24
ISBN 3-00-013619-3, May 2004, 106 pages

High Performance Computing in Chemistry
Johannes Grotendorst (Editor)
Report of the Joint Research Project:
High Performance Computing in Chemistry - HPC-Chem
NIC Series Volume 25
ISBN 3-00-013618-5, December 2004, 160 pages

Zerlegung von Signalen in unabhängige Komponenten: Ein informationstheoretischer Zugang
Harald Stögbauer
NIC Series Volume 26
ISBN 3-00-013620-7, April 2005, 110 pages

Multiparadigm Programming 2003
Joint Proceedings of the
3rd International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'03)
and the
1st International Workshop on Declarative Programming in the Context of Object-Oriented Languages (PD-COOL'03)
Jörg Striegnitz, Kei Davis (Editors)
NIC Series Volume 27
ISBN 3-00-016005-1, July 2005, 300 pages

Integration von Programmiersprachen durch strukturelle Typanalyse und partielle Auswertung
Jörg Striegnitz
NIC Series Volume 28
ISBN 3-00-016006-X, May 2005, 306 pages

OpenMolGRID - Open Computing Grid for Molecular Science and Engineering
Final Report
Mathilde Romberg (Editor)
NIC Series Volume 29
ISBN 3-00-016007-8, July 2005, 86 pages

GALA Grünenthal Applied Life Science Analysis
Achim Kless and Johannes Grotendorst (Editors)
NIC Series Volume 30
ISBN 3-00-017349-8, November 2006, 204 pages

Computational Nanoscience: Do It Yourself!
Lecture Notes
Johannes Grotendorst, Stefan Blügel, Dominik Marx (Editors)
Winter School, 14. - 22 February 2006, Forschungszentrum Jülich
NIC Series Volume 31
ISBN 3-00-017350-1, February 2006, 528 pages

NIC Symposium 2006 - Proceedings
G. Münster, D. Wolf, M. Kremer (Editors)
Symposium, 1 - 2 March 2006, Forschungszentrum Jülich
NIC Series Volume 32
ISBN 3-00-017351-X, February 2006, 384 pages

Parallel Computing: Current & Future Issues of High-End Computing
Proceedings of the International Conference ParCo 2005
G.R. Joubert, W.E. Nagel, F.J. Peters,
O. Plata, P. Tirado, E. Zapata (Editors)
NIC Series Volume 33
ISBN 3-00-017352-8, October 2006, 930 pages

From Computational Biophysics to Systems Biology 2006 Proceedings
U.H.E. Hansmann, J. Meinke, S. Mohanty, O. Zimmermann (Editors)
NIC Series Volume 34
ISBN-10 3-9810843-0-6, ISBN-13 978-3-9810843-0-6,
September 2006, 224 pages

Dreistufig parallele Software zur Parameteroptimierung von Support-Vektor-Maschinen mit kostensensitiven Gütemaßen
Tatjana Eitrich
NIC Series Volume 35
ISBN 978-3-9810843-1-3, March 2007, 262 pages

From Computational Biophysics to Systems Biology (CBSB07) Proceedings
U.H.E. Hansmann, J. Meinke, S. Mohanty, O. Zimmermann (Editors)
NIC Series Volume 36
ISBN 978-3-9810843-2-0, August 2007, 330 pages

Parallel Computing: Architectures, Algorithms and Applications - Book of Abstracts
Book of Abstracts, ParCo 2007 Conference, 4. - 7. September 2007
G.R. Joubert, C. Bischof, F. Peters, T. Lippert, M. Bücker, P. Gibbon, B. Mohr (Eds.)
NIC Series Volume 37
ISBN 978-3-9810843-3-7, August 2007, 216 pages

All volumes are available online at

http:// www.fz-juelich.de/nic-series/.