John von Neumann Institute for Computing




# The IBM eServer pSeries 690 as a Research Instrument for Computer Scientists

Guido Juckeland, Michael Kluge,
Ralph Müller-Pfefferkorn, Wolfgang E. Nagel,
and Bernd Trenkler

published in

*NIC Symposium 2006* ,
G. Münster, D. Wolf, M. Kremer (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **32**, ISBN 3-00-017351-X, pp. 315-322, 2006.

© 2006 by John von Neumann Institute for Computing
Permission to make digital or hard copies of portions of this work for personal or classroom use is granted provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise requires prior specific permission by the publisher mentioned above.

http://www.fz-juelich.de/nic-series/volume32

# The IBM eServer pSeries 690 as a Research Instrument for Computer Scientists

**Guido Juckeland, Michael Kluge, Ralph Müller-Pfefferkorn, Wolfgang E. Nagel, and Bernd Trenkler**

Technische Universität Dresden
Center for Information Services and High Performance Computing
01062 Dresden, Germany
*E-mail: {Guido.Juckeland, Michael.Kluge, Ralph.Mueller-Pfefferkorn, Wolfgang.Nagel, Bernd.Trenkler}@tu-dresden.de*

## 1 Introduction

Currently, there is quite a number of different system architectures of HPC systems available on the market. Often, details of the complex system - from the hardware to the operating system - determine the performance of an application on a specific architecture. Examples for such small but important details are the cache hierarchy or the operating system's scheduling algorithms.

We tested and analyzed some features of the IBM p690 system at the Forschungszentrum Jülich to help users and administrators in the analysis of their applications and machine behavior, thus to optimize performance and system behavior (sections 2, 3 and 4).

Furthermore, the programming paradigms applied in parallel applications introduce an overhead and can be a potential source of performance loss. MPI, as a widely used standard, needs strict rules to be adopted by the developer, for example in the communication between the parallel processes. Assisting the programmer in the process of MPI problem detection can thus be of invaluable help (section 5).

## 2 Examination of the Scheduling Properties on the IBM p690 with the PARbench Environment

Benchmarking in the field of HPC is mostly realized with special programs which run separately on the system. However, utilization of expensive hardware quite often requires running multiple programs on the machine simultaneously in the multiprogramming mode. Competition for resources, runtime conflicts and sometimes even scheduling problems are the consequences. The goal of our study has been to measure the behavior of the machine when workloads compete.

### 2.1 PARbench

The PARbench Benchmark System was developed at Forschungszentrum Jülich in the early 90's. Over the last years, it was enhanced and ported to many parallel machines by our research group at Technische Universität Dresden. PARbench enables the simulation of virtually every workload the user might have in mind and specifies. It is able to

execute many benchmark programs in parallel and record their behavior with regards to time flow and several other parameters. OpenMP is used as the concept for parallelization to support parallel jobs within a chosen benchmark workload.

### 2.2 A Selection of Tests

The IBM p690 system used for these tests was running AIX 5L Version 5.2 as the operating system. The system consists of SMP nodes with 32 processors as the building block for the whole cluster. It uses a thread based scheduling system with priority queues (256 stages). The scheduling algorithm is a fair round robin algorithm with dynamic priorities where each processor has its own queue. There also exists one global queue for all processors. However, this queue must be explicitly activated as it overwrites the system of local queues.

As part of our activities, we have run many different workloads to investigate several aspects of the machine. Here, we will concentrate on one test where we have looked at the situation of parallel jobs running in a multiprogramming environment. Further results can be found in[10,12].

### 2.3 Results

The test scenario mentioned above is a situation where 32 sequential jobs (CPU time: each job is running for 100 seconds) are concurrently generated completely filling one node with 32 CPUs for 100 seconds. This is an ideal situation where the utilization is about 100%. This workload is kept constant now but every second group of four jobs is executed as a parallel job using OpenMP as the parallelization paradigm (each job uses 4 threads). The result is indicated in figure 1.

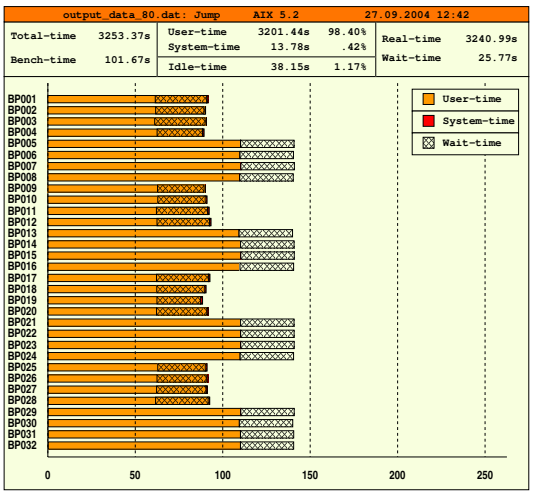



Figure 1. PARbench experiment with 16 jobs with 4 threads each and 16 serial jobs running on 32 processors using the global queue.

It clearly shows that the parallel jobs finished earlier (after about 60 seconds, left end of the overlapped pattern) – which means that in the average they have got more than one CPU allocated over time. The sequential jobs ended after about 140 seconds (right end of the bar). So far, these results are not completely surprising and are acceptable. The work done by each job, however, has been kept constant. On the other hand, it clearly can be seen that the parallel jobs were accounted for only about 85 seconds (right end of the bar for the parallel jobs) – which suggests that they have not used the full CPU time they have used before. At the same time, the sequential jobs were accounted for more than 115 CPU seconds. In sum, the total accounted CPU time stays constant at 3200 CPU seconds (32 times 100 seconds). These different user times for sequential and parallel jobs indicate shortcomings in the accounting system where CPU time used by the parallel jobs is accounted for the sequential ones. This discovery has been reported to IBM and is still under examination.

## 3 BenchIT

Performance analysis of computer systems is an interesting but quite challenging task. A first approach is given by standard benchmarks and their results available on-line for a wide variety of computer systems (e.g. LINPACK[14] or SPEC[15]). Own measurements normally require some detailed knowledge of the system architecture and most of the other machine components. Nevertheless, there are plenty of options for getting measured performance results which are inconsistent, unreliable, and sometimes even incorrect. However, such results are sometimes used to choose the system architecture in the next procurement.

With BenchIT we want to improve the measurement and the comparison of archived performance data. BenchIT offers a uniform and flexible architecture for the measurement and presentation of such data[13,16]. BenchIT consists of two parts for the measurement and the presentation of performance data.

The BenchIT main kernel driver initiates and controls the performance measurement. It repeatedly calls the measurement kernel which implements a measurement algorithm with varying problem sizes (e.g. vector sizes or matrix dimensions). When the processing of the kernel is done with all problem sizes (or a time limit is reached) the data is analyzed, outliers are corrected, and all information is written into a result file.

The results of a BenchIT measurement run are written into a plain ASCII file. It is clear that the result file has also to contain information about the measurement environment as well as the system architecture, since only the result file is uploaded to the BenchIT web server. Only with this additional information the measurement becomes comparable.

The BenchIT web server ( `http://www.benchit.org`) is the key element in the data analysis process. It offers sharing files with different user groups, therefore, it enables the user to compare his results with the ones of colleagues or any other BenchIT user.

The assembly of plots occurs in steps where all available data is filtered in order to contain just the results the user wishes to see. The data is presented using gnuplot – parts of the website are therefore a mere front end to make all gnuplot options available. Plots are shown online or are exported in png, eps or emf format for including them in presentations as well as in articles. Furthermore, plots can be stored, easily accessed, and postprocessed.

One of the main design goals in the development of BenchIT is portability between different platforms. Real portability problems arise in considering the main kernel driver

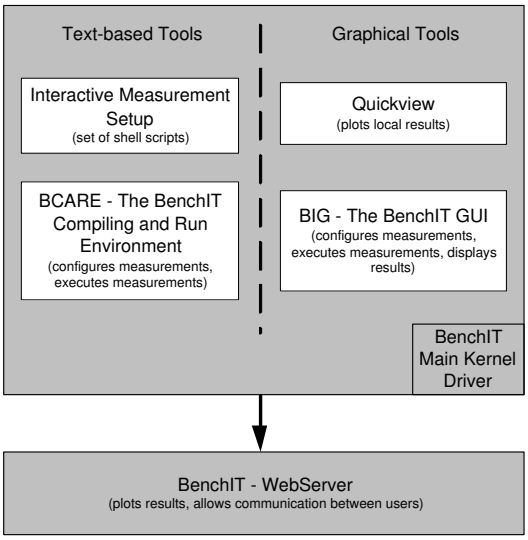



Figure 2. BenchIT components

since the measurements are to run on a large variety of platforms and operating systems. The greatest common denominator among all those systems seems to be a shell, a compiler, and some degree of POSIX-compatibility. Therefore, the whole main kernel driver is steered by a set of shell scripts invoking the system compiler(s) for each measurement run and kernel. Some of the results obtained on the IBM p690 can be found on our website `http://www.benchit.org`.

## 4 EP-Cache: Optimizing Cache Access – Compiler Tests and Source-To-Source Transformations

Usually, a developer focuses on implementing a correct program which solves a problem by using an algorithm. Frequently, applications which do not take the cache hierarchy of modern microprocessors into account achieve only a small fraction of the theoretical peak speed. Fine-tuning a program for better cache utilization has become an expensive and time consuming part of the development cycle. One way to optimize the cache usage of applications are source-to-source transformations of loops. There are a number of known transformations that improve data locality by reusing the data in the cache, such as loop interchange, blocking and unrolling.

Modern compilers claim to use loop transformations in code optimization. In the EP-Cache project (funded by the BMBF contract number 01IRB04) we have tested three FORTRAN90 compilers (IBM xlf for AIX V8.1.1[1], Intel ifc 7.1[2] and SGI MIPSpro 7.3[3]) for loop transformations. In addition, the same source code was optimized manually.

Our measurements (see figure 3 for two compilers) demonstrate that the capabilities of the tested FORTRAN compilers to optimize cache behavior vary. Only MIPSpro7 is able to automatically optimize sequential code in such a way that the resulting speedup is

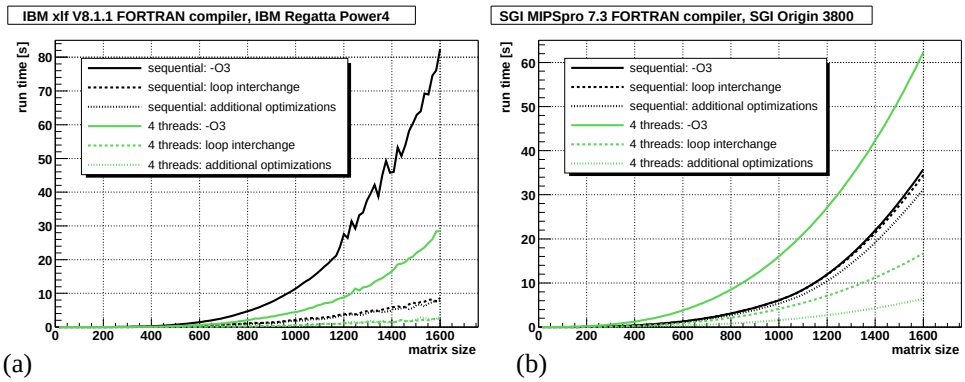



Figure 3. Runtime as function of the matrix dimension (a) on a IBM Regatta p690 system with IBM's xlf for AIX FORTRAN V8.1.1 compiler; the measurement curves of the two manually optimized parallel codes are on top of each other and (b) the SGI Origin 3800 with the MIPSpro 7.3 FORTRAN compiler

comparable with a manual optimization. In the case of parallel OpenMP processing none of the compilers can improve the original source code.

Currently, the only way to deal with cache access problems in FORTRAN programs seems to be manual optimizations, like loop transformations[5]. However, there are three drawbacks in a manual optimization: it is time consuming, error-prone, and can become quite complicated.

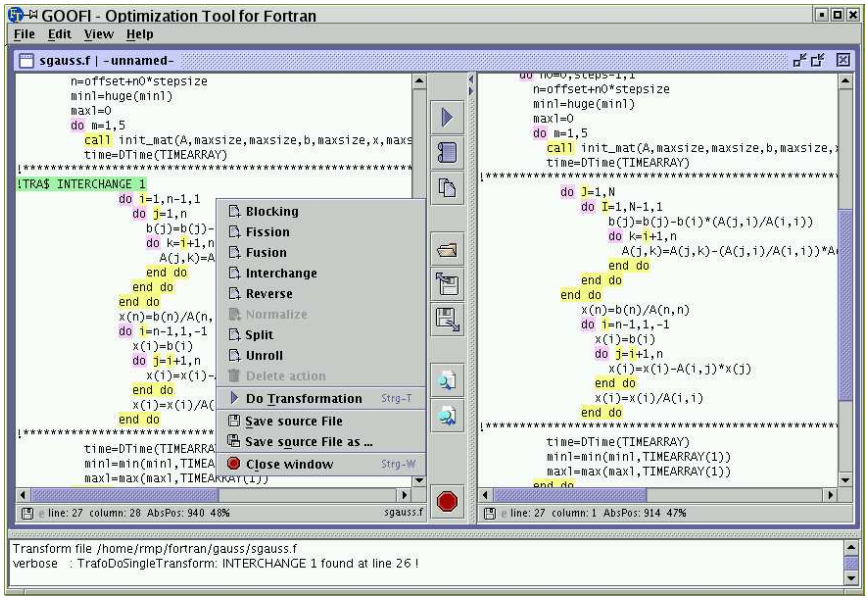



Figure 4. Screenshot of GOOFI with original and transformed source files and the transformation selection window

Therefore, we have developed a tool to assist developers in optimizing their FORTRAN applications: loop transformations are performed automatically on user request. GOOFI (Graphical Optimization Of Fortran Implementations) provides a graphical user interface (figure 4) where the user loads his/her source code (left side of the window) and requests transformations for a loop (by a mouse click). By another mouse click, he/she receives the transformed code, which appears up in the right window of the split screen, making direct visual comparison easily possible. The entire results of these studies and developments were published at EuroPar 2004[4].

## 5 Automatic Scalability Analysis for MPI Programs

Identifying performance problems can be a time consuming and difficult task especially for parallel applications.

### 5.1 Automatic MPI Overhead Detection

Within the MPI Standard, most of the communication between the processes running in parallel is performed by simply exchanging messages between these processes. To understand what is going on during an execution of an MPI program, many tools have been developed. Typically, these tools keep track of the messages within a system and are able to show a timeline of the program activities as well as message statistics and other useful data after the program has terminated. There are many current research activities trying to analyze the behavior of an application automatically.

One of the activities at the Technische Universität Dresden within this research area during 2004 was to automatically find the lines within the source code of an MPI program causing unnecessary waiting time as well as scalability problems[6–8]. For achieving this goal, it was necessary to define a 'normal behavior' for a call to an MPI function. If a communication function is called multiple times under the same conditions it is most likely that the execution time for this MPI function call varies, even under ideal conditions on a dedicated system (see figure 5). To be able to distinguish between the normal variations that will happen everytime from those variations that are caused by a bad parallelization scheme within the users application, those normal variations have to be defined. Once this is accomplished, each call to an MPI function can be inspected, and thereupon the execution time can be named within or beyond the normal variations.

Our approach is based on the assumption that the variations observed by calling the same function multiple times under ideal conditions are statistically distributed. By taking a quantile above $0.9$, the value for a maximum time for a call to an MPI function is found.

At this point we are able to detect unnecessary waiting time within an MPI application. By mapping those waiting times back to the source code level, the user is given helpful information about his program.

### 5.2 MPI Scalability Analysis

The second part of this project was dedicated to the automatic detection of scalability problems. An application that has the same input data but runs on two processors instead on one processor is expected to finish within about half the runtime. However, the amount

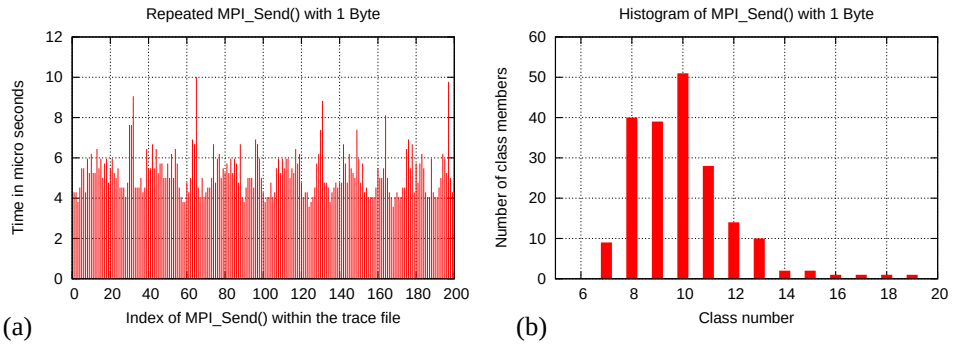



Figure 5. Example of normal variations of a repeated MPI_Send() with a small message size (a) and the histogram by using a class width of $500ns$. Depending on the (user selectable) quantile the method will automatically select a value around 9 Milliseconds as the acceptable maximum for a call to this MPI function call with this attributes.

of work within an application that actually can be parallelized limits the scalability of an application. How an application performs during a parallel program run is also determined by the amount of communication and synchronization between the processes. How the amount of communication changes with increasing number of processors depends on the algorithm used. If the communication increases faster than linear it will result in a scalability problem. This working thesis is used to map the amount of communication back to the line number within the source code of a program where the appropriate MPI function has been called. By fitting a quadratic polynomial to these numbers (one number for each program run) at the source code level it can be expected that the factor for the quadratic term in the polynomial is close to zero. By identifying the MPI calls in the source code where this is not true, a possible scalability issue has been detected.

An architecture for a tool implementing the ideas mentioned above has been proposed and implemented in a prototype. By applying the tool to some ASCI-Benchmarks it was possible to find possible scalability problems and unnecessary waiting time automatically.

## Acknowledgments

We want to thank the German Federal Ministry of Education and Research (BMBF) for the funding of the project EP-Cache (contract 01IRB04).

We thank the Forschungszentrum Jülich (John von Neumann Institute for Computing) for giving us the possibility to use their computing facilities (e.g. IBM Regatta p690 ”Jump”) for our research and development. The fact that the systems is built from 32 CPU SMP Clusters makes it easy to get reproducible results when doing performance analysis. By allocation of one of the clusters in the system one user gets 32 processors exclusively. So the noise usually generated from a multi user mode is practically inexistent. This makes the machine a valuable and easy-to-use research instrument for a computer scientist.

## References


1. XL Fortran for AIX V8.1.1, IBM (2003).
2. Intel Fortran Compiler for Linux Systems, Intel Corporation (2003).
3. MIPSpro Fortran 90, Silicon Graphics Inc. (2003).
4. R. Müller-Pfefferkorn, W.E. Nagel and B. Trenkler; *Optimizing Cache Access: A Tool for Source-To-Source Transformations and Real-Life Compiler Tests*, Euro-Par 2004 - Parallel Processing , Springer, LNCS 3149, 72–81 (2004).
5. J. Blum; Transit: Ein interaktives Werkzeug zur Programmoptimierung mittels Code-Transformationen, FZ Jülich, Technical report No. Jül-3302, November 1996.
6. Michael Kluge; Statistische Analyse von Programmspuren für MPI-Programme. Diploma thesis, December 2004.
7. Michael Kluge, Andreas Knüpfer and Wolfgang E. Nagel; Statistical Methods for Automatic Performance Bottleneck Detection in MPI Based Programs In *Computational Science - ICCS 2005, Volume I*, pages 3307–337, 2005.
8. Michael Kluge, Andreas Knüpfer and Wolfgang E. Nagel; Knowledge Based Automatic Scalability Analysis and Extrapolation for MPI Programs In *11th International Euro-Par Conference 2005*, 176–184, 2005.
9. Sebastian Boesler; Performance-Analyse von Hochleistungsrechnern im Multiprogramming-Betrieb: Untersuchungen auf der SGI Origin. Diploma thesis, Technische Universität Dresden, December 2001.
10. Heiko Dietze; Das PARbench-System: Untersuchungen zum Scheduling von parallelen Programmen auf der IBM p690. Diploma-Thesis, Technische Universität Dresden, November 2004.
11. Wolfgang E. Nagel and Markus A. Linn; Benchmarking parallel programs in a multiprogramming environment: The PARbench system, 1991.
12. H. Dietze, W.E. Nagel and B. Trenkler; Scheduling issues on IBM p690: Performance Analysis with the PARbench Environment. accepted for publication in *Proceedings of Parallel Computing ParCo 2005*, Malaga, Spain.
13. G. Juckeland, S. Börner, M. Kluge, S. Kölling, W. E. Nagel, S. Pflüger, H. Röding, S. Seidl, T. William, and R. Wloch; BenchIT – Performance Measurement and Comparison for Scientific Applications. In *Proceedings of the ParCo 2003*, Dresden, Germany, ISBN 0-444-51689-1.
14. LINPACK. `http://www.netlib.org/benchmark/`.
15. The SPEC Benchmarks. `http://www.spec.org`.
16. G. Juckeland, M. Kluge, W.E. Nagel and S. Pflüger; *Performance Analysis with BenchIT: Portable, Flexible, Easy to Use*, In Proc. of QEST 2004, September 27 - 30, 2004, Enschede, The Netherlands, IEEE Computer Society Order Number P2185, ISBN 0-7695-2185-1.