The amount of parallelism in modern supercomputers currently grows from generation to generation. Further application performance improvements therefore depend on software-managed parallelism: the software must organize data exchange between processing elements efficiently and optimally distribute the workload between them. Performance analysis tools help developers of parallel applications to evaluate and optimize the parallel efficiency of their programs. This dissertation presents two novel methods to automatically detect imbalance-related performance problems in MPI programs and intuitively guide the performance analyst to inefficiencies whose optimization promise the highest benefit.

The first method, the delay analysis, identifies the root causes of wait states. A delay occurs when a program activity needs more time on one process than on another, which leads to the formation of wait states at a subsequent synchronization point. Wait states are the primary symptom of load imbalance in parallel programs. While wait states themselves are easy to detect, the potentially large temporal and spatial distance between wait states and the delays causing them complicates the identification of wait-state root causes.

The delay analysis closes this gap, accounting for both short-term and long-term effects. The second method is based on the detection of the critical path, which determines the effect of imbalance on program runtime. The critical path is the longest execution path in a parallel program without wait states: optimizing an activity on the critical path will reduce the program's runtime. Comparing the duration of activities on the critical path with their duration on each process yields a set of novel, compact performance indicators. These indicators allow users to evaluate load balance, identify performance bottlenecks, and determine the performance impact of load imbalance at first glance by providing an intuitive understanding of complex performance phenomena.

Both analysis methods leverage the scalable event-trace analysis technique employed by the Scalasca toolset: by replaying event traces in parallel, the bottleneck search algorithms can harness the distributed memory and computational resources of the target system for the analysis, allowing them to process even large-scale program runs.

This publication was written at the Jülich Supercomputing Centre (JSC) which is an integral part of the Institute for Advanced Simulation (IAS). The IAS combines the Jülich simulation sciences and the supercomputer facility in one organizational unit. It includes those parts of the scientific institutes at Forschungszentrum Jülich which use simulation on supercomputers as their main research methodology.

IAS Series
Volume 23
ISBN 978-3-89336-940-9

JÜLICH
FORSCHUNGSZENTRUM

JÜLICH
FORSCHUNGSZENTRUM

Characterizing Load and Communication Imbalance

David Böhme

IAS Series

23

Member of the Helmholtz Association

Characterizing Load and Communication Imbalance in Parallel Applications

David Böhme

JÜLICH
FORSCHUNGSZENTRUM

Schriften des Forschungszentrums Jülich

IAS Series Volume 23

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Characterizing Load and Communication Imbalance in Parallel Applications

David Böhme

Schriften des Forschungszentrums Jülich

IAS Series | Volume 23

ISSN 1868-8489 | ISBN 978-3-89336-940-9

Bibliographic information published by the Deutsche Nationalbibliothek.
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Publisher and Distributor:	Forschungszentrum Jülich GmbH Zentralbibliothek 52425 Jülich Phone +49 (0) 24 61 61-53 68 · Fax +49 (0) 24 61 61-61 03 e-mail: zb-publikation@fz-juelich.de Internet: http://www.fz-juelich.de/zb
Cover Design:	Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH
Printer:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2014

Schriften des Forschungszentrums Jülich
IAS Series Volume 23

D 82 (Diss., RWTH Aachen University, 2013)

ISSN 1868-8489
ISBN 978-3-89336-940-9

Persistent Identifier: urn:nbn:de:0001-2014012708
Resolving URL: http://www.persistent-identifier.de/?link=610

Abstract

The amount of parallelism in modern supercomputers currently grows from generation to generation, and is expected to reach orders of millions of processor cores in a single system in the near future. Further application performance improvements therefore depend to a large extend on software-managed parallelism: in particular, the software must organize data exchange between processing elements efficiently and optimally distribute the workload between them. Performance analysis tools help developers of parallel applications to evaluate and optimize the parallel efficiency of their programs by pinpointing specific performance bottlenecks. However, existing tools are often incapable of identifying complex imbalance patterns and determining their performance impact reliably. This dissertation presents two novel methods to automatically extract imbalance-related performance problems from event traces generated by MPI programs and intuitively guide the performance analyst to inefficiencies whose optimization promise the highest benefit.

The first method, the delay analysis, identifies the root causes of wait states. A delay occurs when a program activity needs more time on one process than on another, which leads to the formation of wait states at a subsequent synchronization point. Wait states, which are intervals through which a process is idle while waiting for the delayed process, are the primary symptom of load imbalance in parallel programs. While wait states themselves are easy to detect, the potentially large temporal and spatial distance between wait states and the delays causing them complicates the identification of wait-state root causes. The delay analysis closes this gap, accounting for both short-term and long-term effects. To this end, the delay analysis comprises two contributions of this dissertation: (1) a cost model and terminology to describe the severity of a delay in terms of the overall waiting time it causes; and (2) a scalable algorithm to identify the locations of delays and determine their cost.

The second new analysis method is based on the detection of the critical path. In contrast to the delay analysis, which characterizes the formation of wait states, this critical-path analysis determines the effect of imbalance on program runtime. The critical path is the longest execution path in a parallel program without wait states: optimizing an activity on the critical path will reduce the programs run time. Comparing the duration of activities on the critical path with their duration on each process yields a set of novel, compact performance indicators. These indicators allow users to evaluate load balance, identify performance bottlenecks, and determine the performance impact of load imbalance at first glance by providing an intuitive understanding of complex performance phenomena. Unlike existing statistics-based load balance metrics, these indicators are applicable to both SPMD and MPMD-style programs.

Both analysis methods leverage the scalable event-trace analysis technique employed by the Scalasca toolset: by replaying event traces in parallel, the bottleneck search algorithms can harness the distributed memory and computational resources of the target system for the analysis, allowing them to process even large-scale program runs. The scalability and performance insight that the novel analysis approaches provide are demonstrated by evaluating a variety of real-world HPC codes in configurations with up to 262,144 processor cores.

Kurzzusammenfassung

Der Grad der Parallelverarbeitung in modernen Supercomputern wächst von Generation zu Generation, und wird in naher Zukunft Grössenordnungen von mehreren Millionen Prozessorkernen erreichen. Die Performanz der Anwendungen hängt dadurch immer stärker von der Fähigkeit der Software ab, diesen Parallelismus effizient zu steuern: insbesondere muss der Datenaustausch zwischen den Prozessen effizient organisiert und die Arbeitslast optimal auf die Prozessoren verteilt werden. Leistungsanalysewerkzeuge helfen den Anwendungsentwicklern dabei, die parallele Effizienz ihrer Anwendungen zu evaluieren und Engpässe aufzuspüren. Bisher verfügbare Werkzeuge sind allerdings in der Regel nicht in der Lage, komplexe Formen von Lastimbalancen zu identifizieren und ihre Auswirkungen auf die Leistung zuverlässig zu quantifizieren. Diese Dissertation stellt daher zwei neue Verfahren vor, um in Ereignisspuren von parallelen Programmen automatisch Lastverteilungs-Probleme zu identifizieren und so den Anwendungsentwickler zu den Schwachstellen mit dem größten Optimierungspotential zu leiten.

Das erste Verfahren, die Delay-Analyse, identifiziert die Ursachen von Wartezuständen. Ein Delay (Verzögerung) tritt auf, wenn eine Programmaktivität auf einem Prozess länger dauert als auf einem anderen und so an einem folgenden Synchronisationspunkt einen Wartezustand auslöst. Solche Wartezustände, bei denen Prozessorkapazität brach liegt, sind das Hauptsymptom von Ungleichgewichten in der Lastverteilung. Während Wartezustände an sich einfach zu erkennen sind, gestaltet sich die Identifikation der Ursachen aufgrund der potentiell grossen Distanz zwischen einem Wartezustand und dem verursachenden Delay oft schwierig. Die Delay-Analyse schließt diese Lücke. Dazu definiert die Dissertation erstens eine Terminologie und ein Kostenmodell zur Charakterisierung von Delays, und stellt zweitens einen skalierbaren Algorithmus zur Identifikation der Delays und der Berechnung ihrer Kosten vor.

Die zweite neue Analysemethode basiert auf der Extraktion des kritischen Pfades. Im Gegensatz zur Delay-Analyse, die die Entstehung von Wartezuständen beschreibt, lassen sich durch die Analyse des kritischen Pfades die Auswirkungen von Imbalancen auf die Programmlaufzeit charakterisieren. Der kritische Pfad ist der längste Ausführungspfad in einem parallelen Programm ohne Wartezustände: daher kann nur die Optimierung von Aktivitäten auf dem kritischen Pfad die Programmlaufzeit verkürzen. Darüber hinaus lassen sich durch den Vergleich der Dauer von Aktivitäten auf dem kritischen Pfad mit der durchschnittlichen Ausführungsdauer dieser Aktivitäten auf jedem Prozess kompakte Leistungsindikatoren ableiten, mit denen komplexe Leistungsprobleme intuitiv hervorgehoben werden können. Insbesondere können Lastverteilungsprobleme schnell erkannt und ihr Einfluss auf die Performanz quantifiziert werden. Anders als bisherige, statistik-basierte Ansätze lassen sich die Leistungsindikatoren sowohl für SPMD als auch für MPMD-Programme anwenden.

Beide Verfahren bauen auf der hochskalierbaren Ereignisspur-Analysetechnik des Scalasca-Toolkits auf: durch die parallele Verarbeitung der Ereignisspuren aller Prozesse können die Rechenresourcen und der verteilte Speicher der Zielplattform für die Suche nach Leistungsengpässen selbst herangezogen werden, wodurch die Analyse hochskalierender Programmläufe ermöglicht wird. Der Erkenntnisgewinn durch die neuen Analyseverfahren und ihre Skalierbarkeit werden anhand von Fallstudien mit einer Vielzahl realer HPC-Anwendungen in Konfigurationen mit bis zu 262.144 Prozessen untersucht.

Acknowledgment

I would like to thank Prof. Marek Behr, PhD, President of the German Research School for Simulation Sciences and Scientific Director of the Aachen Institute for Advanced Study in Computational Engineering Sciences (AICES), as well as Prof. Dr. Dr. Thomas Lippert, Director of the Jülich Supercomputing Centre, for the opportunity to carry out my PhD project in these excellent research environments.

This project would not have been possible without the support and guidance of my advisor Prof. Dr. Felix Wolf, to whom I owe deep gratitude. I also thank Prof. Dr. Horst Lichter and Prof. Dr. Wolfgang Nagel for serving as second referees. Furthermore, I am grateful to Dr. Bronis R. the Supinski and Dr. Martin Schulz for hosting me at the Lawrence Livermore National Laboratory and opening additional research directions for my thesis.

Thanks are due to my colleagues at the Jülich Supercomputing Centre and the German Research School for Simulation Sciences for their help and advice on countless occasions. I would particularly like to acknowledge Dr. Markus Geimer for being an invaluable source of technical wisdom, and Marc-André Hermanns for his keen eye for details. Finally, I would like to thank my friends – especially Michael, Peter, Pavel, Zoltan, and Alexandre, to mention just a few – without whom the past years would have been only half as much fun. Last but by no means least, I am deeply grateful to my parents for their continuous love, encouragement, and absolute support during the completion of this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Complementing observation and experimentation, simulation of physical phenomena has become the third pillar of scientific research. However, the never-ending urge to improve scope and accuracy of simulations through more detailed numerical models, higher resolutions, as well as multi-physics and multi-scale approaches leaves simulation scientists hungry for ever more computational power. Unsurprisingly, most of the world's largest supercomputers are dedicated to run advanced scientific simulation codes. Some time ago, simulation codes could benefit from continuously rising microprocessor clock speeds that would automatically improve execution speed with each new hardware generation, but with the shift towards multi-core architectures, improvements of sequential processor speed have come to an end. Hence, as Herb Sutter puts it, "the free lunch is over" [77]: further program speedup now depends to a larger extent on the software's capability to leverage increasing amounts of parallelism. Software performance tuning is therefore more important than ever, and developers rely on powerful performance-analysis tools to identify bottlenecks in their codes. For software that runs on massively-parallel machines, characterizing parallel performance aspects is particularly important.

Providing the basic terms and definitions for the discussion of the two innovative performance analysis methods that form the major contribution of this dissertation, the following sections briefly outline the foundations of parallel computers, of parallel programming models, and of the performance analysis of parallel programs.

1.1 Parallel Computers

No matter how fast a processor is, a group of them working together can probably solve a given problem faster: this relatively obvious idea lead to the development of parallel computers early on. Supercomputers have been using parallel architectures for decades, but due to their complex architecture and difficult programming, the general public preferred sequential machines for everyday tasks. However, since the mid-2000s, when traditional means of increasing single-processor speed hit physical limitations, processor manufacturers adopted parallelism on a broad scale to further increase processor throughput. While parallel computing concepts that were originally applied in supercomputers now found their way into commodity machines, the general shift to parallelism also affects supercomputer architects and programmers: the introduction of parallelism on multiple levels and the rapid increase of overall parallelism often require significant changes to existing programs. This section provides an overview of hardware architectures and programming models found in modern supercomputers.

1.1.1 Hardware Architectures

As stated before, supercomputers today use parallel architectures. We can classify parallel architectures based on the memory architecture as either *shared-memory* or *distributed-memory* architectures. In a shared-memory machine, all processors access the same, global memory, whereas in a distributed-memory system, processors have local, private memory and cannot directly access the memory of remote processors.

A widespread class of shared-memory architectures are symmetric multi-processor (SMP) systems, where multiple processor sockets are placed on a single board and connected to the memory via a shared memory bus. Today, SMP parallelism is also found in multi-core systems, where multiple, typically similar processing units (cores) are placed on a single processor die. Other than traditional SMP systems, the cores in a multi-core processor usually share one or two levels of the processor's cache hierarchy. Contemporary high-end SMP system often have more than one processor socket with multi-core processors in each of them. Since a shared memory bus can become a performance bottleneck, modern systems with multiple processor sockets often connect each processor to its own local memory banks, and realize accesses to non-local memory banks through high-speed serial links. Examples are the AMD Opteron processors with HyperTransport [38] links, or the QuickPath interconnect [41] from Intel. Hardware cache coherency protocols such as MESI ensure cache coherency between the processors. Since memory access times differ depending on wether the requested location is stored in a local or remote memory bank, these architectures are referred to as ccNUMA (cache-coherent non-uniform memory architectures). While a shared-memory architecture allows highly efficient data exchange and synchronization between processors, the limited memory bandwidth still constitutes a bottleneck. Therefore, even ccNUMA systems quickly reach a scalability limit.

In contrast, a distributed-memory system is composed of multiple nodes that are connected through a network, where each node has its own local memory and at least one processor. Processors can only directly access memory on their local node, and have to communicate over the network to exchange data with remote nodes. The network transfer makes data exchange between nodes an order of magnitude slower than accesses to local memory. However, the aggregate memory bandwidth in distributed-memory machines increases with system size, and systems can in principle scale up arbitrarily. Because of their better scaling properties, large-scale supercomputers are typically distributed-memory systems. Common types of distributed-memory architectures are massively-parallel multiprocessor systems (MPP) and clusters. Although there is no sharp distinction between these two types, the term "MPP" generally refers to large-scale systems with propietary network interconnects, often using scalable network topologies such as a torus layout; while systems built mainly from commodity, off-the-shelf components are designated as clusters. Examples of MPP systems include the Blue Gene series from IBM and the Cray XT/XE/XK series; clusters are offered from many server vendors and system integrators such as Dell, HP, or Bull. Clusters are typically built from standard server components, but they often use high-speed networks such as InfiniBand since efficient communication is an important performance factor.

Today, supercomputers virtually exclusively use multi-core CPUs on their nodes. To this end, today's large-scale systems are actually *hybrid* architectures with distributed-memory parallelism between nodes and shared-memory parallelism on the nodes. Another current trend are *heterogeneous* systems. Here, the standard CPUs on the compute nodes are accompanied by additional, throughput-oriented coprocessors (*accelerators*) that can execute vast amounts of (floating-point) operations in parallel and more efficiently than CPUs. A prominent example for a heterogeneous architecture was the Roadrunner supercomputer, which used the heterogeneous IBM Cell processor. Today, graphics hardware vendors such as NVidia succesfully promote the integration of GPUs (graphic-processing units) as accelerators into supercomputers: in the most recent Top 500 [1] list (June 2012), 52 systems use NVidia GPUs. Examples are the Chinese Tianhe-1A (rank 5), Jaguar at Oak Ridge National Laboratory (rank 6), and the Japanese Tsubame 2.0 (rank 14).

1.1.2 Programming Models

Because of the vastly different characteristics of shared-memory and distributed-memory architectures, programmers need to choose a parallelization strategy that best fits their target platform. First and foremost, this means picking an appropriate programming model: typically, some form of multi-threading for shared-memory machines, and message-passing for distributed-memory machines. For hybrid systems, both approaches are also often used complementary in a single program.

Message Passing

The most popular programming model for tightly coupled parallel programs on distributed-memory machines is *message passing*, where processes synchronize and exchange data by sending messages over a message channel. In 1991, the message-passing interface standard (MPI) effort was initiated to create a generic, standardized, and portable message-passing API with language bindings for C, C++ and Fortran. When the first version of MPI was released in 1994, it rapidly gained attraction and is now the de-facto standard for message-passing programs. The latest version, MPI 2.2 [54], was released in September 2009. At its core, MPI provides functions for point-to-point or collective data exchange and synchronization between processes. In addition, MPI also includes interfaces for parallel file I/O and one-sided communication.

The MPI model defines a flat space of processes that each have their own memory address space, i.e., a process can not access another process' memory directly. To exchange information, the processes participating in a parallel program share a common communication context (*communicator*), where each process is assigned a unique *rank* number. Processes can address other processes in the communicator by their rank number and use it to send or receive messages from remote ranks or to participate in collective communication. Thanks to its flexible and generic interface, virtually any parallelization scheme can be implemented efficiently with MPI.

Currently, MPI is the by far most popular choice for programming distributed-memory machines in the HPC area, receiving widespread support from industry, academia, and platform vendors. MPI is installed on virtually every supercomputer, and MPI programs run on any scale from a couple of dozen to hundreds of thousands of processes. A 2009 study from Balaji et al. [5] does identify some scalability bottlenecks in the MPI specification, but concludes that the core functionality will continue to scale to much higher levels. While parallel programs begin to adopt complementary parallel programming paradigms to exploit the increasing node-level parallelism, message passing with MPI remains the tool of choice for implementing data exchange between nodes on distributed-memory machines.

As a very generic programming model for parallel computers, message passing can also be used for programming shared-memory machines. The straightforward approach of running individual MPI processes on each processor core avoids many of the issues related to explicit shared-memory programming, but it has its limitations, too: in particular, multiple MPI processes need more memory than a single multi-threaded process. Also, multi-threading facilitates finer-grained parallelization schemes that can avoid explicit memory copies. As the number of cores per processor increases while the amount of available memory per core is expected to shrink, many codes adopt a hybrid parallelization approach with message-passing between computing nodes and shared-memory multi-threading within a node.

Shared Memory

With the recent rise of multi-core processors, either in stand-alone machines or in the nodes of larger, distributed-memory systems, shared-memory parallelism is now ubiquitous, spawning new interest in viable programming interfaces.

The shared-memory architecture facilitates multi-threading, where multiple threads of execution that access the same memory address space run in parallel. Global memory access from every thread eliminates the need to exchange data explicitly. This flexibility comes at a cost, however: uncoordinated accesses from multiple threads can easily corrupt shared data structures. Ensuring proper synchronization is therefore the foremost task when writing multi-threaded code. On top of that, today's complex NUMA architectures with multiple, partially shared cache levels hold some performance challenges for the ambitious programmer: meeting all data locality requirements to create optimally performing code across different systems can be difficult. As a result, shared-memory programming may look simple compared to message passing at first glance, but in reality, the effort required to create a correct and performant program can be significant.

There are a number of multi-threading programming interfaces available. Operating systems typically provide low-level APIs for creating threads and synchronizing memory access. For example, the POSIX threading interface (Pthreads) is a widely used, platform-independent low-level threading API. Some programming languages, such as Java or the upcoming C++11, also have multi-threading support built in. While low-level threading interfaces force programmers to take care of thread management and synchronization themselves, other shared-memory programming models provide high-level constructs to simplify the development of multi-threaded codes. The tool of choice for HPC codes is typically OpenMP [8] (Open specification for Multi-Processing), a combination of an API, language directives, and environment variables for C, C++ and Fortran programs. By now, all major compilers support OpenMP. OpenMP provides a set of high-level parallelization and synchronization constructs, particularly means to semi-automatically parallelize computational loops. Recently, more advanced constructs such as lightweight tasks have been added. Because OpenMP is also straightforward to integrate into MPI programs, the combination of MPI and OpenMP has become a popular solution for developing hybrid applications.

Partitioned Global Address Space and One-Sided Communication

Beyond message passing, partitioned global address space (PGAS) and one-sided communication are the two other major programming models used in applications for distributed-memory systems. A partitioned global address space presents a virtual, global memory address space to all processes. Each process owns a single section of the global address space. A process can access both local and remote memory sections; an access to remote memory locations will automatically invoke the necessary network operations. The PGAS model is the basis for data-parallel programming languages such as Unified Parallel C (UPC) [86], Co-array Fortran [64], Chapel [15], and X10 [73].

Another alternative to exchanging data on distributed-memory machines is the remote memory access (RMA) or one-sided communication model. Unlike message passing, where all processes affected by a data transfer have to actively participate in the communication operation, a one-sided data transfer can be completely defined and controlled on only one side. The one-sided communication model is especcially useful on systems with hardware remote memory access support, such as InfiniBand network adapters. It is also used to implement PGAS languages. A popular framework for one-sided communication is ARMCI (Aggregate Remote Memory Copy Interface) [62]. The MPI-2 standard also provides a one-sided communication interface.

Heterogeneous Systems

The fundamental differences of throughput-oriented GPUs in comparison to classic CPUs also require dedicated programming models for heterogeneous systems. Currently, the preferred programming interface for the widespread graphics processors from NVidia that implement the Compute Unified Device Architecture (CUDA) [65] is NVidia's own propietary CUDA API and 'C for CUDA' programming language. An open alternative is OpenCL (Open Computing Language) [46], a hardware-independent open standard for programming hardware accelerators. With both CUDA and OpenCL, the kernels (i.e., code executed on the accelerator) need to be implemented as seperate components in a restricted, C99-based programming language.

As alternatives to the low-level CUDA and OpenCL interfaces, the (Open)HMPP [19] and OpenACC [67] initiatives provide higher-level approaches to programming accelerators. Both are based on programming language annotations, so that programmers do not need to switch programming languages to develop code for accelerators.

1.1.3 Execution Models

Flynn [20] classifies parallel architectures by the number of concurrent instruction and data streams available in the architecture or program. In the context of parallel simulation codes, the SIMD (single instruction stream, multiple data streams) and MIMD (multiple instruction streams, multiple data streams) models are most relevant. Typically, both models are used in combination: the SIMD model represents the instruction-level parallelism provided by the vector-operation instruction sets in modern CPUs (such as Intel's SSE), where a single operation is applied to multiple data elements at once. Simultaneously, explicit parallelism in MIMD fashion using multi-threading or message passing is used to execute a program across multiple CPU cores and/or cluster nodes.

The MIMD category can be further classified into SPMD (single program, multiple data streams) and MPMD (multiple programs, multiple data streams) programs. With SPMD parallelism, each processor executes the same program (but with independent instruction pointers, as opposed to SIMD parallelism) operating on different data sets. Most simulation codes in the HPC environment are implemented as SPMD programs. In contrast, MPMD programs (loosely) combine sub-programs that perform entirely different activities. A common example is the master-worker model, which combines a "master" program (which distributes work) and "worker" programs (which execute the work). Another type of HPC programs that lends itself to MPMD-style parallelism are multi-physics codes which simulate a compound system by coupling different, interacting mathematical models and domains. Often, the individual models are themselves implemented as tightly-coupled SPMD programs and run in dedicated *process partitions*. Communication therefore occurs mostly within the process partitions, but to some extend also across partitions to reflect the interactions between different parts of the coupled system. An example for a coupled application that supports a flexible task decomposition in the way outlined here is the Community Earth System Model (CESM) climate project [85]. There, components that simulate different parts of the earth system, such as land surface, atmosphere, sea ice, or the ocean, can work in parallel in various SPMD and MPMD configurations. An additional, separate coupler component enables data transfers between individual components.

1.1.4 Parallel Performance

Writing efficient software for massively parallel computers is a challenging task. Difficulties arise from a number of influence factors, including heterogeneous architectures, complex memory hierarchies, and in particular the ever-increasing amounts of parallelism on multiple levels that need to be exploited. In addition, software-related aspects such as the complexity of large simulation codes themselves, legacy code, or the need to combine multiple parallel programming models further complicate the performance challenge. In short, the combined complexity of hardware and software architectures in parallel computing environments makes it virtually impossible to ensure that a program will run at the desired speed on the target system without systematic performance analysis.

The performance of parallel programs in particular is characterized by both serial and parallel performance aspects. Primarily serial performance aspects include the efficient use of the memory hierarchy and, obviously, the computational efficiency of the single-processor code. Important parallel performance aspects are communication and synchronization efficiency, process/thread mapping, and workload and communication balance.

Communication and synchronization efficiency refers in particular to the ratio of communication and (useful) computation in the program, i.e., the parallelization overhead. Also, an inefficient communication pattern can be a serious performance bottleneck. The second aspect mentioned, process/thread mapping, refers to the placement of (MPI) processes on the machine. In some cases, performance can be improved if processes are placed in such a way that the logical communication relationships between processes match the underlying physical links between the processors. Process mapping is something that is often tricky to accomplish optimally and difficult to observe.

Workload balance, the distribution of workload across processes, is obvoiusly a key factor for parallel performance: optimal speedup can only be achieved if the workload is equally distributed across processors. Likewise, developers need to balance the communication resource requirements – such as the number of communication partners – evenly between the processes. Properties, causes and analysis of load and communication imbalance as well as basic strategies for their remediation are discussed in detail in Chapter 3.

1.2 Performance Analysis

Combined, the parallel performance aspects are a major element of the overall program performance, and they alone determine the scalability of the code. A comprehensive performance evaluation of a parallel program must therefore consider not only serial, but also parallel performance aspects. Hence, serial performance analysis tools such as gprof are insufficient for the analysis of parallel programs. Dedicated tools for parallel programs include specific functionality for the analysis of parallel performance aspects.

1.2.1 Objectives and Requirements

The term *performance analysis*, as used in this thesis, refers less to the analysis of underlying algorithms or numerical methods, but rather to the performance evaluation of a program's implementation. Still, this interpretation does of course not exclude the possibility of reconsidering the choice of a particular algorithm as a consequence of the analysis. The major objectives of performance analysis are:

- Comparing expected with actual performance and scalability of the program
- Identifying performance and scalability bottlenecks

The first objective, comparing expected and actual performance, is necessary when porting applications to new platforms or integrating new algorithms or program features. This step can at least theoretically be partially automated if a formal *performance model* of the program exists. A performance model, such as the LogP [16] model for parallel machines, describes the performance behavior of a program as a function of its major input parameters and the system characteristics. However, few application developers take the time to develop a formal performance model of their program; in most cases, they only have a vague or intuitive understanding of the code's performance. In general, performance modeling is best used in conjunction with performance measurement, and vice versa: the performance model is needed to identify and explain disparities between observed and expected program performance, while measurement data is needed to develop and validate the performance model.

Many parallel programs collect basic performance data themselves, such as the overall wall-clock time or time per iteration of the time-stepping loop. Although this basic information facilitates simple scalability studies and detection of overall performance improvements or degradations, it is typically insufficient for the second objective of performance analysis: identifying performance bottlenecks. Should the program not perform as desired, the performance analyst needs to pinpoint the root causes of the problem. Identifying specific bottlenecks requires detailed information about the program's performance on a low level, which only few programs provide themselves. Dedicated performance-analysis tools are then a crucial aid for application developers to help them with this task. In addition to commercial tools distributed by system vendors or software companies, a vibrant community of developers from academic institutions has created a range of high-quality, open-source performance-analysis tools for parallel applications. Often, the use of these tools leads to significant performance improvements in the target programs.

The requirements of performance tools are determined by the objectives of performance analysis mentioned above: they have to provide insightful data to allow an informed decision on whether the observed performance of a program matches the expectations, and, if necessary, support users in finding performance bottlenecks. To reach these goals, tools employ a variety of techniques, which we will explore in more detail in the following section.

1.2.2 Performance-Analysis Techniques

Tool developers have created a variety of complementary performance analysis techniques. To provide a consistent user experience, performance-analysis toolsets combine components to address three aspects of performance analysis:

- Measurement and data storage
- Data analysis
- Data presentation and visualization

This sections discusses these performance analysis techniques in general. Later, Section 1.2.3 provides an overview of the various tools on the market.

Measurement

The evaluation of program performance requires observation of the program at run time. Performance observation necessitates performance measurement. Measurement is therefore a key aspect of performance analysis, and performance tools distinguish themselves to some extend by what and how they measure.

Tools typically measure a variety of performance-relevant metrics. The most important metrics are certainly time-related ones; others include visit counts of code regions, the number of bytes transferred during communication operations, or the overall number of communications. In addition, modern processors provide a number of counter registers which store performance metrics collected by the CPU itself, for example, the number of instructions, memory accesses, or cache hits/misses. The PAPI library [40] provides an architecture-independent programming interface to these hardware-specific counters, which is employed by many tools to obtain hardware-counter metrics.

Regarding the way performance data is measured, most tools employ one of two basic methods: *sample-based* or *event-based* measurement. With sampling, measurements are taken at certain intervals (the sampling rate), typically triggered by timer interrupts, and associated with the code region of the target program that was visited at the time of the sample. As an advantage, measurement granularity and measurement overhead can easily be tuned by adjusting the sampling rate. Also, the code location of a sample can typically be identified very precisely, down to the level of a single line of code. However, since only snapshots of the execution are taken, there is always a certain degree of uncertainty associated with the measurement results. Especially count-based metrics, such as function visit counts, are imprecise; this is particularly true for functions that run for a significantly shorter time than the sampling interval. Therefore, sampling measurements should be taken over a longer period to collect enough samples for meaningful results.

Event-based measurements are triggered at certain events in the program, such as entering or leaving a code region. Thanks to the deterministic distribution of the measurement points, event-based measurements are complete and precise. On the downside, it is harder to tune the measurement granularity, and, therefore, the measurement overhead. Some performance-analysis techniques, for example, those that rely on a complete record of inter-process interactions, require event-based measurements.

Profiling and Tracing

Performance tools also employ different concepts for the aggregation and storage of performance data. Widely used are *profiling* and *tracing*. In a summary profile, all measurements pertaining to a single (process/source code) location are accumulated over time. The result then only contains aggregate values, for example, the total time spent in a certain code region, but not the precise intervals during which a region was visited. Therefore, profiles are often also called *summary profiles*. As a major advantage, the space required for a summary profile remains roughly constant, irrespective of the program runtime. Profiles of parallel programs usually store performance data for each process (and/or thread) individually.

With tracing, each measurement is stored individually. This technique generates increasing amounts of data over time, but allows a precise reconstruction of the program's dynamic behavior. In a profile, these performance dynamics are lost. There are approaches that combine advantages from both profiling and tracing; for example, Szebenyi et al. describe an on-line compression scheme for iterative applications that preserves performance dynamics in constant-size *time-series profiles* [78]. Still, the advanced analysis techniques presented later in this thesis are only possible with trace data which captures the entire communication behavior in the section of interest. Chapter 2 discusses trace-analysis techniques in greater detail.

Analysis

The second important step performed by performance tools is data analysis, i.e., processing and filtering measurement data in a way that allows an easy overview of the performance behavior, or helps identifying interesting outliers or patterns. Some tools focus merely on presenting the measurement data to the user. These tools typically provide a variety of filters and statistical analysis methods, ranging from simple (e.g., minima, maxima, average) to quite advanced techniques such as cluster analysis. Often, the data-analysis functionality is integrated into the visualization component.

Another group of tools follows a highly analysis-oriented approach that aims to identify high-level performance patterns in the measured data automatically. These *automatic analysis* tools search for known inefficiency patterns in the measurement data, such as wait states in parallel communication or synchronization operations, or calculate high-level metrics such as the critical path. The analysis can take place while the program is running (*on-line*) or after it finished (*post-mortem*). On-line analysis does not require temporary storage for intermediate raw measurement data, but is limited in its capabilities to operations with little communication and computational effort in order to keep measurement dilation at bay.

Post-mortem analysis necessitates intermediate storage of raw measurement data (often traces), but can employ sophisticated, even multi-pass analysis strategies whose complexity is only limited by the user's patience. Instead of presenting measurement data in its entirety, an automatic analysis report pinpoints specific hotspots or bottlenecks a user may want to look into. The compact summary of interesting performance phenomena makes automatic analysis especially valuable for large-scale parallel program runs, where the sheer volume of data produced by a performance measurement presents a challenge for searching bottlenecks manually. Finding the root causes of performance problems (and fixing them) still requires expert knowledge and access to detailed performance data, but automatic analysis helps performance analysts by providing a good starting point for performance investigation. This task becomes even more important with increasing degrees of parallelism.

Presentation and Visualization

A good way to present the collected (and processed) performance data to the user is a key usability aspect of a performance tool. The challenges involved in finding helpful data presentation approaches are manifold. For example, measurements of complex programs at large scale can produce massive amounts of data that need to be made accessible. A good presentation tool should allow users to explore even large quantities of data interactively. Ideally, it also manages the balancing act between providing a compact overview of the program's behaviour as to not drown the user in a plethora of data, and at the same time allowing her to examine interesting findings in detail to investigate root causes of performance problems.

Auto-tuning

While performance analysis shows how far away a program's performance is from the optimal speed it can achieve on the target system, and helps identify specific bottlenecks, it does not make a program faster by itself. Instead, implementing optimizations based on performance-analysis results is where the actual difficulties often start. The potentially elaborate tuning process is still a mostly brain-powered task. However, it is also possible to follow an (often complementary) *auto-tuning* approach, which automatically chooses optimal algorithms or parameter settings for the target system. Dynamic performance tuning environments such as MATE [58] or Active Harmony [81] work fully on-line: they monitor and analyze the application as it runs, and can tune parameters and algorithms at runtime. Alternatively, auto-tuning can be used to automatically create optimized versions of performance-critical libraries by empirically determining optimal settings for performance-critical algorithm parameters, such as matric block size, in a profile-guided feedback process. This approach has been used in the context of the ATLAS [88] and OSKI [87] projects to create automatically tuned BLAS and sparse matrix kernels, respectively. Automatic tuning can greatly reduce the cost of porting libraries to new architectures.

1.2.3 Performance Tools

A wide range of both open-source and commercial parallel performance-analysis tools exist. The TAU (Tuning and Analysis Utilities) performance suite [51] developed at the University of Oregon is a well-known and widely used profiling framework. It supports a wide range of parallel programming models, including MPI, OpenMP, CUDA, Pthreads, and Java Threads, and features a number of instrumentation techniques. TAU's ParaProf visualization component (Figure 1.1) offers a variety of 2D and 3D displays to browse profile data in detail. HPCToolkit [3] from Rice University focuses on collecting profiles using sample-based measurement. It also offers post-mortem analysis methods to highlight program regions with load imbalance. Other open-source profilers for parallel programs are Perfsuite [50] developed at NCSA, and Open|SpeedShop [82] developed by the Krell Institute. Like HPCToolkit, Open|SpeedShop can perform sampling for taking measurements of parallel programs.

Many hardware vendors provide performance-analysis tools for their platforms. Notably, Cray provides the CrayPat [18] tool to collect performance profiles and traces of parallel applications, and the Cray Apprentice2 graphical user interface to analyze profiles and traces recorded with CrayPat interactively. IBM provides the IBM High Performance Computing Toolkit (HPCT) for its pSeries, eSeries, and Blue Gene platforms, which can collect hardware performance counters as well as profiles and traces for MPI and OpenMP programs. With its High Performance Computing Systems Toolkit (HPCST) [39], IBM also offers an automatic analysis tool that can actively search and identify pre-defined bottleneck patterns from performance data, and suggest possible solutions. Intel also provides performance-analysis tools. The Intel VTune Amplifier [43] toolsuite is a performance-analysis tool for Fortran, C and C++ programs on Windows and Linux with a focus on multithreading. The Intel MPI Trace Analyzer [42] (ITAC) is a trace-analysis solution for programs using Intel MPI. It can record parallel event traces and display them in a timeline view.

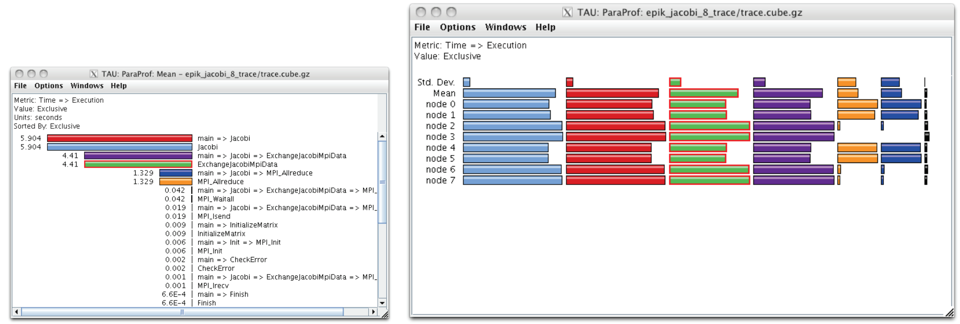
In addition to the vendor-provided tools, a variety of platform-independent trace-analysis tools exist. Paraver [12] from the Barcelona Supercomputing Centre is an extensive tracing and visualization framework. It offers various trace visualizations, such as timeline and statistics displays, as well as combinable semantic and quantitative analysis functions, which can produce displays of high level information such as the global parallelism of the application or load balance in different loops. Vampir [48] from TU Dresden is a well-known, full-featured trace visualization tool, with a particular focus on event timeline displays. Figure 1.2 shows a selection of the performance displays Vampir has to offer. While Vampir's instrumentation and trace collection framework is available as open-source software, the visualization component is distributed under a commercial license. Finally, Scalasca [26] is a scalable, automatic trace-analysis toolset which performs post-mortem event-trace analysis to automatically identify bottlenecks in parallel applications. As the foundation for the work in this dissertation, Scalasca will be discussed in more detail in Section 2.2. In addition to the post-mortem analysis tools presented so far, open-source on-line analysis tools also exist. Notable examples are the Paradyn [55] tool from the University of Wisconsin, Madison, and Periscope [29] developed at TU München.

The ongoing trend to heterogeneous systems also raises the demand for performance-analysis tools that support heterogeneous programs. NVidia provides the comprehensive NVidia visual profiler [66] for CUDA and OpenCL programs. It supports both profiling and trace analysis, and can automatically point out typical performance bottlenecks related to heterogeneous architectures, such as inefficient memory copies between the accelerator device and the host processor. However, the NVidia profiler is limited to programs on a single shared-memory machine. Some vendor-independent tools such as TAU and Vampir recently added support for CUDA and OpenCL programs, and can also analyze combined CUDA/MPI applications. Tool support for the recent HMPP and OpenACC programming interfaces is at this point still limited.

1.3 Contribution of this Thesis

Load and communication (im)balance as a particular important aspect of parallel performance calls for appropriate tool support to help developers recognize and improve imbalance-related performance problems in their codes. However, imbalance can take a variety of shapes, and – as will be discussed in detail in Section 3.7 – recognizing all of them also poses a challenge to performance-analysis tools. Summarizing the findings outlined there, a reliable, generic performance-analysis solution should

- point out program locations where imbalance originates from;
- reveal the underlying imbalance pattern;
- determine the severity of imbalance in terms of its actual performance impact;
- take performance dynamics into account;
- be generically applicable, in particular to both SPMD and MPMD programs;
- work at large scale.



(a) Function profile **(b)** Process profile

Figure 1.1: Performance profile displays in TAU ParaProf. The left picture shows a function profile, with the average time per process spent in each function. The right picture shows the time spent in each function (color-coded) on each process. This display can be used to detect (static) load imbalance.

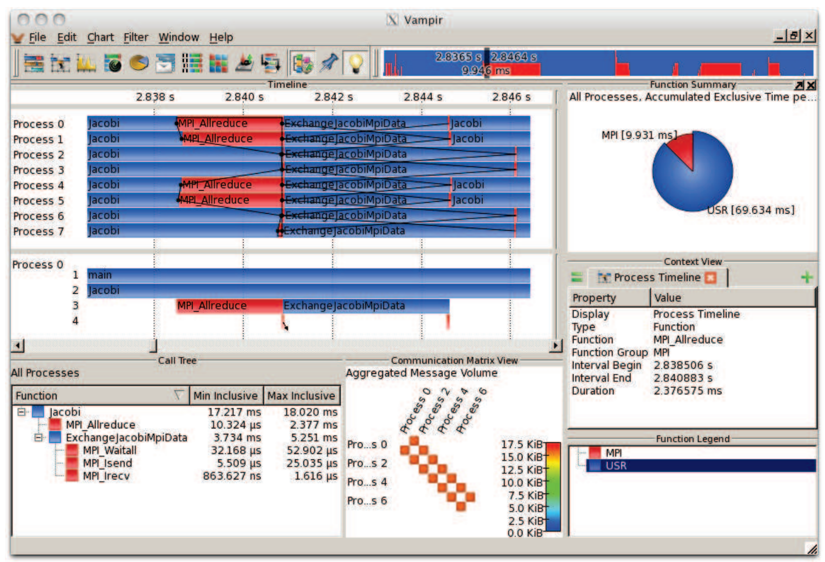


Figure 1.2: Various event trace displays in Vampir. The top left area shows an event timeline, covering a small time interval on eight processes. The process timeline display below shows the activities performed on a single process within the selected interval (here, process 0). Grouped around the timelines are a call-path profile (bottom left), a message statistics profile (bottom center), a function group profile (top right), and a context view giving details on a selected item (in this case, an MPI_Allreduce instance).

However, as an overview of previous work in Section 3.6 points out, currently available imbalance analysis solutions account for one or a subset of these aspects at best, but none of them fulfills all of the requirements listed here. Notably, profiling-based approaches cannot capture dynamic performance effects, while existing trace-based solutions drown users in vast amounts of data, do not scale to relevant system sizes, or do not appropriately adress complex load-balance issues arising in MPMD or hybrid programs. This dissertation introduces two novel methods to automatically detect and characterize imbalance in event traces: the *delay analysis* [10] to identify root causes of wait states, and the *critical-path analysis* [9] that characterizes load imbalance in terms of its performance impact. Together, these methods provide a universally applicable solution for the analysis of imbalance that fulfills all of the requirements listed above.

The delay-analysis approaches load imbalance from its visible effect, using wait states in synchronization points as a starting point to track down the imbalances that caused them in the first place. This approach optimally complements Scalasca's existing wait-state analysis. So far, Scalasca scans event traces to identify wait states in MPI operations or OpenMP constructs. These wait states indicate parallelization bottlenecks resulting from load imbalance or inefficient communication patterns: as such, wait states represent *symptoms* of inefficient parallelism. Moreover, wait states may themselves delay subsequent communication operations and spread further wait states through the system. This propagation effect can create a potentially large temporal and spatial distance between wait states and their original root causes, making it difficult to relate the effect (wait state) to its cause (imbalance) manually. The delay analysis closes this gap by automatically pinpointing the specific performance bottlenecks that cause wait states later on. To that end, the dissertation defines a terminology that describes the formation and propagation of wait states as a result of *delays*, a cost model that ranks delays according to their associated resource waste, and a scalable algorithm to detect and characterize delays.

The second major contribution of this dissertation, the critical-path analysis, illuminates the effect of load imbalance on program run time and resource consumption. The critical path determines those activities in a program that are responsible for the overall run time. While critical-path analysis has been used in performance-analysis tools to identify promising optimization targets for quite some time, the method introduced in this dissertation combines the critical-path with per-process profile data to uncover performance-relevant load-balance issues and determine their performance impact. The analysis creates a set of compact performance indicators that capture the overall load balance and any potential bottlenecks in an intuitive way. Compared to existing solutions that determine load imbalance on the basis of per-process profiles alone, the critical-path based approach has two major advantages: First, it accounts for dynamic effects, such as load shifting between processes over time, that other approaches miss. Second, in addition to the analysis of SPMD programs, it contains specific support for the analysis of complex load imbalance issues in MPMD programs.

Because parallel performance problems such as load imbalance are particularly significant at large scales, it is vitally important that analysis methods are applicable to large-scale program runs. Both the delay analysis and the critical-path analysis are implemented as extensions to the Scalasca performance-analysis toolset, which employs a highly scalable parallel event trace replay technique to analyze traces even from tens of thousands of processes. Scalability studies with up to 262,144 cores confirm that both analysis methods are applicable to large-scale problems. Finally, a number of case studies with real-world examples demonstrate the ability of the delay and critical-path analysis to uncover imbalance-related performance bottlenecks intuitively.

Summary

Since sequential processing speed is unlikely to increase significantly in current and future hardware, software must be designed to exploit parallelism to leverage maximal computing power. However, writing efficient parallel programs is challenging. One particularly important aspect of parallel performance is load and communication balance. The two new analysis techniques introduced in this thesis help developers in quickly identifying imbalance-related performance bottlenecks in large-scale parallel programs.

The dissertation is organized as follows. First, Chapter 2 briefly explains the underlying techniques used in performance-analysis tools. Next, Chapter 3 discusses load and communication balance in parallel programs in general, with a special focus on intricacies posed by certain imbalance patterns that complicate the identification of imbalance. Chapters 4 and 5 introduce the delay analysis and the critical-path analysis, respectively. Both chapters also include case studies which highlight the specific benefits of that technique. Afterwards, Chapter 6 discusses scalability and applicability of both techniques in comparison. Finally, Chapter 7 concludes the dissertation and provides an outlook on future research directions based on this work.

Chapter 2

Event-Trace Analysis

Many interesting phenomena in a program's behavior can only be investigated when the full sequence of its activities (up to a certain granularity) is known. Event tracing is a widely used technique to obtain this information for post-mortem analysis. This section introduces event tracing in general and its application in the Scalasca toolset for automatically identifying performance bottlenecks in particular. After a basic introduction of the Scalasca toolset, the chapter discusses basic principles of event tracing, with a particular focus on the Scalasca event model and references to related work where appropriate.

2.1 Event-Tracing Applications

An event trace represents the program execution as a sequence of events which reflect certain state changes in the program. The events are defined by an event model, which describes the different types of events and data attributes associated with them, as well as the context in which each event type occurs. At runtime, the target program performs measurements and stores event records containing the type, location, (typically) a timestamp, and attributes for all events that occur during execution.

Event tracing has a number of different applications in the context of parallel program analysis. In [49], Kranzlmüller describes the generation of event graphs from trace data to debug communication patterns in message-passing applications. Other projects, e.g. Dimemas [11, 31] or PSINS [83], use event traces to conduct simulations that predict performance and scalability of applications or system software on larger configurations or non-existant systems. Such simulation tools can also help in evaluating procurement decisions or future system designs. In [34], Hermanns et al. introduce a complementary trace-based simulation approach to study the impact of program optimizations. Of course, event tracing is also used for performance analysis. Trace visualizers like Vampir (Figure 1.2) present the recorded execution of a parallel code as a time-line diagram. Each program activity is represented by a rectangle whose length corresponds to its runtime, and message exchanges are shown as arrows between communication activities. Finally, automatic analysis tools like Scalasca perform an automatic bottleneck search on the trace to draw the user's attention directly to inefficiency patterns in the program. Automatic and visual analysis methods complement each very well and are often used in combination, for example to identify performance bottlenecks automatically first and examine them in detail in a time-line visualization tool later on. Open event specifications and trace file formats such as OTF-2, which can be parsed by different tools, facilitate such complementary analyses.

2.2 Scalasca

Scalasca [26] is a profiling and trace analysis toolset for parallel programs which is jointly developed by the Jülich Supercomputing Centre and the German Research School for Simulation Sciences in Aachen. Its distinct feature is the automatic detection of wait states in communication and synchronization operations. Scalasca is specifically targeted to the analysis of parallel application behavior of large-scale programs; compared to its predecessor KOJAK [89], Scalasca is able to analyze massively parallel program runs with tens of thousands of processes. Developed with portability in mind, Scalasca runs on a broad range of current supercomputing platforms and architectures, including IBM Blue Gene L/P/Q series, the Cray XT/E/K series, NEC, and Linux clusters. It supports programs written in C, C++, or Fortran using MPI, basic OpenMP constructs, or both (hybrid programs). Support for further parallel programming models, such as partitioned shared address space languages or remote memory access models, is currently under development.

Scalasca supports two types of measurements. In *summary profile* mode, Scalasca records a runtime profile of the program. The profile report contains various metrics such as time, visit counts, bytes transferred in communication calls, and, optionally, hardware counter metrics for each call path and process and/or thread in the program. Time metrics are further split into the time spent in user code and the time spent in various types of MPI operations or OpenMP workshare, loop, and synchronization constructs. With its *tracing* mode, Scalasca records an event trace of the program, and runs an automatic post-mortem bottleneck search on the trace to identify and classify wait-states in MPI and OpenMP operations. The extended performance report produced by the trace analysis divides MPI time further into waiting and communication time and shows the process and program locations of wait states. Alternatively, the event traces produced by Scalasca can also be converted into various other trace formats or manually inspected with the Vampir trace visualizer.

Scalasca's performance reports store performance data in a three-dimensional (metric, callpath, thread) matrix. Essentially, every entry in this matrix represents the value of a particular metric at a particular program location (identified via its call path) on a particular thread. Even for trace analysis reports, values are always aggregated along the time dimension. Additionally, Scalasca provides a *worst instance tracking* feature, which reports the exact location in the trace where the worst instance of a performance bottleneck or metric has been observed.

Users can explore the reports using the provided graphical user interface (Figure 2.1), dubbed "Cube" in reference to the three dimensions in the report. It features three panes, one for each performance dimension. The left pane shows the hierarchy of metrics found in the report, which includes typical performance profile metrics such as execution time and visit counts as well as the waiting-time metrics produced by the automatic trace analysis. The middle pane shows the distribution of performance data across the program call tree, and the right pane shows the distribution of performance data across processes. For programs that use a regular

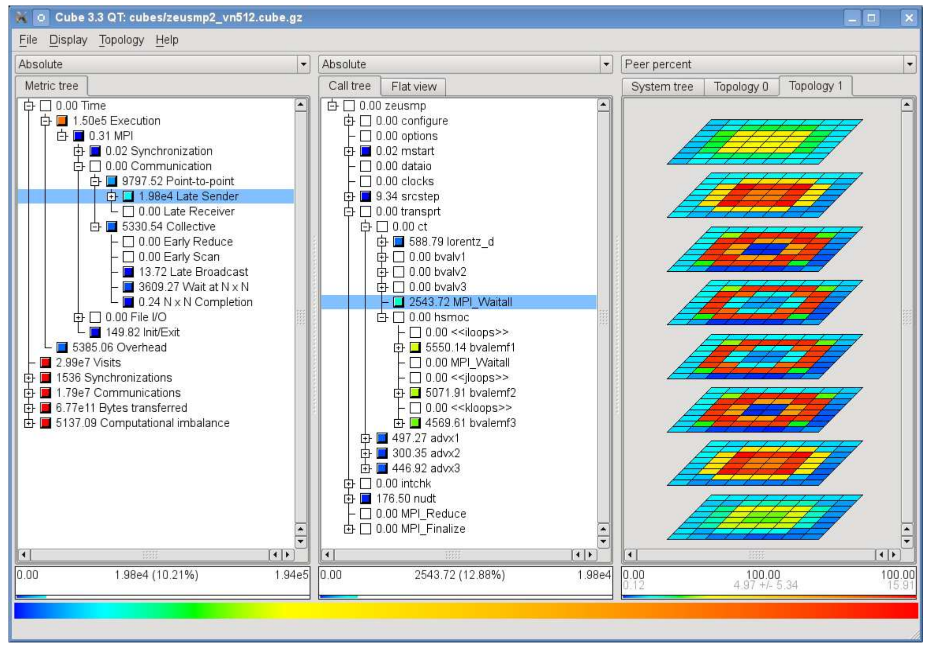


Figure 2.1: A trace-analysis report of the Zeus-MP/2 application with 512 MPI processes in the Scalasca report browser. The left pane shows the performance metrics produced by Scalasca. The middle pane shows the distribution of performance data across call paths for the selected “Late sender” waiting time metric. Finally, the third pane shows the distribution of the late-sender waiting time for the selected `MPI_Waitall` call path across processes. In this case, the processes are arranged in the logical 8x8x8 grid used by the application.

1D, 2D or 3D virtual process topology, Scalasca can record and display the performance data within the application's virtual topology. Besides the HAC approach by Schulz et al. [75], Scalasca is one of very few production-grade tools which support such a visual mapping. As we will see in the example section in Chapter 4, this feature can greatly help in understanding complex performance phenomena.

2.3 The Scalasca Event Model

Scalasca uses event traces as basis for its detailed automatic performance bottleneck search in parallel MPI, OpenMP, and hybrid MPI/OpenMP programs. Therefore, the underlying event model captures all the necessary information to identify performance problems and relate them to the program and machine locations where they occur. A special focus lies on capturing the MPI communication and OpenMP synchronization behavior: the event model has been designed to support Scalasca's automatic event-trace analysis of inefficient communication patterns, but it also facilitates other use cases beyond the automatic bottleneck search. Advanced examples include a performance simulation approach to evaluate performance hypotheses [34], and trace visualization using Vampir or other tools.

Scalasca's event model specifies programming-model independent events, such as events for entering and leaving source code regions, and events that are specific to a certain programming model, e.g. MPI or OpenMP. Every event contains a timestamp and additional information (attributes) related to the action it describes. Table 2.1 lists the event types and their attributes that are relevant for this dissertation.

The programming-model independent Enter (E) and Exit (X) events denote entering and leaving source code regions, respectively. The region entered is specified as an attribute of the Enter event, the region left in the Exit event is implied by assuming that region instances are properly nested. Most commonly, regions are functions or subroutines, but they can also mark basic building blocks such as a loop or a sequence of statements. The nesting of regions makes it possible to reproduce the *call path* of an event in the trace, that is, the sequence of regions entered leading to this event. The call-path information helps users to relate individual findings in the trace to their calling context (e.g., `main/foo/send` or `main/bar/send`). Knowledge of the call path is particularly useful to identify the calling contexts of MPI operations, which are often issued from different places in the program.

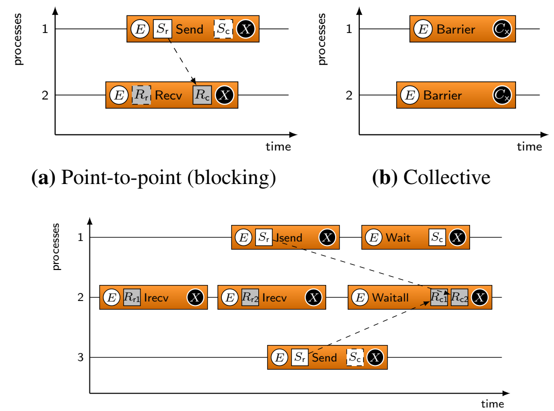
2.3.1 MPI Events

Scalasca's event model contains a number of event types to capture MPI point-to-point and collective communication. Additionally, the model also includes events for MPI one-sided communication, but since one-sided communication is not covered by this thesis, they are not included in this discussion.

The MPI-related events provide a rich amount of information about the respective communication operation. For point-to-point communication, the collected attributes include the number of bytes that were transferred, the communicator for every communication operation, source or

Table 2.1: Event types in the Scalasca event model and their attributes. Each event additionally contains a timestamp.

Symbol	Event type	Attributes
Entering and leaving code regions		
E	Enter	Region entered Hardware counter values (optional)
X	Exit	Hardware counter values (optional)
MPI communication		
S_R	Send request	Destination location Message tag Bytes sent Communicator Request ID (for non-blocking send)
S_C	Send completion	Request ID
R_R	Receive request	Request ID
R_C	Receive completion	Source location Message tag Bytes received Communicator Request ID (for non-blocking receive)
C_X	Collective exit	Collective root location Bytes sent Bytes received Communicator
OpenMP events		
F	Fork	
J	Join	
O_X	OpenMP collective exit	Region ID
L_A	Aquire lock	Lock ID
L_R	Release lock	Lock ID



(a) Point-to-point (blocking) **(b)** Collective

(c) Point-to-point (non-blocking)

Figure 2.2: Typical event sequences for MPI point-to-point communication. White and black circles denote Enter and Exit events, respectively. White and grey squares denote the request and completion events for send and receive operations, respectively. Events represented as dashed squares are implied and not actually stored in the trace.

destination rank, and the message tag. For collective communication operations, the attributes include the communicator, the number of bytes transferred, and the root rank for collective broadcast or reduction-type operations.

MPI point-to-point messaging is modeled using four special communication event types. The point-to-point event records are enclosed by enter and exit events that mark the beginning and end of the MPI call and carry attributes of the specific communication operation at hand. To capture both blocking and non-blocking communication, send and receive operations are modeled with two events each, a request and a completion event. A request event is located at the point where the operation was initiated (e.g., in an `MPI_Send` or `MPI_Isend` region for a send operation, `MPI_Irecv` for a receive operation, or `MPI_Start` for either). Likewise, a completion event is located at the point where the operation was completed (e.g., in `MPI_Recv` or `MPI_Wait`). In the case of non-blocking communication, request and completion event carry a request ID to match a request to its corresponding completion event (and vice versa). For blocking communication calls, send completion and receive request events are not explicitly written, but merely implied to be enclosed in the same MPI call region as the send request or receive completion event. Their (implied) timestamp then corresponds to that of the enclosing send region's exit or receive region's enter event, respectively.

Unlike point-to-point communication events, which are enclosed between enter and exit events of the MPI call, MPI collective operations are modeled as a pair of an enter and a specialized collective exit event on each process that participates in the collective operation. The specialized collective exit event replaces the standard exit event. This exit event record type is used for all MPI collective operations, the specific operation at hand becomes apparent from the name of the MPI call.

Figure 2.2 shows time-line diagrams representing typical event sequences for MPI communication. Figure 2.2a shows a simple example of blocking communication, where send request and receive completion event records are present between the enter and exit events of the MPI calls; the send completion and receive request events are only implied. Figure 2.2c shows a more complex example with non-blocking communication. Here, we find a send request event record in the `MPI_Isend` region on rank 1, and the corresponding send completion event record in a subsequent `MPI_Wait` region. The time line on rank 2 demonstrates how two receive completion event records are enclosed in a single `MPI_Waitall` region, indicating that both communication operations were completed in this MPI call. Example 2.2b shows the representation of MPI collective operations as a combination of Enter and collective Exit events.

2.3.2 OpenMP Events

In a multi-threaded program, each thread writes its own event records. Scalasca's event model includes event types for OpenMP constructs. The fork and join event records denote creation and termination of a new team of threads, respectively. These event records are only written on the master thread. Child thread events only occur between fork and join. OpenMP constructs, such as parallel regions, loops, or (explicit or implicit) barriers, are modeled as a code region using an Enter and a special OpenMP collective exit record. Like the MPI collective exit record, the OpenMP collective exit record replaces the regular exit record and contains all information that the regular exit event would. In addition, there are lock aquire and lock release events, which model e.g. critical sections or the OpenMP locking routines.

2.4 Wait-state Classification in Scalasca

In its current form, Scalasca's automatic trace analysis detects wait states in various MPI and OpenMP communication operations and classifies them by the type of wait-state pattern. This section illustrates these concepts (i.e., what specifically constitutes a wait state in MPI and OpenMP) and describes the different types of wait states detected by Scalasca.

2.4.1 Wait States in MPI Communication

The major focus of Scalasca's wait state analysis is MPI communication. Scalasca characterizes MPI wait states in a generic way, relying only on the communication semantics derived from the MPI standard and not including specific details of any particular MPI implementation. So far, Scalasca detects wait states in point-to-point, collective, and MPI-2 one-sided communication. However, this thesis discusses point-to-point and collective communication only.

The wait-state detection has not been implemented for some non-communication operations where they may also occur (e.g., `MPI_Probe`, file I/O, or certain communicator management operations). Moreover, wait-state detection is not possible in some cases for which the event trace does not contain all necessary information. This problem particularly concerns some collective operations that send a varying number of data elements to each process, such as

Table 2.2: Types of wait states detected by Scalasca in MPI communication operations.

MPI call	**Type of wait states**
MPI_Send, MPI_Ssend	Late Receiver
MPI_Recv	Late Sender
MPI_Sendrecv, MPI_Wait variants	Late Sender *or* Late Receiver
MPI_Probe	*Wait-state detection not implemented*
MPI_Bsend, MPI_Rsend, MPI_Iprobe, MPI_Irecv, MPI_Isend variants, MPI_Start, MPI_Startall, MPI_Test variants	*No wait states possible*
MPI_Bcast, MPI_Scatter, MPI_Scatterv	"Late Broadcast" (Wait at 1-to-N collective)
MPI_Gather, MPI_Gatherv, MPI_Reduce	"Early Reduce" (Wait at N-to-1 collective)
MPI_Scan, MPI_Exscan	Early Scan
MPI_Allreduce, MPI_Allgather, MPI_Alltoall	Wait at NxN collective
MPI_Barrier	Wait at Barrier
MPI_Allgatherv, MPI_Alltoallv	*No wait-state detection possible*

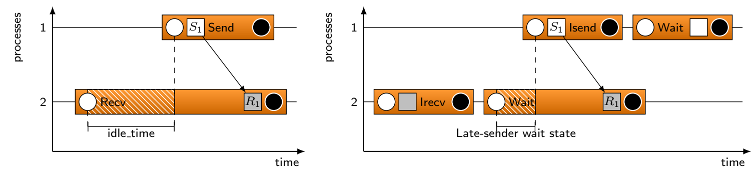
MPI_Alltoallv). Section 6.2.3 explains these limitations in more detail. Table 2.2 shows which type of wait state Scalasca detects in each MPI communication operation. In the following, the wait-state types and their theoretical background are explained in more detail.

Point-to-point Communication

Scalasca detects two basic types of wait states in point-to-point communication: *late sender*, where a message receiver sits idle while waiting for a message to be sent, and *late receiver*, where a sender of a synchronous message waits until the receiver is ready for the message transfer.

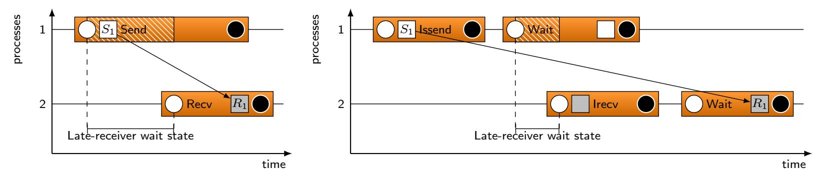
A late-sender wait state occurs when an MPI call that is completing a receive operation blocks because no matching send request had been issued by the time the call was entered. The waiting time corresponds to the time between entering the receive completion function on the receiver side and the time entering the send request function on the sender side. Except for MPI_Test (and its variants), which do not exhibit wait states by defintion, late sender instances can occur in all MPI calls that can contain a receive completion event. Figure 2.3 illustrates different forms of late-sender wait states in blocking and non-blocking MPI calls.

A late-receiver wait state occurs when an MPI call that is completing a synchronous send operation blocks while the receiver is not ready to receive the message (i.e., it has not yet posted a matching receive). The waiting time then corresponds to the time between entering the send completion call on the sender side and the time entering the receive request call on the receiver side. Figure 2.4 illustrates the concept for both the blocking and non-blocking cases.



(a) Late sender (regular) **(b)** Late sender (non-blocking)

Figure 2.3: Late-sender wait states can occur in `MPI_Recv` calls (blocking case, Fig. 2.3a) or any of the `MPI_Wait` calls (non-blocking case, Fig. 2.3b).



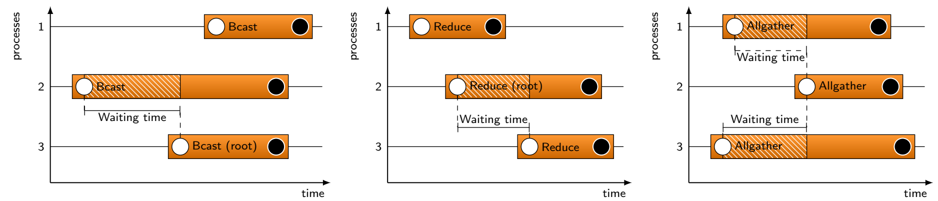
(a) Late receiver (regular) **(b)** Late receiver (non-blocking)

Figure 2.4: Late-receiver wait states occur when messages are sent in synchronous mode and the receiver is not yet ready to receive the message. They appear in `MPI_Send` calls (blocking case, Fig. 2.4a) or any of the `MPI_Wait` calls (non-blocking case, Fig. 2.4b).

It should be noted that this characterization of late-receiver wait states is a heuristic: because it is not easily possible to determine whether a send operation was in fact synchronous (in a generic way without support of the MPI library), some amount of time that was actually used for the data transfer itself may erroneously be classified as late-receiver waiting time if there is a partial overlap between the send and receive calls. However, since data transfer times are typically small in comparison to actual waiting times, the late-receiver heuristic will still correctly identify severe wait states while the error introduced by the misclassification effect is negligible.

Collective Communication

The MPI standard does not explicitely specify synchronization semantics for its collective communication operations. Collective operations may or may not be synchronous, their exact synchronization behavior is implementation-specific. The only exception is the barrier, which always synchronizes all participating ranks. However, the communication semantics do imply some form of synchronization nonetheless: for example, a broadcast operation cannot complete on any rank before the root rank has entered the operation. Scalasca detects and classifies wait states that result from these implicit synchronization requirements. Figure 2.5 shows time-line visualizations of the different wait-state patterns in collective communication Scalasca detects. *Late broadcast* wait states (Figure 2.5a) occur at 1-to-N collective operations such as broadcast or scatter when ranks have to wait for the root process to enter the operations. In N-to-1 operations such as reductions or gather, the root process may incur an *early reduce* wait state (Figure 2.5b) when a non-root process enters late. Finally, for implicitly globally synchronizing all-to-all data exchange operations such as allgather or alltoall, all ranks wait



(a) Late broadcast **(b)** Early reduce **(c)** Wait at NxN

Figure 2.5: Wait states in MPI collective communication.

for the rank that enters last (Figure 2.5c). The same pattern also applies to the barrier.

2.4.2 Wait States in OpenMP Synchronization

Scalasca also identifies wait states at OpenMP synchronization points. However, the current trace replay algorithm only supports a basic execution model where parallel regions are not nested and the number of threads remains constant over the entire execution. It also does not yet support OpenMP 3.0 tasks. Scalasca identifies and classifies wait states that synchronize the entire team of threads, where it distinguishes between *explicit barrier synchronization* for wait states that occur at an explicit synchronization point in the code (i.e., the `barrier` construct), and *implicit barrier synchronization* for wait states that occur at implicit barriers at the end of a parallel loop or a parallel region.

2.5 Trace Analysis Methodology

This section outlines the basic working principle behind Scalasca's scalable wait-state search. Figure 2.6 illustrates the trace analysis workflow in Scalasca. To prepare a program for a trace measurement, it has to be *instrumented* first. Instrumentation inserts measurement code into the program and adds the necessary libraries to perform trace (and runtime summary) measurements. Scalasca supports a variety of instrumentation techniques: MPI events are captured by wrapper functions which are implemented through the PMPI profiling interface [54, Chapter 14]. To instrument user code, Scalasca uses profiling interfaces built into modern compilers that automatically insert measurement code at certain code points, such as entering or leaving a function. An alternative is source-code instrumentation, where the measurement code is directly inserted into the source code; either manually using an instrumentation API or automatically through a source-to-source compiler. Scalasca offers two automatic source-code instrumentors: Opari [57], Scalasca's instrumentor for OpenMP constructs, and a generic, configurable solution based on TAU's program database toolit (PDT) [25]. Both compiler and source-code instrumentation require recompilation of the program. With the experimental binary instrumentor *Cobi* [59], it is also possible to instrument pre-compiled binary executables for Scalasca measurement.

At runtime, the instrumented target program generates event records for the intercepted events and writes them into a thread-local memory buffer. The buffer is flushed to disk at the end

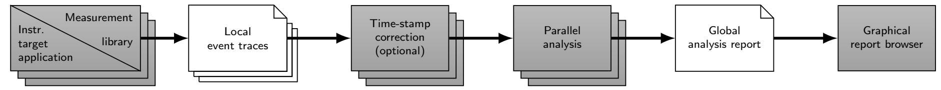


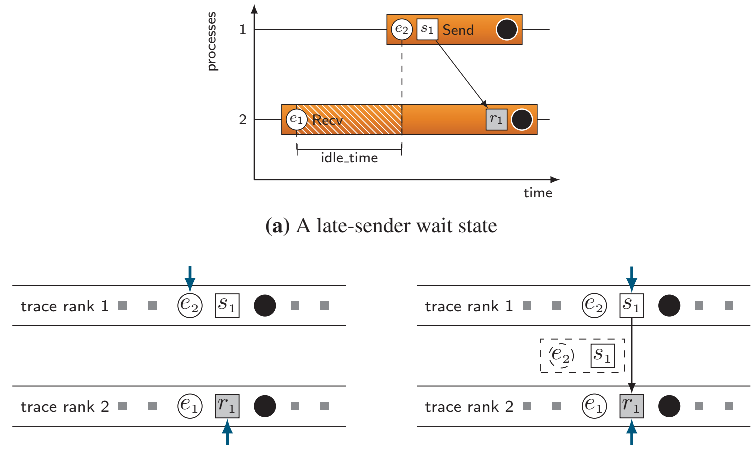
Figure 2.6: Scalasca's parallel trace-analysis workflow. Gray rectangles denote programs and white rectangles denote files. Stacked symbols denote multiple instances of programs or files, run or processed in parallel.

of the execution, or in between if it is full. Users are strongly advised to make the memory buffer large enough to contain the entire trace, since intermediate buffer flushes occur unsynchronized and can heavily perturb the measurement. At the end of the execution, all threads write their trace data in parallel, which allows them to utilize potential parallel I/O capabilities of an underlying (parallel) file system. By default, each thread writes its own trace file. However, metadata handling for a very large number of files can become a bottleneck at large scale. As an alternative, it is possible to write traces using the SIONlib [21] I/O framework, which maps an arbitrary number of virtual files onto a small number of physical files. While the actual event records can easily be recorded and stored individually by each thread in a distributed fashion, the identifiers for objects referenced in the event record attributes, such as code regions, locations, or MPI communicators, need to be globally unique. Therefore, these unique identifiers are created in an additional unification step at the end of the execution using a scalable, parallel reduction approach [24, 23], and written to a global definition file.

After the execution of the target program has finished and trace files have been written to disk, the Scalasca trace analyzer is launched. The analysis runs on the same number of processes and threads as the target program. By making the trace analysis a parallel program in its own right, Scalasca can utilize the distributed memory and processing capabilities of the parallel machine for the trace analysis itself, and therefore achieve excellent scalability. Each analysis thread loads the trace records of one of the original program's threads. To compensate for timestamp errors which result from unsynchronized clocks, an optional, distributed timestamp-correction algorithm [7] can be applied in-memory before the actual analysis. During the actual bottleneck search, the analysis threads traverse their thread-local traces in parallel. When they encounter a synchronization point in the trace, the corresponding analysis threads "replay" the original communication using an operation of similar type (but with different data), where they can exchange information that is necessary for the analysis at hand.

Figure 2.7 illustrates the approach. Consider an event trace of an MPI point-to-point data transfer which incurs a late-sender wait state, as shown in Figure 2.7a. The point-to-point operations are represented by a sequence of function enter, send (or receive, respectively), and function exit event records. For the analysis, the corresponding analysis threads traverse the traces in parallel (Figure 2.7b). When the analysis threads reach the original synchronization point, they initiate a point-to-point operation using the same parameters (communicator, tag, and source/destination) as the original communication, to transfer the timestamp of entering the original send operation to the analysis thread for the receive operation (Figure 2.7c). There, the analyzer compares the local and remote timestamps to determine the amount of waiting time that occured at the receive operation.

Finally, the wait state instances are classified by their type (e.g. *late sender*) and accumulated in a local (type, call-path) matrix. At the end of the trace replay, these local matrices are



(a) A late-sender wait state

(b) Parallel trace replay

(c) Data exchange at synchronization point

Figure 2.7: Parallel trace replay in Scalasca

merged into a global, three-dimensional (type, call-path, thread) matrix. This global report is written to disk and can be explored with the Scalasca report browser (Figure 2.1).

2.6 Tracing Constraints and Requirements

While event tracing offers highly detailed insights into the dynamic performance behavior of parallel programs, it is undeniably also an expensive technique, both in terms of the physical resources and the effort required to set up useful experiments. This section gives an overview of the issues associated with tracing and lists methods to mitigate these problems.

An obvious constraint is the amount of data that is generated. The size of a trace grows (roughly) linearly both with the number of processes and the runtime of a program. Traces of large-scale experiments can occupy a large amount of storage, and their analysis requires significant I/O bandwidth and compute power for data processing. While parallel I/O and distributed processing allows trace analysis to scale well with increasing numbers of processes, the linear increase of trace size with program runtime typically limits the applicability of trace analysis to short-running experiments. To address this issue, several groups developed (lossy) trace compression solutions. ScalaTrace [63] compresses trace data both across the time and process dimension online and can in some cases produce near-constant size communication traces. Knüpfer [47] introduces an offline trace compression approach using CCG (complete call graph) data structures to exploit redundancies in the trace data. These trace compression approaches can often achieve impressive compression rates, but the compression works best only in highly regular, iterative SPMD applications. Currently, Scalasca does not use trace compression. However, as an extension considered for future releases, it is planned to keep trace data in persistent memory segments after the measurement, where the analysis process can read it. This approach can at least avoid the additional overhead for writing and recovering trace data from permanent storage.

Closely related to the problem of large amounts of data in trace experiments is the problem of measurement overhead and dilation. Like any measurement technique, instrumentation alters the behavior of the target program, so that the performance characteristics of the measured program run may be different from its original behavior. The overhead should be kept as low as possible to obtain an accurate representation of the original performance behavior in the measurement.

Both trace size and measurement overhead are directly affected by the measurement granularity. Fine-grained measurements produce a higher event rate which shows more detail, but increase both overhead and trace size; while a too sparse measurement may miss important performance details. Therefore, it is crucial to determine the optimal granularity before running a trace experiment. Scalasca provides means to filter user functions statically (i.e., at compile time) or dynamically (i.e., at runtime), limit MPI events to certain types (e.g., only point-to-point or collective events), or disable tracing entirely for a certain code region. The latter functionality can be used to skip tracing of uninteresting program sections (e.g., the initialization phase). It is highly recommended to apply filters for frequently executed, short-running user functions, since these often incur high measurement overhead and unnecessarily increase the trace size. To identify candidates for filtering, one typically performs a profile measurement first before running a trace experiment.

Another important aspect to consider for event tracing of parallel programs is clock synchronization. While some systems such as the IBM Blue Gene series do have a globally synchronized clock, most parallel architectures, particularly clusters, only provide processor-local clocks for time measurement. Since these processor-local clocks are typically not perfectly synchronized, it is important to account for clock differences when comparing timestamps from different sources. Scalasca uses linear offset interpolation of timestamps to compensate for constant differences between processor clocks and linear clock drifts, and optionally applies an advanced *controlled logical clock* algorithm [7] to remove clock-condition violations (e.g., messages appearing to be received before they were sent) resulting from non-linear clock drifts.

Summary

Event traces record the entire dynamic execution behavior of a program up to (almost) any required level of detail. Thereby, they provide the foundation for in-depth performance analyses using trace visualization tools or automatic bottleneck detection. The Scalasca performance analysis toolset automatically detects patterns of wait states in previously recorded event traces of MPI programs using a parallel, highly scalable trace search approach. This method provides the technical basis for the imbalance characterization methods introduced in this dissertation.

Chapter 3

Characterizing Imbalance

Ideally, the workload in a parallel program is distributed evenly across all available processing elements. However, because perfect balance is often difficult to achieve in practice, load- or communication imbalance is one of the most frequent performance and scalability bottlenecks in parallel programs. Imbalance occurs in various forms and patterns. This chapter discusses typical patterns of imbalance in parallel programs, as well as common causes of imbalance and possible solutions. Moreover, it also demonstrates the difficulties faced by performance analysis tools in identifying imbalance, and finally presents an overview of previous work related to characterizing imbalance in parallel programs.

3.1 Execution Model

Obviously, achieving an optimal distribution of workload across the available processing elements is key for efficient resource usage in parallel computing environments. In a typical HPC environment, where multiple parallel programs may run simultaneously in different partitions of one (or more) large machines, load balance must be ensured on multiple levels: each single parallel program should use the partition it is assigned to efficiently, while the job scheduler must achieve a good balancing of jobs across the machine to optimize the overall machine utilization. The concepts developed in this dissertation primarily target (im)balance within a single parallel program, with a particular emphasis on tightly coupled simulation codes. This section outlines the particular execution model of parallel programs that the discussions in the next chapters are based on.

While the model aims to be as generic as possible, some restrictions for the sake of simplicity do apply. First, the basic model for the discussion of basic imbalance concepts is restricted to programs with a single level of parallelism, in particular single-threaded message-passing programs. Therefore, without loss of generality, we use the term "process" to describe a single participant of a parallel program. Likewise, "processor" describes a single *processing element*, which can either be a traditional single-core CPU, or one core of a multi-core CPU. The model assumes that each process occupies a dedicated processor. Therefore, the model does not cover execution modes where processes (partially) share a single processor, for example through overloading/timesharing or SMT/HyperThreading modes.

We can model the execution of a parallel program in the form of a time line diagram with separate time lines for each process, as shown in Figure 3.1. We assume that the entire parallel

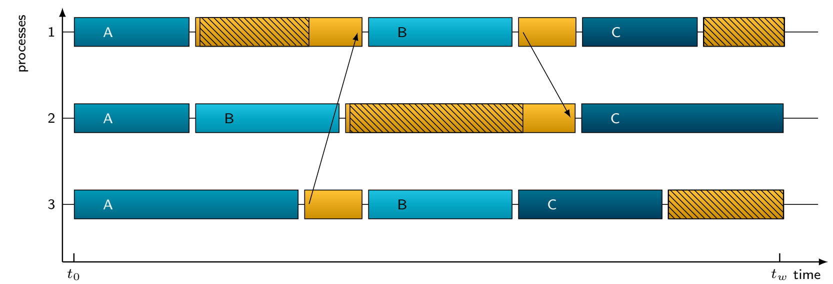


Figure 3.1: Time line model of parallel program execution. Each rectangle represents an activity. Arrows between processes denote communication; the hatched areas inside communication activities represent wait states.

resource is allocated from program start t_0 until program end t_w. This is a reasonable assumption in the context of current HPC systems, reflecting the practice employed by virtually all job schedulers. Thus, the total resource consumption of the program corresponds to the number of processes P multiplied by the total (wall-clock) runtime, $t_w - t_0$. In the following, the term *wall-clock time* refers to fractions of the length of execution, and *allocation time* refers to fractions of the resource consumption. While the wall-clock time is always a value between 0 and t_w, the allocation time can be as large as $P \cdot (t_w - t_0)$. Conceptually, processes that finish earlier than t_w idle in a wait state until the last process finishes (pseudo-synchronization). Additional wait states may occur at synchronization points during execution due to load or communication imbalance.

Boxes on a process time line represent the activities that this process performs. The length of the box represents the duration of the activity. An *activity* in this context corresponds to a single execution of a particular function or otherwise instrumented code section. The activities a typical parallel program performs fall into various categories, such as floating-point and integer operations, memory accesses, I/O, or communication and synchronization operations. For simplicity, we classify these operations into just three super-categories: *work* includes all operations that bring the actual computation forward, *parallelization overhead* comprises extra operations necessary for the parallel execution (particularly communication and synchronization), and *wait states* denote time intervals where processing elements sit idle while they wait to synchronize with another processing element. In Figure 3.1, the colored activity boxes represent work, the gray boxes represent communication or synchronization, and the hatched gray areas represent wait states.

3.2 Load and Communication Imbalance

Now, we can depict the *workload* of a processor in terms of the execution time that it needs to complete a piece of work. Note that the execution time not only depends on the amount of work, but also on the processing speed. The total workload of a program then corresponds to the total allocation time that the processing elements spend performing (useful) work. Ideally, the workload in a parallel program is uniformly distributed across the processors, i.e., each process has the same workload. If this is not the case, a (work)load imbalance occurs. (In the following, the term *load imbalance* is used as a shortcut for *workload imbalance*). Load imbalance is characterized by workload deviation on one or more processes. In a parallel program with P processes, the workload deviation δ_p of a process p corresponds to the difference between the process' workload w_p and the average workload per process:

$$\delta_p = w_p - \frac{1}{P}\sum_{i=1}^{P} w_i$$

Note that the workload deviation can be positive or negative. A positive deviation is sometimes referred to as *overload*, while a negative deviation is (somewhat awkwardly) called *underload*. Load imbalance exists if one or more processes have a non-zero workload deviation. Hence, workload *deviation* is a property of individual processes, whereas load *imbalance* is a global property of the program or sub-program. It is often useful to examine load imbalance for single modules or subroutines individually; however, let us first look at imbalance in the program as a whole.

In comparison to perfectly balanced workload, load imbalance adversely affects performance. This performance impact determines the severity of the imbalance. We can describe the performance impact of imbalance both in terms of its impact on the program runtime (i.e., wall-clock time), and in terms of its impact on resource consumption (i.e., allocation time). Assuming a parallel program with P processes and a single, global synchronization at the end of the execution and no further parallelization overhead, the runtime impact I_t of load imbalance compared to a perfectly balanced execution corresponds to the maximum workload deviation:

$$I_t = \max_p \left(w_p - \frac{1}{P}\sum_{i=1}^{N} w_i \right)$$

As another side of the same medal, the allocation-time impact of the imbalance can simply be expressed in terms of its runtime impact by multiplying it with the number of processes:

$$I_T = P \cdot I_t$$

Note that these simple formulas only apply to trivial cases. However, the following sections describe various effects that influence the performance impact in real-world programs which are not fully covered by these formulas. As a result, determining the actual performance impact of an imbalance usually requires more sophisticated methods.

In addition to (work)load imbalance, programs may also exhibit imbalance in operations that constitute parallelization overhead, especially communication. Besides the pure amount of time spent in communication operations, the amount of data exchanged and the number of communication partners of a process are useful metrics to characterize communication costs and balance. The same properties that characterize load imbalance also apply to communication imbalance. In fact, workload and communication imbalance can be interrelated, and balancing algorithms need to take communication balance as well as workload balance into account.

3.3 Patterns of Imbalance

Imbalance in parallel programs occurs in a large variety of forms. We can study imbalance patterns with respect to the distribution of workload or communication across processors, and in iterative programs also with respect to its evolution and fluctuation over time. The pattern has a significant impact on the severity of an imbalance, and may also complicate its detection. This section demonstrates imbalance patterns that occur in parallel programs and explains their implications. First, we examine load imbalance in real-world example programs in a case study with the SPEC MPI benchmark codes. Then, Subsection 3.3.5 introduces some more complex imbalance patterns specific to MPMD codes.

3.3.1 Experiment Setup

To study imbalance "in the wild", we ran experiments with various codes, specifically those in the SPEC MPI2007 benchmark suite [76] and the PEPC particle physics code [28]. All of the experiments were performed on the Juropa cluster system at the Jülich Supercomputing Centre, a system comprised of 2208 compute nodes with two quad-core Intel Xeon X5570 (Nehalem-EP, 2.93GHz) processors each, 24 GB of memory per node, and a QDR InfiniBand interconnect with a non-blocking fat-tree topology.

The SPEC MPI2007 benchmarks consist of various real-world MPI applications that represent a broad range of application domains and parallelization schemes. We performed runs with 256 MPI processes for each of the eleven SPECMPI applications for which a large reference input data set ("lref") is available, and additionally included the 104.milc benchmark for which only a medium-size configuration is provided.

3.3.2 Distribution across Processes

Examining the distribution of workload (and workload deviation) across processes reveals important insights into the characteristics of a load imbalance. Figure 3.2 visualizes the workload deviation observed for each of the SPEC MPI applications. For each process, a vertical line originating at a virtual, horizontal line that represents the average workload shows the workload deviation on that process. Hence, lines pointing upwards from the average-workload line represent overloaded processes, and lines pointing downwards represent underloaded processes. The green horizontal line on top of each plot shows the total runtime of the program, thus providing a nice indicator of the overall overhead. Large spikes in the imbalance plots suggest severe imbalance, whereas small deviation from the average workload baseline indicates well-balanced workload. We can see that even the most well-balanced programs in the case study (122.tachyon, 125.RAxML, and 126.lammps) still exhibit a small amount of imbalance. Other programs, in particular 104.milc, 121.pop2, and 129.tera_tf have large spikes which indicate severe imbalances.

Figure 3.3 illustrates the workload distribution in the SPEC MPI benchmarks in the form of workload histograms. The filled curves and spikes indicate how many processes finish their

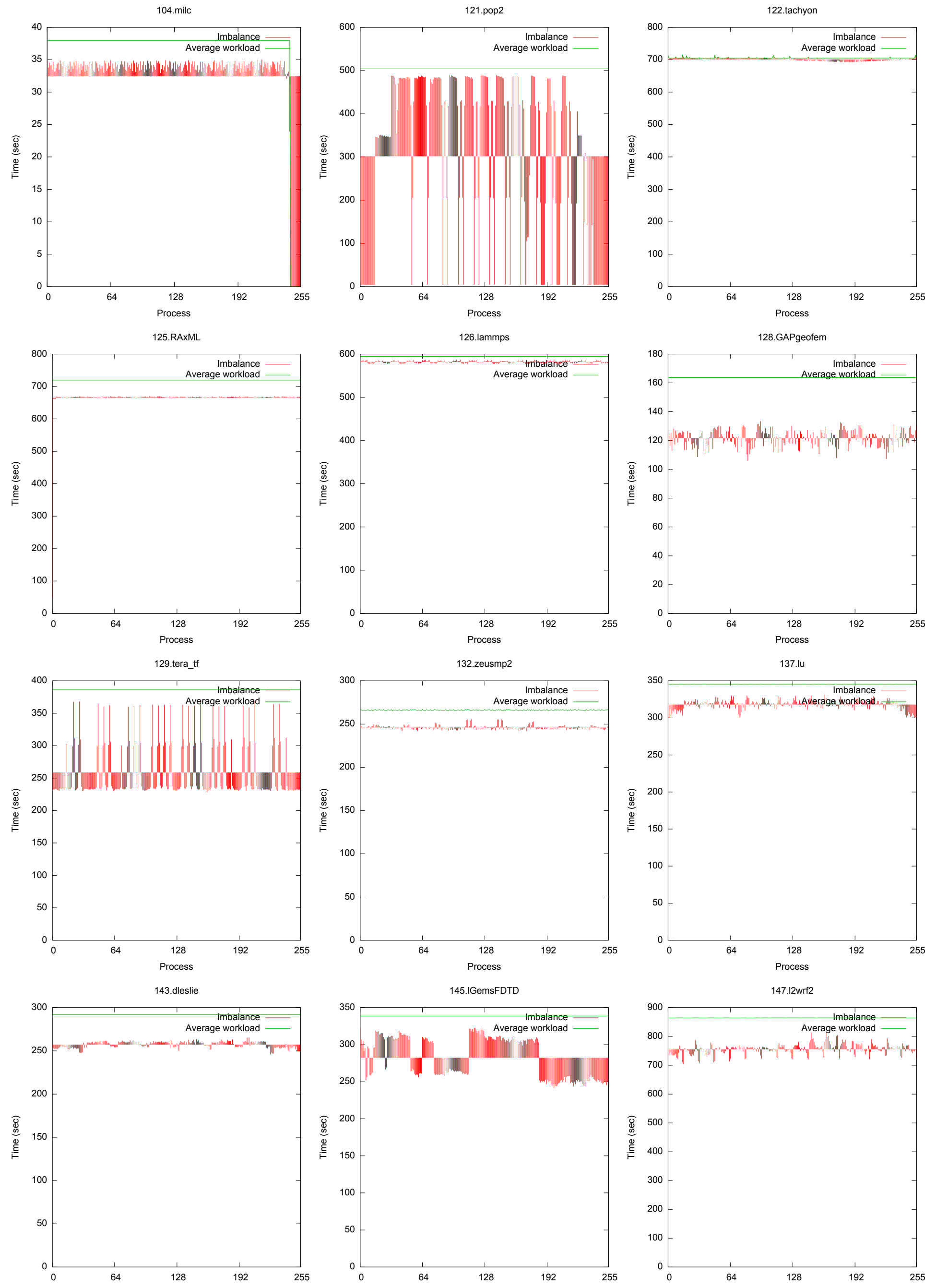


Figure 3.2: Workload imbalance per processes in the SPEC MPI benchmarks.

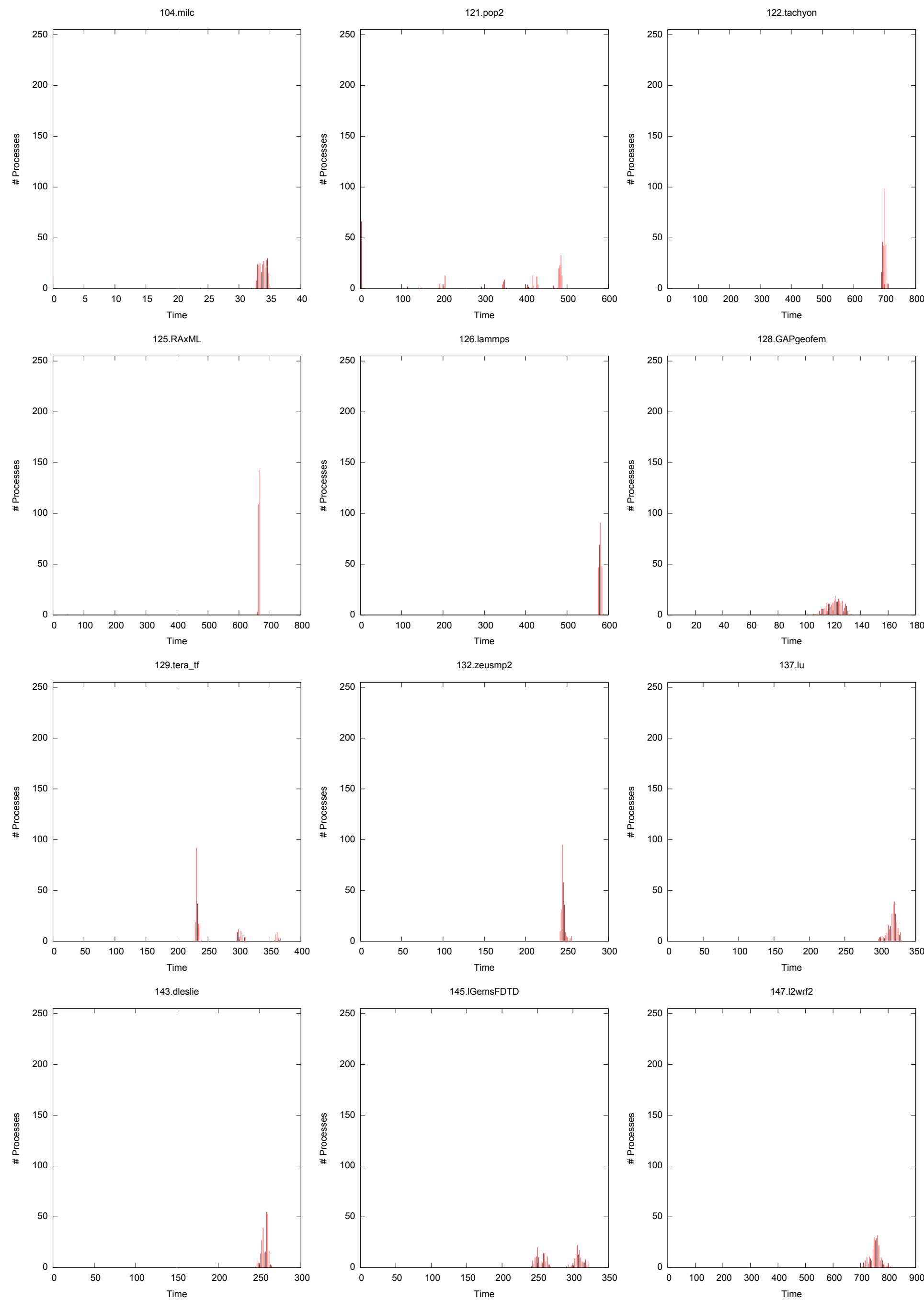


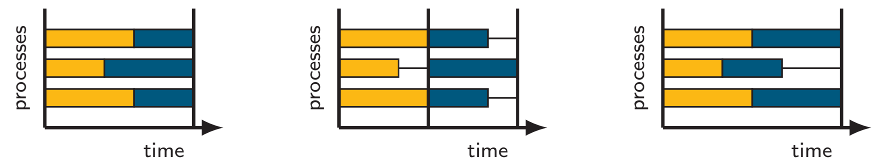
Figure 3.3: Process workload histograms of the SPEC MPI benchmarks.

work in a certain amount of time, which helps to identify clusters and specific distribution patterns. In the workload histograms, a well-balanced workload appears as a single, sharp spike. We can see such a spike for 122.tachyon, 125.RAxML, and 126.lammps. For the remaining programs, we find a few distinct workload distribution patterns. In some cases (128.GAPgeofem, 137.lu, and 147.l2wrf2) the workload is (nearly) normally distributed, which appears as a filled gaussian curve in the histogram plots. The width of the curve then indicates the severity of the imbalance. For the remaining cases, we find multiple distinct clusters or spikes in the workload histograms, which means there are two or more groups of processes with (sometimes vastly) different workloads. In 145.lGemsFDTD, we find two distinct, almost equally large clusters of processes with similar workloads. However, in the other cases, one or more outliers or discrete smaller spikes in the workload histograms indicate significantly different workloads on a minority of the processes. In some cases (104.milc, 121.pop2, and 143.dleslie), that minority is underloaded, while in others (129.tera_tf and 132.zeusmp2) it is overloaded. Most notably, in 129.tera_tf, two small groups of processes have a 25% and a 50% larger workload than the remaining processes, respectively. For 104.milc, the imbalance plot (Figure 3.2) shows that a small group of the allocated processes actually does not participate in the computation at all. Likewise, we find multiple clusters of significantly underloaded processes in 121.pop2. To summarize, we observed the following important workload distribution patterns:

- Normally distributed workload
- Two or more clusters of processes with similar workload, often with
  - a minority of underloaded processes, or
  - a minority of overloaded processes.

The workload distribution pattern is a good indicator of the original cause of an imbalance. Because perfect load balance is almost impossible to achieve, pretty much all parallel algorithms lead to at least some imbalance in a gaussian distribution pattern. There is not much a programmer can do to prevent a minimal amount of imbalance, and, unless it grows too large, also little need to. However, multiple clusters of processes with different workloads typically indicate a deeper algorithmic problem or inefficient domain decomposition. Such an imbalance should be adressed and analyzed. Sections 3.4 and 3.5 discuss causes of load balance and their solutions in detail.

The distribution pattern also has a significant influence on the performance impact of an imbalance. For example, a minority of overloaded processes (or, in the extreme case, only one) can have an enourmous performance impact when other processes idle while waiting for the overloaded ones to finish, whereas the impact of an underloaded minority may be negligible. Moreover, if the maximum workload deviation remains constant, the allocation time impact of an overloaded minority rapidly increases with increasing scale. In contrast, the performance impact of the imbalance may actually decrease with higher scale in case of an underloaded minority.



(a) Cancellation of opposing imbalances

(b) Aggravation of opposing imbalances due to synchronization

(c) Aggravation of similar imbalances

Figure 3.4: Interference of superimposing imbalances in different activities

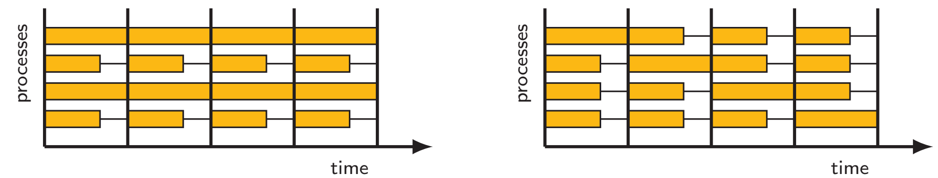
3.3.3 Interference between Imbalances

For performance analysis, it is useful to study not only global imbalance, but identify individual program locations that exhibit imbalance. However, to determine the actual performance impact of an imbalance, it is necessary to take interference between imbalances occuring at different locations into account. The interference patterns shown in Figure 3.4 illustrate why. Similar to constructive or destructive interference effects of superimposing waves, superimposition of imbalances with similar or different workload distribution patterns attenuate or aggravate their overall performance impact. For example, Figure 3.4a shows a case where two imbalances with opposing workload distribution patterns cancel each other out in such a way that neither of the imbalances produces a negative performance impact. However, as Figure 3.4b shows, synchronization breaks the superimposition, so that the performance impact of two imbalances can add up even if they have opposing workload distribution patterns. Obviously, multiple imbalances with a similar workload distribution pattern in different activities also lead to a larger overall performance impact (Figure 3.4c). Because of these interference effects, estimations of the performance impact of individual imbalances need to be based on the actual overall resource waste that occured.

3.3.4 Transformation over Time

The iterative nature of typical simulation codes adds another dimension to the formation and evolution of imbalances: the severity of an imbalance can increase or decrease, and the workload distribution pattern may change over time. Interestingly, these aspects have been largely neglected by previous work. Therefore, workload or communication imbalance distribution patterns that change over time expose a significant weakness in traditional, profile-based approaches to load-balance analysis.

Often, the distribution of workload across processes is determined in the beginning of an iterative program and remains constant over the entire runtime, that is, each process performs the same amount of work in every iteration. In this case, the workload distribution pattern and any potential imbalance therein is *static*. Figure 3.5a illustrates such a static imbalance: the workload distribution is the same in each iteration, that is, each process retains the same amount of workload deviation over the entire runtime. However, for some problems – for example, simulation programs that apply adaptive mesh refinement – the workload processed by each



(**a**) Static imbalance (**b**) Dynamic imbalance

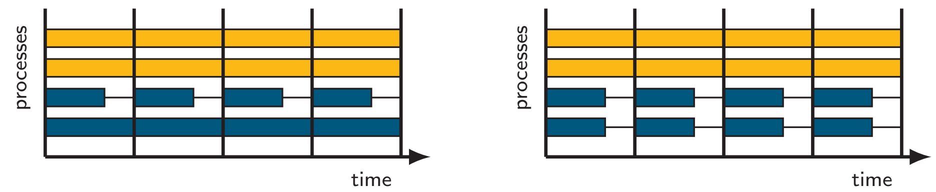
Figure 3.5: Evolution of workload distribution patterns over time. An imbalance is static if the workload distribution pattern remains constant over time, and dynamic if the distribution pattern changes over time.

process may change between iterations. This leads to a *dynamic* workload distribution, where also the workload deviations may be different in each iteration. Analyzing dynamic imbalance can be a challenge: approaches that use profiling may underestimate the performance impact of dynamically changing imbalance, or in rare cases even fail to detect it at all. The (admittedly extreme) example in Figure 3.5b illustrates why. Here, a program performs multiple iterations of a computation with global synchronizations in between. There is one overloaded process in each iteration, but the excess workload shifts to another process in every iteration, so that the total, aggregate workload of each process is the same. An execution profile, where data is aggregated along the time axis, will therefore not indicate any load imbalance at all, in spite of its significant performance impact. More precisely, the profile may reveal that a considerable amount of allocation time is wasted, but it is not possible to relate this information to a specific imbalance. Analyzing dynamic imbalance requires means to trace changes of program behavior over time, which makes it a considerably more complex undertaking than the detection of static imbalance.

An example of a dynamic communication imbalance was found in the PEPC particle physics code [28]. Analyses conducted by Szebenyi et al. [79] using time-series profiles and event traces showed that a small subset of processes engaged in significantly more point-to-point communication operations than others. Moreover, this imbalance grew steadily, and the communication overload moved to neighboring processes over time. Further investigation revealed that an adaptive load balancing scheme employed by PEPC did indeed balance the computational load well, but by doing so assigned a large number of particles to only a few processes, which then induced the communication imbalance.

3.3.5 Imbalance in MPMD Programs

The use of domain decomposition as primary means for work distribution in most tightly-coupled parallel programs facilitates the SPMD parallelization model, where the same program is executed on all processors. However, as programmers need to explore new ways to exploit the growing parallelism, we can assume that more programs begin to adapt hybrid task/data decomposition schemes in MPMD-style models as outlined in Section 1.1.3. In a typical execution model, different parts of a combined multi-physics application are implemented as individual, tightly-coupled SPMD-parallel components, which run in parallel next



(a) Intra-partition imbalance. Imbalance occurs within the blue partition.

(b) Inter-partition imbalance. Imbalance occurs between the yellow and blue paritions, but not within.

Figure 3.6: Intra- and inter-partition imbalance. The process space is partitioned into two (yellow and blue) partitions, which perform different activities.

to each other in individual *process partitions*, and (typically infrequently) exchange data between each other either directly or via a separate coupler component. These MPMD programs exhibit multi-level parallelism, so that we also have to consider imbalance on multiple levels. Imbalance can occur within each process partition, but also on a global level when the wall-clock time spent in individual partitions before an inter-partition data exchange occurs varies. To distinguish these effects, we call imbalance within a process partition *intra-partition imbalance*, and imbalance between partitions *inter-partition imbalance*. Figure 3.6 illustrates these concepts. Inter-partition imbalance (Figure 3.6b) characterizes imbalances between different process partitions, i.e., a (wall-clock) time difference between two partitions before they engage in a global data exchange. In contrast, intra-partiton imbalance (Figure 3.6a) characterizes imbalance within a single SPMD partition.

This extended classification of imbalance in MPMD programs allows detailed conclusions about the causes and nature of the imbalance, in particular when we study the performance impact of imbalance in individual subroutines. Notably, approaches based on the aggregate average execution time per process do not characterize imbalances in individual subroutines correctly in MPMD environments.

3.4 Causes of Imbalance

A variety of reasons can cause load or communication imbalance in parallel programs. On a high level, we can distinguish between *program-internal* and *program-external* causes.

3.4.1 Program-Internal Causes

Program-internal causes of imbalance comprise all factors influencing the workload and communication distribution that are a direct result of the program's own actions, produced either by the algorithms used or the runtime configuration. Most importantly, this includes the choice of the problem decomposition approach (e.g., task or domain decomposition) and its implementation. Scientific and engineering simulation codes often use domain decomposition, and must ensure to decompose the computational domain in a way that places a similar workload on each process. This can be a challenging task. For once, the structure of the computational domain may not be trivially divisible by the exact number of processes available. In many codes, the physical objects being simulated have complex, highly irregular structures that are difficult to decompose equally in the first place, e.g., machine parts in a fluid dynamics simulation, or the earth's land or ocean surface in a weather or climate simulation. Moreover, effects caused by the dynamic physical processes being simulated in the program may shift workload from one part of the computational domain to another, eventually causing a load imbalance or requiring continuous re-balancing as the simulation progresses.

In many cases, communication imbalance can also be attributed to process-internal causes. For example, processes may be assigned a different number of communication partners depending on their position in the simulated domain. A frequent communication imbalance pattern occurs in programs using a regular 1D, 2D or 3D domain decomposition and a stencil communication pattern: depending on the boundary conditions, processes assigned to the border of the computational domain may have fewer communication partners than those assigned to the interior. However, this usually affects only a minority of the processes and has a negligible performance impact. More serious communication imbalances can occur in engineering codes which operate on complex geometries. Here, the decomposition approach used to map the mesh or graph input data onto processes must not only consider the workload balance, but also balance the number of communication links required and the amount of data transferred per process in the resulting decomposition.

Obviously, in addition to load and communication balance, developers must also consider other performance-relevant factors when optimizing a program, such as the communication pattern, memory usage, or implementation complexity. In many cases, one may take a small load imbalance into account in favor of, e.g., a more scalable communication pattern, or a simpler algorithm.

3.4.2 Program-External Causes

Aside from the program's implementation and configuration itself, program-external factors imposed by the hardware or runtime environment may also lead to imbalance. Heterogeneous systems that combine nodes with different processing speeds are an obvious source of imbalance, but such systems are rarely used to run tightly-coupled simulation codes. A parallel program that is expected to run in a heterogeneous environment usually includes measures to compensate for different processing speeds. Even if the amount of data to process is perfectly distributed across a homogeneous machine, imbalance may still occur: often, the actual data values being processed have a strong influence on the workload if, for example, the convergence speed of a numerical algorithm depends on the values being used.

An important cause of program-external imbalance is system noise. System noise is induced by external processes that compete for (CPU) resources with the parallel program. Since HPC systems typically do not run multiple user processes on a single processor, the term usually describes background activities performed by the operating system or runtime environment. When system noise interrupts program execution in an uncoordinated manner at random time intervals, it can have a significant performance impact; especially on programs performing regular collective communication or global synchronizations. This effect was pointed out by Petrini et al. in their now famous 2003 paper “The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q” [69]. There, they found that at large scale, system noise randomly delayed a small number of processes in nearly every iteration, which accumulated to a significant overall delay. In [35], Hoefler et al. conduct detailed simulations to characterize the performance impact of various noise patterns at different scales.

Program-external factors can also influence communication performance and, therefore, cause communication imbalance. Notably, network contention can occur in cluster systems that run multiple jobs simultaneously which share network resources (e.g., links, switches, or even network adapters), so that communication activity in one job can slow down communication in another. The distance between communication partners with respect to the network topology creates a hierarchy of bandwidths and latencies which also affects communication times: message transfers between distant nodes can take significantly longer than communication between processors that are close together (e.g., on the same node) or even use shared caches.

3.5 Eliminating Imbalance

Effective methods and algorithms to partition computational domains among processes so that workload is balanced and communication is minimized are essential prerequisites for creating scalable parallel simulation programs. Since the general problem of optimally distributing workload is NP-hard [68, “Load Balancing”], all practical approaches employ heuristics. Several load balancing algorithms with different compromises in terms of the effort required versus the quality of the resulting workload distribution exist. Common startegies are recursive bisection, space-filling curves, and graph partitioning.

Recursive bisection is a divide-and-conquer method, which splits the object graph into two approximately equal parts, and continues to recursively subdivide each half individually until a minimum partition size is reached. Bisection is a fast way of providing good partitions, and can exploit parallel processing for the partitioning itself. However, the benefit depends on the specific bisection algorithm used, which greatly influence partitioning quality and computational cost.

A space-filling curve is a mathematical function that maps a line onto the unit square, or the entire N-cube in N dimensions. There are many space-filling curves, well-known ones used for load balancing include the the Peano curve, the Morton curve, or the Hilbert curve. The space-filling curve produces a one-dimensional representation of N-dimensional data. Once the data has been linearized, the curve just needs to be split into equally-sized pieces. Moreover, a good choice of space-filling curve preserves locality, and therefore minimize communication costs.

The graph partitioning approach is based on a graph model of the communication pattern of the program, with nodes representing discrete units of work, and weighted edges representing communication. A load-balancing graph-partitioning algorithm splits the graph into (ideally) equally-sized partitions while minimizing the total amount of communication across partition boundaries. Because the general graph-partitioning problem is again NP-hard, heuristics are applied. The advantage of graph-partitioning load-balancing approaches is their flexibility and generality, as object communication graphs can be constructed for any problem, irrespective of the particular application domain. On the downside, graph-partitioning approaches are computationally expensive, particularly for a large number of partitions.

In addition to the balancing algorithm itself, developers also need to decide *when* load balancing needs to be applied. With *static* problems, for example an ocean model in a climate code, which does not change its shape during execution, it is sufficient to determine a balanced workload distribution only once in the beginning. In other cases, particularly engineering problems where areas requiring more work change as the simulation progresses, require a *dynamic* load-balancing strategy, i.e., a re-balancing of workload in regular intervals at runtime. Obviously, these factors also affect the choice of the load balancing algorithm. For static problems, a computationally expensive load-balancing algorithm that creates a high-quality distribution is likely to pay off, whereas the runtime overhead of load balancing needs to be kept at a minimum for dynamic problems.

Even without using advanced load-balancing algorithms in the code, some simple strategies can help mitigating imbalance. Notably, increasing the overall workload or decreasing the number of processes assigned to a problem often reduces the allocation-time impact of load imbalance and improves parallel efficiency. Of course, downscaling is not always desired. An entirely different approach to (quite literally) reduce imbalance costs was presented by Rountree et al. in [72]: based on the notion that some load imbalance is unavoidable, their Adagio system identifies underloaded processes and reduces their processor clock speed. While this approach does not reduce the runtime impact of load imbalance, it attempts to reduce the energy consumption of the system without loss of performance.

Increasing awarness of the impact of program-external sources of imbalance, particularly system noise, led to the design of noise-reduced or noise-free HPC software environments. Some concepts were already applied on the ASCI-Q. This includes obvious optimizations such as deactivating all unnecessary background services, but also more advanced ones such as co-scheduling the execution of the remaining background services between all nodes, so that the background activity runs simultaneously on all processes and does not delay them at random. Modern massively parallel systems such as the Blue Gene or Cray XT/E/K series of Supercomputers often use minimal, noise-free OS kernels instead of a full-featured Linux OS on their compute nodes.

3.6 Related Work

As an important performance factor in parallel programs, load and communication imbalance should be a priority target for the performance analysis of parallel applications. Most of the currently available parallel performance-analysis tools shown in Section 1.2.3 focus on data collection and visualization, and rely on their users' abilities to identify performance bottlenecks and imbalance manually with the help of statistical analyses and graphical visualizations. However, several production and research tools also offer automatic identification and quantification of imbalance. This section provides an overview of previous work and lists specific tools that support the analysis of imbalance, and discusses the general strengths and weaknesses of the traditional profiling and tracing approaches used in current performance-analysis tools with respect to imbalance analysis.

One problem shared by virtually all performance-analysis tools for parallel programs is their inability to quantify the impact of process-external effects. This is because measurement points are bound to the process under observation rather than the processor it is running on. Instrumentation-based approaches take measurements only at events triggered by the process itself, while sampling-based approaches usually take measurements only when the process under observation is actually running. Therefore, profiles or traces never explicitly outline the time not spent in the target process; instead, this time is subsumed in the activities that were interrupted by the operating system. In principle, operating system monitoring is possible, but it requires an instrumented OS kernel, which is typically not available on production-grade parallel systems. In the context of parallel application monitoring, this approach was used in an experimental setup with KTAU [60] to capture kernel performance data. In [61], Nataraj et al. incorporated the KTAU kernel observations into user-level traces of MPI programs to characterize the influence of kernel operations (i.e., system noise) on the parallel application performance.

Calzarossa et al. [13, 14] calculate processor dissimilarities (local imbalances) from per-process summary profiles and use these to derive dispersion indices for activities (work or communication) and code regions (subroutines, loops, or statements). The dispersion indices represent a measure of the imbalance within an activity or code location. They then define scaled indices of dispersion that take the wall-clock time spent in the activity or code location into account, and rank the scaled dispersion indices to identify the most suitable candidates for performance tuning. CrayPat [17] calculates imbalance metrics from summary profiles and visualizes them graphically in a graphical user interface. Notably, their imbalance metrics include the *imbalance percentage*, which represents the relative "badness" of an imbalance; and *imbalance time*, which represents the potential runtime savings (i.e., the runtime impact). Both approaches calculate imbalance metrics for individual program locations. However, they do not incorporate the effects of interference between imbalances in different program locations when characterizating the severity of an imbalance. Tallent et al. [80] avoid this pitfall using an approach that is more related to the root-cause analysis presented in this dissertation, but based on summary profiles instead of event traces. Their work is discussed in more detail in Section 4.5.

Analysis approaches based on summary profiles, as employed by the tools mentioned above, are useful, lightweight instruments to detect (static) imbalance. Statistics derived from per-process workload data (such as maximum, minimum, mean and average workload as well as the standard deviation) provide good indicators for the presence and severity of imbalance. Most importantly, as pointed out in Section 3.2, the difference between maximum and average per-process workload serves as a measure for the runtime impact of an imbalance, assuming the imbalance is static. Moreover, suitable visualizations of the per-process data – such as the imbalance plots or the workload histograms shown in Section 3.3.2 – help in identifying workload distribution patterns. The biggest drawback of profile-based approaches is their inability to determine the performance impact of dynamically changing imbalance accurately.

While static imbalance can be easily identified, the aggregation of performance data over time hides imbalance that dynamically shifts between different processes, so that the performance impact of such an imbalance can be severely underestimated when it is derived from a profile. Recognizing this problem, Huck et al. [37] use phase profiles to uncover otherwise hidden inefficiencies. Because the phase profiles lack information on the amount of wait states, Huck uses the application simulator Dimemas [11] to determine the theoretical "ideal" wall-clock time the program main loop would take if all communication finished instantaneously. The fraction between this ideal execution time and the maximum aggregate per-process workload represents the "micro load imbalance" proportion resulting from dynamic workload shifts that classic summary profiles miss. However, because phase profiles do not retain the exact ordering of events, Huck notes that his simulation approach may be less accurate than trace analysis in the presence of large amounts of point-to-point communication. Moreover, the phase profiling approach requires users to explicitly instrument the program's main loop. Huck's work is one of very few load-balance analysis approaches that supports the analysis of non-SPMD programs: by clustering the performance data before the analysis, he can apply the analysis to each process partition in a composite MPMD program individually. However, examining results from different process partitions separately misses potential imbalance issues that may be introduced by the process partition composition itself.

Approaches using event tracing or phase profiling retain temporal information that allows the identification of dynamic imbalance and determine its performance impact accurarely, but at the cost of higher storage space requirements. To adress the storage implications of event traces capturing the full two-dimensional process-time space, Gamblin et al. [22] apply wavelet transformations borrowed from signal processing to obtain fine-grained but space-efficient time-series load balance measurements for SPMD codes. The resulting data can be visually examined in a graphical 3D browser that allows users to spot both static and dynamic imbalances easily, but the approach does not facilitate a quantitive analysis of the performance impact of an imbalance. It is also limited to the analysis of SPMD codes.

In conclusion, we find that each of the existing load (im)balance analysis solutions lacks in at least one area needed for an all-encompassing imbalance analysis. In particular, quantifying the exact performance impact of dynamic imbalance reliably using a generic approach that is applicable to any parallel program appears to be difficult to accomplish.

3.7 Requirements for Imbalance Analysis

Generic and reliable performance-analysis solutions that support the characterization of imbalance must be able to detect the various forms and patterns of imbalance which may occur in parallel programs, as discussed in this chapter. To summarize, these are

- Workload (and communication) distribution across processes, specifically
  - Normal (gaussian) distribution
  - Overloaded minority
  - Underloaded minority

- Interference effects
  - Cancellation
  - Aggravation
- Temporal variation of workload distribution
  - Dynamic imbalance
  - Static imbalance
- Multi-level imbalance, e.g. in MPMD codes
  - Intra-partition imbalance
  - Inter-partition imbalance

Moreover, determining the severity of an imbalance in a specific subroutine must be based on its actual performance impact in terms of wall-clock or allocation time waste. From these findings, we can extrapolate the following requirements for a reliable, generic automatic imbalance analysis solution:

- Support developers by relating performance waste caused by imbalance to the program locations where it originates from. To assist in the search for the underlying algorithmic causes of an imbalance, make the imbalance pattern distinguishable.
- Quantify the severity of an imbalance in terms of its actual performance impact. In particular, take interference effects of imbalances with opposing distribution patterns into account.
- Take the dynamic shift of imbalance between processes over time in iterative programs into account.
- Present results in a compact, easily understandable, intuitive way.
- Generic applicability: a solution should work for both SPMD and MPMD programs
- Work at large scale; both with respect to the measurement and analysis itself and with respect to the visualization of results.

Creating a tool that follows all the requirements listed above is a challenging task. In fact, while the significance of each of these requirements has also been recognized by other research groups before, currently no single solution fulfills all of the requirements.

Summary

Load and communcation balance is one of the most important parallel performance characteristics, and as such it should be an important agenda item in performance evaluations of parallel programs. Imbalance can emerge from various program-internal or program-external factors. Depending on the cause, imbalance occurs in a variety of patterns, both with respect to the distribution of load across processes and its evolution over time, and to a great extent, this pattern determines the performance impact of the imbalance. However, complex imbalance patterns also complicate the detection and quantification of imbalance. Specifically, simple, profile-based analysis solutions cannot capture dynamically changing imbalance patterns in iterative programs reliably.

To optimally assist developers in identifying imbalance bottlenecks in their codes, performance-analysis tools need to identify and highlight imbalanced activities in the code and accurately quantify their performance impact. A survey of previous work revealed that no existing solution currently fulfills all of these requirements. The delay analysis and the critical-path analysis, which will be presented in the following chapters, close this gap.

Chapter 4

Identifying Root Causes of Wait States

Wait states – periods where processes or threads sit idle at synchronization points – are a good indicator for inefficient parallelism. While wait states are originally caused by workload or communication time differences (delays) prior to synchronization points, propagation effects may spread them across the entire system, and thereby severely increase the performance impact of the original delay. The large distance between symptom and cause of wait states also complicates the search for their root causes. This chapter introduces the delay analysis as a solution to identifying such problems. As a solution to identifying such problems, this chapter introduces the delay analysis, which maps delay costs in terms of waiting time onto the program locations where wait states are caused. The delay analysis revisits an idea from earlier work by Meira Jr. et al. [52, 53], but unlike Meira's approach, the delay analysis highlights the long-term effects of wait state propagation explicitly and works in a much more scalable way.

The chapter is organized as follows: First, Section 4.1 establishes a terminology and a general model to describe the formation of wait states. Section 4.3 outlines the implementation of the delay analysis within Scalasca's event-trace analysis framework. Section 4.4 then presents several case studies that demonstrate the findings and enhanced insight provided by the delay analysis. Finally, Section 4.5 discusses prior and related work.

4.1 A Wait-State Formation Model

Load- or communication imbalance in a parallel program typically leads to wait states at subsequent synchronization points. Moreover, in the presence of complex point-to-point process interactions, a long chain of events may separate wait states from their root causes. Understanding how and where wait states form and propagate is vitally important for finding a reasonable starting point for their reduction. This section introduces the wait-state formation model underlying the delay analysis. It describes the creation of wait states from imbalances (delays) and their propagation through the system, and provides a cost model that attributes the observed waiting time back to their root causes.

While Section 4.2 describes the complete formal model, this section first outlines the basic concepts and terminology using a simple example. Figure 4.1 shows the execution of a parallel

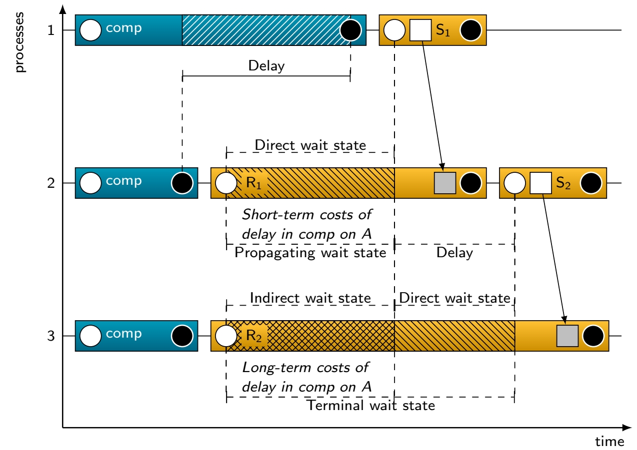


Figure 4.1: Formation of wait states in a parallel program. Regions labeled S and R denote send and receive operations, respectively; hatched areas represent wait states. Delay in `comp` on process 1 causes a wait state in R_1 on process 2, which propagates to R_2 on process 3. The costs of the delay correspond to the sum of the *short-term* costs (for the wait state it causes directly) and *long-term* costs (for the wait states it causes indirectly).

program in the form of a time-line diagram, where the activities performed by a process are shown as rectangles on the time line, and the length of the recangle represents the time spent in the activity. Edges between two process time lines denote communication or synchronization. The example demonstrates how wait states form and propagate, as we will see in the following.

4.1.1 Wait States

Wait states are intervals through which a process sits idle. They occur within communication operations when one process waits for synchronization with another process. Hence, leaving the wait state requires progress on another process. A typical example is the late sender wait state, where the receiver of a message has to wait until the sender initiates the message transfer. The *amount* of a wait state (or *waiting time*) is the length of the interval it covers. In Figure 4.1, both processes 2 and 3 exhibit wait states (shown as hatched areas).

An important effect that contributes to the formation of wait states is *wait-state propagation*: a wait state may itself delay a subsequent communication operation and thereby cause more wait states later on. Hence, wait states originating from a single source can propagate through the entire system. With this propagation effect in mind, wait states can be classified in two different ways depending on where we start analyzing the chain of causation that leads to their formation.

Starting from the cause, we can distinguish direct and indirect wait states. A *direct* wait state is a wait state that was caused by some "intentional" extra activity that does not include waiting time itself. For example, wait state R_1 in Figure 4.1 is a direct wait state because it was caused by excess computation in function `comp` on process 1. However, since the wait state R_1 itself delays the send operation S_2, the excess computation is also indirectly responsible for the wait state in R_2 on process 3, which is therefore denoted as *indirect* wait state. In other words, an indirect wait state is a wait state that was only indirectly caused by "intentional" activities, while its immediate cause is propagation of some previous wait state. On the other hand, process 3 in Figure 4.1 also exhibits a direct wait state from communication imbalance: the actual receipt of the message on process 2 delays the dispatch of the message to process 3.

Looking at wait-state formation starting from the effect, we can distinguish between wait states at the end of the causation chain and those in the middle. A wait state which sits at the end of the causation chain and, thus, does not propagate any further is called a *terminal* wait state. In Figure 4.1, the wait states on process 3 are terminal because they do not propagate further. In contrast, the wait state R_1 on process 2 is a *propagating* wait state because it is responsible for the wait state on process 3. Both classification schemes fully partition the set of wait states, but each in different ways. For example, a terminal wait state can be direct or indirect, but it can never be propagating. Terminal wait states are often found at global synchronization points.

4.1.2 Delay

A delay is the original root cause of a wait state, that is, an interval that causes a process (or thread) to arrive belatedly at a synchronization point, which in turn causes one (or more) other processes or threads to wait. As such, the term "delay" refers to the act of delaying rather than the state of being delayed. A delay is not necessarily of computational nature and may also include communication. For example, an irregular domain decomposition can easily lead to excess communication when processes have to talk to different numbers of peers. However, a delay itself does not include any wait states. Wait states that are direct causes of subsequent wait states would be classified as propagating wait states. In Figure 4.1, delay occurs in function `comp` on process 1, where excess computation leads to a wait state on process 2 at the next synchronization point. In addition, the actual message receipt on process 2 also constitutes a delay.

4.1.3 Costs

A delay may be small compared to the amount of wait states it causes. To identify the delays whose remediation will yield the highest benefit, we need to know their overall influence on waiting time. This notion is expressed by the delay costs: The costs of a delay are the total amount of wait states it causes. However, it is often also useful to study the direct effects of a delay in isolation. Therefore, we distinguish between the amount of wait states a delay causes directly and the amount it causes indirectly through wait-state propagation, and divide the delay costs into short-term and long-term costs. Short-term costs cover direct wait states, whereas long-term costs cover indirect wait states. The total delay costs are simply the sum of the two. The costs are not named "direct" and "indirect" costs because of their established meaning in business administration. In Figure 4.1, the amount of the wait state R_1 constitutes the short-term delay costs of the delay in `comp` on process 1, and the amount of the wait state it indirectly causes in R_2 on process 3 constitutes its long-term delay cost.

Note that the costs of a delay can be much larger than the delay itself. For example, a delay that lets many processes idle simultaneously in a collective operation can incur significant short-term delay costs. In particular, however, wait-state propagation in programs with complex communication chains can lead to huge long-term delay costs, an effect that can be observed in various examples in Section 4.4.

The primary result of the delay analysis is a mapping of the costs of a delay onto the call paths and processes where they occur, offering a high degree of guidance in identifying promising targets for load or communication balancing.

4.2 Delay Cost Calculation

As we have learned so far, the delay costs of a call path describe its overall impact on the formation of wait states. This section shows how exactly delay costs are computed for a generic parallel program.

4.2.1 Program Model

We can model a parallel program as as set of processes P, with each process executing a sequence of activities in parallel. We assume that each process occupies a single processing element exclusively, and executes exactly one activity at any time. A tuple $(p,i) \in P \times \mathbb{N}$ denotes the ith activity executed by process p. The function $\mathrm{Enter} : P \times \mathbb{N} \to \mathbb{R}$ denotes the time an activity started executing, while $\mathrm{Exit} : P \times \mathbb{N} \to \mathbb{R}$ denotes the time at which it finished executing.

An activity represents the single execution of a particular piece of program code, for example, one instance of a function invocation. The program location can be identified through its call path. The function $\mathrm{Callpath} : P \times \mathbb{N} \to C$ determines the call path executed by an activity (p,i) on process p in the set of call paths C.

4.2.2 Synchronization Points and Wait States

For the two-sided communication models studied here, each process participating in a data transfer or synchronization operation must actively invoke a communication activity. Communication activities that are non-local can only complete when another process reaches a corresponding activity. One example is an `MPI_Recv` call, which can only complete after the remote process sent a matching message using one of the `MPI_Send` variants. Ideally, both processes should begin their corresponding communication activities at the same time.

If the completion of an activity (p,i) on process p depends on a remote activity (q,k) being executed on another process q, *and* process p enters (p,i) before process q enters (q,k), these two activities constitute a *synchronization point*. Hence, a synchronization point $S = ((p,i),(q,k))$

is a tuple $(P \times \mathbb{N}) \times (P \times \mathbb{N})$ of two activities (p,i) and (q,k) for which the following conditions hold:

- Completion of activity (p,i) *depends* on the execution of activity (q,k). This property is derived from the semantics of the underlying communication system.
- Activity (p,i) starts earlier than activity (q,k)

$$\text{Enter}(p,i) < \text{Enter}(q,k).$$

- There is some overlap between the two activities

$$\text{Exit}(p,i) > \text{Enter}(q,k).$$

For the time interval between entering (p,i) on process p and entering (q,k) on process q, activity (p,i) is in a *wait state*. The amount of waiting time in activity (p,i) – denoted as $\omega(p,i)$ – is given by

$$\omega(p,i) = \text{Enter}(q,k) - \text{Enter}(p,i).$$

4.2.3 Synchronization Intervals

A wait state occurs when one process enters an activity on which another process depends late. That is, the wait state in activity (p,i) at a synchronization point $S = ((p,i),(q,k))$ occurs because process q starts executing activity (q,k) later than process p starts executing (p,i). Anything that led process q to arrive late must have occured in the interval before synchronization point S, but after the previous synchronization point between the same two processes.

An interval between two subsequent synchronization points is called a *synchronization interval*. Formally, a synchronization interval $\zeta_{(S',S)} = ((p,i',i),(q,k',k))$ is a composite tuple $((P \times \mathbb{N} \times \mathbb{N}) \times (P \times \mathbb{N} \times \mathbb{N}))$ for which the following conditions hold:

- $S' = ((p,i'),(q,k'))$ and $S = ((p,i),(q,k))$ are synchronization points between processes p and q
- S' and S are subsequent synchronization points, that is, there is no other synchronization point between the same two processes within $\zeta_{(S',S)}$

$$\nexists S'' = \big((p,i''),(q,k'')\big) \text{ where } i' < i'' < i \wedge k' < k'' < k.$$

All direct causes of the wait state in synchronization point $S = ((p,i),(q,k))$ will be found within the corresponding synchronization interval $\zeta_{(S',S)} = ((p,i',i),(q,k',k))$ on process q.

4.2.4 Causes of Wait States

We distinguish two types of direct wait-state causes: delays and wait-state propagation. However, it is in general not possible to isolate the specific cause(s) of the wait state. Therefore, we assign the blame for a wait state equally to all possible wait-state causes (i.e., delays and propagating wait states) in the synchronization interval, with each identified cause receiving an amount of blame that is proportional to its severity. The following subsections explain in detail how propagating wait states and delays are characterized.

Wait-state propagation

A propagating wait state is a wait state that itself delays subsequent communication. Thereby, waiting time propagates to further wait states later on. In the synchronization interval $\zeta = ((p,i',i),(q,k',k))$ associated with a wait state in activity (p,i), wait states that occur on process q (i.e, in activities (q,l) where $k' < l < k$ and $\omega(q,l) > 0$) as a result of communication with other processes are considered *propagating wait states*, because they delay process q's arrival at the synchronization point $((p,i),(q,k))$. As wait states do not perform any "useful" work, we consider propagating wait states to be direct causes of the resulting wait state.

Delays

Delays represent non-waiting excess times on one process within a synchronization interval that lead to a wait state. For the synchronization interval $\zeta = ((p,i',i),(q,k',k))$, the combined excess time in activities on process q which is not spent in similar activities on process p represents delay which contributes to the wait state in (p,i).

Two processes can perform any number or kinds of activities within the synchronization interval. For example, a process q may execute shorter, but more activities within the synchronization interval than process p, which in total takes more time than on p. Also, the precise position of an activity within the synchronization interval is irrelevant for its contribution to the delay: activities at the beginning of a synchronization interval may just as well be a cause of the wait state than those at the end. Therefore, delay is not a property of the individual activities in the synchronization interval, but of the program locations (i.e., call paths) that are executed.

Direct delays in a synchronization interval $\zeta = ((p,i',i),(q,k',k))$ are, in essence, all call paths whose execution without wait state takes more time on process q than on process p. This includes load imbalance, where process q simply spends more time in a call path than process p, but also additional call paths which are only executed on process q but not on p within the synchronization interval.

The delay of a call path in a synchronization interval is determined by comparing the execution time of that call path within the synchronization interval on both processes. Waiting time is excluded from the execution time comparison and instead categorized as propagating waiting time. Therefore, the execution time of a call path c within an interval of activities $[i',i]$ on process p, denoted as $d_{(p,i',i)}(c)$, is given by

$$d_{(p,i',i)}(c) = \sum_{j=i'+1\dots i-1 \,|\, \mathrm{Callpath}(p,j)=c} \mathrm{Exit}(p,j) - \mathrm{Enter}(p,j) - w_{(p,j)} \tag{4.1}$$

The excess execution time $\delta_\zeta(q,c)$ of a call path c within the synchronization interval $\zeta = ((p,i',i),(q,k',k))$ is then the (positive) difference between the total execution time of c on process q and process p:

$$\delta_\zeta(q,c) = \begin{cases} d_{(q,k',k)}(c) - d_{(p,i',i)}(c) & \text{if } d_{(q,k',k)}(c) > d_{(p,i',i)}(c) \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

Note that the execution time difference $d_{(q,k',k)}(c) - d_{(p,i',i)}(c)$ can be negative for some call paths. This happens when the delaying process q spends less time in these call paths than the waiting process p, while the combined time of all activities executed on q in the synchronization interval still exceeds that of process p. Since the execution time difference in these call paths can obviously not contribute to the waiting time (instead, it only mitigates higher excess times in other call paths), they are disregarded as wait-state causes. In contrast, any call path with an excess execution time greater than 0 within ζ constitutes delay, and is therefore considered a direct cause of the wait state in (p,i).

4.2.5 Delay Costs

While the excess execution time $\delta_\zeta(q,c) > 0$ tells if call path c is a delay within synchronization interval ζ, it does not yet represent the actual amount of waiting time for which the delay is responsible. There are primarily three reasons for this:

- The effect of the excess time in call path c may be mitigated by excess time in another call path on the waiting process.
- The same excess time in an activity can be responsible for wait states on other processes as well (this is particularly important for collective communication).
- The excess time does not cover the indirect effects through wait-state propagation onto wait states further away.

To accurately reflect the contribution of each program location to the formation of wait states, we map *delay costs* equal to the amount of waiting time they cause in a specific way onto the program locations where excess time in a synchronization interval leads to wait states. Moreover, to distinguish direct effects of excess time on the wait state at the end of the synchronization interval and indirect effects through propagation of that wait state, we divide the costs into *short-term* and *long-term* costs. Short-term costs cover the direct effects, long-term costs cover the indirect effects. The total delay costs match the entire amount of waiting time in the program, and their distribution reflects the amount to which delays in each program location are responsible for wait states. Hence, the delay costs represent a mapping from a process/call-path location to the amount of delay costs in that location:

$$\text{Short-term cost} : P \times C \to \mathbb{R}$$
$$\text{Long-term cost} : P \times C \to \mathbb{R}$$

Delay costs are proportionally distributed among the call paths with delay found in a synchronization interval in such a way that the sum of all direct wait-state causes (i.e., propagating wait states and delays) in that interval matches the corresponding target cost. The short-term target cost for a synchronization interval belonging to the wait state in activity (p,i) is the waiting time $\omega(p,i)$ itself. The long-term target cost for the interval is the wait state's *propagation costs* $\varphi(p,i)$, which represents the total amount of waiting time caused by propagation of the wait state in (p,i) later on.

The total delay costs of a call path c on process p are the sum of the costs of (q,c) in each synchronization interval ζ where (q,c) constitutes delay (i.e., $\delta_\zeta(q,c) > 0$). The costs are calculated as follows:

$$\text{Short-term cost}(q,c) = \sum_{\zeta=((p,i',i),(q,k',k))} \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}\, \delta_\zeta(q,c)\ \omega(p,i)$$
$$\text{Long-term cost}(q,c) = \sum_{\zeta=((p,i',i),(q,k',k))} \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}\, \delta_\zeta(q,c)\ \varphi(p,i)$$

The fraction $\frac{1}{\hat{\delta}_\zeta+\hat{\omega}_\zeta}$ is a scaling factor to map the length of each individual delay proportionally (with respect to all direct wait-state causes) to the target costs. Here, $\hat{\delta}_\zeta$ and $\hat{w}_\zeta$ represent the aggregated length of all delay excess times and the total length of all wait states in synchronization interval $\zeta = ((p,i',i),(q,k',k))$, respectively:

$$\hat{\delta}_\zeta = \sum_{c\in C} \delta_\zeta(q,c) \qquad \hat{\omega}_\zeta = \sum_{l=k'+1}^{l<k} \omega(q,l)$$

The propagation costs $\varphi(q,l)$, which are used to determine the long-term delay costs in the synchronization interval assigned to a wait state in an activity (q,l), represent the sum of both the direct and indirect contributions of that wait state to all further wait states later on. To calculate the propagation costs, we therefore recursively aggregate the short-term and long-term contributions of this wait state in all synchronization intervals where it constitutes a propagating wait state:

$$\varphi(q,l) = \sum_{\zeta=((p,i',i),(q,k'<l,l<k))} \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}\, \omega(q,l)\ (\omega(p,i) + \varphi(p,i))$$

Propagation costs are assigned to propagating wait states in a similar fashion as delay costs are assigned to delays. However, while short-term and long-term delay costs correspond to the waiting time $\omega(p,i)$ or the propagation costs $\varphi(p,i)$ of the affected synchronization intervals, respectively, the propagation costs of the propagating wait state correspond to the sum of the two, and thereby capture the propagating wait state's direct and indirect effects. The recursive nature of calculating propagation costs from propagation costs of the newly caused wait states thus provides the means to incorporate long-distance effects in the calculation of long-term delay costs.

4.2.6 Wait-state Propagation Characteristics

To make wait-state propagation itself observable, we also classify wait states by their position in the propagation chain. Therefore, we use two different wait-state classifications:

- A partition into propagating versus terminal wait states indicates which wait states propagate further and which do not.
- A partition into direct versus indirect wait states indicates how much waiting time is a result of wait-state propagation, and how much is directly caused by delays.

The following subsections show how these partitions are determined.

Propagating and terminal waiting time

The propagating waiting time reflects the amount of waiting time to which a wait state itself is directly responsible for wait states later on. As explained earlier, the "blame" for a wait state is distributed proportionally among all its potential direct causes in the associated synchronization interval. Therefore, for a wait state in an activity (p,i) with the associated synchronization interval $\zeta = ((p,i',i),(q,k',k))$ which is partially caused by a propagating wait state in activity (q,l), $k' < l < k$, a specific portion of the waiting time in activity (q,l) is classified as propagating waiting time. Similar to the calculation of short-term delay costs in the synchronization interval, we use the scaling factor $\frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}$ to determine this portion.

Since a wait state can constitute a propagating wait state in multiple synchronization intervals, the final portion of propagating waiting time assigned to this wait state is the maximum portion of propagating waiting time determined for it in any synchronization interval. Hence, for a wait state in an activity (q,l), the portion of propagating waiting time is calculated as follows:

$$\text{Propagating waiting time}(q,l) = \max_{\zeta=((p,i',i),(q,k'<l,l<k))} \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}\ \omega(q,l)\ \omega(p,i)$$

The portion of waiting time which is not propagating waiting time is classified as terminal waiting time, as is the entire waiting time in any wait state that does not propagate at all.

Direct and indirect waiting time

The partitioning of waiting time into direct and indirect waiting time indicates to which amount a wait state's direct causes were delays or wait-state propagation, respectively. This partitioning is again determined by the principle of distributing the blame for the wait state proportionally among all direct wait state causes. Hence, to calcualte the fraction of indirect waiting time (i.e., the portion caused by wait-state propagation) in a wait state in an activity (p,i), we apply the scaling factor $\frac{1}{\hat{\delta}_\zeta+\hat{\omega}_\zeta}$ of the associated synchronization interval $\zeta = ((p,i',i),(q,k',k))$ to the total amount of waiting time ω_ζ on the delaying process q in ζ. To obtain the absolute amount of indirect waiting time in activity (p,i), we then apply this factor to the waiting time $w(p,i)$:

$$\text{Indirect waiting time}(p,i) = \frac{1}{\hat{\delta}_\zeta + \hat{\omega}_\zeta}\, \hat{\omega}_\zeta\ \omega(p,i)$$

Accordingly, any waiting time which is not indirect waiting time is classified as direct waiting time.

4.3 A Scalable Delay Analysis Approach

The reference implementation of the delay analysis is based on Scalasca's trace analysis framework, utilizing its parallel trace replay technique. Other than the simple wait-state search performed so far, the delay detection and cost aggregation requires a significantly enhanced, multi-step analysis process. This section describes the underlying algorithms and concepts used in the reference implementation, in particular the extended trace-replay workflow and the backward replay step that performs the delay detection and cost accounting. Finally, it discusses the presentation of the delay analysis results in Scalasca's report visualization component.

4.3.1 Trace Replay

To ensure scalability, the delay analysis follows the same parallelization strategy as Scalasca's pure wait state analysis does, leveraging the principle of replaying the communication operations recorded in the trace in parallel. The basic principle has already been outlined in Section 2.5 and is explained in detail in [26]. Here, we examine the specific characteristics of the algorithm that are important for the discussion of the delay analysis implementation.

The trace replay runs on the same number of processes as the target program. Specifically, the analysis of an MPI program will run with the same number of MPI ranks as the original program. Each MPI rank of the analysis program processes the trace file of the corresponding rank in the target program. It does not have direct access to a trace of another rank. The analysis processes can exchange data at communication events encountered in the trace. For example, at an MPI collective communication event, the corresponding analysis processes can exchange data. At point-to-point communication events, the analysis process processing the original sender's trace sends a message to the rank processing the original receiver's trace, using similar message parameters (communicator, destination rank, and tag) but different data.

The trace replay can be performed either in forward or backward direction. Unsurprisingly, a forward replay processes the trace from the beginning to the end. In contrast, a backward replay processes a trace backward in time, from its end to its beginning, and reverses the roles of senders and receivers. Hence, for MPI point-to-point events in the trace, messages are now sent from the original receiver to the original sender. The parallel backward replay was originally applied in Scalasca in the context of the scalable timestamp synchronization [7]. As we will see in the following, the backward replay also suggests an elegant solution for the calculation of delay costs.

4.3.2 Delay Detection

Other than the pure wait state analysis, the delay analysis requires multiple replay passes over the trace. First, a set of setup passes performs preparatorial steps required for the delay analysis, in particular the synchonization point detection. The actual delay analysis is then performed in a backward replay. Starting from the endmost wait states, this allows delay costs to travel from the place where they materialize in the form of wait states back to the place where they are caused by delays. Finally, an additional pass classifies wait states with respect to wait-state propagation into the classes direct vs. indirect and propagating vs. terminal, respectively. Overall, the entire analysis now consists of several replay passes:

1. Three preparatorial replays that perform the wait-state detection and annotate communication events with information on synchronization points and waiting times incurred.
2. The backward replay that performs the actual delay analysis. For all wait states detected during the preparatorial replays, it identifies and classifies the delays causing them and calculates their cost.
3. A post-processing forward replay that divides wait states into the propagating vs. terminal and direct vs. indirect classes.

Synchronization-point detection

Figure 4.2 presents a detailed overview of the five replay passes. During the first three replay passes, the analysis processes annotate (i) each wait state with the corresponding amount of waiting time, and (ii) each synchronizing MPI event (synchronization point) with the rank of the remote process involved. The annotations will be needed later to identify the synchronization intervals where delays occured.

A synchronization point is a communication event of synchronizing nature, that is, any communication event where a wait state occurs on at least one of the participating processes. For the delay analysis to work correctly, synchronization points must be marked on each process participating in a synchronizing communication operation. Note that any information aquired on one process is not automatically available to other processes, but has to be transferred there

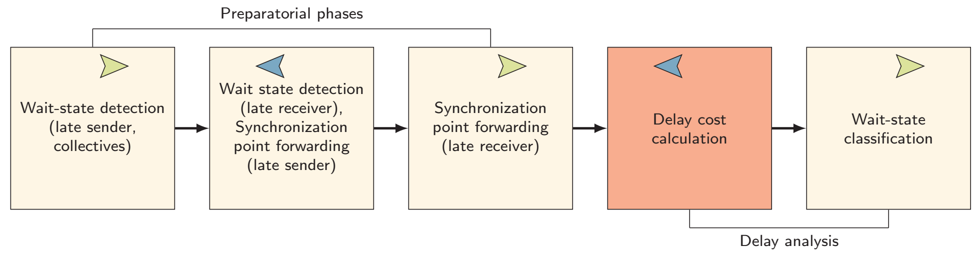


Figure 4.2: Trace analysis replay passes performed for the delay analysis. The green and blue darts within the boxes indicate wether the stage performs a forward or backward replay. The box marked in red indicates the backward replay stage that performs the delay detection and cost accounting.

explicitly. Moreover, point-to-point data transfers can only follow the original communication paths found in the trace along the direction (forward or backward) of the current replay. Therefore, annotating a point-to-point operation as a synchronization point requires two replay passes: one *wait-state detection* pass to identify and mark the wait state on the process where the wait state occured, and a second *synchronization-point forwarding* pass in opposite direction to mark the corresponding communication event on the process where the wait state was caused. Because late-receiver wait states are identified in a backward replay, a total of three replay passes is required to annotate all point-to-point synchronization points. Of the three preparatorial replay phases, the first (forward) one annotates communication events that incur or inflict wait states in MPI collective communication and those that incur point-to-point late-sender wait states. The second (backward) replay annotates synchronization information to communication events that inflict late-sender wait states as well as those that incur late-receiver wait states, and the third (forward) replay finally annotates those events that inflict late-receiver wait states.

Delay identification

The actual delay analysis is performed during the backward replay in the fourth analysis stage, which is marked in a darker color in Figure 4.2. Whenever the annotations indicate a wait state identified during the preparatorial stage, the algorithm determines the corresponding synchronization interval, identifies the delays and propagating wait states causing the waiting time, and calculates the short-term and long-term delay costs.

As defined in Section 4.2, a *synchronization interval* covers the time between two consecutive synchronization points of the same two ranks where runtime differences can cause wait states at the end of the interval. Whereas the communication event associated with the wait state marks the end point of the interval, its beginning is defined by the previous synchronization point involving the same pair of ranks. As the communication operations are reenacted in backward direction in the course of the algorithm's execution, the costs are successively accumulated and transferred back to their source.

Figure 4.3 illustrates the delay analysis for a late-sender wait state. The example exhibits a delay in activity $\texttt{comp}_2$ and a propagating wait state induced by some influence external to

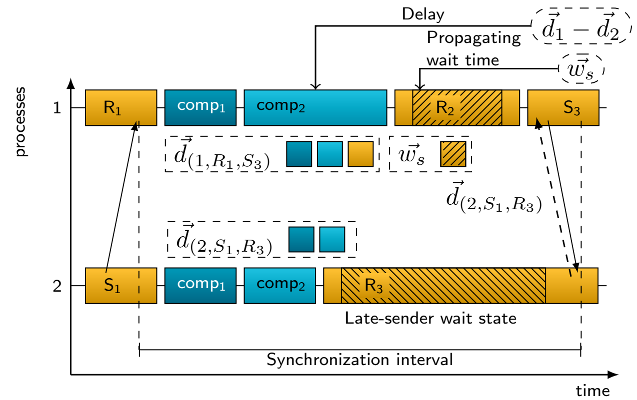


Figure 4.3: Delay detection for a late-sender wait state via backward replay. Original messages are shown as solid arrows, whereas messages replayed in reverse direction are shown as dashed arrows.

the scene in communication activity R_2 of the sender (i.e., process 1). This causes a late-sender wait state in the communication activity R_3 of the receiver (i.e., process 2). To identify the delay during our analysis, both sender and receiver determine the total accumulated time except waiting time spent in each call path within the synchronization interval, as depicted by Equation (4.1). The communication operations at the end of the interval are excluded. These time durations are stored in *time vectors* $\vec{d}_{(1,R_1,S_3)}$ on the sender and $\vec{d}_{(2,S_1,R_3)}$ on the receiver, respectively. In addition, the sender also determines its waiting-time vector $\vec{\omega}_s$, which contains the amount of waiting time in each (communication) call path visited on the sender within the interval. This is necessary to distinguish delay from propagating wait states. The receiver sends its time vector $\vec{d}_{(2,S_1,R_3)}$ via the reversed communication (dashed arrow) to the sender, which calculates the delays for each call-path region by subtracting the corresponding values in the time vectors, as described by Equation (4.2), and stores the results in the delay vector $\vec{\delta}$. The delay vector now contains the execution time differences (excluding waiting time) between sender and receiver for all call paths that exhibit delay. In the subsequent steps, the algorithm calculates the actual delay costs for the current synchronization interval.

Delay cost accounting

In the next step, the algorithm determines the short- and long-term costs of the detected delay and maps them onto the (call path, process) tuples where the delay occurred. For the example in Figure 4.3, the short-term costs simply correspond to the amount of direct waiting time incurred by process 2 in R_3. The amount of direct waiting is obtained by dividing the overall waiting time in R_3, which is transferred to process 1 during the backward replay, into direct and indirect waiting time at the ratio of the amount of delay in $\hat{\delta}$ versus the amount of waiting time in $\hat{\omega}_s$, respectively. The short-term costs are then mapped onto the delaying call paths by distributing the amount of direct waiting time proportionally across all call paths involved in the delay. Likewise, propagating waiting time is mapped onto the call paths suffering wait

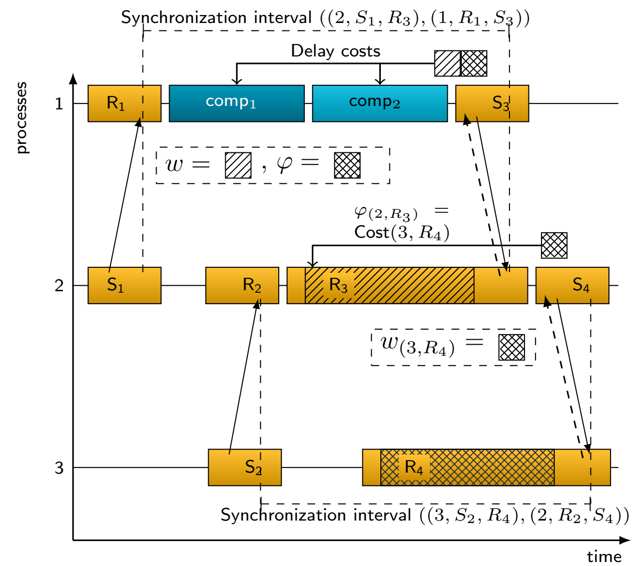


Figure 4.4: Source-related accounting of long-term costs via backward replay and successive accumulation of indirectly induced waiting time. The waiting time $w_{(3,R_4)}$ first travels to its immediate cause, wait state R_3 on process 2, from where it is propagated further to its ultimate cause, the delay on process 1.

states in the synchronization interval on process 1 by proportionally distributing the amount of indirect waiting time across the call paths in $\vec{\omega}_s$.

To calculate the long-term costs of the detected delay, we need to know the total amount of waiting time that was indirectly caused via propagation. Therefore, communication events where waiting time was detected are further annotated with a propagation factor φ, which represents the costs indirectly caused by this wait state later on. These propagation factors are initialized with zero and updated in the course of the backward replay. The long-term costs are propagated backwards by transmitting the propagation factor of a wait state back to the delaying process, where it is used to calculate long-term delay costs and to update the propagation factors of wait states present in the synchronization interval. In this way, the delay costs are successively accumulated as they travel backward through the communication chain until they reach their root cause(s). Hence, we can accurately incorporate distant effects into the calculation of the overall delay costs in a highly scalable manner.

The more complex example in Figure 4.4 illustrates the data flow necessary to accomplish the source-related accounting of long-term costs. Here, delays in region instances $\texttt{comp}_1$ and $\texttt{comp}_2$ on process 1 cause a wait state in R_3 on process 2, which in turn delays communication with process 3, resulting in another wait state in R_4 on process 3. The backward replay starts at the wait state in R_4 on process 3. The waiting time $w_{(3,R_4)}$ of this wait state is transmitted to process 2 via reverse communication. There, the propagation factor $\varphi_{(2,R_3)}$ of the wait state in R_3 is updated to account for the amount of waiting time caused by its propagation. Next, both R_3's waiting time $w_{(2,R_3)}$ and its propagation factor $\varphi_{(2,R_3)}$ are transferred to process 1, where they are mapped onto the initial delay in $\texttt{comp}_1$ and $\texttt{comp}_2$ in synchronization interval $((2,S_1,R_3),(1,R_1,S_3))$. The waiting time $w_{(2,R_3)}$ represents the short-term costs, and the propagation factor $\varphi_{(2,R_3)}$ represents the long-term costs.

Cost accounting for MPI collective communication

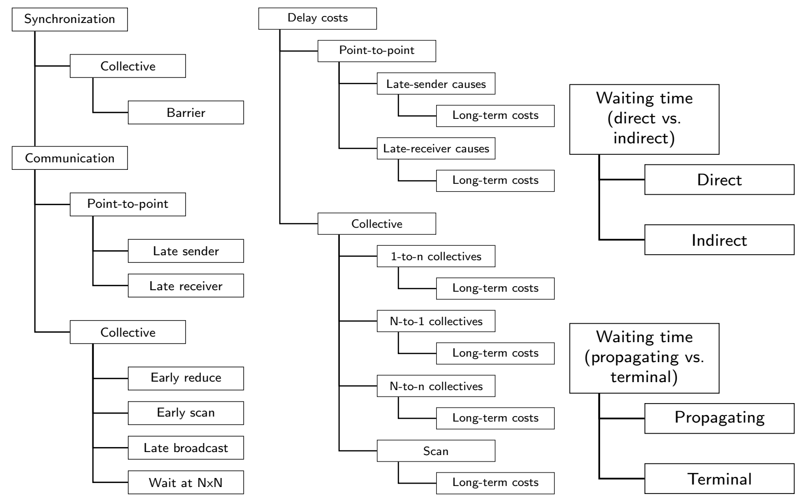
The general principle of the backward-replay based accounting method also applies to collective operations, but with some subtle differences.

Since MPI does not specify explicit synchronization semantics for most collective communication operations, the delay analysis follows implicit synchronization semantics that can be derived from the MPI standard, in the same way as Scalasca's pure wait-state analysis (Section 2.4.1). Hence, for n-to-1 and n-to-n communication and synchronization operations, such as barrier, all-to-all or (all)gather/(all)reduce, delay costs are assigned to the last process that enters the operation (because the operation can only complete after all processes have entered it). For 1-to-n communications (broadcasts), delay costs are assigned to the root process of the operation (because all processes have to wait for the root to send its data, but may exit before all other processes have received the data). In contrast to the point-to-point case, the time vector of the delaying process is broadcast to all processes participating in the operation. Now, every process determines the delaying call paths and calculates the delay costs for the amount of waiting time occurring locally on that process. The individual cost contributions are then accumulated in a reduction operation and finally assigned to the delaying process.

4.3.3 Visual Representation of Delay Costs

As already outlined in Section 2.2, Scalasca stores performance data that was measured or inferred from the automatic trace analysis in a three-dimensional [metric, call path, process] structure. Metrics are organized in a hierarchy, with each additional level providing further detail to a more general metric. A prominent example is the time hierarchy. On the top level, this hierarchy encompasses the entire allocation time used by the program. A sub-level denotes the portion of allocation time spent in MPI communication, which is then further subdivided into the time spent in point-to-point or collective communication. The waiting times detected by the wait-state analysis are also incorporated into this hierarchy. Figure 4.5a shows the organization of the MPI waiting time subhierarchy of a Scalasca wait-state analysis report. Each wait-state pattern (late-sender, late-receiver, late broadcast, etc.) is represented by an individual metric node.

Since the assignment of delay costs onto delaying call paths and threads and the classification of wait states into either direct/indirect or propagating/terminal ones is orthogonal to the performance pattern subdivision in the existing time hierarchy, the new metrics calculated by the delay analysis form additional metric hierarchies next to the existing ones. While the wait-state patterns in the time hierarchy mark the program locations where the wait states occur, the delay cost metrics mark the program locations where the delays that cause them occur. The overall amount of delay costs is equal to the total amount of wait states. Because delay costs directly complement the wait states, it is useful to subdivide delay costs according to the



(a) Wait-state hierarchy **(b)** Delay cost hierarchy **(c)** Wait-state classification

Figure 4.5: Metric hierarchies of wait states and associated delay costs in Scalasca report displays.

wait-state patterns (late sender, late receiver, etc.) that Scalasca distinguishes. This way, users can easily see which delays are responsible for which type of wait state. The resulting delay cost hierarchy (Figure 4.5b) reflects this subdivision. Furthermore, it distinguishes long-term and short-term costs of the delays responsible for each wait-state pattern. When a delay metric node is expanded, the node itself shows the short-term delay costs, while the long-term costs are shown explicitly in the “Long-term costs” sub-nodes. Users can collapse the sub-nodes to obtain the overall delay costs assigned to a program location.

The classification of wait states into propagating vs. terminal and direct vs. indirect wait states is shown in additional metric hierarchies (Figure 4.5c). Because the two classification schemes are orthogonal (since they divide the same wait states in different ways), both form independent top-level metric hierarchies. For simplicity reasons, these hierarchies are not very deep – in fact, they only have two sub-nodes each (direct/indirect and propagating/terminal, respectively), according to the scheme.

Figure 4.6 shows the representation of delay costs in Scalasca’s report browser Cube. The left pane contains (among other metric hierarchy) the delay cost hierarchy. The center pane shows which call paths exhibit the delay costs selected on the left. Finally, the right pane illustrates the distribution of these delay costs across the processes. The values for each process can be listed in text form in a simple system hierarchy tree, or color-coded in a graphical display of either the machine topology or, for applications that arrange their processes in a logical one-, two-, or three-dimensional grid, the logical application topology. Often, the logical topology display helps relating a performance-data distribution pattern to specific concepts in the underlying algorithm. In the example in Figure 4.6, the arrangement of processes in the virtual three-dimensional topology used by the application illustrates that delays located in a spherical region in the center of the virtual domain exhibit particularly high delay costs.

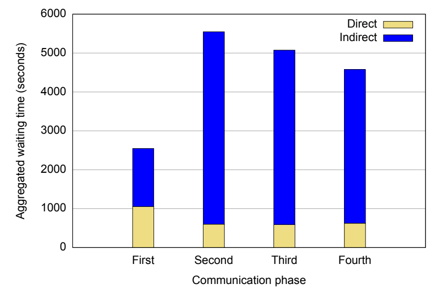
Figure 4.6: Visualization of delay costs in the Scalasca report browser. The left pane shows the delay cost hierarchy, where the current selection marks the short- and long-term costs of delays causing late-sender wait states. The middle pane shows the selected delay costs in all call paths. The right pane shows the distribution of delay costs for the selected call path across all processes.

4.4 Evaluation

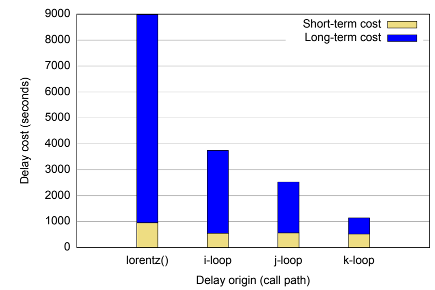
The delay analysis has been implemented in a fully working prototype within the Scalasca framework and applied to various HPC simulation codes. Technical characteristics of the underlying trace replay approach, which is used by both the delay and critical-path analysis, will be discussed later in Chapter 6. The case studies presented in this section highlight the insights the delay analysis offers into the formation of wait states and their root causes. We examine three different MPI codes: the astrophysics simulation code Zeus-MP/2 [32], the sea ice model of the Community Earth System Model [85], and the plasma-physics code "Illumination" [27, 28]. All measurements were taken on the 72-rack IBM Blue Gene/P computer Jugene at the Jülich Supercomputing Centre.

4.4.1 Zeus-MP/2

The first case study is the astrophysical application Zeus-MP/2. The Zeus-MP/2 code performs hydrodynamic, radiation-hydrodynamic (RHD), and magnetohydrodynamic (MHD) simulations on 1, 2, or 3-dimensional grids. For parallelization, Zeus-MP/2 decomposes the computational domain regularly along each spatial dimension and uses non-blocking MPI point-to-point communication to exchange data between neighboring cells in all active directions of the computational domain. This study examined the 2.1.2 version on 512 processes simulating a three-dimensional magnetohydrodynamics wave blast, based on the "mhdblast_XYZ"



(a) Composition of waiting time in different communication phases of Zeus-MP/2



(b) Short- and long-term costs of delays in four dominant call paths of Zeus-MP/2

Figure 4.7: Distribution of short- and long-term delay costs and direct and indirect wait time in Zeus-MP/2.

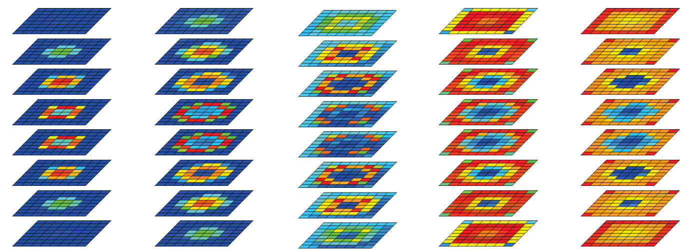
(a) Computation **(b)** Waiting time **(c)** Delay costs

Figure 4.8: Distribution of computation time, waiting time, and total delay costs in Zeus-MP/2 across the three-dimensional computational domain.

example configuration provided with the distribution. The number of simulated time steps was limited to 100 in order to constrain the size of the recorded event trace.

The wave-blast simulation requires 188,000 seconds of allocation time in total, 12.5% of which is waiting time. Most of this waiting time (75.6%) can be attributed to late-sender wait states in four major communication phases within each iteration of the main loop, in the following denoted as first to fourth communication phase. As Figure 4.7a shows, the dominant part of the waiting time in these communication phases is indirect. Regarding the root causes of the waiting time, the delay analysis identified four call paths as major origins of delay costs: the `lorentz` subroutine and three computational loops within the `hsmoc` subroutine, which are referred to as i-loop, j-loop and k-loop in the following. Within the main loop, the lorentz() subroutine is placed before the first communication phase, the i-loop before the second, and the j-loop and k-loop before the third and fourth communication phases, respectively. Figure 4.7b illustrates the mapping of short- and long-term delay costs onto the call paths responsible for the delays. Especially the `lorentz` routine and the i-loop region exhibit a high ratio of long- versus short-term delay costs, indicating that delays in these call paths indirectly manifest themselves as wait states in later communication phases.

The visualization of the virtual process topology in the Scalasca report browser allows us



(a) Delay costs

(b) Direct waiting time in 1st comm. phase

(c) Indirect waiting time in 1st comm. phase

(d) Indirect waiting time in 2nd comm. phase

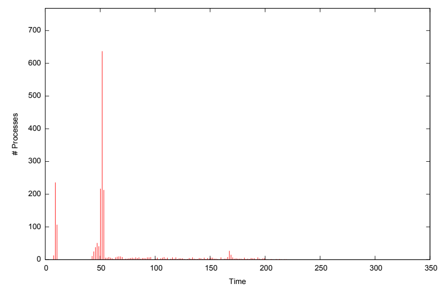
(e) Indirect waiting time in 3rd comm. phase

Figure 4.9: Propagation of wait states caused by delays in the `lorentz` subroutine. These delays cause direct wait states in the first communication phase, which induce indirect wait states at the surrounding layer of processes and travel further outward during the second and third communication phase.

to study the relationship between waiting and delaying processes in terms of their position within the computational domain. Figure 4.8a shows the distribution of workload (computational time, without time spent in MPI operations) within the main loop across the three-dimensional process grid. The arrangement of the processes in the figure reflects the virtual process topology used to map the three-dimensional computational domain onto the available MPI ranks. Obviously, there is a load imbalance between ranks of the central and outer regions of the computational domain, with the most underloaded process spending 76.7% (151.5 s) of the time of the most overloaded process (197.4 s) in computation. Accordingly, the underloaded processes exhibit a significant amount of waiting time (Figure 4.8b). Examining the delay costs reveals that almost all the delay originates from the border processes of the central, overloaded region (Figure 4.8c). The distribution of the workload explains this observation: Within the central and outer regions, the workload is relatively well balanced. Therefore, communication within the same region is not significantly delayed. In contrast, the large difference in computation time between the central and outer region causes wait states at synchronization points along the border.

These findings indicate that the majority of waiting time originates from processes at the border of the central topological region. Indeed, visualizing direct and indirect wait states separately confirms the propagation of wait states. Figure 4.9 shows how delay in the `lorentz` subroutine at the border of the central region causes direct wait states in the surrounding processes during the first communication phase, which in turn cause indirect wait states within the next layer of processes and propagate further to the outermost processes during the second and third communication phase.

Unfortunately, resolving the imbalance issues in the Zeus-MP/2 case study was out of the scope of this thesis. The high delay costs assigned to the major computational kernels of the program clearly show that an imbalanced workload is the root cause of the wait states. Zeus-MP/2 uses a static, regular domain decomposition to distribute workload across processes,



(a) Computation time histogram

(b) Distribution of computation time in the virtual domain

Figure 4.10: Computation time histogram (left) and distribution across the process grid in the 2D application domain show a significant overload on processes assigned to polar regions. Additional imbalance is introduced by assigning processes to land-only grid points, which do not participate in the computation at all.

which produces the inefficiencies we observed when the workload within different regions of the application domain varies. To solve this problem, the developers need to adopt a load-balancing scheme, possibly one of the approaches mentioned in Section 3.5.

4.4.2 CESM Sea Ice Model

In the second case study, the delay analysis was applied to an early version of the sea ice component of the Community Earth System Model (CESM) climate code. The measured configuration used a 1° dipole grid of the earth and a Cartesian grid decomposition on 2048 MPI processes.

This configuration suffers from severe load imbalances. The first of these is a result of the domain decomposition in the form of a uniform, high-resolution grid of the earth. Since the sea ice model obviously only applies to ocean regions, processes assigned exclusively to land regions do not participate in computation or communication at all. Also, processes assigned to regions where land and ocean overlap have less workload than processes assigned to all-ocean regions. A second load imbalance exists between processes assigned to open ocean regions and those assigned to sea ice (i.e., polar) regions, which have a significantly higher workload than the others.

The workload histogram in Figure 4.10a illustrates the severity of the imbalances. In addition, the mapping of computation times onto the 2D grid used by the application in Figure 4.10b visualizes the imbalance in the application context. In the histogram, we find a first cluster of processes that has nearly no workload, which corresponds to the processes assigned to land grid points. The largest cluster of processes, apparently the ones assigned to open ocean regions, receive about 52 seconds of computational work. However, a relatively small but widely spread out group of processes – the ones assigned to sea ice regions – receive workloads of up to 200 seconds.

As a result of the imbalance, many processes suffer late-sender wait states in the point-to-point nearest-neighbor MPI data exchange following the computation. Figure 4.11a illustrates the distribution of these wait states across the computational grid. Essentially, all processes assigned to open ocean regions incur wait states; the median waiting time is 106 seconds. Note that processes assigned to land-only grid points do not participate in the data exchange at all, and therefore do not incur late-sender wait states. The delays responsible for the late-sender wait states are located on the border between sea ice and open ocean processes, as we can see in Figure 4.11b. However, wait states on most processes assigned to open ocean areas are a result of wait-state propagation. The classification of wait states into direct versus indirect and propagating versus terminal wait states performed by the delay analysis allows us to observe the propagation effect directly. First of all, the distribution of indirect wait states in Figure 4.11d confirms that wait states on most processes are indeed produced by propagation. Direct wait states are only located on processes immediately surrounding the sea ice regions in the process grid, as can be seen in Figure 4.11c. From the distribution of propagating wait states shown in Figure 4.11e, we can see that wait states propagate both southwards from the northern polar region as well as northwards from the southern polar region towards the equatorial region of the domain. Terminal wait states (Figure 4.11f) are primarily located on processes assigned to the equatorial ocean regions and on shores, where the propagation chain ends.

Similar to the Zeus-MP/2 case study before, the CESM sea ice model illustrates how delays on a few processes spread wait states over a significant part of the process space. The distinction between propagating and terminal wait states makes these effects transparent. Moreover, the example also shows how the visual mapping of performance data such as delays and wait states onto the virtual process topology used by the application greatly improves the understanding of performance phenomena.

It should be noted that the experiments were run at a larger scale than typically used in production runs. The limited scalability of the regular domain decomposition for the sea ice calculation is known, and is expected to produce significant imbalance for large-scale runs. Currently, the developers investigate better load balancing schemes, in particular space-filling curves, to achieve better balance at large scale.

4.4.3 Illumination

The last case study presented here is Illumination, a 3D parallel relativistic particle-in-cell code for the simulation of laser-plasma interactions [28, 27], where the delay analysis was able to shed light onto an otherwise obscure performance phenomenon. The code uses MPI for communication and I/O. In addition, it employs the Cartesian topology features of MPI to simplify domain decomposition and the distribution of work, allowing the code to be easily executed with different numbers of cores. The three-dimensional computational domain is mapped onto a two-dimensional logical grid of processes. As in the case of Zeus/MP2 and the CESM sea ice model, the logical topology can be conveniently visualized in the Scalasca report browser.

This study examined a benchmark run over 200 time steps on 1024 processors. The traditional wait-state analysis showed that the application spent 55% of its runtime in computation and 44% in MPI communication, of which more than 90% was waiting time in point-to-point

(a) Total late-sender wait states

(b) Delay costs

(c) Direct late-sender wait states

(d) Indirect late-sender wait states

(e) Propagating late-sender wait states

(f) Terminal late-sender wait states

Figure 4.11: Wait-state propagation in the CESM sea ice model visualized by mapping propagation metrics onto the 2D process grid used by the application. Wait states are created by delays on processes at the border of sea ice and open ocean regions, after which they propagate northwards from the south and southwards from the north, respectively (bottom left) before they terminate on processes assigned to the equatorial and coastal regions (bottom right).

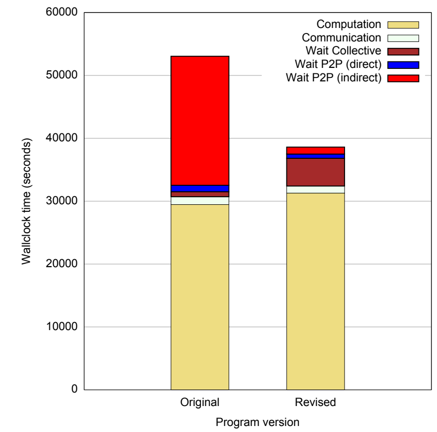


Figure 4.12: Runtime composition and quantitative comparison of the original versus the revised version of Illumination. In the revised version, the indirect waiting time was significantly reduced and wait states were partially shifted from point-to-point to collective communication. A slight increase in computation time was caused by additional memory copies needed in the context of the switch to non-blocking communication.

communication (Figure 4.12). In particular, a large amount of time – 91% of the waiting time, and 36% of the overall runtime – was spent in *late-receiver* wait states. There was also a notable computational load imbalance in the program's main loop, as shown in Figure 4.13a by the distribution of computation time across the process grid, where processes within a circular inner region obviously need more time than those outside. The measured computation time varies between 16 and 22 seconds per process.

An investigation using the delay analysis quickly revealed the root cause of the late-receiver wait states. First of all, the analysis shows that here, too, most of the waiting time (95%) is a result of propagation. Examining the delay cost metric leads us to the beginning of the propagation chain. The analysis shows that, interestingly, the delays carrying the highest costs do not originate from the computational load imbalance in the main loop: they is only responsible for 10% of the observed waiting time. Instead, we find the highest delay costs assigned to the MPI operations performing the data exchange. In fact, more than 75% of the waiting time originate from delays caused by the communication itself. As in the case of ZeusMP/2 and the CESM sea ice model, a visual analysis of the performance metrics mapped onto the virtual domain of the application helps to understand the underlying performance phenomena. Figure 4.13c shows the topological distribution of the total delay costs across the 2D domain of the application. The delay costs form a pattern of horizontal stripes which bears little resemblance to the pattern of the computational imbalance. Moreover, the distribution pattern of short-term delay costs in the communication call paths that carry the highest delay costs, shown in Figure 4.13d, reveals that with the exception of the border processes, the underlying problem occurs across *all* processes of the two-dimensional grid. Together, these findings suggest that the main problem is actually an inefficient communication pattern as such: in a sense, the communication impedes itself.

The data exchange between neighboring processes in Illumination was implemented using a sequence of blocking MPI point-to-point calls. However, this approach imposes a strict order in which messages are processed, so that even if another communication on a process would be able to complete, any delay or wait state early on in the message chain will block the remaining outstanding communications. This is especcially true when the MPI implementation uses synchronous communication, as observed in the example. When exchanging the blocking MPI communication routines in the code with their non-blocking counterparts and using MPI_Waitall to complete outstanding requests in an arbitrary order, the waiting time is substantially reduced. Against the background of the delay analysis results, this seems now plausible because wait states in one operation no longer delay subsequent communication calls. Another performance analysis was performed with the revised version of the code. As Figure 4.12 illustrates, this version indeed shows a significant performance improvement. More than 80% of the program runtime is now consumed by computation, 11% by wait states in collective communication, and only 5% by wait states in point-to-point communication.

The computational load imbalance remains, however, the delay costs now clearly identify delay within the computational part of the main loop as the major cause of the waiting time. The distribution of delay costs in the topology display (Figure 4.13e) now resembles a ring pattern around the inner imbalanced region of the domain, and accentuates the computation time gradient between the overloaded inner and underloaded outer region. Hence, the delay analysis confirms the load imbalance as the single root cause of the bulk of waiting time and, thus, indicates that the waiting time cannot be significantly reduced any further without actually resolving the load imbalance itself.

4.5 Related Work

A number of research projects target the detection and analysis of imbalance as a source of inefficiency in parallel programs, but only few of these cover wait-state propagation or consider the link between wait states at synchronization points and imbalances causing them. While the large body of prior work covering imbalance analysis in general has already been adressed in Section 3.6, this section discusses prior work which is conceptually related to the delay analysis in particular.

The delay analysis approach is inspired by the work of Meira Jr. et al. in the context of the Carnival system [52, 53]. Using event traces of program executions, Carnival identifies the differences in execution paths leading up to a synchronization point and explains the waiting time in terms of those differences. This waiting time analysis is implemented as a pipeline of four independent tools: The first stage extracts execution steps (i.e., activities) from the trace file. These activities are combined to execution paths for every instance of waiting time during the second stage. The third stage partitions the activities into equivalence classes so that for every class only one representative needs to be stored. The fourth stage finally isolates the differences between matching paths, yielding one or more characterizations for each program location that exhibits wait states.

In comparison, the delay analysis is similar to Meira Jr.'s approach on a conceptual level, but offers far greater scalability through its parallel design. Moreover, the theoretical foundation underlying the delay analysis includes a concise terminology and cost model that requires only a few powerful, orthogonal concepts to explain the most important questions related to the formation of wait states. Although Meira Jr.'s characterization of wait states through differences in execution paths incorporates wait-state propagation implicitly, it does not explicitly distinguish between long-term and short-term effects. It also does not provide means to analyze wait-state propagation itself, which the delay analysis does through its distinction between propagating/terminal and direct/indirect wait states.

Again leveraging the post-mortem analysis of event traces, Jafri [44] applies a modified version of vector clocks to distinguish between direct and indirect wait states. However, neither does his analysis identify the responsible delays, as the delay analysis does, nor does his sequential implementation address scalability on large systems. While his sequential implementation may take advantage of a parallel replay approach to improve its scalability, the general idea of using vector clocks to model the propagation of waiting time from one process to another, which is inherently a forward analysis, may be confronted with excessive vector sizes when waiting time is propagated across large numbers of processes.

In [80], Tallent et al. describe a method integrated in HPCToolkit [3] to detect load imbalance in call-path profiles that attributes the cost of idleness (i.e., wait states) occuring within globally balanced call-tree nodes (balance points) to imbalanced call-tree nodes that descend from the same balance point. Tallent's approach also uses a highly scalable parallel program to perform the load-imbalance detection automatically in a post-mortem analysis step. However, whereas the delay analysis operates on comprehensive event traces, Tallent's imbalance analysis is based on aggregate call-path profiles that do not preserve temporal information, which is necessary to locate the precise causes of wait states. As an advantage, HPCToolkit's creation of call-path profiles through asynchronous sampling requires less storage space and often allows easier control of the measurement overhead for complex programs than Scalasca's recording of event traces based on direct instrumentation. Also, the automatic analysis of the (much smaller) aggregate profiles in Tallent's implementation requires significantly less computational resources than the elaborate trace replay approach employed by the delay analysis. Nevertheless, the delay analysis identifies the causes of wait states precisely, whereas Tallent's approach merely identifies suspects by correlating global wait states with call paths that exhibit global, static imbalance. Other than the delay analysis, Tallent et al.'s profile-based solution cannot correctly characterize dynamic imbalance. Moreover, their approach does not integrate the effect of wait-state propagation into its calculation of imbalance costs. It also neither distinguishes imbalances by the type of wait state they cause, nor does it identify the machine location (process or threads) that are responsible for a delay. Overall, Tallent's work represents a lightweight method to uncover imbalance in parallel codes, but the profile-based approach cannot reach the precision of the equally scalable trace-based delay analysis.

Summary

The delay analysis is a novel method to identify and characterize sources of parallel inefficiency in MPI programs. It maps the allocation-time costs of wait states back onto delays that originally caused them. Wait states are idle times that occur when processes fail to reach a synchronization point simultaneously; delays are excess runtimes in certain program locations on one process that lead to a wait state on another process later on. The delay analysis determines the locations of delays as well as their costs, which represent the total amount of waiting time the delay is responsible for. Moreover, the delay analysis incorporates wait-state propagation effects into the calculation of delay costs. By characterizing wait states with respect to their position in the propagation chain, it also provides means to study wait-state propagation itself.

The prototype implementation leverages Scalasca's parallel trace replay approach, using a backward replay to follow wait-state propagation chains from the endmost wait states back to their root causes. Other than related profile-based approaches, the trace-based delay analysis captures performance dynamics accurately, while the parallel algorithm allows it to handle much larger scales than conceptually similar trace-based solutions.

(a) Computation time distribution (original)

(b) Wait states (original)

(c) Total delay costs (original)

(d) Short-term delay costs (original)

(e) Total delay costs (revised)

(f) Wait states (revised)

Figure 4.13: Performance metrics mapped onto the virtual process grid used by Illumination indicate wait-state root causes. A mismatch between the workload distribution pattern (a) and the delay cost pattern (c,d) indicates that wait states (b) are not caused by the workload imbalance. The version with a revised data exchange algorithm shows significantly reduced delay costs (e) and wait states which now match the workload imbalance pattern.

Chapter 5

Critical-Path Based Performance Analysis

In addition to detecting the root causes of wait-states, we also investigated the critical path as an instrument to locate and characterize imbalance. As a result, we developed novel methods to identify the critical-path of a parallel program and automatically distill useful insights into compact performance indicators. This chapter describes concepts, implementation, and evaluation results of the critical-path analysis. First, Section 5.1 delineates scope and motivation behind the critical-path based approach against the background of previous and alternative solutions. Section 5.2 describes the theory and concepts of the approach in detail. Section 5.3 describes the prototype implementation within the Scalasca trace-analysis framework, then, Section 5.4 discusses advantages and application of the critical-path analysis in practice in two detailed case studies. Finally, Section 5.5 provides an overview of previous and related work.

5.1 Revisiting Critical-Path Analysis

Originally developed as a tool for planning, scheduling, and coordinating complex engineering projects [45], which usually involve concurrent activities, the notion of the critical path is also helpful in understanding the performance of parallel programs. Critical-path analysis models the execution of a parallel program as a *program activity graph* (PAG), which represents the duration and precedence order of individual program activities. The *critical path* is the longest path through the graph. Therefore, the *critical activities*, which are those on the critical path, determine the overall program runtime. The critical path itself does not have any wait states but may induce them in other execution paths. Increases in the time for critical activities prolong the overall execution. Conversely, shortening activities that are *not* on the critical path only increases waiting time, but does not improve the program runtime. Therefore, the critical path identifies promising optimization candidates, because only optimizations in critical activities may actually reduce the program runtime. However, the overall runtime reduction of such an optimization may be smaller than the reduction of the activity, as the change may shift the critical path to include other activities. Thus, while knowledge of the critical path can identify activities for which optimizations will prove worthwhile, knowledge of the impact of those optimizations on the critical path can guide optimization efforts more precisely.

In spite of its apparent utility and numerous earlier research works, as of today critical-path analysis plays only a minor role in the performance analysis of supercomputing applications. To some extent, this lack of adoption can be explained with a failure to exploit and interpret critical-path data in a user-friendly and informative way. The SPMD paradigm used by many, if not most parallel programs sets supercomputing applications apart from typical engineering projects for which the critical path was first invented. For such programs, where every process executes a nearly identical sequence of activities, the information obtained from the critical path by itself seemed to offer little additional benefit compared to more lightweight conventional profiles. Also, the sheer length of the critical path in realistic programs makes it an unwieldy data structure to analyze manually with reasonable effort. However, the most important factor for the limited role of critical-path analysis in contemporary performance-analysis tools is probably the lack of a scalable and reliable method to extract the critical path in large-scale programs.

Despite these shortcomings, the fundamental properties of the critical path remain. A critical path measurement contains essential information for the optimization and load balance of parallel codes. Most importantly, it retains dynamic performance characteristics which can be used to identify dynamic imbalance. However, we need new ways to interpret and to analyze it without sacrificing scalability.

A new, highly scalable algorithm to extract the critical path from event traces based on Scalasca's event-trace replay approach allows us to revisit the critical path as basis for intuitive and meaningful performance metrics. To that end, we use the critical path to derive several compact *performance indicators* that illuminate the relationship between critical and non-critical activities to guide the analysis of complex load-imbalance phenomena intuitively. Similar to economic indicators such as consumer price index or gross domestic product, which characterize things as complex as a nation's economic well-being in terms of a few numbers, performance indicators improve the understanding of labyrinthine program behavior without letting the user drown in a flood of performance details. The main difference to classic performance metrics such as the number of messages is the higher level of abstraction that these indicators provide. While also offering insight into classic SPMD programs, the indicators especially suit programs with a *multiple program multiple data* (MPMD) structure, which is popular among the increasing number of multi-physics codes.

5.2 Critical-Path Analysis Concept

This section describes the basic concept of the critical path and the performance indicators built upon it. In essence, our approach combines critical-path data with per-process summary profile data. The critical path provides an overview of the most time-consuming activities, but does by itself not capture important parallel performance characteristics such as parallel efficiency or load balance. In contrast, summary profiles do not capture dynamic effects that characterize a program's execution. By combining critical-path and summary profiles, we can characterize load balance and highlight typical parallelization issues more reliably than by using summary profiles alone. The combination leads to a set of compact *performance* *indicators*, which provide intuitive guidance about load-balance characteristics that quickly draw attention to potentially inefficient code regions.

Our novel analysis concept produces two groups of performance data structures. The first group, the *critical-path profile* and the *critical-path imbalance indicator*, characterizes the impact of program activities on wall-clock time. In addition, the imbalance indicator captures how much time in a call path is lost due to load imbalance. The second group, called *performance impact indicators*, characterizes the influence of program activities on allocation time. These indicators are especially useful for the analysis of MPMD programs. In particular, they classify load imbalance in MPMD programs by origin, and can distinguish if resource waste is a result of an uneven distribution of workload between process partitions, or of imbalance among processes performing the same activity within a single partition. The following subsections explain the performance indicators in detail. Before, however, we re-examine the basic idea of the critical path.

5.2.1 The Critical Path

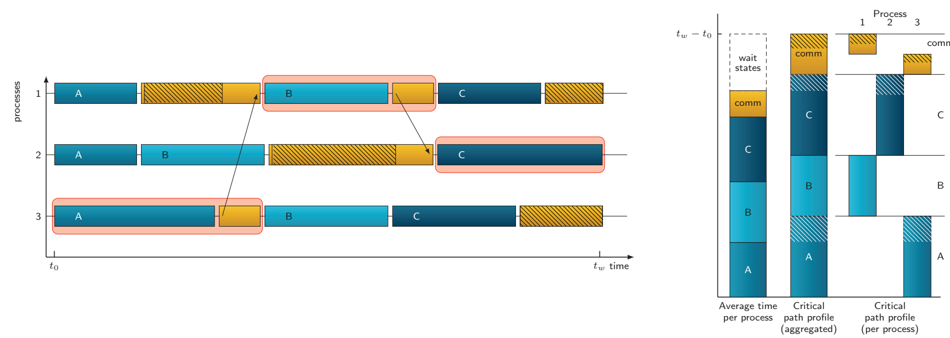
To explain the concept of the critical path, we can again use the program execution model introduced in Section 4.2, which describes the execution of a parallel program as a set of processes P which execute sequences of activities in parallel. A tuple $(p,i) \in P \times \mathbb{N}$ denotes the ith activity executed by process p. The function $\mathrm{Callpath} : P \times \mathbb{N} \to C$ identifies the program location (i.e., the call path in the set of call paths C) excuted by an activity. Together, the activities form a *program activity graph*, which connects subsequent activities on the same process through *sequence edges*, and communication activities on different processes that form a synchronization point through *communication edges*. The critical path is the longest sequence in the program activity graph that does not include wait states. Activities on the critical path are called *critical activities*.

The critical path determines the runtime of the program: any increase in a critical activity will increase the overall program runtime. An optimization on the critical path may decrease runtime, but the improvement is not guaranteed since a different sequence of activities may become the critical path instead. In contrast, optimizing any activity that is not on the critical path only increases waiting time, but does not affect the overall runtime.

Note that the critical path is not necessarily unique, that is, a program activity graph may have multiple different critical paths. However, this effect rarely occurs in practice. All critical-path based performance metrics and indicators in the analysis approach at hand are only determined for a single critical path. Should multiple critical paths exist, the extraction algorithm described in Section 5.3 selects one of them.

5.2.2 Critical-Path Profile

The *critical-path profile* represents the total amount of (wall-clock) time that each call path and each process of the program spends on the critical path. Mathematically, the critical path profile is defined as a mapping $P \times C \to \mathbb{R}$ that assigns to each combination of call path and process the time attributable to the critical path.



(a) SPMD program time-line diagram. Each rectangle represents an activity. Arrows between processes denote communication; the hatched areas inside communication activities represent wait states. Highlighted activities mark the critical path. Processes 1 and 3 end with wait states due to pseudo-synchronization.

(b) Average activity time and critical-path profile (aggregated and per process); hatched areas denote critical-path imbalance.

Figure 5.1: The critical path in an SPMD program run (a) and the associated critical-path profile (b), showing both the total contribution of program call paths to the critical path (middle) and the individual contributions of each process (right). The profile also shows the amount of critical imbalance in each call path.

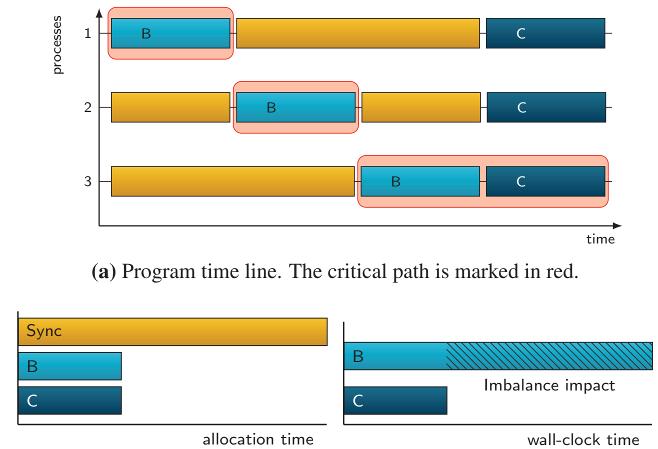
Figure 5.1b illustrates the critical-path profile for the example program shown in Figure 5.1a. The critical-path profile can provide a simple overview of the most time-consuming subroutines, as shown in the middle bar, but also allows detailed insight into each process's time share on the critical path, as illustrated by the bars on the right of Figure 5.1b.

5.2.3 Critical-Path Imbalance Indicator

The critical-path imbalance indicator ι is the difference between a call path's contribution to the critical path and the average time spent in the call path across all processes. Precisely, we define it for a critical-path call path c as:

$$\iota(c) = \max\left(d_{cp}(c) - \operatorname{avg}(c), 0\right)$$
$$\operatorname{avg}(c) = \frac{1}{|P|}\sum_{p\in P} d_p(c)$$

where $d_{cp}(c)$ denotes the time that call path r contributes to the critical path, $d_p(c)$ represents the total time without wait states spent in c on process p, and P is the set of processes. Since an imbalance only affects overall execution time if the time on the critical path is larger than the average, the imbalance indicator only includes positive values. Figure 5.1b illustrates the concept graphically. The critical-path imbalance is the hatched area of the critical-path profile bars. Call path B exhibits no critical-path imbalance since it is perfectly balanced. Call paths A and C as well as the communication activities exhibit critical-path imbalance, indicating some inefficient parallelism in these activities.



(a) Program time line. The critical path is marked in red.

(b) Summary profile of the allocation time spent in each activity.

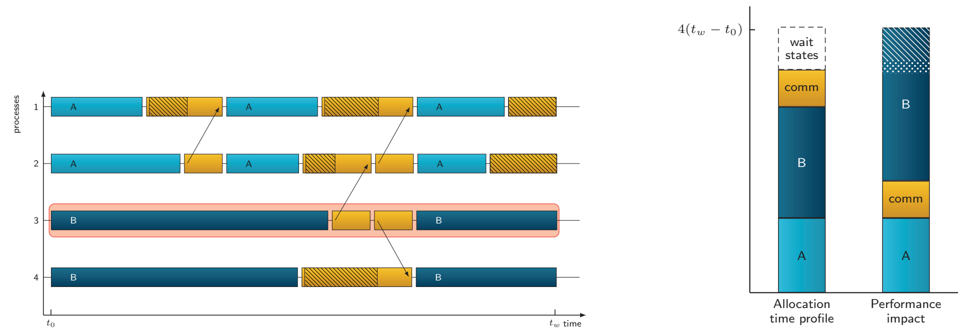
(c) Critical-path profile of the wall-clock time spent in each activity.

Figure 5.2: Analysis of dynamic performance effects: The serialized execution of function B, as seen in the time line (a), goes unnoticed in a summary profile (b), but is correctly identified as a performance bottleneck by the critical-path imbalance indicator (c).

Essentially, the amount of critical-path imbalance in a program corresponds to the wall-clock time that is lost due to inefficient parallelization, compared to a perfectly balanced program. Thus, the critical-path imbalance indicator provides clear guidance in discovering parallelization bottlenecks. Also, the ratio of critical-path imbalance and the total time of a call path on the critical path provides a useful load imbalance metric. Thus, it provides similar guidance as prior profile-based load imbalance metrics (e.g., the load-imbalance percentage metrics defined in CrayPat [17]), but the critical-path imbalance indicator can often draw a more accurate picture. The critical path retains dynamic effects in the program execution, such as shifting of imbalance between processes over time, which summary profiles simply cannot capture. Thus, purely profile-based imbalance metrics regularly underestimate the actual performance impact of a given load imbalance. As an extreme example, consider a program like the one in Figure 5.2, where a function is serialized across all processes but runs for the same amount of time on each. Purely summary-profile based metrics would not show any load imbalance, whereas the critical-path imbalance indicator correctly characterizes the function's serialized execution as a performance bottleneck.

5.2.4 Performance-Impact Indicators

In contrast to the critical-path imbalance indicator, which determines the impact of imbalances in terms of wall-clock time, the performance impact indicators characterize the allocation-time impact of imbalances. The main application for these indicators is the analysis of MPMD programs. For such programs, the overall performance impact of executing program call paths which only run on a subset of all processes cannot be sufficiently expressed in terms of wall-



(a) Time-line diagram of an MPMD program run, showing two partitions that execute different activities; highlighted activities mark the critical path, hatched areas represent wait states.

(b) Allocation time profile and performance impact; the hatched area represents inter-partition imbalance costs, the dotted area intra-partition imbalance costs.

Figure 5.3: Parallel profile and performance impact indicators for an MPMD program.

clock time alone. Instead, we need to determine their allocation-time impact. Moreover, as described in Section 3.3.5, MPMD programs exhibit more complex, multi-level imbalance characteristics not found in SPMD programs. In particular, we can categorize imbalance into *intra-* and *inter-partition* imbalance to distinguish between imbalance within a single SPMD process partition, or imbalance between different process partitions, respectively. The performance impact indicators identify and characterize imbalance with respect to these categories to provide improved insight into load balance issues in MPMD programs.

Performance impact

As mentioned above, the performance impact indicators determine the total allocation time impact that is attributable to a program call path. This concept describes the performance impact of a program call path as the sum of the total allocation time spent executing the call path itself and its *imbalance costs*: the allocation time lost due to any excess time spent in that call path on the critical path.

The calculation of the imbalance costs is based on the notion that the activities on the critical path determine the runtime of the program, and, therefore, also its resource consumption / allocation time. We also explicitly distinguish between the resources consumed by communication and computation activities directly and the amount that wait states occupy. Then, we map the allocation time occupied by wait states onto those program call paths which take more time on the critical path than on the process where the wait state occured. To allow additional insights for MPMD programs, we further divide imbalance costs assigned to a program location into intra- and inter-partition imbalance costs.

The time-line diagram in Figure 5.3a illustrates the concept of the performance impact indicators in the context of an MPMD program run. Here, one partition of the processes executes activities in call path A and another partition executes activities in call path B, with the critical path running entirely on process 3. In this example, we classify the underload in call path B on process 4 as intra-partition imbalance costs, and the resource consumption due to the wait states on processes 1 and 2 as inter-partition imbalance costs. Figure 5.3b shows the parallel allocation time profile of the program on the left and the performance impact on the right. Since call path B is the only one on the critical path, it accumulates the entire critical-path imbalance costs; hence, these costs are added as imbalance costs onto the performance impact of call path B. In this example, inter-partition imbalance costs (hatched area) are roughly three times as large as the intra-partition imbalance costs (dotted area). As a result, serial optimizations of call path B or rebalancing the partitions to assign more processes to B (or fewer processes to call path A) promise greater benefits with respect to resource consumption than load-balancing the execution of call path B within its partition.

The critical-path based imbalance impact cost indicator is conceptually related to the delay analysis. However, other than the delay costs, the mapping of imbalance costs onto critical activities does not necessarily reflect the direct causes of wait states. Instead, the imbalance cost indicators are a heuristic to highlight optimization possibilities that are most likely to improve performance. However, the calculation of imbalance costs based on the critical path allows us to distinguish intra and inter-partition costs, which is not easily possible with the delay analysis. Therefore, the performance impact indicators provide better insight into the specific imbalance characteristics found in MPMD programs.

Calculating the imbalance costs

We can easily compute the imbalance costs in parallel based on the knowledge of the critical-path profile. Each process calculates its local portion of the imbalance costs, which are then aggregated using a global reduction operation. Basically, on each process, we identify call paths that take more time on the critical path than on this process and assign imbalance costs to these call paths.

On each process, we first determine the critical-path excess time $\delta_p(c)$ for each critical-path call path c:

$$\delta_p(c) = \max\left(d_{cp}(c) - d_p(c), 0\right)$$

Here, $d_{cp}(c)$ represents the aggregate time a call path c spends on the critical path across all processes, and $d_p(c)$ represents the aggregate execution time of the call path on process p. Because only positive critical-path excess times contribute to the overall performance impact, negative values are discarded. Hence, the critical-path excess time of a critical-path call path c on process p shows how much more time the call path occupies on the critcal path than on the local process p. Imbalance costs are then mapped onto the critical-path call paths proportionally to their excess times in such a way the total costs match the total waiting time $\hat{w}_p$ on the process:

$$\text{Imbalance cost}(p,c) = \frac{1}{\hat{\delta}_p}\,\delta_p(c)\,\hat{w}_p$$

Here, $\hat{\delta}_p$ corresponds to the sum of the excess times of all critical-path call paths for process p:

$$\hat{\delta}_p = \sum_{c \in C} \delta_p(c)$$

Overall, the aggregate imbalance costs correspond to the allocation time lost to imbalance, which equals the waiting time.

If the critical-path call path c does not occur on the local process at all, its imbalance costs are considered inter-partition imbalance costs; otherwise they are considered intra-partition imbalance costs. Most useful are the typically the total intra- or inter-partition imbalance costs of a call path c, which are simply the sum of the respective imbalance costs of c across all processes:

$$\text{Intra-partition imbalance cost}(c) = \sum_{\{p \in P \,|\, d_p(c) > 0\}} \frac{1}{\hat{\delta}_p} \delta_p(c)\, \hat{w}_p$$

$$\text{Inter-partition imbalance cost}(c) = \sum_{\{p \in P \,|\, d_p(c) = 0\}} \frac{1}{\hat{\delta}_p} \delta_p(c)\, \hat{w}_p$$

5.3 Scalable Critical-Path Detection

As indicated in the introduction, determining the critical path of a parallel program run is a non-trivial task. The difficulties in obtaining this performance structure are twofold: First, the critical path is a global structure, which makes it difficult to extract in a scalable way. Second, the exact course of the critical path can only be fully determined at the end of the execution, which makes it difficult to capture the critical path using an on-line monitoring tool. Hence, finding a reliable and scalable approach to extract the critical path is a crucial requirement for making critical-path analysis a feasible performance-analysis solution on today's large-scale systems. A novel algorithm developed in the course of this thesis to extract the critical path based on Scalasca's parallel post-mortem trace replay approach is a major contribution to fulfill this goal. In fact, Scalasca's trace replay method allows a particularly simple and straightforward solution of the problem. In the following, we will examine this algorithm in detail.

The algorithm extracts the critical path from a previously recorded event trace of the execution of a parallel program using a backward replay. In a nutshell, the algorithm first determines the endpoint of the critical path (which is straightforward), and follows it backwards along the synchronization points where it shifts between processes. As a prerequisite, wait states and synchronization points that occur in the program need to be known before the critical path search in the backward replay. Therefore, the algorithm utilizes the synchronization point information provided by the preparatorial replay phases (the first to third phase in Figure 4.2)

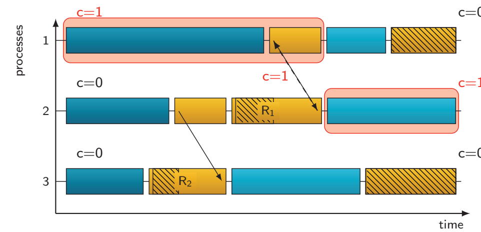


Figure 5.4: Detection of the critical path in the backward replay. The critical path (marked in red) ends on process 2, where the replay starts. The critical-path token then c moves to process 1 at the synchronization point in R_1.

for the delay analysis. The actual critical-path extraction is then performed alongside the delay analysis in the main backward replay stage (that is, the fourth stage shown in Figure 4.2).

For MPI programs, the critical path runs between MPI_Init and MPI_Finalize. As the backward replay starts at the end of the trace, the algorithm first determines the MPI rank that entered MPI_Finalize last. This point marks the endpoint of the critical path. Since the critical path only runs on a single process at any given time, we set a flag on the corresponding analysis process to signal its "ownership" of the critical path.

The remainder of the critical-path search exploits the lack of wait states on the critical path. As the backward replay progresses, the critical path remains on the flagged process until it reaches a communication event in the trace which has been marked as a wait state. In this case, the critical path continues at the corresponding communication event on the process that is responsible for the wait state (i.e., the communication peer). Therefore, the algorithm transfers critical-path ownership to this process via backward communication replay. In the case of a point-to-point communication event, the replay uses a point-to-point message for the ownership transfer; in the case of a collective communication event, it uses a reduction operation.

Figure 5.4 illustrates the backward replay, which starts from the end of the trace shown on the right. Since process 2 finished execution last, it owns the final critical path segment. Moving backwards through the trace, we find a wait state at communication event R_1. Now, the critical-path flag moves to the wait state's origin on process 1 using a point-to-point message transfer from the original receiver to the original sender. During the course of the replay, the processes that own the critical-path flag accumulate their individual contributions to the critical-path profile locally. While in principle our approach can capture the entire dynamic structure of the critical path, Scalasca currently collects and reports only the critical-path profile. However, future extensions that make use of the dynamic structure are conceivable, such as a trace time-line display with critical-path highlighting.

After Scalasca completes the critical-path extraction, we derive our performance indicators. The nature of this task lends itself to parallel computation. We therefore accumulate the global critical-path profile using a global reduction operation and distribute it to all processes, which then calculate their individual contributions to each indicator.

Overall, the parallel backward trace replay leads to an exceptionally scalable and straightforward critical-path extraction algorithm. Crucial for the correctness of the critical-path analysis

Table 5.1: PEPC profile statistics for "tree_walk" and "sum_force"

	tree_walk	sum_force
Runtime per process		
Minimum	14.44 s	20.15 s
Average	15.07 s	20.63 s
Maximum	16.68 s	21.17 s
Imbalance time (maximum-average)	1.61 s	0.54 s
Critical-path time	20.78 s	21.29 s
Critical-path imbalance (crit.time-average)	5.71 s	0.66 s

is the accurate detection of all synchronization points at which the critical path may shift to another process. With only very few exceptions, which will be discussed in Section 6.2.3, the algorithm does correctly identify all synchronization points. Notably, this also includes "late receiver" synchronization points resulting from synchronous send operations. As shown later in Section 5.5, many prior approaches do not handle this case correctly.

5.4 Evaluation

We evaluate the critical-path analysis using two selected case studies to demonstrate the application of the performance indicators and their advantages compared to alternative solutions. The first case study, PEPC, demonstrates the critical-path profile and the critical-path imbalance indicator, and highlights how the critical-path based metrics provide a much better characterization of load imbalance than profiles alone. The second case study, ddcMD, is an MPMD program. Here, we demonstrate how the MPMD imbalance classification by the performance impact indicators help understand the complex performance characteristics of the MPMD program configuration. All experiments were perfomed on the IBM Blue Gene/P supercomputer Jugene at the Jülich Supercomputing Centre.

5.4.1 PEPC

PEPC ("Pretty Efficient Parallel Coulomb-solver") is an SPMD N-body simulation code developed by the Simulation Laboratory Plasma Physics at the Jülich Supercomputing Centre [70]. We used the 1.4 benchmark version of the code for our experiments and simulated 1.6 million particles over 20 time steps on 512 cores. Then, we conducted a trace analysis to obtain the critical-path profile, the performance indicators, and a summary performance profile.

The measured wall-clock execution time for the example configuration is 70.6 seconds. The major fraction of this time is spent in two code regions: a computational tree-balancing loop ("tree_walk") and the calculation of particle forces ("sum_force"). Table 5.1 summarizes runtime and critical-path profile data for these two code regions and the associated load imbalance metrics. The summary profiles suggest that the "sum_force" calculation takes around 5 seconds longer than the "tree_walk" loop. However, even though no single process spends more than 16.7 seconds in the “tree_walk” loop, the critical-path profile reveals that both code regions are responsible for around 21 seconds of the total wall-clock time. The iterative nature of the program and load imbalance in the “tree_walk” loop explains this discrepancy. Since PEPC employs a dynamic load-balancing scheme, the load imbalance is not static. Instead, load maxima appear on different processes in different iterations. Due to global synchronizations within each iteration, the per-iteration load maxima accumulate on the critical path, so that the total impact of the “tree-walk” loop on program runtime is actually larger than the time it consumes on any process. As shown in the artificial example in Figure 5.2 earlier, runtime profiles cannot capture this effect and underestimate the loop’s runtime influence. The critical-path based metrics clearly provide more insight; in particular, the critical-path imbalance indicator shows that the load imbalance in “tree_walk” wastes 5.7 seconds of wall-clock time.

These results demonstrate that the critical-path imbalance indicator estimates the influence of load imbalance on program runtime with higher accuracy than indicators based on summary profiles alone do, making it a highly valuable tool to assess load balance and to detect parallelization bottlenecks.

5.4.2 ddcMD

To demonstrate the functionality of our critical-path analysis with an MPMD code, we performed experiments with the molecular dynamics simulation code ddcMD [30]. This code partitions simulation components in a heterogeneous decomposition according to their scaling properties to circumvent the scaling problems of long-range interaction force calculations [71]. ddcMD uses a particle-particle/particle-mesh algorithm that divides the force calculation into an explicit short-range pair-interaction piece, and a long-range piece that is solved in Fourier space. While the short-range interaction scales well, the long-range force calculation does not. In ddcMD, this problem is solved by calculating the long-range forces (mesh calculation) on a small partition of around 5-10% of the available processes, and the short-range pair interactions (particle calculation) on a partition of the remaining processes. Both tasks can run in parallel, but must be carefully load-balanced in order to achieve high efficiency. In particular, the mesh calculation should run slightly faster than the particle calculation, so that the mesh processes do not block the large group of particle processes.

Load balance between the particle and mesh tasks can be tuned in various ways. Obviously, the partition sizes themselves are crucial parameters. However, on systems with a highly regular network interconnect, such as the Blue Gene/P, the placement of processes on the system is also an important factor for achieving optimal efficiency in ddcMD, which leaves only few reasonable options for the partition sizes. Thus, a useful strategy is to first choose a fixed partitioning scheme and process mapping that optimally fits the Blue Gene’s network topology, and then fine-tune the load balance between particle and mesh tasks using two other parameters that significantly impact efficiency: the inverse screening length α and the mesh size δ. By changing α and δ one can shift workload from mesh tasks to the particle tasks and vice-versa without loss of accuracy. Increasing the mesh size increases the workload of the mesh processes, and the accompanying decrease in α reduces the workload of the particle processes.

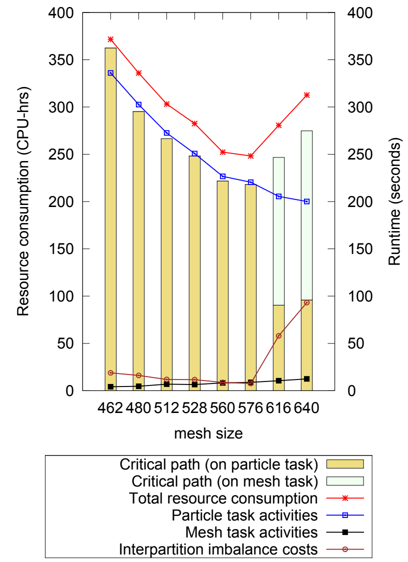


Figure 5.5: Influence of mesh size δ and inverse screening length α on ddcMD performance. With small mesh sizes, the critical path is located entirely in the particle task partition. With large mesh sizes, the critical path switches to the mesh task partition.

Our experiments study the influence of various combinations of α and δ (with $\alpha\delta$ kept constant) on program performance. We run simulations of 38 million particles on 4,096 cores of Blue Gene/P, with fixed partition sizes of 3,840 cores for the particle tasks and 256 cores for the mesh tasks. The application itself runs for 374 seconds in the worst case. Our parallel trace analysis (including trace I/O and report collation) on the 4,096 processes runs another 285 seconds, 113 seconds of which is for the parallel replay.

Figure 5.5 summarizes the analysis results. The bars in the background show the duration of particle and mesh calculations on the critical path. The line at the top (shown in red) represents the overall resource consumption (i.e., critical-path length multiplied by the number of processes), the remaining lines visualize the attribution of resource consumption to the two activity classes and the inter-partition imbalance costs (this graph does not show intra-partition imbalance costs). The spectrum of configurations ranges from small mesh sizes that place little workload on the mesh tasks and a large workload on the particle tasks on the left to large mesh sizes that shift workload from particle to mesh tasks on the right. On the left of the spectrum, the critical path visits only particle tasks while it shifts to some of the mesh tasks on the right.

As expected, the workload of mesh tasks increases with larger mesh sizes, while the workload of the particle tasks decreases due to the complementary change in α. Since the overall resource demand of the particle calculations is much higher than the demand of the mesh tasks, the shift in $\alpha\delta$ combinations in favor of lower α values also leads to an overall decrease of resource consumption, and, consequently, runtime. The runtime difference between particle and mesh tasks also shrinks, further decreasing resource consumption and improving resource utilization. The inter-partition imbalance costs clearly illustrate this effect. For small mesh sizes, the inter-partition imbalance costs represent the resource loss due to the excess time on the particle tasks, which leads to wait states among the mesh tasks. The inter-partition imbalance costs and the resources employed for mesh computations exhibit a noticeable gap that indicates that the resources wasted by wait states are larger than those used by mesh computations. At a mesh size of 576, we reach an optimal configuration.

Once the mesh size exceeds that threshold, the performance characteristics change completely. Now, the mesh calculation takes longer than the particle calculation, so that the critical path now visits the mesh tasks. While the resources consumed by the particle computations continue to decrease, the particle tasks must now wait for the mesh calculations to finish, which leads to an overall increase in runtime and total resource consumption. Considerable resources are wasted since the large number of processes in the particle partition must wait for the mesh computations, as indicated by large inter-partition imbalance costs that originate from the mesh tasks. With further increases of the mesh size, the resource waste due to the poor workload distribution between the two partitions grows rapidly.

Figure 5.5 shows that the load balance between the particle and mesh partitions is key to efficient execution of ddcMD. In contrast, intra-partition imbalance costs (not shown in Figure 5.5) only vary between 3 and 5% of the overall resource consumption, which indicates that the particle- and mesh-workloads themselves are well balanced.

In conclusion, the ddcMD example clearly demonstrates the usefulness of our performance indicators for characterizing load balance in MPMD programs. They provide appropriate guidance for choosing an optimization strategy. For example, the mesh sizes of 528 and 616 points lead to roughly the same overall runtime, but our performance indicators reveal radically different performance characteristics. With a 528 point mesh size, some remaining inter-partition imbalance costs of the particle computations suggests optimization by shifting workload from the particle to the mesh tasks. In the other case, the occurrence of mesh tasks on the critical path and the high inter-partition imbalance costs of the mesh computations indicate an uneven workload distribution between the two process partitions that leads to wait states among the particle processes.

5.5 Related Work

Several researchers have explored critical-path analysis as a means to identify performance problems and to steer optimizations. For example, Barford and Crovella [6] use the critical path to study the performance of TCP in the context of HTTP transactions, while Tullsen and Calder [84] reduce dependencies in binary codes based on critical-path information. In the context of parallel message-passing programs, the critical path was used in several prior tools to describe the actual computational demands of an application. ParaGraph [33] is an early tool that could visualize the critical path of a parallel program in a space-time diagram. Miller et al. [56] describe a distributed algorithm to capture the critical path. Schulz [74] describes techniques to extract the critical path from a message-passing program in the form of a graph, which can then be used for further post-mortem analysis. Hollingsworth [36] employs a sophisticated online mechanism to extract a critical-path profile from message-passing programs His approach uses dynamic program instrumentation to capture information during the program execution, which is then aggregated per function to reduce the memory required to store the results.

Extracting the critical path is a difficult problem. The online techniques used by Schulz and Hollingsworth use *piggybacking* (i.e., attaching additional data to MPI messages sent by the program) to transfer data about the critical path between processes at runtime. However, the piggybacking approach can only pass information in the program's communication direction, that is, from sender to receiver. Therefore, they can not correctly determine the critical path when wait states occur at synchronizing send operations (i.e., late receiver wait states), where information would need to be passed from the receiver to the sender.

Alexander et al. [4] already pointed out that the critical path itself is not overly expressive. To address this problem, they compute *near-critical paths* through search algorithms on execution graphs. They weigh each program edge with the computational complexity of the corresponding program section. However, their algorithm requires global searches across the entire graph, which is not scalable. Further, the algorithm results in a large number of paths that are considered near-critical. Each near-critical path typically varies little from the primary critical path, which reduces its usefulness.

Summary

The critical path is a powerful and expressive abstraction of the performance of a parallel program. However, critical-path techniques play only a minor role in current performance-analysis tools. In part, this minor role arises from the difficulty of isolating the critical path. Here, Scalasca's parallel post-mortem event-trace analysis is ideally suited to extract the critical path reliably and in a scalable manner.

More importantly, prior work has not fully exploited the information provided by critical-path analysis. In particular, the structure of the graph is either exposed in its entirety or lost in aggregated metrics. However, the critical path proves to be a valuable basis for identifying and quantifying load imbalance. The combination of summary profile metrics with a critical-path profile produces compact performance indicators that retain some of the dynamic information that the critical path provides. As demonstrated by the PEPC case study, this dynamic information is essential for accurately quantifying the performance impact of dynamic imbalances. In addition, the performance impact indicators help developers to recognize the specific nature of imbalance problems in MPMD programs.

Chapter 6

Comparative Study

Both the delay analysis and the critical-path analysis pinpoint performance bottlenecks related to imbalance and quantify their performance impact with respect to resource waste. In that sense, both techniques target similar performance analysis problems. This chapter examines both methods in comparison, discussing their differences and similarities, and outlines the use of both techniques in a real-world example study. Moreover, this chapter also covers general strengths and limitations of the event model and event-trace replay technique that provide the foundation for both the delay analysis and the critical-path analysis, and compare them to techniques used in related tools. Finally, a scalability study demonstrates the applicability of the trace analysis techniques to large-scale programs.

6.1 Functional Aspects

In some way, the delay analysis and the critical-path based performance indicators represent two different approaches to solving a similar problem, namely, characterizing imbalance in parallel programs. Looking back at the requirements for reliable imbalance analysis developed in Section 3.7, this section demonstrates that both approaches are viable solutions for identifying and characterizing imbalance, and points out their individual strenghts and weaknesses.

6.1.1 Functional Comparison

This section summarizes the basic functional properties of the delay and critical-path analysis. Both methods identify and characterize causes of imbalance, but do so from different angles.

Especially the delay costs and critical-path based performance impact indicators describe conceptually similar performance properties: both characterize the performance impact of inefficient parallelism (specifically, imbalance) in terms of the allocation time lost due to the imbalance. Basically, both approaches map the total amount of wait states found in the program onto call paths from which they originate. However, the mappings they produce may be slightly different. These differences are the result of the different goals and base assumptions underlying both approaches. The delay analysis determines delay costs individually for each synchronization interval, pinpointing any delay which leads to a wait state at a synchronization point. In contrast, the critical path characterizes the global performance impact of program activities on the program as a whole. Therefore, the critical-path based approach only pinpoints inefficiencies which have a *global* runtime effect. In particular, this implies that critical-path imbalance impact costs are exclusively assigned to call paths that appear on the critical path, whereas delay costs can also be assigned to call paths off the critical path. Therefore, the delay analysis determines the influence of delays on all wait states, while the critical-path imbalance characterizes their impact on the overall resource consumption. As a result of the different underlying principles, the delay analysis recreates the direct causal connection between wait states and their causes, whereas the critical-path imbalance indicator produces merely a correlation between wait states and extra-activities on the critical path.

While the delay costs and critical-path imbalance costs are conceptually closely related, each brings unqiue specializations. Choosing the best tool for a given task therefore depends on the specific performance issue to be investigated. The critical-path based performance impact indicators allow the distinction of intra- and inter-partition imbalance costs, which helps users to categorize the nature of imbalance problems in MPMD programs. This distinction would not easily be possible with the technical approach underlying the delay analysis. In contrast, the delay analysis gives detailed insights into wait-state formation with its distinction of propagating/terminal and direct/indirect wait states, as well as long-term and short-term delay costs. This distinction, with wait states as starting point, is in turn not possible with the critical-path based approach.

Finally, the critical-path imbalance indicator distinguishes itself from the other two methods by indicating imbalance costs in wall-clock time instead of allocation time. Although this property limits the usefulness of the indicator to SPMD programs, it produces typically very intuitive and easyly interpretable results.

6.1.2 Suitability

Section 3.7 outlined basic requirements that an accurate, generic automatic imbalance analysis solution should fulfill in order to detect the most important imbalance patterns. In summary, they should

- pinpoint imbalanced call paths in the program,
- make the imbalance pattern / load distribution transparent,
- quantify the severity of the imbalance in terms of its actual performance impact,
- take interference between different imbalances into account,
- take dynamic shift of imbalance over time into account,
- be applicable to any (e.g., SPMD and MPMD) parallelization scheme.

In the following, we will see how the delay analysis and the performance indicators based on the critical path fulfill these requirements.

Pinpointing imbalance problems

Being one of the main objectives of the delay and critical-path analysis methods in the first place, both approaches obviously help pinpointing locations where imbalance occurs. Users can identify program and process locations where imbalance leads to wait states from the delay costs associated to them; likewise, the critical-path imbalance indicator and imbalance cost indicators highlight program locations where imbalance has a significant performance impact. Due to Scalasca's instrumentation approach, program locations can be identified up to the level of functions or subroutines (with additional call-path information), which may be refined by adding additional, manual instrumentation to the source code.

Identifying imbalance patterns

Being able to recognize the pattern of an imbalance easily helps developers in understanding the underlying performance problem. For static imbalances, the basic workload distribution pattern can already be determined from the distribution of computation time in the Scalasca report browser. The visualization of the distribution of delay costs across processes helps isolating imbalance problems even better, as it accentuates high computation time gradients that are the root causes of wait states. Moreover, delay costs capture the full severity of dynamic imbalances, which may be diluted in the computation-time profile data visualization because of the data aggregation. Specifically, a minority of overloaded processes is easily identifyable in the delay-cost visualization (as, for example, in the Zeus/MP2 case presented in Section 4.4.1). The opposite case of an underloaded minority of processes can be just as easily identified by visualizing the distribution of wait-states themselves, a capability which Scalasca already posessed prior to adding the delay and critical-path analysis methods.

As seen in the CESM sea ice and Zeus/MP2 case studies in Section 4.4, Scalasca's ability to visualize performance data such as delay costs and wait-state propagation within the application's logical process topology is extremely helpful in relating the performance observations to internal causes of imbalance. Unfortunately, visualizing the evolution of imbalance patterns over time is at this point not straightforward within Scalasca's report browser Cube, because data in the performance reports is aggregated along the time dimension. It is only possible to create virtual call-paths that represent different iterations or timespans using additional instrumentation, which then allows users to study the evolution of performance characteristics – including delays and the critical path – over time. In the future, it is also concievable to combine the analysis techniques with different visualization approaches, for example, to examine the critical path in a time-line visualization tool such as Vampir.

Quantifying imbalance

As pointed out in Section 3.7, quantifying the severity of imbalances in different program locations in terms of the actual resource waste they cause should be one of the most important characteristics of an accurate imbalance analysis solution. In particular, the quantification needs to incorporate aggravation or attenuation effects resulting from superimposition of different imbalances, as shown in Section 3.3.3. In this regard, both the delay analysis and the critical-path based performance indicators present significant improvements over previous work. Specifically, delay costs are calculated based on the actual waiting time that occured and are only assigned to (positive) delays; therefore, the delay analysis accurately quantifies the actual contribution of imbalance in individual program and process locations to the formation of wait states even in the presence of interference with other imbalances. Likewise, the imbalance impact performance indicators based on the critical path also determine the performance impact of imbalances based on the actual resources lost to wait states, which guarantees that interference effects are correctly taken into account. Only the simpler critical-path imbalance indicator may overestimate the actual performance impact of an imbalance in the rare case of interfering imbalances with opposing workload distribution patterns, but it will nonetheless identify imbalances correctly when they have a negative impact.

Capturing performance dynamics

The next item on the list of requirements above states that an accurate imbalance analysis solution needs to account for performance-dynamic effects, i.e., dynamically changing workload distribution patterns in iterative programs, when quantifying the performance impact of imbalance. As discussed in Section 3.7, only methods that retain temporal information, such as time-series profiles or event traces, can capture such effects. In contrast to traditional profiling-based solutions, both the delay analysis and critical-path analysis methods operate on event traces, which capture the entire dynamic execution behavior of a program section of interest. Because the delay analysis is applied to each individual wait state, the delay costs implicitly cover dynamic effects. Likewise, the critical path retains dynamic information from which the performance indicators derived thereof can more accurately describe the impact of an imbalance even in the presence of dynamically changing workload distribution patterns. This advantage has been demonstrated in the PEPC case study in Section 5.4.1, where the critical-path imbalance indicator could quantify the impact of the dynamic imbalance in that code much better than regular runtime profiles alone.

Universal applicability

An important requirement for a generic performance analysis method is its universal applicability to a broad range of programs. In contrast to prior imbalance characterization schemes which are limited to SPMD-style programs, such as the ones from Tallent [80], Calzarossa [13], or Gamblin [22], the delay and critical-path analyses produce useful results for any parallelization scheme implementable in MPI, in particular including MPMD-style programs. Delay costs characterize the contribution of an imbalance to the formation of wait states irrespective of the parallelization scheme. Moreover, the critical-path based imbalance impact indicators even provide explicit support for the investigation of complex imbalance problems in MPMD decompositions. One exception is the simple wall-clock time based critical-path imbalance indicator: its definition includes the average time per process spent in a call path, which only has a useful meaning for SPMD-style program decompositions where each call path appears on every process.

While the delay and critical-path analysis concepts introduced in this thesis target message-passing programs only, it should be noted that the general approach is applicable to other parallel programming models as well. Indeed, message passing is a fundamental mechanism underlying most parallel programming models. For example, data exchange through shared memory buffers guarded with synchronization constructs in a shared-memory program can be interpreted as a message transfer. These similarities facilitate a simple adaption of the basic critical-path and delay analysis principles to, for example, the fork-join or task-based parallelism in OpenMP programs.

6.2 Technical Aspects

Both the delay and critical-path analyses share the same technological foundation, namely, Scalasca's event-trace analysis framework. This section discusses the various technical characteristics that influence accuracy and usability of the two analysis methods presented in this thesis. In this regard, we examine Scalasca's trace recording and analysis framework in general as well as both analysis methods individually with respect to their general applicability, the accuracy of the imbalance quantifications, and the scalability of the analysis algorithms. All results are based on case studies of a range of real-world example applications and benchmarks.

6.2.1 Trace-Analysis Applicability

As already outlined in Section 2.6, the event trace recording and replay approaches required for the delay and critical-path analyses are resource-intensive performance analysis methods compared to lightweight approaches such as profiling. Nonetheless, it is typically straightforward to obtain targeted event traces and perform Scalasca analyses for virtually all MPI programs even at large scale. To demonstrate the practicality of Scalasca's analysis approach on a broad set of real-world HPC codes, this section presents experiences gained while performing trace analyses of the SPECMPI 2007 benchmark suite. Later, Section 6.2.2 also demonstrates the scalability of the trace analysis; first, we discuss the measurement overheads or storage requirements of the trace analysis method.

Measurement and storage overhead

In combination with the study of imbalance patterns in Section 3.3, additional trace analysis experiments of the SPEC MPI2007 benchmark suite were performed on the Juropa cluster system using the setup described in Section 3.3.1. All benchmarks were configured to use a large ("lref") reference input data set (except for 104.milc, which lacks a large input data set and uses a medium one instead) and 256 MPI processes each. The "lref" runs took between 5 and 15 minutes on the Juropa cluster. For the event tracing experiments, all programs have been instrumented to record MPI events and user functions using instrumentation calls injected by the compiler. Custom filters have been created for each application to omit frequently executed but otherwise unimportant functions from the trace at runtime. With the

Table 6.1: Trace sizes of SPEC MPI2007 experiments

Benchmark	Functions		Trace size	
	Total	Filtered	Uncompressed	Compressed
104.milc	255	41	1.29 GB	0.45 GB
121.pop2	1443	29	20.64 GB	6.2 GB
122.tachyon	413	195	0.04 GB	0.01 GB
125.RAxML	403	5	1.51 GB	0.57 GB
126.lammps	1581	12	0.48 GB	0.19 GB
128.GAPgeofem	66	2	33.37 GB	8.4 GB
129.tera_tf	83	10	0.95 GB	0.35 GB
132.zeusmp2	198	1	1.38 GB	0.46 GB
137.lu	47	1	9.62 GB	3.1 GB
143.dleslie	36	10	30.36 GB	9.1 GB
145.lGemsFDTD	280	13	16.56 GB	4.5 GB
147.l2wrf2	2380	76	12.42 GB	4.3 GB

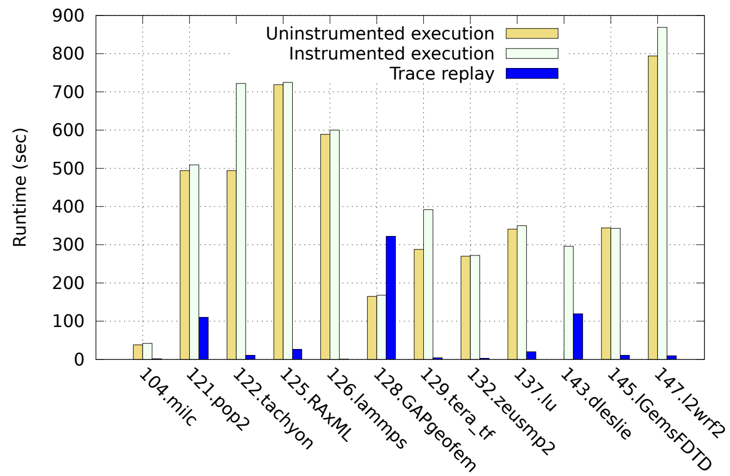


Figure 6.1: Uninstrumented and instrumented runtimes and trace replay time for the SPECMPI 2007 benchmarks. (Juropa cluster, 256 processes)

exception of 128.GAPgeofem, for which the number of iterations was decreased from 10,000 to 1,500 due to a massive amount of MPI point-to-point events, trace measurements could then be performed for all of the original reference benchmark configurations, producing between 170MiB and 9GiB of compressed trace data per application.

Table 6.1 shows the amount of trace data generated for each application, the total number of functions in the codes, and the number of functions that were filtered at runtime (i.e., excluded from the trace). The number of functions refers to the functions which have been automatically instrumented for measurement. This includes all functions defined by the program itself and the MPI operations it calls, but no functions from other external libraries. Obviously, as we can see from the total number of functions, the codes in the SPEC MPI2007 suite differ significantly in their complexity, with 147.l2wrf2 being the by far most complex code with more than 2380 functions. The size of the traces being created depends on the number of events generated during the program execution. As we can see in the table, trace sizes can vary greatly between different programs. Trace size matters for two reasons: first, it determines the buffer size required to store the (uncompressed) trace data in memory during measurement; second, the trace data needs to be temporarily stored on disk for the analysis. By default, the trace data will be written to disk in compressed form. As shown in the table, the compression reduces the amount of data by a factor of three to four.

Even more important than the trace size is the measurement overhead. The perturbation of the application's actual performance due to the instrumentation is of particular concern for the validity of the measurement. As a rule of thumb, a measurement should ideally not dilate the execution by more than 10% for best results. Figure 6.1 compares the runtime of the uninstrumented benchmark executables of the SPEC MPI2007 programs with their runtimes while recording the trace (excluding the additional overhead of writing trace data to disk at the end of the execution, which does not dilate the measurement itself). The comparison shows that the measurement dilation is acceptably low in most cases; only 122.tachyon, 129.tera_tf and 147.l2wrf2 show more than 10% dilation.

To keep the amount of trace data manageable and the measurement overhead at bay, some care needs to be taken when configuring trace experiments. Scalasca measurements can be configured to exclude unimportant but frequently executed functions from being recorded during the measurement using a filter file. Therefore, it is recommended to perform a profile experiment first to get an estimation of the expected amount of trace data and identify candidates for filtering before recording a trace. When filters need to be applied, it is important to find a good compromise between the expressiveness of the measurement results on the one hand and the associated measurement dilation and storage overhead on the other hand. As we can see from the number of filtered functions in Table 6.1, only few functions need to be filtered to reduce the overhead to an acceptable level in most cases. However, the benefit of runtime filtering can be limited for programs which heavily rely on very small, very frequently executed functions. With 122.tachyon, even heavy filtering could not reduce measurement dilation below 30%. We also find this pattern frequently in modern C++ codes. These programs often require more sophisticated instrumentation methods than filtering to produce optimal measurement results, for example explicit user-defined instrumentation of interresting code regions, sampling, or binary instrumentation.

Trace-analysis overhead

With Scalasca's parallel trace analysis approach, a certain amount of time of a user's resource allocation is required to perform the automatic trace analysis, which makes the resources required for the trace analysis itself an important factor to consider. The time required for the trace analysis does not depend on the time spent in the target application, but rather on the number of events generated per process, in particular the number of communication events. This is because communication events usually trigger more expensive actions in the trace replay; in particular, they activate the message transfers between analysis processes. Moreover, because of the number of different replay stages and analyses performed, a single communication event in the original application usually triggers multiple communication actions in the analysis. Therefore, the trace analysis time is usually negligible compared to the runtime of the target application for programs with a large computation-to-communication ratio, but the trace analysis might also take longer than the original program run time for extremely communication-intense programs.

However, the SPECMPI study shows that such extreme analysis times are exceptional. Figure 6.1 also depicts the time for the combined forward and backward parallel trace replays, including wait-state search, delay analysis, and critical-path analysis. For the communication-intensive 128.GAPgeofem application, the trace analysis takes almost twice as long as the original application run. Otherwise, the analysis time is only notable compared to the application runtime in the case of 121.pop2 and 143.dleslie; in all other cases, it is negligible.

6.2.2 Scalability

Since many performance problems in parallel programs only start to appear or become significant at large scale, it is especially important that performance analysis methods scale as well. Previous work [90] already demonstrated the scalability of Scalasca's wait-state search on configurations with up to 294,144 MPI processes. This section presents additional results that include the delay and critical-path analysis. For the scalability experiments, traces of the Sweep3D benchmark application [2] have been recorded and analyzed on the Blue Gene/P system Jugene at the Jülich Supercomputing Centre. Sweep3D is an MPI benchmark performing the core computation of a real ASCI application, a 1-group time-independent discrete ordinates neutron transport problem. It calculates the flux of neutrons through each cell of a three-dimensional grid (i, j, k) along several possible directions (angles) of travel. The angles are split into eight octants, corresponding to one of the eight directed diagonals of the grid. Sweep3D uses an explicit two-dimensional decomposition (i, j) of the three-dimensional computational domain, resulting in point-to-point communication of grid-points between neighboring processes, and reflective boundary conditions. A wavefront process is employed in the i and j directions, combined with pipelining of blocks of k-planes and octants, to expose parallelism. The benchmark ran in weak scaling mode with a constant problem size of $32 \times 32 \times 512$ cells per process.

The experiments were run with three different trace analysis configurations: the first configuration combines wait-state analysis, delay analysis, and critical-path analysis; the second configuration performs wait-state and delay analysis without critical-path analysis; and the last configuration performs wait-state and critical-path analysis without delay analysis. Note that the wait-state analysis is a prerequisite for both the delay analysis and the critical-path analysis, which is why it is always included. Moreover, the experiments also included measurements of the pure wait-state analysis for reference.

Figure 6.2 shows the wall-clock time needed for the trace analysis in the various configurations. The 32-fold doubling in the number of processes and the resulting large range of times necessitates a log-log scale in the plot. We can see that the trace replay in the complete configuration with wait-state, delay and critical-path analysis takes only slightly more time than the

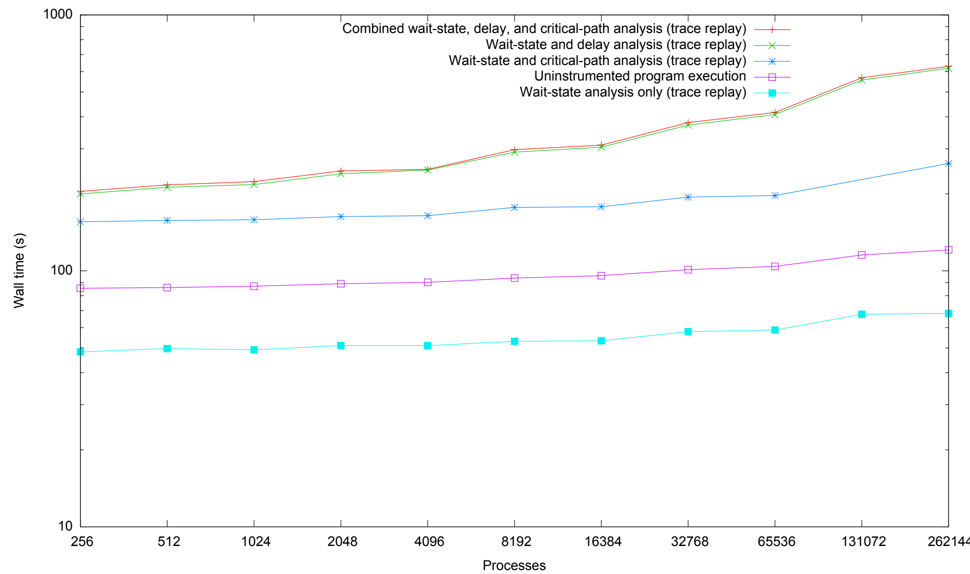


Figure 6.2: Scalability of the delay and critical-path analysis methods for the Sweep3D benchmark on Blue Gene/P.

configuration that includes wait-state and delay analysis, but not the critical-path analysis. In contrast, the configuration with only wait-state and critical-path analysis needs considerably less time than the complete configuration. Also, the critical-path analysis scales slightly better than the delay analysis, since the time difference between configurations with and without the delay analysis increases with the number of processes. Hence, the delay analysis is clearly the more expensive one of the two methods. Nonetheless, it has been shown that both methods perform well at scales present only in today's largest machines, being able to perform detailed trace analyses on 262,144 processes in a matter of minutes.

6.2.3 Limitations

Overall, the delay and critical-path analyses aim to provide detailed and accurate results for all programs based on point-to-point and collective MPI operations. However, there are some limitations and open issues of the underlying event model, which prevent the analyses from producing complete results in a few cases.

First and foremost, the accuracy of both the delay and critical-path analysis critically depends on the ability to correctly reconstruct all synchronization points of the original program run from the event trace. However, for certain MPI operations, the event trace produced by Scalasca does not contain the necessary information. Notably, of all point-to-point and collective communication operations, Scalasca does not record complete synchronization information for `MPI_Probe` or some of the collective operations that send or receive varying amounts of data on each process (specifically, the "-v" or "-w" variants of all-to-all collective communication operations, i.e. `MPI_Allgatherv`, `MPI_Alltoallv`, and `MPI_Alltoallw`). For these operations, users can specify a zero amount of data to be transferred on some processes. Therefore, not all of the processes need to actually participate in the data exchange, and hence synchronize with the others. In case of these collective operations, recording the complete synchronization information would require $\mathcal{O}(P)$ storage space per process for an operation on a communicator with P processes; which is of course prohibitively large and therefore unlikely to be realizable. The missing synchronization information in `MPI_Probe` on the other hand seems like an omission in the event model specification.

In addition to the operations mentioned so far, Scalasca's event model currently also lacks synchronization information for the (implicitly collective) communicator creation operations (e.g., `MPI_Comm_create` or topology creation operations such as `MPI_Cart_create`), and for synchronizing file I/O requests or operations (e.g., `MPI_File_open`).

The incomplete synchronization information implies that Scalasca cannot identify wait states inside these routines, even though they may possibly and quite likely occur there. Moreover, for programs that use any of these MPI operations, the delay and critical-path analysis may not produce accurate results. In case of the delay analysis, the problems are twofold: first, the analysis can obviously not determine delays that cause unidentified "hidden" wait states in the first place. Second, synchronization intervals for wait states after a hidden wait state may not be determined correctly. In this case, the detected synchronization interval will span across the hidden synchronization point, so that delays which occur after this point are still identified correctly. However, any waiting time which occurs at the hidden synchronization point itself will erroneously be classified as communication time. Nevertheless, it should be noted that even if not all synchronization points are identified, the delay analysis will typically still produce usable results for the remaining, successfully detected wait states. In contrast, the critical-path detection suffers severely from unidentified synchronization points: simply speaking, the detection algorithm may "miss a turn" and reconstruct the critical path incorrectly when a synchronization point was not identified. Therefore, results should be interpreted carefully when one of the MPI operations for which Scalasca lacks synchronization information appears on the critical path. Of these operations, the collective "-v"-operations are probably the most popular ones, and therefore pose the most severe restriction to the critical-path analysis. While communicator and topology creation operations are also quite widespread, they typically occur only in the initalization phase, so that the critical path will then still be reconstructed correctly for the remainder of the program.

Aside from the issues in detecting synchronization points mentioned so far, the delay and critical-path analysis (and, in fact, also the wait-state analysis) do at this point generally not work for MPI programs that use inter-communicators [54, Chapter 6.6 "Inter-Communication"] or dynamically spawn new processes [54, Chapter 10 "Process Creation and Management"]. Moreover, delay and critical-path analysis do not support MPI one-sided communication [54, Chapter 11 "One-Sided Communications"] yet. However, since only very few MPI programs make use of these functions, these constraints do not constitute severe limitations to the applicability of the delay and critical-path analysis in practice.

Summary

This chapter examined the capabilities of the delay and critical-path analyses with respect to the analysis of imbalance in comparison with each other. While the delay and critical-path based approach are based on different underlying concepts and focus on different analysis questions, both methods meet the previously established criteria for useful imbalance analysis solutions. In addition, both methods share the backward trace replay technique implemented in the extended Scalasca trace analysis framework. As demonstrated by the SPEC MPI2007 case studies and the Sweep3D scalability experiments, this trace-replay technique can be applied to a broad range of programs at very large scale.

Chapter 7

Conclusion & Outlook

This chapter provides a conclusion of the concepts developed in this dissertation, a summary of the open ends, and finally gives an outlook into new research areas to follow up on this work.

The central theme of developing new performance analysis techniques to detect and characterize imbalance in parallel programs was motivated by a lack of fully versatile and scalable solutions for this purpose in currently available tools. As a first step towards this goal, we first investigated the overall approach to analyzing imbalance itself. To that end, we identified and categorized the various imbalance patterns that occur in parallel programs, and derived requirements for their proper identification and characterization in a performance analysis tool. In particular, this survey showed that the characterization of imbalance needs to be based on the overall performance impact it has. The performance impact of imbalance is determined both by its magnitude and its pattern, and manifests itself in the form of wait states, which may occur in a large distance from their original cause. Moreover, imbalance analysis tools must consider dynamic changes of imbalance patterns over the course of the execution, which requires them to capture such performance dynamics. Overall, we found that the analysis of event traces provides the best foundation for the analysis of imbalances in parallel programs, whereas solutions based on summary profiles may not be able to detect and characterize imbalance properly. As a result, two new imbalance analysis solutions based on the scalable, automatic event trace search approach in the Scalasca performance analysis toolset have been developed.

The first of these new solutions, the delay analysis, identifies delays that cause wait states at subsequent synchronization points and determines the cost of these delays in terms of the overall waiting time they cause. Notably, the approach explicitly incorporates the long-distance impact of delays through wait-state propagation effects. In contrast to the serial algorithm used in the Carnival system – where the concept was first introduced by Meira et al. – the parallel algorithm introduced in this thesis is much more scalable, and facilitates the analysis of wait-state propagation itself by classifying wait states in terms of their propagation behaviour. The second new analysis method is based on the detection of the critical path. While there have been prior approaches that used the critical path in the context of parallel performance analysis, to the author's knowldege, this thesis is the first to point out its usefulness for the identification and characterization of load or communication imbalance. Because of their ability to retain dynamic performance characteristics in a compact structure, the resulting critical-path based imbalance identification methods pinpoint imbalanced program portions intuitively and capture dynamic imbalance much more accurately than approaches based on summary profiles.

Of course, the work on the topics is not finished with the end of the dissertation. So far, the existing implementations have only prototype status. As part of third-party funded projects, they are currently being integrated into the production-level Scalasca software package. Moreover, while the general approach of delay and critical-path analysis is applicable to many parallel programming models, the current implementation is limited to MPI-1 communication primitives (i.e., MPI point-to-point and collective communication). Support for additional programming models is therefore a primary work target. Work on support for the MPI-2 one-sided communication interface is currently underway, moreover, an early prototype with support for OpenMP and hybrid MPI/OpenMP programs has already been implemented. Here, studying the interaction between process-local imbalance and inter-process imbalance in hybrid programs that combine multi-threading with message passing presents a particularly interesting new research direction.

Further improvements in the analysis workflow can be achieved by combining the automatic analysis with other trace-based tools, such as the graphical Vampir trace browser. Scalasca already tracks the worst wait-state instances it finds and connects to third-party tools such as Vampir, where users can visually examine the affected region in the trace. This concept could be easily extended to also pinpoint the corresponding delay instances. Taking this approach even further, it is also thinkable to highlight the entire critical path in a trace timeline view in Vampir.

Finally, the imbalance detection methods should be embeddded into a broader optimization framework. Like all performance analysis tools, the critical-path and delay analyses help identifying and characterizing imbalances, but leave the task of actually removing them to the application developer. Ideally, the imbalance characterization should lead to specific suggestions for improvements. In general, however, this goal is difficult to accomplish, because solving real-world load or communication imbalance problems typically requires extensive modifications of the underlying algorithms and decomposition schemes, or even a complete redesign of the program. Similarly, “performance bugs” as another cause of inefficient parallelism are usally unique by their nature, and do not lend themselves to standardized solution approaches. Moreover, performance analysis methods are generally unaware of the semantics of code sections which exhibit imbalance, which further reduces the possibilities of generating specific solution advice on an individual basis automatically.

Nonetheless, strengthening the value of the analysis results for the user by providing guidelines to remediate the detected imbalances remains an intriguing research idea. Such a system could utilize a knowledge base of load-balancing strategies and provide suggestions based on parameters such as severity and temporal or spacial distribution patterns of any major imbalances which have been detected. Thinking further, this system could be embedded into an auto-tuning framework, where it could guide an automatic selection of pre-implemented decomposition strategies or load-balancing parameters based on the amount, locations, and patterns of imbalance which have been observed. Applied to MPMD programs, for example, the system could automatically adjust partition sizes based on the amount of inter-partition imbalance to minimize the overall imbalance costs.

Bibliography

[1] Top500 supercomputer sites. `http://www.top500.org`, June 2012.

[2] Accelerated Strategic Computing Initiative. The ASCI SWEEP3D Benchmark Code. `http://www.ccs3.lanl.gov/pal/software/sweep3d/sweep3d_readme.html`, 1995.

[3] Laksono Adhianto, S. Banerjee, Michael W. Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, April 2010.

[4] Cedell A. Alexander, Donna S. Reese, James C. Harden, and Ronald B. Brightwell. Near-Critical Path Analysis: A Tool for Parallel Program Optimization. In *Proceedings of the First Southern Symposium on Computing*, 1998.

[5] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on a Million Processors. In *Proceedings of the 16th European Parallel Virtual Machine and Message Passing Interface Conference*, volume 5759 of *Lecture Notes in Computer Science*, pages 20–30, September 2009.

[6] P. Barford and M. Crovella. Critical Path Analysis of TCP Transactions. *ACM SIGCOMM Computer Communication Review*, 31(2):80–102, 2001.

[7] Daniel Becker, Rolf Rabenseifner, Felix Wolf, and John Linford. Scalable timestamp synchronization for event traces of message-passing applications. *Parallel Computing*, 35(12):595–607, December 2009. Selected papers from the 14th European PVM/MPI Users Group Meeting.

[8] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.1*, 2011.

[9] David Böhme, Bronis R. de Supinski, Markus Geimer, Martin Schulz, and Felix Wolf. Scalable critical-path based performance analysis. In *Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS), Shanghai, China*, May 2012.

[10] David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. Identifying the root causes of wait states in large-scale parallel applications. In *Proc. of the 39th International Conference on Parallel Processing (ICPP, San Diego, CA, USA)*, pages 90–100. IEEE Computer Society, September 2010.

[11] BSC Performance Tools Team. Dimemas. `http://www.bsc.es/plantillaA.php?cat_id=475`.

[12] BSC Performance Tools Team. Paraver. `http://www.bsc.es/plantillaA.php?cat_id=485`.

[13] Maria Calzarossa, Luisa Massari, and Daniele Tessera. Load Imbalance in Parallel Programs. In *Parallel Computing Technologies*, Lecture Notes in Computer Science, pages 197–206, 2003.

[14] Maria Calzarossa, Luisa Massari, and Daniele Tessera. A methodology towards automatic performance analysis of parallel applications. *Parallel Computing*, 30(2):211–223, February 2004.

[15] Cray Inc. *Chapel Language Specification Version 0.91*, 2012.

[16] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Proc. of the 4th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '93)*, 1993.

[17] Luiz DeRose, Bill Homer, and Dean Johnson. Detecting Application Load Imbalance on High End Massively Parallel Systems. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes In Computer Science*, pages 150–159. Springer, 2007.

[18] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. Cray Performance Analysis Tools. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, 2008.

[19] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A hybrid multi-core parallel programming environment. In *Proc. of the First Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

[20] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[21] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. Scalable massively parallel I/O to task-local files. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC09), Portland, OR, USA*. ACM, November 2009.

[22] Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Rob Fowler, and Daniel A. Reed. Scalable load-balance measurement for SPMD codes. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[23] Markus Geimer, Marc-André Hermanns, Christian Siebert, Felix Wolf, and Brian J. N. Wylie. Scaling performance tool MPI communicator management. In *Proc. of the 18th European MPI Users' Group Meeting (EuroMPI), Santorini, Greece*, volume 6960 of *Lecture Notes in Computer Science*, pages 178–187. Springer, September 2011.

[24] Markus Geimer, Pavel Saviankou, Alexandre Strube, Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie. Further improving the scalability of the Scalasca toolset. In *Proc. of PARA 2010: State of the Art in Scientific and Parallel Computing, Part II: Minisymposium Scalable tools for High Performance Computing, Reykjavik, Iceland, June 6–9 2010*, volume 7134 of *Lecture Notes in Computer Science*, pages 463–474. Springer, 2012.

[25] Markus Geimer, Sameer S. Shende, Allen D. Malony, and Felix Wolf. A Generic and Configurable Source-Code Instrumentation Component. In Gabrielle Allen, Jarek Nabrzyski, Ed Seidel, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *Proc. of the International Conference on Computational Science (ICCS), Baton Rouge, LA, USA*, volume 5545 of *Lecture Notes in Computer Science*, pages 696–705. Springer, May 2009.

[26] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Abraham, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, April 2010.

[27] Michael Geissler, S Rykovanov, Jörg Schreiber, Jürgen Meyer-ter Vehn, and G D Tsakiris. 3D simulations of surface harmonic generation with few-cycle laser pulses. *New Journal of Physics*, 9(7):218, 2007.

[28] Michael Geissler, Jörg Schreiber, and Jürgen Meyer-ter Vehn. Bubble acceleration of electrons with few-cycle laser pulses. *New Journal of Physics*, 8(9):186, 2006.

[29] Michael Gerndt and Michael Ott. Automatic performance analysis with periscope. *Concurrency and Computation: Practice and Experience*, 22(6):736–748, 2010.

[30] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz. Extending Stability Beyond CPU Millennium: A Micron-Scale Atomistic Simulation of Kelvin-Helmholtz Instability. November 2007.

[31] Juan Gonzalez, Marc Casas, Miquel Moreto, Judit Gimenez, Jesus Labarta, and Mateo Valero. Simulating Whole Supercomputer Applications. *IEEE Micro*, 31(3):32–45, June 2011.

[32] John C. Hayes, Michael L. Norman, Robert A. Fiedler, James O. Bordner, Pak Shing Li, Stephen E. Clark, Asif Ud-Doula, and Mordecai-Mark MacLow. Simulating Radiating and Magnetized Flows in Multi-Dimensions with ZEUS-MP. *Astrophysical Journal Supplement*, 165:188–228, 2006.

[33] Michael T. Heath, Allen D. Malony, and Diane T. Rover. The Visual Display of Parallel Performance Data. *IEEE Computer*, 28(11):21–28, November 1995.

[34] Marc-Andre Hermanns, Markus Geimer, Felix Wolf, and Brian J. N. Wylie. Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In *Proceedings of the 17th International Conference on Parallel, Distributed, and Network-Based Processing*, February 2009.

[35] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Supercomputing 2010*, November 2010.

[36] Jeffrey K. Hollingsworth. An Online Computation of Critical Path Profiling. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '96)*, 1996.

[37] Kevin A. Huck and Jesus Labarta. Detailed Load Balance Analysis of Large Scale Parallel Applications. In *Proceedings of the 39th International Conference on Parallel Processing*, pages 535–544, September 2010.

[38] HyperTransport Consortium. *HyperTransport I/O Technology Overview*. HyperTransport Consortium, June 2004.

[39] IBM. *IBM High Productivity Computing Systems Toolkit*. IBM T.J. Watson Research Center, September 2009.

[40] ICL UoT. Performance Application Programming Interface (PAPI). `http://icl.cs.utk.edu/papi/index.html`, Jan 2012.

[41] Intel Corp. *An Introduction to the Intel QuickPath Interconnect*, Jan 2009.

[42] Intel Corp. Intel trace analyzer and collector. `http://software.intel.com/en-us/intel-trace-analyzer`, 2012.

[43] Intel Corp. Intel VTune Amplifier XE. `http://software.intel.com/en-us/intel-vtune-amplifier-xe`, 2012.

[44] Hassan M. Jafri. Measuring causal propagation of overhead inefficiencies in parallel applications. In *Proceedings of the 19th IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 237–243, November 2007.

[45] James E. Kelley, Jr and Morgan R. Walker. Critical-Path Planning and Scheduling. In *Proc. of the Eastern Joint IRE-AIEE-ACM Computer Conference, Boston, Massachusetts*, pages 160–173. ACM, December 1959.

[46] Khronos OpenCL Working Group. *The OpenCL Specification*. Khronos OpenCL Working Group, 2011.

[47] Andreas Knüpfer. *Advanced Memory Data Structures for Scalable Event Trace Analysis*. PhD thesis, Technische Universität Dresden, December 2008.

[48] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008.

[49] Dieter Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, Johannes Kepler Universität Linz, September 2000.

[50] Rick Kufrin. PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux. In *Proc. of the 6th International Conference on Linux Clusters: The HPC Revolution*, April 2005.

[51] Allen D. Malony and Sameer Shende. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[52] Wagner Meira Jr., Thomas J. LeBlanc, and Virgílio A. F. Almeida. Using cause-effect analysis to understand the performance of distributed programs. In *SPDT '98: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 101–111, New York, NY, USA, 1998. ACM.

[53] Wagner Meira Jr., Thomas J. LeBlanc, and Alexandros Poulos. Waiting time analysis and performance visualization in Carnival. In *SPDT '96: Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, pages 1–10, New York, NY, USA, 1996. ACM.

[54] Message Passing Interface Forum, Knoxville, Tennessee. *MPI: A Message-Passing Interface Standard*, 2.2 edition, June 2009.

[55] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[56] Barton P. Miller, Morgan Clark, Jeff K. Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Trans. on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[57] Bernd Mohr, Allen D. Malony, Sameer S. Shende, and Felix Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23(1):105–128, August 2002.

[58] Anna Morajko, Tomàs Margalef, and Emilio Luque. Design and implementation of a dynamic tuning environment. *Journal of Parallel and Distributed Computing*, 67(4):474 – 490, 2007.

[59] Jan Mußler, Daniel Lorenz, and Felix Wolf. Reducing the overhead of direct application instrumentation using prior static analysis. In *Proceedings of Europar 2011, Bordeaux, France*, volume 6852 of *LNCS*, pages 65–76. Springer, September 2011.

[60] Aroon Nataraj, Allen D. Malony, Sameer Shende, and Alan Morris. Kernel-level measurement for integrated parallel performance views: the KTAU project. In *Proc. of the 2006 IEEE Conference on Cluster Computing*, pages 1–12, September 2006.

[61] Aroon Nataraj, Alan Morris, Allen D. Malony, Matthew Sottile, and Pete Beckman. The ghost in the machine: Observing the effects of kernel operation on parallel application performance. In *Supercomputing 2007*, pages 1–12, 2007.

[62] Jarek Nieplocha, Vinod Tipparaju, Manoj Krishnan, and Dhabaleswar Panda. High performance remote memory access comunications: The ARMCI approach. *International Journal of High Performance Computing and Applications*, 20(2):233–253, 2006.

[63] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. ScalaTrace: Scalable Compression and Replay of Communication Traces for High Performance Computing. *Journal of Parallel and Distributed Computing*, 69(8):696–710, August 2009.

[64] Robert W. Numrich. and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[65] NVIDIA Corp. CUDA parallel computing platform. `http://www.nvidia.com/object/cuda_home_new.html`, 2012.

[66] NVIDIA Corp. NVidia Visual Profiler. `http://developer.nvidia.com/cuda/nvidia-visual-profiler`, 2012.

[67] OpenACC. *The OpenACC Application Programming Interface*, November 2011.

[68] David Padua, editor. *Encyclopedia of Parallel Computing*, volume 2. Springer, 2011.

[69] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of the ACM/IEEE Supercomputing 2003 Conference*, 2003.

[70] S. Pfalzner and P. Gibbon. *Many-Body Tree Methods in Physics*. Cambridge University Press, New York, September 2005.

[71] D. F. Richards, J. N. Glosli, B. Chan, M. R. Dorr, E. W. Draeger, J.-L. Fattebert, W. D. Krauss, T. Spelce, F. H. Streitz, M. P. Surh, and J. A. Gunnels. Beyond Homogeneous Decomposition: Scaling Long-Range Forces on Massively Parallel Systems. November 2009.

[72] Barry Rountree, David Lowenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. Adagio: making DVS practical for complex HPC applications. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*, 2009.

[73] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. *X10 Language Specification Version 2.2*. IBM, 2012.

[74] Martin Schulz. Extracting Critical Path Graphs from MPI Applications. In *Proceedings of the 7th IEEE International Conference on Cluster Computing*, September 2005.

[75] Martin Schulz, Joshua A. Levine, Peer-Timo Bremer, Todd Gamblin, and Valerio Pascucci. Interpreting performance data across intuitive domains. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 206 –215, sept. 2011.

[76] Standard Performance Evaluation Corporation. SPEC MPI2007 benchmark suite, 2007.

[77] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[78] Zoltán Szebenyi, Felix Wolf, and Brian J. N. Wylie. Space-Efficient Time-Series Call-Path Profiling of Parallel Applications. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC09), Portland, OR, USA*. ACM, November 2009.

[79] Zoltán Szebenyi, Brian J. N. Wylie, and Felix Wolf. Scalasca Parallel Performance Analyses of PEPC. In *Proc. of the 1st Workshop on Productivity and Performance (PROPER) in conjunction with Euro-Par 2008, Las Palmas de Gran Canaria, Spain*, volume 5415 of *Lecture Notes in Computer Science*, pages 305–314. Springer, 2009.

[80] Nathan R. Tallent, Laksono Adhianto, and John Mellor-Crummey. Scalable Identification of Load Imbalance in Parallel Executions using Call Path Profiles. In *Supercomputing 2010*, New Orleans, LA, USA, November 2010.

[81] Christian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *Proc. of the ACM/IEEE Conference on Supercomputing (SC02)*, November 2002.

[82] The Krell Institute. Open—SpeedShop. `http://www.openspeedshop.org`, 2012.

[83] Mustafa M. Tikir, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, August 2009.

[84] D. Tullsen and B. Calder. Computing Along the Critical Path. Technical report, UC San Diego, 1998.

[85] University Corporation for Atmospheric Research (UCAR). The Community Earth System Model. `http://www.cesm.ucar.edu/`, Februar 2012.

[86] UPC Consortium. *UPC Language Specifications V1.2*, 2005.

[87] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *SciDAC 2005 Proceedings (Journal of Physics)*, Juli 2005.

[88] R. Clint Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2001.

[89] Felix Wolf, Bernd Mohr, Jack Dongarra, and Shirley Moore. Automatic analysis of inefficiency patterns in parallel applications. *Concurrency and Computation: Practice and Experience*, 19:1481–1496, February 2007. Special Issue *Automatic Performance Analysis*.

[90] Brian J. N. Wylie, David Böhme, Bernd Mohr, Zoltán Szebenyi, and Felix Wolf. Performance analysis of Sweep3D on Blue Gene/P with the Scalasca toolset. In *Proc. 24th Int'l Parallel & Distributed Processing Symposium and Workshops (IPDPS, Atlanta, GA)*. IEEE Computer Society, April 2010.

1. **Three-dimensional modelling of soil-plant interactions: Consistent coupling of soil and plant root systems**
   by T. Schröder (2009), VIII, 72 pages
   ISBN: 978-3-89336-576-0
   URN: urn:nbn:de:0001-00505

2. **Large-Scale Simulations of Error-Prone Quantum Computation Devices**
   by D. B. Trieu (2009), VI, 173 pages
   ISBN: 978-3-89336-601-9
   URN: urn:nbn:de:0001-00552

3. **NIC Symposium 2010**
   Proceedings, 24 – 25 February 2010 | Jülich, Germany
   edited by G. Münster, D. Wolf, M. Kremer (2010), V, 395 pages
   ISBN: 978-3-89336-606-4
   URN: urn:nbn:de:0001-2010020108

4. **Timestamp Synchronization of Concurrent Events**
   by D. Becker (2010), XVIII, 116 pages
   ISBN: 978-3-89336-625-5
   URN: urn:nbn:de:0001-2010051916

5. **UNICORE Summit 2010**
   Proceedings, 18 – 19 May 2010 | Jülich, Germany
   edited by A. Streit, M. Romberg, D. Mallmann (2010), iv, 123 pages
   ISBN: 978-3-89336-661-3
   URN: urn:nbn:de:0001-2010082304

6. **Fast Methods for Long-Range Interactions in Complex Systems**
   Lecture Notes, Summer School, 6 – 10 September 2010, Jülich, Germany
   edited by P. Gibbon, T. Lippert, G. Sutmann (2011), ii, 167 pages
   ISBN: 978-3-89336-714-6
   URN: urn:nbn:de:0001-2011051907

7. **Generalized Algebraic Kernels and Multipole Expansions for Massively Parallel Vortex Particle Methods**
   by R. Speck (2011), iv, 125 pages
   ISBN: 978-3-89336-733-7
   URN: urn:nbn:de:0001-2011083003

8. **From Computational Biophysics to Systems Biology (CBSB11)**
   Proceedings, 20 - 22 July 2011 | Jülich, Germany
   edited by P. Carloni, U. H. E. Hansmann, T. Lippert, J. H. Meinke, S. Mohanty, W. Nadler, O. Zimmermann (2011), v, 255 pages
   ISBN: 978-3-89336-748-1
   URN: urn:nbn:de:0001-2011112819

9. **UNICORE Summit 2011**
Proceedings, 7 - 8 July 2011 | Toruń, Poland
edited by M. Romberg, P. Bała, R. Müller-Pfefferkorn, D. Mallmann (2011), iv, 150 pages
ISBN: 978-3-89336-750-4
URN: urn:nbn:de:0001-2011120103

10. **Hierarchical Methods for Dynamics in Complex Molecular Systems**
Lecture Notes, IAS Winter School, 5 – 9 March 2012, Jülich, Germany
edited by J. Grotendorst, G. Sutmann, G. Gompper, D. Marx (2012), vi, 540 pages
ISBN: 978-3-89336-768-9
URN: urn:nbn:de:0001-2012020208

11. **Periodic Boundary Conditions and the Error-Controlled Fast Multipole Method**
by I. Kabadshow (2012), v, 126 pages
ISBN: 978-3-89336-770-2
URN: urn:nbn:de:0001-2012020810

12. **Capturing Parallel Performance Dynamics**
by Z. P. Szebenyi (2012), xxi, 192 pages
ISBN: 978-3-89336-798-6
URN: urn:nbn:de:0001-2012062204

13. **Validated force-based modeling of pedestrian dynamics**
by M. Chraibi (2012), xiv, 112 pages
ISBN: 978-3-89336-799-3
URN: urn:nbn:de:0001-2012062608

14. **Pedestrian fundamental diagrams: Comparative analysis of experiments in different geometries**
by J. Zhang (2012), xiii, 103 pages
ISBN: 978-3-89336-825-9
URN: urn:nbn:de:0001-2012102405

15. **UNICORE Summit 2012**
Proceedings, 30 - 31 May 2012 | Dresden, Germany
edited by V. Huber, R. Müller-Pfefferkorn, M. Romberg (2012), iv, 143 pages
ISBN: 978-3-89336-829-7
URN: urn:nbn:de:0001-2012111202

16. **Design and Applications of an Interoperability Reference Model for Production e-Science Infrastructures**
by M. Riedel (2013), x, 270 pages
ISBN: 978-3-89336-861-7
URN: urn:nbn:de:0001-2013031903

17. **Route Choice Modelling and Runtime Optimisation for Simulation of Building Evacuation**
by A. U. Kemloh Wagoum (2013), xviii, 122 pages
ISBN: 978-3-89336-865-5
URN: urn:nbn:de:0001-2013032608

18. **Dynamik von Personenströmen in Sportstadien**
by S. Burghardt (2013), xi, 115 pages
ISBN: 978-3-89336-879-2
URN: urn:nbn:de:0001-2013060504

19. **Multiscale Modelling Methods for Applications in Materials Science**
by I. Kondov, G. Sutmann (2013), 326 pages
ISBN: 978-3-89336-899-0
URN: urn:nbn:de:0001-2013090204

20. **High-resolution Simulations of Strongly Coupled Coulomb Systems with a Parallel Tree Code**
by M. Winkel (2013), xvii, 196 pages
ISBN: 978-3-89336-901-0
URN: urn:nbn:de:0001-2013091802

21. **UNICORE Summit 2013**
Proceedings, 18th June 2013 | Leipzig, Germany
edited by V. Huber, R. Müller-Pfefferkorn, M. Romberg (2013), iii, 94 pages
ISBN: 978-3-89336-910-2
URN: urn:nbn:de:0001-2013102109

22. **Three-dimensional Solute Transport Modeling in Coupled Soil and Plant Root Systems**
by N. Schröder (2013), xii, 126 pages
ISBN: 978-3-89336-923-2
URN: urn:nbn:de:0001-2013112209

23. **Characterizing Load and Communication Imbalance in Parallel Applications**
by D. Böhme (2014), xv, 111 pages
ISBN: 978-3-89336-940-9
URN: urn:nbn:de:0001-2014012708

The amount of parallelism in modern supercomputers currently grows from generation to generation. Further application performance improvements therefore depend on software-managed parallelism: the software must organize data exchange between processing elements efficiently and optimally distribute the workload between them. Performance analysis tools help developers of parallel applications to evaluate and optimize the parallel efficiency of their programs. This dissertation presents two novel methods to automatically detect imbalance-related performance problems in MPI programs and intuitively guide the performance analyst to inefficiencies whose optimization promise the highest benefit.

The first method, the delay analysis, identifies the root causes of wait states. A delay occurs when a program activity needs more time on one process than on another, which leads to the formation of wait states at a subsequent synchronization point. Wait states are the primary symptom of load imbalance in parallel programs. While wait states themselves are easy to detect, the potentially large temporal and spatial distance between wait states and the delays causing them complicates the identification of wait-state root causes.

The delay analysis closes this gap, accounting for both short-term and long-term effects. The second method is based on the detection of the critical path, which determines the effect of imbalance on program runtime. The critical path is the longest execution path in a parallel program without wait states: optimizing an activity on the critical path will reduce the program's runtime. Comparing the duration of activities on the critical path with their duration on each process yields a set of novel, compact performance indicators. These indicators allow users to evaluate load balance, identify performance bottlenecks, and determine the performance impact of load imbalance at first glance by providing an intuitive understanding of complex performance phenomena.

Both analysis methods leverage the scalable event-trace analysis technique employed by the Scalasca toolset: by replaying event traces in parallel, the bottleneck search algorithms can harness the distributed memory and computational resources of the target system for the analysis, allowing them to process even large-scale program runs.

This publication was written at the Jülich Supercomputing Centre (JSC) which is an integral part of the Institute for Advanced Simulation (IAS). The IAS combines the Jülich simulation sciences and the supercomputer facility in one organizational unit. It includes those parts of the scientific institutes at Forschungszentrum Jülich which use simulation on supercomputers as their main research methodology.

IAS Series
Volume 23
ISBN 978-3-89336-940-9

JÜLICH
FORSCHUNGSZENTRUM