Publication Series of the John von Neumann Institute for Computing (NIC)
NIC Series Volume 17

John von Neumann Institute for Computing (NIC)

Felix Wolf

Automatic Performance Analysis on Parallel Computers with SMP Nodes

NIC Series Volume 17 ISBN 3-00-010003-2

Central Institute for Applied Mathematics

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

Publisher: NIC-Directors
Distributor: NIC-Secretariat
Research Centre Jülich
52425 Jülich
Germany
Internet: www.fz-juelich.de/nic
Printer: Graphische Betriebe, Forschungszentrum Jülich

NIC Series Volume 17 ISBN 3-00-010003-2

Automatic Performance Analysis on Parallel Computers with SMP Nodes

Von der Fakultät für Elektrotechnik und Informationstechnik der
Rheinisch-Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften
genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Felix Wolf

aus München

Berichter: Universitätsprofessor Dr. rer. nat. Friedel Hoßfeld
Universitätsprofessor Christian H. Bischof, Ph.D.

Tag der mündlichen Prüfung: 27.11.2002

Abstract

Parallel computers with SMP nodes provide both multithreading and message passing as their modes of parallel execution. This thesis addresses the complexity of the performance problems that can arise in these systems by formally characterizing the problems in terms of execution patterns that represent situations of inefficient behavior. These patterns are specified as compound events which are input for an automatic analysis process that recognizes and quantifies the inefficient behavior in event traces. Mechanisms that hide the complex relationships within compound-event specifications allow a simple description of complex inefficient behavior on a high level of abstraction.

The analysis process automatically transforms event traces into a scalable representation of performance behavior, allowing a fast and easy identification of performance bottlenecks on varying levels of granularity along the dimensions of problem type, call graph, and process or thread. The uniform mapping of performance behavior onto the corresponding fraction of execution time enables the convenient correlation of different performance behavior using only a single integrated view. A modular analysis architecture separates the performance-problem specifications from the actual analysis process, simplifying the extension and customization of predefined performance problems to meet individual (e.g., application-specific) needs.

To demonstrate the methodology in real parallel-programming environments, it was applied to the programming interfaces MPI, OpenMP, and their combination. To show the methodology's usefulness in practice, the performance-tool prototype EXPERT was implemented and successfully tested for several real-world applications.

Kurzfassung

Parallelrechner mit SMP-Knoten bieten sowohl Multithreading als auch Message-Passing als parallele Programmiermodelle. Diese Dissertationsschrift behandelt die potenziellen Leistungsprobleme solcher Systeme mit Hilfe einer formalen Beschreibung von Ausführungsmustern, die Situationen ineffizienten Verhaltens repräsentieren. Die Muster werden als Verbundereignisse spezifiziert und dienen als Eingabe für einen automatischen Analyseprozess, der das ineffiziente Verhalten in Ereignisspuren nachweist und quantifiziert. Mechanismen zur Kapselung komplexer Beziehungen innerhalb der Verbundereignisspezifikationen erlauben eine einfache Beschreibung komplexen ineffizienten Verhaltens auf hohem Abstraktionsniveau.

Der Analyseprozess transformiert Ereignisspuren automatisch in eine skalierbare Repräsentation des Leistungsverhaltens, die eine schnelle und einfache Identifizierung von Leistungsengpässen auf beliebigen Granularitätsstufen entlang der Dimensionen Problemtyp, Aufrufpfad, und Prozess oder Thread erlaubt. Die einheitliche Abbildung des Leistungsverhaltens auf den entsprechenden Anteil der Ausführungszeit ermöglicht den mühelosen Vergleich unterschiedlichen Verhaltens in einer einzigen integrierten Darstellung. Eine modulare Analysearchitektur separiert die Spezifikationen der Leistungsprobleme vom eigentlichen Analyseprozess, was die Erweiterung und Anpassung vordefinierter Leistungsprobleme an individuelle (z.B. anwendungsspezifische) Bedürfnisse gestattet.

Zur Verwendung in realen parallelen Programmierumgebungen wurde dieser Ansatz auf die Programmierschnittstellen MPI, OpenMP und deren Kombination angewandt. Zum Nachweis der Praxistauglichkeit wurde das Leistungsanalysewerkzeug EXPERT prototypisch implementiert und erfolgreich anhand realer Anwendungen getestet.

Acknowledgments

This thesis was written at the Zentralinstitut für Angewandte Mathematik (ZAM) of Forschungszentrum Jülich. I would like to thank my advisor, Prof. Dr. Friedel Hoßfeld, who has the chair of Technical Informatics and Computer Science at Aachen University of Technology and who was director of the ZAM during my time as a PhD student, for giving me the opportunity to work in this excellent and supportive environment and for his equally generous and wise guidance throughout the course of my PhD. Also, I would like to thank Prof. Dr. Christian Bischof for serving as the second referee. In addition, I would like to acknowledge Dr. Rüdiger Esser, head of the ZAM department Programming Techniques, for his continuous support of my thesis project.

It was a great privilege to work with Dr. Bernd Mohr, and I would like to express my deep gratitude for his creative influence and his selfless support.

It was a pleasure to carry out my thesis research as a part of the IST working group APART, whose members contributed many ideas, created an inspiring atmosphere at various meetings, and are great colleagues. In particular, I would like to thank Prof. Dr. Allen Malony for carefully reviewing my thesis prior to its submission and for all his motivating comments during the last three years. I am also grateful to Prof. Dr. Michael Gerndt for his constant encouragement and his many helpful suggestions, especially during the early stages of my thesis research. I also would like to thank Dr. Luiz DeRose for creating the opportunity of a visiting research studentship at the IBM T. J. Watson Research Center.

Special thanks go to Janet Carter-Sigglow, who spent long hours polishing the language of my thesis manuscript. Thanks are also due to Dr. Bernhard Steffen and Dr. Peter Weidner, who took care of its mathematical correctness. I am indebted to Prof. Dr. Rudolf Berrendorf and Ulrich Detert, who jointly built up the ZAMpano SMP cluster, which served as the primary development platform of the prototype described in this thesis. I am grateful to Wolfgang Frings and Reiner Vogelsang for helping me conduct my experiments, to Deutsches Klima Rechenzentrum, the Institut für Chemie und Dynamik der Geosphäre IV, and the Institut für Festkörperforschung for giving me access to their applications, to Dietrich Bartel for his assistance in graphical questions, and to Arpad Kiss for providing the basic tree-browser implementation of the prototype's user interface. And finally thanks to all other colleagues who provided support and helped me complete this thesis.

I had the pleasure of writing my thesis at the ZAM in parallel with Stefan Birmanns, who shared with me all the ups and downs of being a PhD student. Finally, I feel a deep sense of gratitude to my parents and to Ilka Müller for their patience and their support during the PhD period.

Felix Wolf
December 2002, Jülich, Germany

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the last few decades parallel computing has proved to be an essential tool for the solution of complex scientific and economic problems. The numerical simulation of physical, chemical, and biological processes provides insight into phenomena that either cannot be addressed by analytical or experimental methods or that require experiments that are too expensive or dangerous. Parallel computing also plays a key role in achieving and preserving scientific and thus economic competitiveness. As more powerful computing resources become available, grand-challenge applications, such as protein-structure prediction and weather prediction, will become reality.

A parallel computer effectively multiplies the performance of single processors. Unfortunately, real applications frequently fail to sustain even a major fraction of the theoretical performance limit that is possible on a given parallel machine. The reason for this gap between peak and real performance lies in the complex interactions among the hardware, system software, programming interface, and algorithm. Understanding the effects of these interactions is crucial for optimizing parallel programs and thus for a better utilization of the available computer hardware.

1.1 Architectures of Parallel Computers

Parallel computers are computers with multiple processors that are able to work jointly on one or more tasks at the same time. One common way to classify parallel computers is based on memory architecture. There are two major classes: *shared memory* and *distributed memory*.

Shared-memory machines, which are also referred to as *symmetric multiprocessors* or *shared-memory multiprocessors* (SMPs) [41], have symmetric access to one shared address

space and are controlled by one operating-system image. This makes it possible, for example, to suspend a process on one processor and to resume it on another processor without copying or moving its address space.

SMPs that share one physical memory belong to the class of UMA (*Uniform Memory Access*) computers and provide symmetric and equally fast access to all addresses of the shared address space. Examples are CRAY T90, IBM 390, SUN E10000.

SMPs that provide a shared address space based on physically distributed memory [31] have variable access times to a memory address depending on the physical distance to that address. These machines are called NUMA (*Non-Uniform Memory Access*) computers. ccNUMA (*cache coherent Non-Uniform Memory Access*) computers are similar to NUMA computers but provide a mechanism for local buffering of remote memory contents in a cache after the first access so that subsequent accesses to the same memory location can be much faster. Cache-coherence protocols ensure that modifications of cached or original data occur consistently. Examples are SGI ORIGIN 2000 and HP V-Class.

Distributed-memory parallel systems, which are often referred to as as *massively parallel processors* (MPPs) when larger numbers of processors are used, do not provide a shared address space. Each memory is local to one processor and not accessible from another processor. Message passing is used to move data between processors. However, some systems provide mechanisms to access remote memory locations on the hardware level. Examples are CRAY T3E and IBM RS/6000-SP.

1.2 Coupled SMP Systems

In the past, MPP systems dominated the scientific computing market, but they claimed only a minor share of the industrial market. In contrast, SMP systems, which are frequently used as database servers, gained increasing popularity both in research and industry. For this reason, more powerful and cheaper SMP systems are likely to become available in the future. However, single SMP systems will not be able to meet the performance requirements of many large-scale applications. Coupling multiple SMP systems is one way to increase the number of processors and thus to provide sufficient computing power to handle such large-scale high-performance problems.

Hoßfeld et al. [40] distinguish between *parallel computers with* SMP *nodes* and *clustered* SMPs. Parallel computers with SMP nodes are tightly coupled over a dedicated network and present themselves to the user as one uniform computer system. Clustered SMPs are only loosely coupled, for example, over a local area network. Both types of architecture are called *coupled* SMP *systems*. By the nature of their memory-system architecture, coupled SMP systems are also distributed-memory systems because memory is distributed across multiple SMP nodes.

Hence, in contrast to single SMPs, coupled SMPs introduce an additional level in the memory-hardware architecture, which now forms a hierarchy of distributed shared memories. Unfortunately, this memory hierarchy further complicates the performance behavior and makes parallel programming more difficult. While the expected economic advantages argue for coupled SMP solutions to high-performance computing, the complex hardware structure of coupled SMPs creates a strong need for programming tools that provide assistance in writing efficient codes for these platforms.

However, coupled SMPs are interesting for another reason as well. This class of computers implements a very general architectural concept, which contains other architectures, such as distributed and shared memory, and, of course, simple sequential architectures, as specializations. So most of the programming tools that apply to coupled SMPs can be used for these subclasses, too.

1.3 Interconnection Networks

The different nodes of a coupled SMP system communicate over an interconnection network. The network performance has a major influence on the overall performance of the system. There are a variety of network topologies that differ in node degree, network diameter, and bisection width.

The nodes of clustered SMPs are often connected with a *local-area network* (LAN) or a *wide-area network* (WAN). In this context, LAN technologies, such as Ethernet, FDDI, ATM, and HiPPI, which are described in more detail in [71], come into operation.

In principle, LAN technologies can also be used to equip parallel computers with SMP nodes, but in most cases these computers use *system-area networks* (SANs), which have been developed specifically to provide better bandwidth and latency by circumventing operating-system protocol stacks. Examples of general-purpose SANs frequently found in the PC-cluster area are Myrinet [60] and SCI (Scalable Coherent Interface) [43].

1.4 Programming Models

The choice of programming models for coupled SMPs is influenced by the hierarchical memory architecture, which provides shared memory inside single nodes and distributed memory across different nodes. In principle, a shared address space across all SMP nodes is technically feasible but it usually requires sophisticated hardware or software solutions, such as reflective memory [44] or TreadMarks [2], respectively.

For this reason, the primary programming model for coupled SMPs is *message passing* because it provides a simple way to communicate across node borders. Inside single nodes, programs may alternatively use the *shared-memory* model. If both message-passing and shared-memory programming are used for a coupled-SMP program, it is commonly referred to as a *hybrid* programming model.

A distinctive feature of the shared-memory model is that it provides implicit communication over shared-memory locations, whereas message passing requires communication to be made explicit using dedicated operations. Common to both is that each processor executes different control flows, which corresponds to the *multiple-instruction stream – multiple-data stream* (MIMD) model in Flynn's classification [25]. Often the instructions come from the same program, in which case the whole computation is a *single-program, multiple-data* (SPMD) computation.

The following subsections give a brief introduction to all three programming models.

1.4.1 Message Passing

Message passing is mainly used on distributed-memory architectures. A message-passing program runs multiple processes, where each process owns one private address space. Communication among different processes takes place only by sending and receiving messages. The messages may be sent either via a network or using shared memory locations if available. Communication between two processes occurs either two-sidedly, where both participating processes have to invoke an operation, or one-sidedly, where only one process has to invoke an operation.

The MPI (*Message Passing Interface*) communication library [52, 53] defines a de facto standard for message passing and is available on most parallel computers. The latest version, MPI 2.0, supports all traditional message-passing features, such as point-to-point communication and collective communication, advanced features, such as process topologies and one-sided communication, but also features that go beyond pure message passing, such as parallel IO.

1.4.2 Shared Memory

A shared-memory program consists of a collection of *tasks*, which are assigned to asynchronously working threads. To accomplish these tasks, all threads have access to a shared address space. Synchronization utilizes specific mechanisms, such as locks and barriers, to implement coherent control of shared-memory access.

The shared-memory programming model comes in three varieties. UNIX System V supports *shared segments*, which provide a mechanism to define shared memory segments and

map them onto the virtual address space of different processes. Programs based on *threads* first create one master thread and later fork additional threads depending on the work to be distributed. In this case, all threads share the same address space and the programmer uses synchronization primitives for sharing memory. The third approach is sequential-program parallelization. Here, the programmer inserts directives or pragmas that assist the compiler in automatically parallelizing computation-intensive code sections.

OpenMP (*Open specifications for Multi Processing*) [61, 62] is a widespread programming interface for scientific shared-memory programming. It defines directives, pragmas, and library calls to control the parallelization of loops and other code sections in Fortran, C, and C++ programs. Execution of an OpenMP program starts with one master thread, which creates a team of slave threads as soon as a parallel region has been entered. After leaving this region, the team terminates and sequential execution resumes. Synchronization is accomplished either implicitly or explicitly by certain directives, pragmas, or library calls. OpenMP implementations are usually based on a low-level thread library.

1.4.3 Hybrid Model

Coupled SMP systems can be programmed using a hybrid combination of message-passing and shared-memory techniques, where shared-memory is used for data sharing inside single nodes and message passing is used for communication across different nodes. Most significant in this context is the combination of MPI and OpenMP. In this case, there is usually one MPI process per SMP node, and OpenMP parallelization can occur in each process. If the application needs to call MPI routines from multiple threads belonging to the same process, a thread-safe MPI implementation is required.

1.5 Automatic Performance Analysis

The process of investigating the performance behavior of an application and finding the reasons for limited performance is called *performance analysis*. It usually precedes any modification of the source code that is intended to optimize or tune the program. Both activities form a cycle that must often be repeated many times until the application delivers the desired performance.

Performance analysis includes several complicated and time-consuming tasks. The developer usually compares a hypothesis of performance, which may be based on a performance model, to objective observations of the run-time behavior. To do so requires instrumenting and monitoring the application. To draw reasonable conclusions from the collected performance data, the data may need several postprocessing steps. Finally, the developer searches through the data, tries to (dis)prove the hypothesis, and thinks about ways to improve the

application's performance behavior. Clearly, performance analysis demands a significant fraction of the overall time required for development and appropriate programming tools could both save time and improve the quality of this process.

Although during the last few decades many achievements have been made, the current situation still suffers from the lack of a software infrastructure that supports all these steps in a satisfactory, automatic manner. Powerful tools, such as VGV [38], provide valuable assistance in analyzing the performance of MPI and OpenMP programs by visualizing the run-time behavior and calculating statistics over the performance data. However, the developer is still required to filter out relevant parts from a huge amount of low-level information and map that information onto the application-program abstractions without tool support. Furthermore, many approaches, such as OPAL [30], are compiler- or language-dependent and, thus, restricted in their portability.

Automating the process of analyzing the performance means automatically delivering the information that is necessary to understand the reasons for inefficient program behavior. Thus, it aims at both reducing the amount of work that is left to the software developer and providing information that cannot be derived manually. In particular, the identification of performance problems, their classification by kind and severity, and their localization in the source code should be addressed.

1.6 Contribution of this Thesis

The kind of performance data available has a great influence on the expressiveness of the performance problems that can be detected. Summary information, as collected by profiling tools, is sufficient to detect a multitude of frequently occurring performance problems. However, there are performance problems that are not visible in this kind of information. In contrast, event traces allow the reconstruction of the dynamic behavior in terms of single events and provide a more detailed view.

This thesis presents a novel approach to analyzing the performance of parallel applications based on event traces. Its strength lies in its ability to allow a deeper but more intuitive insight into performance behavior than is provided by traditional tools. This is achieved through an automatic transformation of fine-grained but low-level performance data, whose analysis is time-consuming and may require a high learning effort when based on such tools, into a more abstract and more expressive view accessible through a simple but flexible user interface.

The thesis describes the automatic transformation of event traces into a three-dimensional property-oriented performance space (Figure 1.1). The approach covers event traces that are generated from MPI, OpenMP, or hybrid applications. Hence, it is especially well

suited for parallel computers with SMP nodes. The performance space presents the performance behavior along three dimensions: performance property, node within the dynamic call graph, and location on the machine, such as SMP node or process.

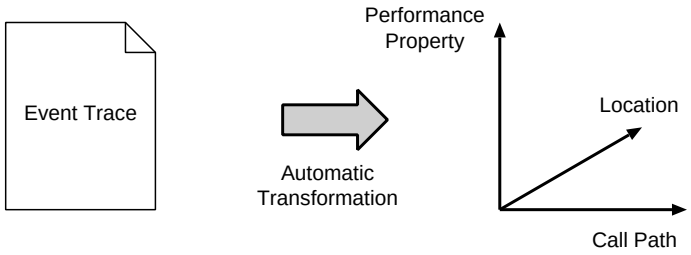

Figure 1.1: Transformation of event traces into a property-oriented performance space.

The performance-property dimension describes the kind of performance behavior. The call-graph dimension describes both the source-code location and the execution phase during which a performance behavior (i.e., property) occurs. Finally, the location dimension gives information on the distribution of performance across different processes or threads. Each dimension is arranged in a hierarchy, which allows the representation of performance behavior on different levels of granularity and pays attention to the hierarchical hardware and software structure of coupled SMPs. Each point in the representation is mapped onto the corresponding fraction of program execution time, allowing the convenient correlation of different behavior along multiple dimensions using only a single integrated view.

The performance properties to be analyzed mostly refer to common situations resulting from a suboptimal usage of the underlying programming model, such as a process waiting for a message from another process. Specification of performance properties is done in terms of *compound events* composed of simple events as recorded in the trace file. A layer of abstraction based on the grouping of related events makes the specifications simple and easy to extend. The resulting specifications serve as input for an automatic analysis process that is responsible for detecting the corresponding compound events in event traces.

Characterization of performance behavior is based on complex event patterns in conjunction with their location in a multi-dimensional structure. This provides both a technique of abstracting from low-level events to meaningful performance situations and a precise way of associating such situations with a place in the source code, an execution phase, and a control-flow point. Since the approach mainly refers to the programming model instead of specific hardware elements, it also provides a high degree of portability.

To accomplish this, the thesis defines a framework for formally specifying compound events that characterize performance behavior on a very high level of abstraction. By looking for such compound events in an event trace, it is possible to prove that particular performance problems are present in an application.

The framework identifies two categories of abstractions whose instances provide a basis for easily specifying compound events. The abstractions represent entities of the different programming models, such as MPI collective operations or OpenMP parallel constructs, and are useful for measuring their influence on performance behavior. The resulting specifications can be easily transformed into an appropriate detection algorithm. Examples are shown of how the approach applies to MPI, OpenMP, and the hybrid combination of both - the most relevant programming models for coupled SMPs.

The implementation of the automatic performance tool EXPERT for MPI, OpenMP, and hybrid applications proves the feasibility of this approach. The comprehensive behavioral classification incorporated in EXPERT explains a multitude of problems in terms of previously specified compound events. Extensibility mechanisms open the classification up to adding new behavior classes, if the predefined ones are not sufficient. EXPERT also offers a display technique based on multiple tree browsers, allowing the user to conveniently navigate through the performance space. Colors assist in identifying performance problems and bottlenecks, and help in investigating them on the most appropriate level of detail. The trees are interconnected so that the user can view one dimension with respect to a selection in another dimension.

Most of the ideas contributed by this thesis apply to coupled SMPs in general. Unfortunately, event tracing is rarely applicable to clustered SMPs because it requires a level of clock synchronization that cannot usually be provided by this class of computing environments. For this reason, parallel computers with SMP nodes are the primary target of the approach taken in this thesis.

1.7 Document Organization

The thesis is structured in two parts. The first part is more theoretical and concentrates on the notion of compound events as a means to describe situations of inefficient behavior. The second part is more practical and deals with the design of a real tool based on the compound-event method.

Chapter 2 provides an overview of the performance analysis of parallel applications. After discussing the drawbacks of traditional methods, an introduction to the problem of automating this task is given. Chapter 3 describes the method of using compound events to automatically detect inefficient behavior in event traces and how the method is applied to MPI, OpenMP, and hybrid applications. The design of an automatic performance-tool prototype based on the compound-event method is presented in Chapter 4. Here, issues such as event-trace generation, abstraction mechanisms, visualization of performance behavior, and extension mechanisms are discussed. Particular emphasis is put on the representation of performance behavior in a multidimensional data structure. To demonstrate that the performance problems addressed here are of practical relevance and that they can be easily

located using the present approach, the prototype is applied to four real-world test cases in Chapter 5. To draw a larger picture of research in the field and to distinguish the approach presented here from others, Chapter 6 contains a survey of related work. Finally, Chapter 7 summarizes the thesis research and comments on future work in automatic performance analysis.

Chapter 2

Automatic Performance Analysis

This chapter gives an introduction to the performance analysis of parallel applications and to the problem of automating this task. Reasons for the existence of complex performance behavior in parallel systems are reviewed. The performance indices and bounds used to quantify performance behavior are defined. A general model of the performance analysis process is then presented along with a survey of different kinds of performance data used in this process. Finally, the concept of a property-oriented performance space is introduced as the foundation of an automated analysis process that can overcome some of the current limitations in performance-analysis methods.

2.1 Complexity in Parallel Systems

The complexity in current parallel systems is a result of the interfaces and interactions between different functional layers:

- Application
- Parallel programming interface
- Operating system
- Hardware

The hardware of today's modern parallel architectures combines sophisticated processor architectures together with multi-layered memory hierarchies and advanced network technologies. The operating system makes the hardware resources accessible to applications through mechanisms, such as process management, memory management, and IO. The parallel programming interface defines the way parallelism is presented to the programmer

and how parallelism is enabled in the system. It comprises compilers, run-time systems, and parallel libraries including those that encapsulate complicated communication mechanisms among the different processors. Finally, the application itself maps structures of the application domain to constructs of the programming language and the parallel programming interface. For this reason, it may need intricate data distribution strategies and associated communication patterns. Often, the understanding of these mutual relations may be further complicated as a result of compiler optimizations that create a distorted picture of the application's source code.

In addition to the complexity within single layers of a parallel system, there is a complexity in the interactions among different layers. For example, an action in one layer may trigger an action in a lower layer or may be a reaction on behalf of an event occurring in a lower layer. For this reason, there are long and interrelated sequences of actions and their (side) effects in parallel systems.

The complexity of single layers as well as the causal connections between different layers of parallel systems are the reason for complex performance behavior and the limited ability of application developers to understand inefficiency in their programs.

2.2 Performance Indices and Bounds

Assessment of a system's performance requires an appropriate measure for drawing a comparison among different systems. Malony [51] identifies three quantitative performance indices for evaluating computer systems: *productivity* (i.e. throughput), *responsiveness* (i.e., turnaround or response time), and *utilization*. In the context of analyzing a parallel application's performance, responsiveness is the index of choice. Whenever an application's performance is classified as good, it has a satisfactory response time. For the non-interactive applications considered here response time is equivalent to execution time.

Speedup expresses the performance of a parallel application in terms of the time necessary for its sequential execution. The speedup for a given number of processors n is defined as the quotient of sequential and parallel execution time:

$$speedup(n) := \frac{T_{sequential}}{T_{parallel}(n)}$$

In general, the speedup can never grow more than linearly and exceed the *ideal speedup* of n unless there are side effects of parallel execution. For example, modern parallel architectures with cache-based memory hierarchies can achieve superlinear speedup as a result of memory allocation effects. The parallel *efficiency* provides a measure of the actual degree of speedup in relation to the ideal speedup:

$$efficiency(n) := \frac{speedup(n)}{n}$$

Amdahl [1, 41] formulates an upper limit of *speedup* based on the sequential part of a program, that is, the fraction of workload α that cannot be divided and distributed across multiple processors:

$$speedup(n) = \frac{n}{1 + (n-1)\alpha}$$

This is known as *Amdahl's law*. It implies that the best speedup that can be expected is upper bounded by α. Amdahl's law is a fundamental relationship in parallel-performance analysis because it points to the central issue of *scalability*, which characterizes the dependence of performance on the number of processors and the degree of problem parallelism. One metric that has been proposed to quantify scalability as the size of the problem changes is *scaled speedup* [34].

Paying attention to hardware utilization is sometimes more appropriate to highlight performance losses. Riley and Gurd [67] derive their notion of performance bounds from the hardware's peak performance as the upper limit. In their view performance of an application can be judged "in terms of the resource utilization it achieves ... while performing useful computation (that which is strictly necessary to solve the application problem at hand)." Note that the restriction to useful computation ties hardware utilization to the speedup criterion.

2.3 Performance Analysis Process

Once a parallel application is free of computational errors, its code usually needs to be optimized. This requires knowledge of which parts of the program are responsible for what kind of inefficient behavior. Performance analysis is the process of identifying those parts, exploring the reasons for their unsatisfactory performance, and quantifying their influence on the overall performance. The information gained through this process should suggest measures that could be taken to tune the application.

Performance analysis and tuning form a cycle that frequently has to be repeated many times until the performance reaches a satisfactory level. After that, the application is ready to run in production mode. Pancake [63] presents a conceptual framework that describes this cycle from the application developer's perspective in the form of five questions that must be answered to accomplish performance improvement:

1. *Identification*: Is there a performance problem? What are the symptoms?
2. *Localization*: At what point in execution is performance going wrong? What is causing the problem to occur?
3. *Repair*: What about the application must be changed to fix the problem? [Perform the repair.]

4. *Verification*: Did the "fix" improve the performance? [If not, optionally undo the repair, then go back to (2).]

5. *Validation*: Is there still a performance problem? [If so, return to (1).]

Note that the question of when the tuning cycle should end is nontrivial. How is the application developer able to determine whether the performance is satisfactory? Should the cycle continue until the performance comes close to theoretical bounds, such as ideal speedup or optimal hardware utilization? In practice, the cycle ends when the application developer either runs out of time or out of ideas. Sometimes, tradeoffs between different execution parameters impose further constraints on the decision about satisfactory performance.

Malony [51] emphasizes the importance of the scientific method of "systematic testing of hypotheses through controlled measurement of observable phenomena, analysis of collected data, and modeling of empirical results" to the process of performance analysis and describes it in the context of experimental computer science. He delineates an idealized model of a parallel-performance–evaluation environment (Figure 2.1), which highlights the process of successive refinement of a hypothesis about performance behavior based on observation and accumulation of knowledge. This model is used here as a foundation for the following discussion of the performance analysis process, which corresponds to question 1 and 2 above.

Performance analysis starts with an initial *performance hypothesis* (e.g., too much time used for communication) based on system and program characteristics, which may include the results of any kind of static analysis. The hypothesis may be further supported by performance prediction based on runs under a different configuration, simulation, or a performance model of the application. Performance models play an important role especially in scalability analysis.

In response to the hypothesis, the experimental performance observation (e.g., monitoring communication) follows. The observation is constrained by certain observational capabilities and is usually performed with the support of programming tools. Since the run-time behavior of an application may be influenced by several different parameters, such as the number of allocated CPUs or the selected input-data set, the hypothesis also may refer to the performance as a function of one or more of these parameters. In this case, the observation may include a whole series of experiments. In addition, it may be feedback driven, that is, the performance data are analyzed online and influence the way the experiment is conducted.

The resulting *empirical performance data* are now subject to postprocessing, which may include matching them with a performance model and making them accessible through presentation. In this way, the data can be used to refine (or disprove) the initial hypothesis or contribute to the *stored performance knowledge* to be used in later hypotheses. Of course, hypothesis refinement includes both a more specialized identification (question

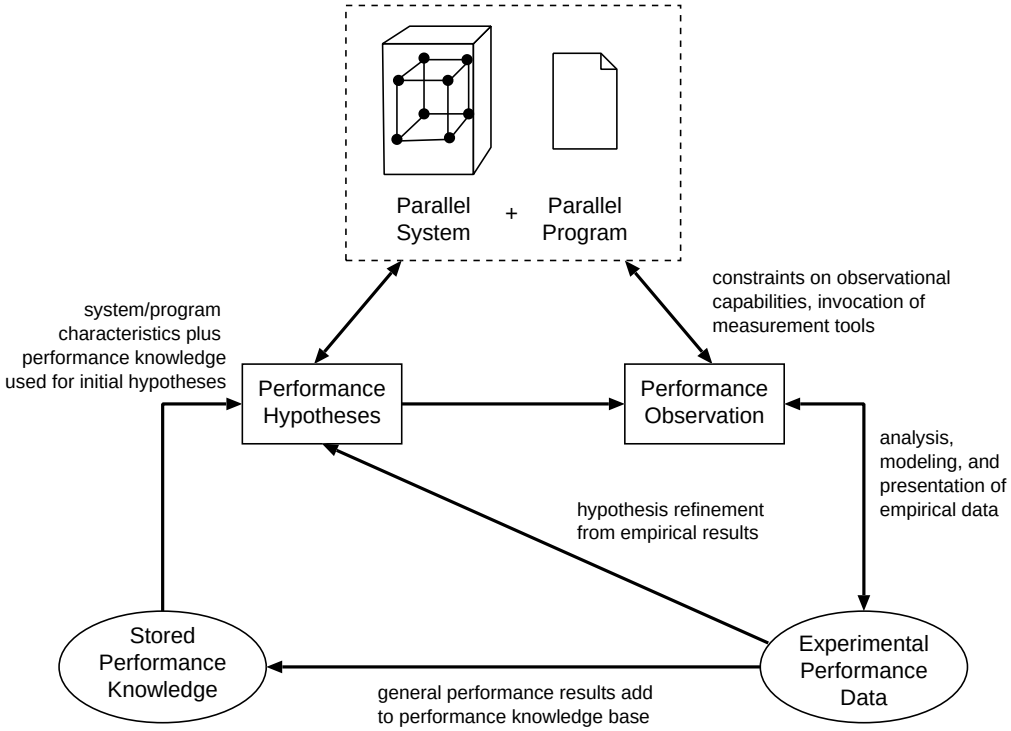

Figure 2.1: Idealized performance-analysis environment from [51].

1) and a more specialized localization (question 2). Depending on the results, the cycle of hypothesis creation and observation may begin again. The process ends when further hypothesis refinement becomes impossible due to a lack of new performance data.

2.4 Performance Data

Performance data associate program entities with performance-related behavioral characteristics. Program entities are either static or dynamic. For example, source-code regions are static entities, whereas instances of those regions or paths within the dynamic call graph are dynamic entities. The characteristics are either qualitative or quantitative. Qualitative characterization refers to the occurrence or the order of certain events, whereas quantitative characterization is usually achieved by relating the number of certain event occurrences to intervals of program execution that represent certain program entities.

Performance data may differ in the level of abstraction they provide both with respect to the behavioral characteristics and with respect to the program entities they refer to. Characterization may occur, for example, either in terms of simple events, such as clock

cycles, or in terms of more complex behavior, such as lock competition. Program entities may represent either simple pieces of source code or entities of the application domain. Observational performance data are usually generated on a low abstraction level and in a later step may be mapped to a higher abstraction level. Unmapped performance data are called *raw* performance data. The most common types of raw observational performance data are *profiles* and *event traces*.

Each type of performance data provides a certain view of the performance behavior. Usually the behavior is described along several dimensions, such as time and location. For this reason, the view defined by a certain type of performance data will be called a *performance space*.

2.4.1 Profiles

Profiles map accumulated performance metrics (e.g., number of clock cycles, number of function calls, or number of cache misses) onto program entities. For example, a profile may contain the fraction of execution time spent in different functions of the program. Typical methods for profile generation are *sampling* and *instrumentation*.

Sampling is a statistical approach of periodically observing the program execution under the control of an interval timer and deriving performance metrics for program parts based on these observations. For instance, the GNU profiler gprof [24] determines the time fraction spent in different functions of the program based on sampling. Besides plain execution times, gprof estimates the execution time of a function when called from a distinct caller only. However, since the estimation is based on the number of calls from this caller, it can introduce significant inaccuracies in cases where the execution time is highly dependent on the call site.

In contrast to sampling, instrumentation inserts code directly into the program so that the program itself is able to trigger actions upon occurrences of certain program-level events (e.g., function calls). For instance, the TAU performance-measurement framework [69, 70] provides the ability to create execution-time and hardware-counter profiles based on routine-, basic-block-, and statement-level instrumentation.

Profiles are useful to generate a rough overview of an application's performance characteristics while introducing only limited perturbation of run-time behavior and requiring only moderate storage.

2.4.2 Event Traces

Event traces are collections of individual run-time events recorded during program execution. The information recorded for an event includes at least a time stamp, the location

(e.g., the process or node) where the event happened, and the event type. Depending on the type, additional information may be supplied, such as the function name for function-call events. Message-event records typically contain details about the current message (e.g., the source or destination location and the message tag). In order to keep instrumentation simple, the information included in such an event record is usually restricted to the data available at the location where and at the moment when the event occurs.

Events are recorded at the point of their occurrence. For this reason, an application needs instrumentation to intercept and store away the desired events; that is, additional code needs to be inserted at program locations where their occurrence can be detected. To keep intrusion low, the event records are initially written into a memory buffer. Upon buffer overflow or program termination, the events are written to a file. Event traces generated independently for each location must be merged and sorted according to their time stamps. Systems that rely only on local clocks have to adjust the time stamps with respect to chronological displacements and clock drifts.

Limitations of event tracing may result from both the huge amount of data being produced and the perturbation of the program execution. This is true in particular when the density of recorded events is high. Because it is difficult to predict when this will occur, instrumentation has to be carried out very carefully and should be selective; that is, it should record only a small subset of all possible events.

The advantages of event traces result from the spatial and temporal relationships among individual events. This allows the reconstruction of an application's run-time behavior and thus can provide more detailed evidence of performance problems. In particular, the ability to visualize program execution using event-trace browsers have made tracing a widely accepted technique especially for message-passing programs.

For instance, VAMPIR [3] (Figure 2.2) provides a flexible display for event traces of message-passing programs. The VAMPIR event model defines event types for entering and leaving a region, for sending and receiving a message, and for executing a collective communication operation. VGV [38], the next-generation of VAMPIR, is based on an extended VAMPIR event model that supports hybrid applications as well.

2.5 Instrumentation

Instrumentation is the process of inserting extra code into a program to observe its execution or performance. Often instrumentation is used to make measurements for these purposes. Shende [69] distinguishes three dimensions of classifying instrumentation and measurement:

1. *How* are performance measurements defined and instrumentation alternatives chosen?

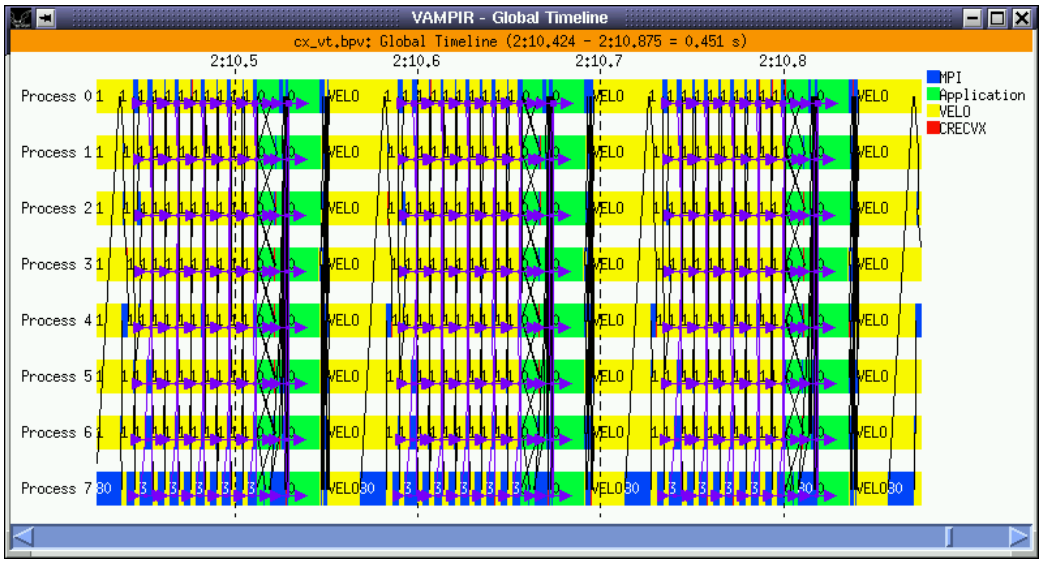

Figure 2.2: Event-trace visualization using VAMPIR.

2. *When* is performance instrumentation added and/or enabled (precompile time, compile time, link time, run time)?

3. *Where* in the program performance measurements are made (granularity and location)?

The first question addresses the selection of phenomena to be observed. It includes, for example, the choice among different metrics (e.g., time or cache misses).

The second question deals with the maintenance of the user's level of abstraction. Running a program requires moving it through several transformation steps: preprocessing, compilation, linkage, and execution or interpretation. Each transformation corresponds to a different level of representing a program's contents: source code, object code or library, executable or byte code, and run-time image. Although each level offers the opportunity to add instrumentation to the program, each level provides different information to be measured. In particular, the user's abstractions may be represented differently on each level. For example, the source code allows access to language-specific abstractions, which may be hidden in the binary representation. However, binary instrumentation of the run-time image allows instrumentation to be carried out at run time (sometimes referred to as *dynamic instrumentation*) and thus to be controlled by feed-back, which provides an excellent way of reducing intrusion. Note that both approaches may impose restrictions on the portability either across different languages or across different machines.

Programs exhibit a hierarchical structure consisting of different, often nested, elements, such as modules, functions, and statements. The third question classifies instrumentation according to the level within the program at which the instrumentation takes place, such as function entry and exit, statement, or instruction. The decision on the best places for

adding instrumentation is governed by the tradeoff between the demand for expressive performance data and the desire to avoid program perturbation.

As an example, the OPARI [57, 58] source-to-source translator instruments OpenMP constructs on the source-code level to capture performance-relevant events, such as entering a parallel region. Since OpenMP defines only the semantics of directives, not their implementation, there is no equally portable way of capturing those events on a different level. However, because OPARI supports all languages for which OpenMP is defined, it is still independent of a specific programming language. As a performance interface, OPARI defines only the types of events to be observed, the selection of information to be measured upon their occurrence is left to the user.

In contrast, Dyninst [11] is a C++ class library for instrumenting the run-time image of multiple processes running on the same machine. It allows the insertion of code snippets, including calls to dynamically loaded modules, at function entries and exits as well as before and after function calls. Because Dyninst requires neither recompiling nor restarting the application, it is well suited for feedback-driven online instrumentation. DPCL [16] is a dynamic instrumentation system based on Dyninst that is integrated with a parallel environment to provide simplified instrumentation of parallel applications.

The TAU [69, 70] performance-measurement framework overcomes the restrictions imposed by single-level instrumentation by allowing instrumentation at multiple levels. An instrumentation API allows the manual insertion of instrumented annotations in the source code. TAU also provides automatic preprocessor-level instrumentation by replacing calls to library routines with instrumented ones. In addition, TAU is able to automatically instrument the source code of C, C++, and Fortran programs using a preprocessor based on the PDT [50] toolkit. Besides compiler-level instrumentation based on a specific optimizing compiler, TAU supports the interception of MPI-specific events, such as message dispatch and receipt, using an interposition wrapper, which is linked between the application and the original MPI library. Finally, instrumentation using Dyninst allows the insertion of extra code at run time.

2.6 Performance Properties

Parallel applications may exhibit a large variety of different performance behaviors. For this reason, a general approach to performance analysis requires a terminology that can be used to refer to performance behavior independent of its specific characteristics.

Fahringer et al. [21] propose the notion of *performance properties* (e.g., load imbalance, communication, cache misses, redundant computations, etc.), which characterize a specific performance behavior of a program and can be checked by a set of conditions. Conditions

are associated with a *confidence* value (between 0 and 1) indicating the reliability in proving the existence of a performance property. In addition, for every performance property a *severity* measure is provided, whose magnitude specifies the importance of the property in relation to other properties. Note that a performance property does not necessarily denote negative, that is, inefficient behavior.

Fahringer et al. further define a *performance problem* as a performance property whose severity exceeds a user- or tool-defined threshold. The unique *performance bottleneck* is defined as the most severe performance property. If the bottleneck is not a performance problem, then the program's performance is considered to be acceptable and does not require any further tuning.

On the one hand, the concept of severity helps to distinguish between important and negligible performance problems during the performance tuning process. The purpose of the severity is to map arbitrarily complex behavior onto a general but simple metric, which provides the ability to draw comparisons with respect to the presence of very different performance properties in an application. For this reason, the notion of performance properties is a useful key concept for performance-analysis frameworks.

On the other hand, severity offers only a simplified view of the performance behavior. The severity arranges all performance properties in a linear order with the most severe (i.e., the bottleneck) on top. However, it does not take into account the various relationships, such as specialization and generalization, that may exist among different properties. If performance analysis were to pursue the goal of identifying the most worthwhile candidate property for optimization, it might be insufficient to sort performance properties only by one criterion while ignoring inter-property relationships.

For example, suppose a program has two and only two similar properties (e.g., overhead and synchronization overhead), of which one is more general (i.e., overhead) than the other one (i.e., synchronization overhead). Suppose also the general property's severity is higher, that is, it is the bottleneck. Note that the latter assumption is natural because the more general property includes the other one's behavior as a subset. Although the more general property has a higher severity because the total overhead is bigger than the overhead caused only by synchronization, in view of the inclusion relationship, the more specific property might be more interesting because it reveals more about its causes. Therefore, an application developer might pay more attention to this less general property, in particular, if synchronization overhead represents a major fraction of the total overhead.

Another criticism targets the definition of performance problems in terms of a threshold because the application or tool developer does not necessarily have an idea of a precise and useful threshold. Sometimes the developer just wants to spend a certain amount of time on optimization and tries to make the best achievements possible in that time. This might be another reason to look for more specific performance properties because their causes are more obvious compared to more general ones.

Finally, as already anticipated by Fahringer et al., the performance behavior is actually multi-dimensional. Fahringer and et al. express this in their parameterization of performance properties, which allows the consideration of a property with respect to only a specific source-code region or function call. Regarding these parameters as further dimensions leads to a very general representation of performance behavior.

A *property-oriented performance space* is defined as a multi-dimensional space with the performance property as its first dimension. The other dimensions represent static or dynamic entities related to an aspect of program execution a performance property may refer to. The definition of the remaining dimensions is very general and may include parts of the source code, dynamic run-time objects, or intervals of the execution time. The performance behavior in such a space is represented by data indicating the extent (i.e., severity) to which a certain performance property is present with respect to entities of the other dimensions. For example, a program may spend five percent of the overall execution time on a property *synchronization overhead* in function *foo* on process *zero*. Here, property *synchronization overhead*, function *foo*, and process *zero* are coordinates of a point in a property-oriented performance space, and the severity of that point is given as five percent. Similar to mapping single points onto a severity, it is possible to map sets of points onto a severity. For example, instead of considering the synchronization time for function *foo*, it should be possible to consider the synchronization time for the whole program. Thus, the severity is a mapping that maps a subset of the performance space onto a numeric value that makes it comparable to other subsets. The advantage of a property-oriented performance space is that it provides the ability to represent performance behavior along multiple dimensions in a data structure that is independent of the semantics of specific performance properties. In addition, the mapping of whole subsets instead of single points onto a severity value allows performance analysis on varying levels of detail.

In this manner, *performance problems* and *performance bottlenecks* can be considered as subsets of the performance space that are mapped onto a high severity and a very high severity, respectively. Of course, they are typically associated with a negative performance property, that is, one that denotes inefficient behavior.

Note that this characterization of both terms clearly refers not only to a class of behavior but also to the program entities that behavior is associated with. In the example above, the synchronization time in function *foo* and process *zero* might be considered as a performance problem. In addition, this characterization is very flexible because it allows inter-property relationships to be taken into account and a problem to be analyzed in the context of a more general problem.

2.7 Automatic Performance Analysis

Automating the process of performance analysis requires a model of the expected results of that process. In general, automation of performance analysis may cover all activities involved in that process. Riley and Gurd [67] roughly divide these into two categories:

- Gathering of data
- Search process

They describe the search process based on the notion of performance properties as a "systematic examination of performance data gathered for an application in order to identify performance properties in relation to regions of the application source code." The search process requires the performance properties to be defined in terms of conditions referring to performance data and includes query formulation and execution. The gathering of the necessary data requires experiment planning and execution management of instrumented runs of the program.

Justification for the above distinction can be found in the difference between raw (i.e., low-level) performance data and high-level performance data that present the performance behavior on a higher level of abstraction. The nature of raw performance data is determined by the nature of common monitoring techniques, which usually gather data in the form of profiles or event traces. Traditional performance tools support the search process mainly by providing low-level views of these performance data types. These views typically include textual or graphical - often interactive - displays, such as tables or bar charts of profiling information, time-line diagrams of event traces, and statistical analyses. The following tools exemplify common techniques of presenting profiles and event traces to the user.

The Apprentice [14] performance tool visualizes execution-time profiles of message-passing programs on the CRAY T3E in the form of bar-chart views (Figure 2.3). Apprentice shows time profiles on the program, routine, and basic-block level. Each bar is divided into sections by the use of a different color indicating a different type of activity, such as parallel processing, communication overhead, or IO. Starting from an arbitrary activity bar, the user can navigate through the call graph in both directions of a calling relationship to obtain profile information on subroutines as well as call sites.

Xprofiler [42] is graphical front end for the GNU gprof profiler [24] with the ability to present gprof output as a call-graph diagram (Figure 2.4). Each node is displayed as rectangle, whose width and height represent the execution time including and excluding called routines, respectively. The arcs are labeled with the number of times a node was visited. In addition to the call-graph view, an annotated source-code view displays profiles for individual source lines.

VAMPIR [3] visualizes event traces of message-passing programs by showing a time line for each process (Figure 2.2) indicating its current execution state by color. Arrows point-

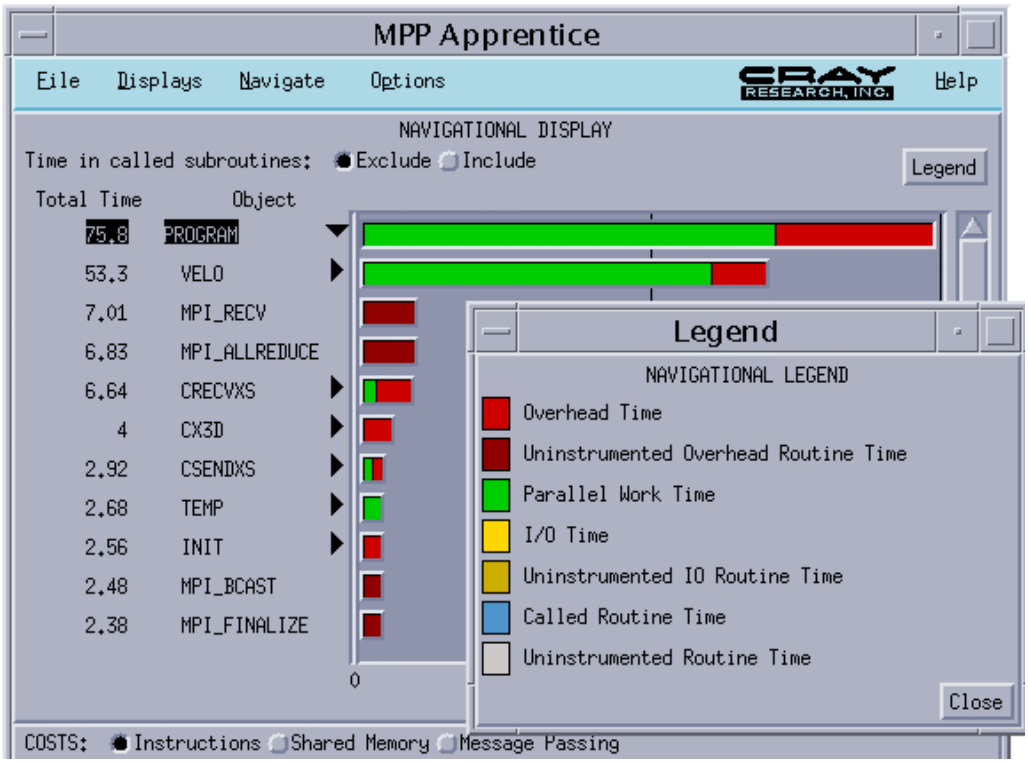

Figure 2.3: Apprentice profile browser.

ing from one time line to another time line represent point-to-point messages sent between processes, whereas connected lines covering multiple processes indicate collective communication. VAMPIR's zooming capability allows the user to examine the run-time behavior on an arbitrary level of temporal granularity. In addition to clicking on single items in the representation to obtain more detailed information, the user can look at statistics for the interval displayed.

The drawbacks of these low-level views are manifold. First, the user is confronted with a potentially large amount of data, which has to be searched manually for the presence of performance properties. This often includes manual comparison of different aspects of program behavior displayed in different unrelated views. Second, the views provided by current tools usually present program behavior in terms of low-level metrics that do not help the user in deciding whether performance improvement is possible, how performance can be improved, and whether an optimization effort would be worth the investment. Third, as a result of the multitude of different view options offered by some tools, a lot of training may be necessary before a tool can provide valuable assistance in performance analysis.

The search process as specified by Riley and Gurd describes a transformation of raw performance data into a two-dimensional performance space of performance property by source-code region. The difference between this performance space and low-level views is the characterization of performance behavior in terms of abstract performance properties that

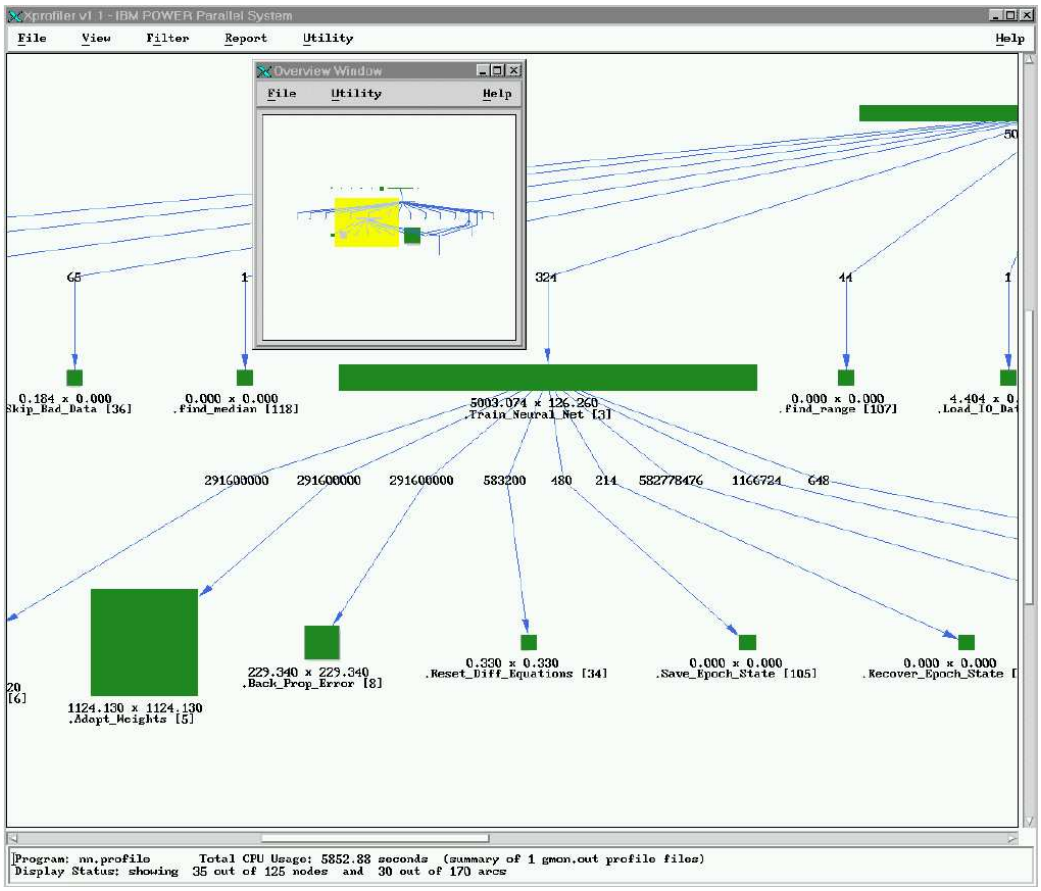

Figure 2.4: Xprofiler call-graph diagram.

explain misbehavior on a higher level of abstraction.

However, there is no logical reason to restrict the performance space to only two dimensions. For example, the performance of a function might be poor only when called from a distinct caller or at a distinct location (e.g., process or node); or a performance property may evolve over time as the application moves among different execution phases. Therefore, it might be reasonable to consider the dynamic call graph or the execution phase as additional dimensions. In general, the search process can be regarded as the transformation of raw performance data into a general multi-dimensional high-level performance space that may be made up of various dimensions depending on the purpose of the analysis. Note that if raw performance data is thought of as being represented in a low-level performance space, the search process can be regarded as a transformation from a low-level space into a high-level space.

This thesis regards the automatic search process as an automatic transformation of low-level performance data into a multi-dimensional property-oriented performance space. The benefit of this viewpoint is a more general model of performance behavior that is able to

take into account the state of the program at the time a specific performance property is present. This may provide a better understanding of the preconditions that lead to a certain kind of behavior.

Chapter 3

Specification of Performance Behavior

This chapter presents a novel approach to analyzing performance properties of parallel applications based on event traces. It defines a framework for formally specifying compound events that characterize performance-relevant behavior. The framework allows the creation of abstract building blocks that represent concepts of the underlying programming model and therefore provide an easy means to specify complex compound events representing inefficient behavior. Using these specifications, it is possible to automatically locate inefficiencies in parallel applications by looking for occurrences of the corresponding compound events in event traces. This will be demonstrated for MPI, OpenMP, and their combination. Finally, to show that the approach is also suitable for existing performance-analysis frameworks, extensions are proposed to integrate it into ASL (APART Specification Language), a language for the formal specification of general performance properties.

3.1 Rationale

Effective automatic performance analysis requires formal methods for specifying performance properties that characterize a specific performance behavior. The strength of specifying inefficient behavior in terms of compound events stems from its ability to describe the behavior on a high level of abstraction directly related to the programming model. The resulting specifications can then serve as a basis for performance tools that are able to prove the presence of complex performance properties in a parallel application without user intervention.

The kind of performance data available has a great influence on the expressiveness of the performance properties that can be defined. Summary information, as collected by profiling tools, is sufficient to describe a multitude of frequently occurring performance properties. However, there are performance properties that are not visible in this kind

of information. A more detailed view of a parallel application's behavior can be gained by using event traces because event traces preserve the spatial and temporal relationships among individual events, allowing the reconstruction of an application's dynamic behavior. By looking for compound events in an event trace, it is possible to prove that particular performance properties are present in an application.

A *compound event* representing a performance property is a set of primitive events, which are called its *constituents*. Compound events that relate to the programming model expose complex relationships among their constituents reflecting their model-relevant actions. For instance, sending a message and receiving it are interconnected by a relationship derived from the message-passing programming model. Because programming models differ in their operational semantics, it is difficult to devise a general formal representation of compound events that is powerful enough to express the complex compound events across all programming models.

To overcome this problem, the thesis identifies two categories of abstractions that can be used to provide programming-model–specific building blocks, on top of which a general specification of compound events is possible. The abstractions represent entities of the different programming models, such as MPI collective operations or OpenMP parallel-region constructs, and are useful for measuring their influence on performance behavior. The resulting specifications can be easily transformed into an appropriate detection algorithm.

3.2 System Observation Based on Events

Because a computer changes its state in discrete intervals (i.e., clock cycles), it is possible to model the dynamic behavior of any program execution as a sequence of atomic actions. The finest temporal granularity of actions happening in a computer system is a clock cycle. However, in practice, measuring the time of each action by software requires several clock cycles. Thus, the temporal resolution of atomic actions that can be observed is much lower.

An event characterizes an atomic action happening at a distinct location and at a distinct point in time. It is the smallest entity that can constitute the dynamic view of a parallel application. However, performance analysis is frequently interested in non-atomic activities (e.g., sending a message), which require a set of events to characterize them. Often a non-atomic activity is described in terms of its start or end events, which can usually be associated with a distinct point in time. The location of an event is determined by the location of the control flow causing the associated change in the state. The location of an event may be logical or physical (e.g., a process or a CPU, respectively).

Event tracing regards the execution of a program as a sequence of events representing actions relevant to the purpose of the observation. Therefore, the selection of event types to be observed defines the view of program execution an event trace can provide. An *event*

model defines the formal properties of that view. It comprises a set of event types with an associated set of attributes and constraints defining correct event ordering.

An event type is defined by a set of attributes. In most cases, there are event types that share a subset of their attributes. For this reason, it is convenient to create a type hierarchy containing concrete event types derived from abstract event types that isolate commonalities. In the following, concrete event types are written in small Roman letters, whereas abstract event types are written in small italics.

Each event traced has a location loc (e.g., the MPI process) as well as a wall-clock time stamp $time$. It is useful to define an abstract event type *Event* constituting the root of the type hierarchy. All event types are derived from *Event*. The set of locations involved in an event trace is called L.

An event type t is defined by a set of attributes $\{a_1, \ldots, a_{n_t}\}$. A subset of these attributes may be associated with more general base types. Subsequently, the notation $e.attr$ is used to refer to an attribute $attr$ of an event e. Table 3.1 summarizes all event attributes used in this document, including those that will be introduced in later sections when dealing with parallel programming models.

Table 3.1: Summary of event attributes.

Attribute	Description
$cedgeptr$	least recent Enter event visiting the preceding call path
$cnodeptr$	least recent Enter event visiting the same call path
$csite$	call site
$enterptr$	Enter event of the enclosing region instance
loc	location
reg	region
$time$	time stamp
MPI	
com	communicator
$dest$	destination location of a message
len	message length
$recvd$	bytes received during a collective operation
$root$	root location of a collective operation
$sendptr$	Send event to a given Receive event
$sent$	bytes sent during a collective operation
src	source location of a message
tag	message tag
OpenMP	
$lock$	a lock object used for synchronization
$lockptr$	*Sync* event that performed the last change of a lock's ownership status

Definition 3.1 (Event Trace). An *event trace* is a finite indexed set of events $E := \{e_1, \ldots, e_{n_e}\}$. The indexing reflects the time-sequenced order of event records in the trace file:

1. $i < j \quad \Rightarrow \quad e_i.time \leq e_j.time$
2. $i < j \quad \wedge \quad e_i.loc = e_j.loc \quad \Rightarrow \quad e_i.time < e_j.time$

Also, the indexing defines a linear order $<$:

$$\forall \quad 1 \leq i, j, \leq n_e : \quad i < j \quad \Leftrightarrow \quad e_i < e_j$$

$\diamond$

The conditions require the events to be in chronological order and allow only events at different locations to happen simultaneously. Depending on the programming model being used by an application, there may be additional constraints as well.

In the following, the abbreviation E_t denotes those events of an event trace E that are of type t:

$$E_t := \{e \in E | \quad type(e) = t\}$$

3.3 Event-Model Enhancement

To be able to express complex relationships among the constituents of a compound event, the event model of system observation can be extended by creating instances of two different categories of abstractions:

- State sequences
- Pointer attributes

The process of creating event abstractions from a given event model is called *event-model enhancement*. The resulting model is called the *enhanced event model*. To distinguish the original model from the enhanced model, the original model is called the *basic event model*. State sequences and pointer attributes are formally defined in the following subsections. Concrete examples can be found in Sections 3.4, 3.5, and 3.6.

The concepts of state sequences and pointer attributes were previously used by the author to design the initial version of the EARL trace-analysis language [75], which targeted the analysis of point-to-point communication in message-passing programs. This chapter provides both a generalization and a refinement of these concepts and a much broader event-analysis coverage, including MPI collective communication, OpenMP, hybrid programming, and call-path analysis.

3.3.1 State Sequences

Compound events representing performance properties often exhibit some form of locality within the event trace. That is, the constituents of such a compound event are not arbitrary subsets of the trace, but share some context. This context is represented by the state of the parallel system at the time when the compound event occurs. In most cases, this state refers to a set of ongoing activities in contrast to activities that are already finished.

An event happening in a parallel system indicates a change in its state, thus events can be regarded as state transitions. An event trace can be seen as a sequence of state transitions starting at an initial state and changing into the next state, event by event, until a final state is reached after the last event. The state entered as the result of an event is a useful abstraction when specifying compound events that represent inefficient behavior.

The overall state of a parallel system is characterized by different aspects. For example, one aspect might be the set of messages being transferred at a given moment, another aspect might be the dynamic call stack of a process or thread. Such a state aspect can be conveniently characterized in terms of the events that caused that aspect's state. Thus, it becomes possible to describe state information using only events and sets of events. For example, the set of messages being transferred at a given moment can be represented by the set of send events of these messages, and the dynamic call stack can be represented by the set of function-call events.

Model enhancement defines for each of theses aspects a *state sequence* that describes the evolution of that aspect over time. Corresponding elements of all state sequences (i.e., those that correspond to the same event) form a vector, which is called the *overall state*. The evolution of the overall state over time is described by a vector of state sequences, which is called the *overall state sequence*. A state sequence is inductively defined by a transition operator. The transition operator is applied to the current overall state and the next event to compute the next state in the sequence and, thus, a part of the next overall state. Note that computing the next state from the preceding overall state allows the definition of relationships across different state sequences.

Definition 3.2 (State Sequence). A *state sequence* $\mathfrak{S}$ of an event trace $E = \{e_1, \ldots, e_{n_e}\}$ is a finite indexed set of subsets of E:

$$\mathfrak{S} = \{\mathfrak{S}_0, \ldots, \mathfrak{S}_{n_e}\}$$

$\mathfrak{S}_0$ is called the *initial state* of $\mathfrak{S}$ and is always the empty set. $\mathfrak{S}_{i>0}$ is called a *state* of the event e_i and does not contain any events happening later than e_i:

$$\begin{aligned} \mathfrak{S}_0 &:= \emptyset \\ \mathfrak{S}_i &\subseteq \{e \in E | \quad e \leq e_i\}, \qquad 1 \leq i \leq n_e \end{aligned}$$

The vector of all *state sequences* defined for an enhanced event model is called the *overall state sequence*:

$$\vec{\mathfrak{S}} = (\mathfrak{S}^0, \ldots, \mathfrak{S}^{n_s})$$

The vector of all states with index i is called the *overall state* i:

$$\vec{\mathfrak{S}}_i = (\mathfrak{S}_i^0, \ldots, \mathfrak{S}_i^{n_s}), \qquad 0 \leq i \leq n_e$$

Thus, a state $\mathfrak{S}_i^j$ describes the aspect j after the occurrence of event e_i. The union of all states with index i contains all events that are part of the overall state. It is called the *flat overall state* Γ_i:

$$\Gamma_i := \bigcup_{k \in \{1,\ldots,n_s\}} \mathfrak{S}_i^k, \qquad 0 \leq i \leq n_e$$

A state sequence $\mathfrak{S}^j$ (i.e., an aspect of the overall state) is inductively defined by a transition operator $\mathfrak{s}^j$. A transition operator is applied to an event e_i and the previous overall state $\vec{\mathfrak{S}}_{i-1}$ to compute the state $\mathfrak{S}_i^j$.

$$\begin{aligned} \mathfrak{S}_0^j &:= \emptyset \\ \mathfrak{S}_i^j &:= \mathfrak{s}^j(e_i, \vec{\mathfrak{S}}_{i-1}), \qquad 1 \leq i \leq n_e \end{aligned} \tag{3.1}$$

A transition operator is defined by a set of transition functions $\mathfrak{s}_t^j$, one for each event type t. If e_i in (3.1) is an event of type t, then $\mathfrak{S}_i^j$ is computed using the transition function for type t:

$$\mathfrak{s}^j(e_i, \vec{\mathfrak{S}}_{i-1}) := \mathfrak{s}_t^j(e_i, \vec{\mathfrak{S}}_{i-1}), \quad \text{if} \quad type(e_i) = t, \qquad 1 \leq i \leq n_e \tag{3.2}$$

If there is no explicitly defined transition function for an event type t, $\mathfrak{s}_t^j$ is assumed to leave the state unchanged (i.e., to be the identity function). A transition function defined for an abstract base event type covers all derived event types. If a transition operator defines several transition functions defined along a path in the type hierarchy, they are all applied in the order defined by inheritance starting with the most general type. That is, the transition function effectively applied is a composition of all the transition functions defined along the path ranging from the root to the type of the current event. For example, consider two types b and d, where d is a descendant of b. If there are two transition functions $\mathfrak{s}_b$ and $\mathfrak{s}_d$ and e_i is an event of type d, then:

$$\begin{aligned} \vec{\mathfrak{S}}'_{i-1} &:= (\mathfrak{S}_{i-1}^0, \ldots, \mathfrak{s}_b^j(e_i, \vec{\mathfrak{S}}_{i-1}), \ldots, \mathfrak{S}_{i-1}^{n_s}) \\ \mathfrak{S}_i^j &:= \mathfrak{s}_d^j(e_i, \vec{\mathfrak{S}}'_{i-1}) \end{aligned}$$

A transition function $\mathfrak{s}_t^j$ may add or remove events from a state. An event added to $\mathfrak{S}_{i-1}^j$ must be an element of the (flat) overall state Γ_{i-1} or it must be e_i itself. It follows that the range of $\mathfrak{S}_i^j$ is limited in the following way:

$$\mathfrak{S}_i^j \subseteq \Gamma_{i-1} \cup \{e_i\}, \qquad 1 \leq i \leq n_e \tag{3.3}$$

$\diamond$

State sequences are abstractions used to provide context information for the constituents of a compound event. They separate activities that are still going on with respect to a certain point in time from activities that are already completed with respect to that point in time. As will be shown later, state sequences are especially useful to represent abstractions of the different programming models.

The flat overall state Γ_i of a given event e_i contains all the events that are related to activities that are still going on. As will be explained later, the flat overall state plays an important role when implementing an enhanced model.

3.3.2 Pointer Attributes

Another useful abstraction is a link connecting related events, so that one can navigate from one event to another related event. An example is a link from the event of receiving a message back to the corresponding event of sending it. This mechanism permits navigation along a path of related events and the definition of relationships among the constituents of a compound event using such paths. A natural way of representing such links is to provide event attributes with pointer semantics. Pointer attributes are the second category of abstractions considered here. They are added to the attributes $\{a_1, \ldots, a_{n_t}\}$ already defined in the basic model for an event type t.

Definition 3.3 (Pointer Attribute). A *pointer attribute* ptr is a mapping that maps an event of a particular type t from an event trace E onto a non-future event from E:

$$\begin{aligned} ptr : E_t &\rightarrow E \cup \{null\} \\ e_i &\mapsto e_i.ptr \end{aligned}$$

The pointer attribute ptr of an event e_i is defined as a function of the attributes $a_1, \ldots, a_{n_t}$ defined in the basic model for t and the preceding overall state $\vec{\mathfrak{S}}_{i-1}$. Its range is limited to e_i and the previous flat overall state Γ_{i-1}:

$$\begin{aligned} e_i.ptr &:= f_{ptr}(e_i.a_1, \ldots, e_i.a_{n_t}, \vec{\mathfrak{S}}_{i-1}) \\ e_i.ptr &\in \Gamma_{i-1} \cup \{e_i, null\} \end{aligned} \tag{3.4}$$

◇

To indicate the absence of a meaningful event, a pointer attribute may carry the $null$ value in certain situations. Pointer attributes depend on the attributes defined in the basic model and on the overall state immediately before the event under consideration. Also, similar to state sequences, the range of pointer attributes is limited to e_i and the previous flat overall state Γ_{i-1} if they are not $null$. From this it follows that pointer attributes never point to future events. Note that the function f_{ptr} may refer to pointer attributes of previous events including ptr itself.

Note that it is possible to use pointer attributes in state-sequence definitions in the same way that state sequences are used in pointer-attribute definitions, that is, using a pointer attribute $e_i.ptr$ to define a state $\mathfrak{S}_i$. Since $e_i.ptr$ may be defined only using the overall state $\vec{\mathfrak{S}}_{i-1}$ and, in addition, $\forall 0 \leq j \leq n_s : \mathfrak{S}_0^j = \emptyset$, it is ensured that $\mathfrak{S}_i$ is still well defined.

3.3.3 Implementation of an Enhanced Model

The calculation of both state sequences and pointer attributes occurs inductively. That is, starting from $\vec{\mathfrak{S}}_0 = (\emptyset, \ldots, \emptyset)$, an implementation of a model computes the pointer attributes of e_1. After that, it calculates $\vec{\mathfrak{S}}_1$, and, in a next step, the pointer attributes associated with e_2 and so forth.

To simplify an implementation, the model-enhancement framework requires that it is possible to compute both the overall state and pointer attributes of an event e_i without accessing any events other than e_i itself and those contained in the flat overall state preceding e_i, that is without accessing any events other than $\Gamma_{i-1} \cup \{e_i\}$.

Definition 3.4 (Working Set). The *working set* Δ_i of an event e_i is the union of the preceding flat overall state and the event itself:

$$\Delta_i := \Gamma_{i-1} \cup \{e_i\}$$

$\diamond$

The name "working set" is used to refer to locality. That is, in each step an implementation only needs to remember a small subset $\Delta_i \subseteq E$, since all functions involved in these calculations refer at most to the current event and the overall state immediately prior to that event. This is a very important property of the framework because it allows a tool to sequentially traverse the trace file from the beginning to the end, and to compute all the abstractions (i.e., state sequences and pointer attributes) solely based on a subset of events to which not more than one event can be added at every step. This helps to avoid expensive file accesses that would otherwise become necessary if the whole file was needed for every computation. Although this requirement limits the ability to define fully general abstractions, it is not a hard restriction because it excludes only events belonging to activities that have already been completed and therefore are usually not related to an event's context.

The working-set requirement has already been anticipated by the definition templates for state-transition functions (3.1 - 3.2) and pointer attributes (3.4) because both templates refer only to structures containing elements of the working set. Note that, when calculating abstractions for an event, pointer-attribute values can be compared and copied without accessing the events they point to because pointer attributes only need to carry references to these events instead of the events themselves. Using the event index as a reference even allows a comparison with respect to the relative position within the trace file.

The concept of model enhancement is not only a convenient method for defining compound events. Since the locality exhibited by compound events is exploited through restricting the potential event accesses to the working set when calculating abstractions, it also provides a foundation for efficient search methods to detect compound-event instances in an event trace.

Sections 3.4 and 3.5 exemplify the concept of model enhancement by applying it to MPI and OpenMP. In both cases, a basic event model is developed and then enhanced by adding instances of the two categories of abstractions. Finally, the two models are merged into one single model to describe hybrid applications in Section 3.6 .

3.4 Model Enhancement: MPI

The MPI message-passing communication library specifies communication operations to be explicitly invoked by an application to exchange messages among processes. The library provides operations for point-to-point and operations for collective communication involving more than two processes.

The current event model, which is covered in this section, does not yet address the advanced features which are included in the latest version of MPI, MPI 2 [53]. To give an overview, a brief discussion of these features follows in Section 3.4.3.

3.4.1 Basic Event Model

The execution of an MPI application involves a set of locations at which events may happen. To keep the model simple, the location of an event occurring during execution of an MPI application is defined as the triggering MPI process, which can be described using the rank in MPI_COMM_WORLD. Thus, the location is a number from $L := P = \{0, \ldots, n_p - 1\}$, where n_p is the total number of processes.

The static view of an MPI application comprises a set of regions. A region is a code section of a parallel program. It may be a function, a loop, or just a basic block. One execution of a region forms a region instance. It is assumed that a region instance may be exited only after all enclosed region instances have been exited, that is, the entries and exits of region instances occurring at the same location form a correct parenthesis expression.

The event types Enter and Exit indicate that a code region has been entered or exited, respectively. They both are derived from the same abstract base type *RegionEvent* that provides a region attribute reg which denotes the region entered or left. In addition, Enter events carry an attribute $csite$, which gives information on the source-code location (i.e., the call site)

from which the new region is entered. Note that transitions between regions, such as entering a loop, do not necessarily involve function calls. In this case, the call site is just the last location of the source region that has been executed before entering the destination region. Note that providing call-site information requires instrumentation of call sites.

Since sending and receiving point-to-point messages are activities that can be easily separated from their triggering functions, the event types Send and Receive represent the message dispatch and the message receipt, respectively. Send provides an attribute $dest$ for the message's destination location, and *Receive* provides an attribute src for the message's source location. Of course, sending and receiving a message are non-atomic activities that may take a while. For this reason, a Send event denotes only the start of sending a message, whereas a *Receive* event denotes the end of receiving a message. This ensures that a Send event never occurs after the corresponding Receive event. Note that the duration that can be derived from both events of a message is the most pessimistic estimation.

The message properties themselves are accommodated in an abstract base event type *MsgEvent*, which has attributes containing the message tag tag, the communicator com, and the message length len. Usually, events of these types are placed in between the Enter and Exit events of the corresponding MPI routine. As already mentioned, message events are constrained in their order within the event trace such that a Send never occurs after its matching Receive.

Modeling MPI collective operations is more involved because here two different aspects are linked very closely. First, a collective operation is executed in parallel on different locations. That is to say, a collective-operation instance is actually a set of single region instances. Second, a collective operation involves communication, but the detailed structure of this communication is usually hidden behind the MPI implementation.[1] This makes it difficult to explicitly model the communication events occurring. Therefore, a hybrid event type MPICExit denotes the exit of a collective operation. It is derived from Exit and also provides attributes characterizing the collective communication. These attributes include the number of bytes sent $sent$ from the event's location and the number of bytes and received $recvd$ by the event's location, the root location of the collective operation $root$, if there is any, and the communicator com. The communicator can be considered as a link connecting the single region instances that constitute a whole collective-operation instance. This is because it determines the set of locations involved in that operation instance. So for each participating location (i.e., each location in the communicator's group of processes), the call of an MPI collective operation results in an Enter event for calling it and in an MPICExit event for leaving it.

The complete type hierarchy is depicted in Figure 3.1 using UML [10] notation. For convenience, full attribute names are used in the figure. Note that in order to keep the hierarchy

[1] Pure barrier synchronization is considered as a special case of an MPI collective operation where the amount of data transferred is zero.

tree simple, the hybrid nature of MPICExit is not expressed explicitly by multiple inheritance relationships.

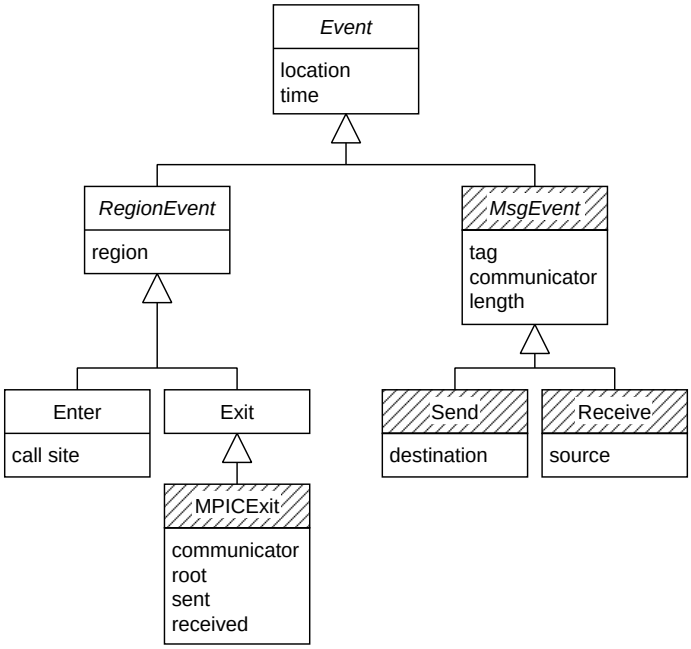

Figure 3.1: Basic event model for MPI applications. Hatched boxes represent MPI-specific event types.

The communicator connects the different events making up a whole collective-operation instance because it defines the group of locations executing that instance. C denotes the set of all communicators created during program execution. The group $Group(c)$ of a communicator $c \in C$ is a subset $Group(c) \subseteq L$ of the set of all locations so that $Group(\text{MPI\_COMM\_WORLD})$ is equal to L.

Note that this model can be implemented using very simple instrumentation technology, such as MPI interposition libraries.

3.4.2 Enhancement

This subsection describes simple abstractions that can be used to specify performance-relevant compound events occurring in MPI applications. The event type hierarchy motivates the description of the activities performed by an MPI application at a given moment in terms of three different higher-level concepts:

- Region instances

- Messages
- Collective-operation instances

First, it is shown that state sequences are a suitable means to describe the set of currently active region instances and message transfers. After that, it is demonstrated that this is also true for collective-operation instances. In addition, pointer attributes prove to be a handy instrument to link the single events constituting (collective) region instances and messages together. As a prerequisite, three auxiliary functions are defined.

Definition 3.5. Let $F \subseteq E$ be a subset of the event trace and $l \in L$ a location:

$$\begin{aligned} mostrcnt(F) &:= \{e \in F | \quad \neg\exists f \in F : f.time > e.time\} \\ leastrcnt(F) &:= \{e \in F | \quad \neg\exists f \in F : f.time < e.time\} \\ haveloc(F, l) &:= \{e \in F | \quad e.loc = l\} \end{aligned}$$

◇

The first two functions return those events from a set of events F that happened most or least recently. $haveloc()$ returns the events that have a specific location. For convenience, the set of events returned by these functions is allowed to be also treated as a single event if the returned set contains exactly one unambiguous element.

State Sequences

The region instances being executed at a certain moment can be easily represented by the set of Enter events that determine their beginnings.

Definition 3.6 (Region Stack). The *region stack* $\mathfrak{R}^l$ of a location $l \in L$ is a state sequence that collects the Enter events of active region instances at location l. Its transition operator $\mathfrak{r}^l$ is defined by the following transition functions:

$$\mathfrak{r}^l_{Enter} : \mathfrak{R}^l_i := \begin{cases} \mathfrak{R}^l_{i-1} \cup \{e_i\} & \text{if} \quad e_i.loc = l \\ \mathfrak{R}^l_{i-1} & \text{else} \end{cases}$$

$$\mathfrak{r}^l_{Exit} : \mathfrak{R}^l_i := \begin{cases} \mathfrak{R}^l_{i-1} \setminus mostrcnt(\mathfrak{R}^l_{i-1}) & \text{if} \quad e_i.loc = l \\ \mathfrak{R}^l_{i-1} & \text{else} \end{cases}$$

◇

The first function $\mathfrak{r}^l_{Enter}$ is responsible for adding Enter events representing active region instances to the region stack and the second function $\mathfrak{r}^l_{Exit}$ is responsible for removing them from the region stack as soon as the corresponding region instances are completed.

The messages currently being transferred are best characterized by the set of their respective Send events.

Definition 3.7 (Message Queue). The *message queue* $\mathfrak{M}^{s,d}$ of a pair of locations $s, d \in L$ is a state sequence that collects the Send events of messages underway from s to d. Its transition operator $\mathfrak{m}^{s,d}$ is defined by the following transition functions:

$$\mathfrak{m}^{s,d}_{Send} : \mathfrak{M}^{s,d}_i := \begin{cases} \mathfrak{M}^{s,d}_{i-1} \cup \{e_i\} & \text{if} \quad e_i.loc = s \quad \wedge \quad e_i.dest = d \\ \mathfrak{M}^{s,d}_{i-1} & \text{else} \end{cases}$$

$$\mathfrak{m}^{s,d}_{Receive} : \mathfrak{M}^{s,d}_i := \begin{cases} \mathfrak{M}^{s,d}_{i-1} \setminus \{e_j\} & \text{if} \quad e_i.src = s \quad \wedge \quad e_i.loc = d \\ \mathfrak{M}^{s,d}_{i-1} & \text{else} \end{cases}$$

$$\text{where} \quad e_j := leastrcnt(\{e \in \mathfrak{M}^{s,d}_{i-1} | \quad e.tag = e_i.tag \quad \wedge \quad e.com = e_i.com\})$$

◇

The first function $\mathfrak{m}^{s,d}_{Send}$ adds Send events of messages that have just been sent, and the second function $\mathfrak{m}^{s,d}_{Receive}$ removes Send events of messages that have just been received. This means that the set of messages currently being transferred is always up to date. Matching Send and Receive events is done using the standard MPI-messaging semantics (pp. 30-34 of [52]), which requires the Send event of a given Receive event to be the least recent event with matching tag and communicator in the message queue for traffic from the message's source location to its destination location (i.e., the location of the Receive event).

The events involved in collective operations form another class of related events that are important in the context of MPI performance properties. A complete collective-operation instance is depicted in Figure 3.2. The figure shows the time lines of all locations (i.e., processes) that are involved in this instance as well as the time lines of two locations that are not involved. The involved locations together form the group that is associated with the communicator of the operation call. This group is a subset of all possible locations L. Entering and leaving the corresponding MPI function are represented by Enter and MPICExit events. An explanation of the arrows pointing from the right to the left follows later.

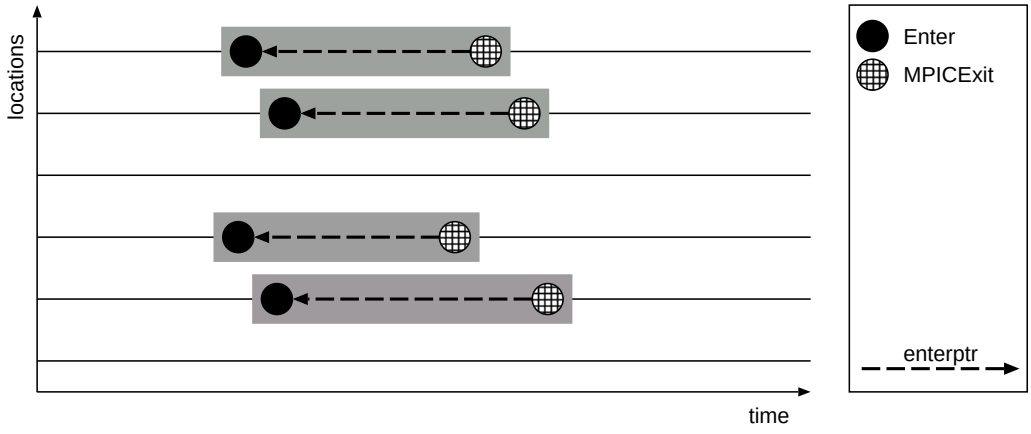

Figure 3.2: An MPI collective-operation instance.

Capturing the events belonging to a collective-operation instance can be done by defining an appropriate state sequence for each communicator. The basic idea is to accumulate the MPICExit events belonging to a collective-operation instance until all events belonging to that instance have been collected. After completion the corresponding events can be removed. This idea exploits the fact that the collective-operation instances executed within the same communicator are never interleaved, that is, collective operations must be executed in the same order by all members of the communicator's group.

Definition 3.8 (Collective-Operation Queue). The *collective-operation queue* $\mathfrak{C}^c$ of an MPI communicator $c \in C$ is a state sequence that collects the MPICExit events of active collective-operation instances of the communicator c. Its transition operator $\mathfrak{c}^c$ is defined by the following transition functions:

$$\mathfrak{c}^c_{MPICExit} : \mathfrak{C}^c_i := \begin{cases} \mathfrak{C}^c_{i-1} \cup \{e_i\} & \text{if} \quad e_i.com = c \\ \mathfrak{C}^c_{i-1} & \text{else} \end{cases}$$

$$\mathfrak{c}^c_{Event} : \mathfrak{C}^c_i := \begin{cases} \mathfrak{C}^c_{i-1} \setminus Inst & \text{if} \quad \forall l \in Group(c) : |haveloc(\mathfrak{C}^c_{i-1}, l)| \geq 1 \\ \mathfrak{C}^c_{i-1} & \text{else} \end{cases}$$

$$\text{where} \quad Inst := \bigcup_{l \in Group(c)} leastrcnt(haveloc(\mathfrak{C}^c_{i-1}, l))$$

◇

An MPICExit event is added to $\mathfrak{C}^c_{i-1}$ by applying the function $\mathfrak{c}^c_{MPICExit}$ if c is the communicator of the MPICExit event ($e_i.com = c$). After all MPICExit events belonging to a collective-operation instance have become elements of $\mathfrak{C}^c$, they are removed by applying $\mathfrak{c}^c_{Event}$.

Note that Event is more general than MPICExit, which has an important effect on the order in which the two functions are applied. $\mathfrak{c}^c_{Event}$ is always applied before $\mathfrak{c}^c_{MPICExit}$. When the last event e_i of a collective-operation instance is reached, $\mathfrak{c}^c_{Event}$ is applied first. However, at this moment, the last event has not yet become part of $\mathfrak{C}^c_i$ so that $\mathfrak{c}^c_{Event}$ is without any effect. After that, $\mathfrak{c}^c_{MPICExit}$, is applied and e_i is added to $\mathfrak{C}^c_i$. Now, the complete instance is a subset of $\mathfrak{C}^c_i$ and the condition in $\mathfrak{c}^c_{Event}$ allowing the removal of this instance is satisfied. Finally, after proceeding to the next event, $\mathfrak{c}^c_{Event}$ is applied again and the complete instance is removed.

To access the events belonging to a collective-operation instance, another auxiliary function is defined that can be used to isolate these events.

Definition 3.9 (mpicoll()). Let $e \in E$ be an event from the event trace:

$$mpicoll(e) := \begin{cases} Inst & \text{if} \quad type(e) = \text{MPICExit} \quad \wedge \\ & \quad \forall l \in Group(e.com): \quad |haveloc(\mathfrak{c}_i^{e.com}, l)| \geq 1 \\ \emptyset & \text{else} \end{cases}$$

$$\text{where} \quad Inst := \bigcup_{l \in Group(e.com)} leastrcnt(haveloc(\mathfrak{c}_i^{e.com}, l))$$

◇

If e is an MPICExit event that completes a collective-operation instance, then $mpicoll()$ returns all MPICExit events belonging to that instance. Otherwise the empty set is returned. How to access the corresponding Enter events will be explained later when dealing with pointer attributes.

Enhancing the model in this way means considering a collective operation essentially as a set of single region instances. This viewpoint has the advantage that it allows OpenMP parallel constructs to be treated in a similar way. Thus, it provides a very general idea of a collective operation.

Pointer Attributes

Both region instances and messages can be represented by pairs of matching events. A region instance is characterized by its Enter and Exit events, and a message is characterized by its Send and Receive events. For this reason, it would be reasonable to provide a link connecting both sides of each pair; that is to say, a link from the Exit event to its corresponding Enter event and a link from the Receive event to its corresponding Send event. The direction of these links follows Definition 3.3, which prohibits links from pointing into the future. However, the relationship connecting an Exit event with its matching Enter event is actually a specialization of a general relationship between an arbitrary event and the Enter event of the region instance enclosing it. Therefore, a link should connect an arbitrary event with the Enter event of its enclosing region instance; that is, with the event that was "at the top" of the region stack immediately before the event happened. The following pointer attributes are defined using conditions that are similar to those already used for the definition of state sequences.

Definition 3.10 (enterptr). The $enterptr$ attribute is a pointer attribute for an arbitrary event $e_i \in E$ that points to the Enter event of the region instance in which the event e_i took place:

$$e_i.enterptr := \begin{cases} mostrcnt(\mathfrak{R}_{i-1}^{e_i.loc}) & \text{if} \quad \mathfrak{R}_{i-1}^{e_i.loc} \neq \emptyset \\ null & \text{else} \end{cases}$$

◇

Definition 3.11 (sendptr). The $sendptr$ attribute is a pointer attribute for a Receive event $e_i \in E_{Receive}$ that points to its corresponding Send event:

$$e_i.sendptr := \; leastrcnt(\{e \in \mathfrak{M}_{i-1}^{e_i.src,e_i.loc} | \;\; e.tag = e_i.tag \quad \wedge \quad e.com = e_i.com\})$$

$\diamond$

The corresponding Send event is the least recent event with matching tag and communicator in the message queue for traffic from the message's source location to its destination location, that is, the location of the Receive event.

The meaning of these pointer attributes is illustrated in Figure 3.3. Here, events taking place at two different locations are shown along their time lines. The upper location performs two nested region instances. During the inner region instance (indicated by the dark-gray bar) a message is sent to the lower location.

The auxiliary function $mpicoll()$ (Definition 3.9) delivers all MPICExit events belonging to the same collective-operation instance. The Enter events of that instance can be accessed by following the *enterptr* attributes originating from the returned events as depicted in Figure 3.2. Thus, it is easy to access the whole set of events constituting a collective-operation instance.

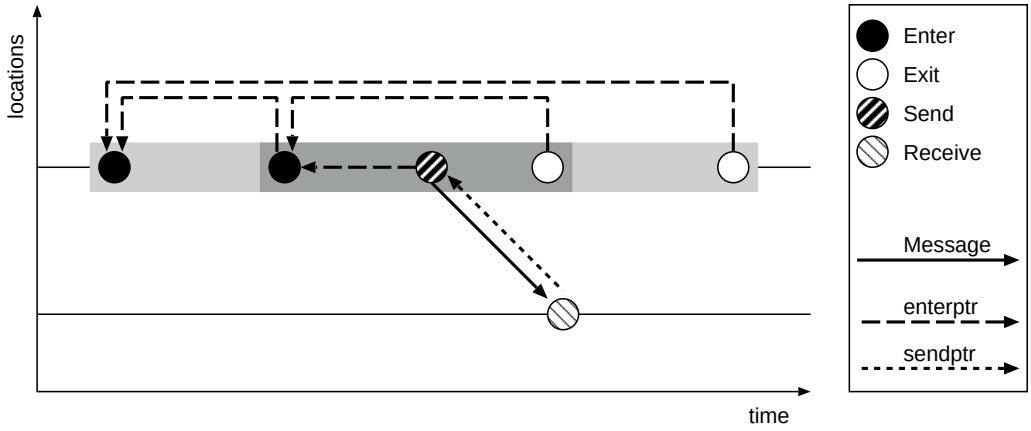

Figure 3.3: References provided by pointer attributes.

Note that the pointer attributes from Definition 3.10 and 3.11 can be used to abbreviate the definitions of $\mathfrak{R}^l$ and $\mathfrak{M}^{s,d}$. Although state sequence and pointer attribute definitions would reference each other, they would remain well defined for the reasons mentioned earlier.

3.4.3 MPI 2

In addition to simple point-to-point communication and collective communication among the members of a group of processes as provided by the first version of MPI, the MPI Forum

decided to make more advanced features, such as parallel file IO, *Remote Memory Access* (RMA), and dynamic process management, part of MPI and publish it as the latest version MPI 2 [53].

Parallel file IO was devised to increase the performance of file IO operations. The IO features of MPI 2 not available in traditional UNIX IO include noncontiguous access in both memory and file, collective IO operations, use of explicit offsets to avoid separate seeks, shared file pointers, non-blocking IO, portable and customized data representations, and hints for the implementation and file system.

Whereas traditional point-to-point communication combines communication and synchronization by requiring each side to explicitly invoke an operation and to supply message parameters, RMA is based on a separation of the two concerns by allowing a process to access another process's memory without that process's explicit participation. However, to maintain consistency of memory accesses each RMA epoch is embraced by specific synchronization operations depending on the RMA access category.

Similar to PVM [29], MPI 2 allows the dynamic startup of processes. This feature serves two goals: the dynamic adjustment of the number of processes to work on the problem at hand and the connection of two MPI applications started separately. Central to this feature are intercommunicators, which distinguish between a local and a remote group of processes.

3.5 Model Enhancement: OpenMP

The OpenMP interface for shared-memory programming offers a set of directives that can be inserted into the source code to instruct the compiler to parallelize code sections, such as loops. In addition, the application may call OpenMP library functions to control the parallel environment or to perform lock synchronization.

3.5.1 Basic Event Model

The execution of an OpenMP application follows the fork-join model. The program starts with a master thread, which creates a team of slave threads when entering a parallel region. The team is terminated after leaving the parallel region and only the master thread resumes its execution. Thus, the locations of OpenMP events are threads.

However, when using nested parallelism by allowing slave threads to create subteams, the unique identification of a thread is no longer possible because the OpenMP library treats every team as a separate name space for thread identifiers. That is, when a new subteam has been created the application can only ask for the identifier of a thread relative to the name space of that subteam. For this reason and due to the large number of applications

that do not make use of this feature, nested parallelism is ignored here and it is assumed that there is no nested execution of parallel regions.

If nested parallelism is ignored, threads can be uniquely identified by their thread number. Thus, each location is an element of $L := T = \{0, \ldots, n_t - 1\}$, where n_t is the total number of threads. The thread number can be obtained using an OpenMP library call. The master thread always has the thread number 0. For simplicity, it is assumed that the team size used in parallel regions remains constant during the entire program execution and is not dynamically adjusted.

A thread with thread number t that is created before and terminated after a parallel region is considered to be the same location as a thread with the same thread number t that is created before and terminated after another parallel region. The thread is assumed to be suspended instead of really being terminated. Thus, the total number of locations only depends on the maximum number of threads running simultaneously.

The static view of OpenMP applications is similar to that of MPI applications in that an OpenMP application's source code is also made up of regions. However, apart from regions, such as functions, loops, and basic blocks, which can be found in MPI applications as well, there are regions that are defined by enclosing them with directives. The resulting regions are called OpenMP *constructs*.

Thus, the basic event model for OpenMP includes Enter and Exit events as the basic MPI model does. However, an OpenMP application may use multiple OpenMP constructs of the same type at different places in the source code. For this reason, it is assumed that the region attribute of OpenMP *constructs* both makes different regions of the same type distinguishable and also encodes the construct type. This is expressed by a function $regtype()$ which can be applied to the region attribute of Enter events and which provides the required distinction among different types of OpenMP constructs.

In contrast to traditional MPI, where all processes start from the very beginning[2], OpenMP starts with only one master thread, which forks into parallel execution only after reaching the first parallel region. Also, after finishing a parallel region the slave threads are terminated and the master thread is the only one that continues execution. To identify the points in time when the execution switches from serial to parallel mode and vice versa, the OpenMP event model defines two event types Fork and Join that are placed immediately before the start of parallel execution and after the end of parallel execution, respectively. They both inherit from a common base type Team.

In OpenMP applications, the threads of a team execute parallel constructs collectively, that is, those constructs that are intended to be simultaneously executed by multiple threads are executed by all threads and in the same order. Thus, the execution of OpenMP parallel constructs follows the same rules concerning the order and participation as MPI collective

[2]Note that the latest version MPI 2 allows process creation during run time as an advanced feature.

operations do. The difference to MPI is that the group of locations is not represented by a communicator but by a team of threads. For pure OpenMP applications without nested parallelism this is the set of all threads. Figure 3.4 shows the execution of a parallel-region construct. The master thread generates Fork and Join events immediately before and after this parallel-region construct.

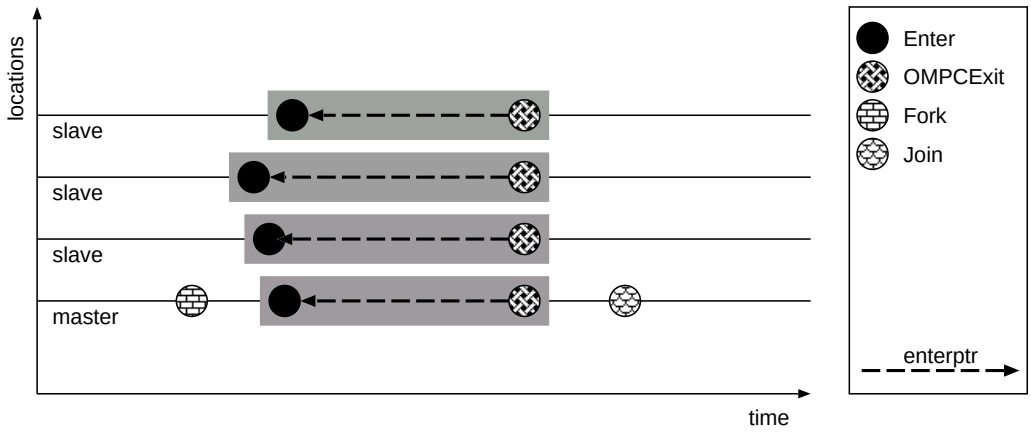

Figure 3.4: Collective execution of an OpenMP parallel region.

To let an event trace reflect the collective execution of parallel constructs, their execution is finished by generating an OMPCExit event instead of a regular Exit event. Parallel constructs are those OpenMP constructs that are executed by multiple threads to exploit or control parallelism. They are listed below:

- parallel
- (parallel) do/for
- (parallel) sections
- (parallel) workshare
- single
- barrier (implicit and explicit)

Note that an implicit (i.e., compiler-generated) barrier associated with a parallel construct is considered to be a separate construct and is treated the same way as an explicit (i.e., user-specified) barrier, except that its source-code location cannot be determined explicitly. So when executing a parallel construct with implicit barrier, the control flow is assumed to perform a nested execution of a parallel construct without an implicit barrier but with an enclosed execution of an “explicit” barrier at its end. Again, OMPCExit is a specialization of Exit. However, this time the specialization does not carry any additional attributes, it denotes only a more specialized context.

The OpenMP event model also includes lock synchronization. There is one type Alock for the event of acquiring a lock and one type Rlock for the event of releasing a lock. Of course, an Alock event always occurs before its corresponding Rlock event. Similar to MPI point-to-point event types, these event types actually describe a non-atomic activity, so both events represent only one small point in time of the actual duration. Therefore, an Alock denotes the first moment after the lock has been acquired, and an Rlock denotes the first moment after the lock has been released. That is, the thread is in possession of the lock only between the Alock and the corresponding Rlock event.

To identify the lock they refer to, both types carry an attribute *lock* that contains an identifier of the lock object they operate on. This attribute is inherited from a common base type *Sync* indicating an event related to lock synchronization. *Sync* events are usually placed in between the Enter and Exit events of the corresponding OpenMP library functions. The complete basic event model for OpenMP is depicted in Figure 3.5.

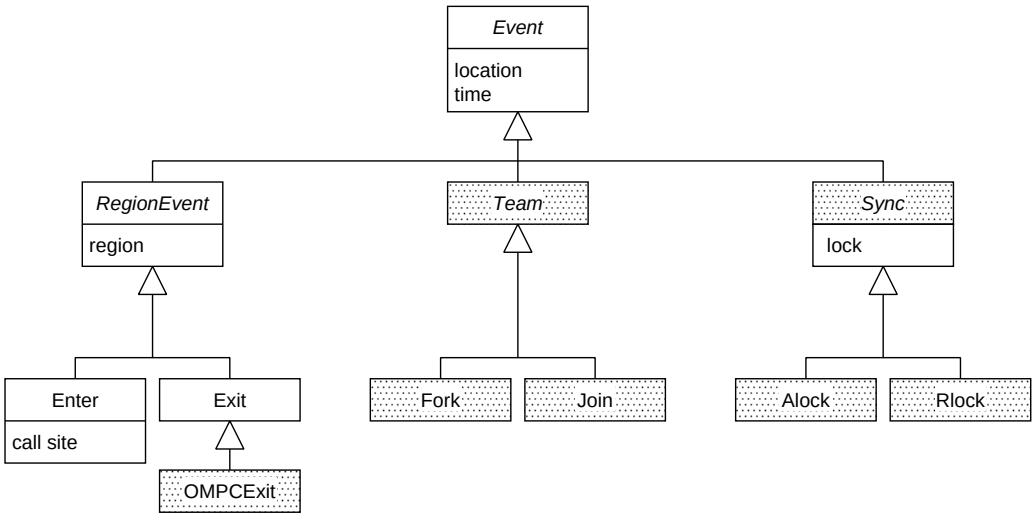

Figure 3.5: Basic event model for OpenMP applications. Spotted boxes represent OpenMP-specific event types.

3.5.2 Enhancement

Some of the abstractions defined for MPI apply to OpenMP as well. These include the region stack and the $enterptr$ attribute that refer to the control flow as it appears in applications of both programming models. And as already anticipated, the treatment of parallel-construct instances is similar to that of MPI collective-operation instances. However, a feature different from MPI is lock synchronization. Here, a way to track the ownership history of a lock object is presented.

Also, the region-stack concept is extended to take into account the fact that a slave thread is a copy of an already running program in contrast to a program starting from the very beginning. Again, the definition of auxiliary functions and constants simplifies the definitions presented in this section.

Definition 3.12. Let $l \in L$ be a location:

$$\begin{aligned} master &:= 0 \\ isslave(l) &:= \begin{cases} true & \text{if} \quad l \neq 0 \\ false & \text{else} \end{cases} \end{aligned}$$

◇

The constant $master$ just identifies the unique master thread, which has the thread number zero, whereas $isslave()$ is a predicate that indicates whether a location is not the master thread, that is, whether its thread number is not equal to zero.

State Sequences

The capture of parallel-construct instances follows the same principle as the capture of MPI collective-operation instances does. As already mentioned, parallel constructs are executed by all threads and in the same order. Since there is only one group of locations (i.e., the team of threads $= L$), only one state sequence for the whole team is defined.

Definition 3.13 (Parallel-Construct Queue). The *parallel-construct queue* $\mathfrak{P}$ is a state sequence that collects the OMPCExit events of active OpenMP parallel-construct instances. Its transition operator $\mathfrak{p}$ is defined by the following transition functions:

$$\begin{aligned} \mathfrak{p}_{OMPCExit} &: \mathfrak{P}_i := \mathfrak{P}_{i-1} \cup \{e_i\} \\ \mathfrak{p}_{Event} &: \mathfrak{P}_i := \begin{cases} \mathfrak{P}_{i-1} \setminus Inst & \text{if} \quad \forall l \in L : |haveloc(\mathfrak{P}_{i-1}, l)| \geq 1 \\ \mathfrak{P}_{i-1} & \text{else} \end{cases} \end{aligned}$$

$$\text{where} \quad Inst := \bigcup_{l \in L} leastrcnt(haveloc(\mathfrak{P}_{i-1}, l))$$

◇

An OMPCExit event is added to $\mathfrak{P}_{i-1}$ by applying the function $\mathfrak{p}_{OMPCExit}$. As soon as all OMPCExit events belonging to a parallel-construct instance are elements of $\mathfrak{P}_{i-1}$, they are removed by applying $\mathfrak{p}_{Event}$. A function $ompcoll()$ to access the events belonging to the same parallel-construct instance can be defined in a way analogous to $mpicoll()$ in Definition 3.9.

When an OpenMP application forks into parallel execution, it creates one or more copies of the master thread. The slaves do not start their execution at the main function, instead they start at the entry of a parallel region. Since the region stack from Definition 3.6 collects events from only one location, the first candidate to be collected for a slave would be the Enter event of the parallel region. However, somebody might be interested in the call path leading to that parallel region and so it seems reasonable to let a slave inherit the region stack of its master. Therefore, the region stack is extended in such a way that slaves always get the region stack of their master upon their creation.

Definition 3.14 (Inherited Stack). The *inherited stack* $\mathfrak{I}^l$ of a location $l \in L$ is a state sequence that collects the Enter events of active region instances at location l. If l is a slave thread, $\mathfrak{I}^l$ inherits the stack of its master upon creation. The transition operator $\mathfrak{i}^l$ of $\mathfrak{I}^l$ is defined by the following transition functions:

$$\mathfrak{i}^l_{Enter} \;:\; \mathfrak{I}^l_i \;:=\; \begin{cases} \mathfrak{I}^l_{i-1} \cup \{e_i\} & \text{if} \quad e_i.loc = l \\ \mathfrak{I}^l_{i-1} & \text{else} \end{cases}$$

$$\mathfrak{i}^l_{Exit} \;:\; \mathfrak{I}^l_i \;:=\; \begin{cases} \mathfrak{I}^l_{i-1} \setminus mostrcnt(\mathfrak{I}^l_{i-1}) & \text{if} \quad e_i.loc = l \\ \mathfrak{I}^l_{i-1} & \text{else} \end{cases}$$

$$\mathfrak{i}^l_{Fork} \;:\; \mathfrak{I}^l_i \;:=\; \begin{cases} \mathfrak{I}^{master}_{i-1} & \text{if} \quad isslave(l) \\ \mathfrak{I}^l_{i-1} & \text{else} \end{cases}$$

$$\mathfrak{i}^l_{Join} \;:\; \mathfrak{I}^l_i \;:=\; \begin{cases} \emptyset & \text{if} \quad isslave(l) \\ \mathfrak{I}^l_{i-1} & \text{else} \end{cases}$$

$\diamond$

The first two functions $\mathfrak{i}^l_{Enter}$ and $\mathfrak{i}^l_{Exit}$ work exactly as the functions of the previously defined region stack do. They just collect Enter events and remove them upon the occurrence of their corresponding Exit events. The difference comes with the third and fourth function. In the case of the master thread, $\mathfrak{i}^l_{Fork}$ and $\mathfrak{i}^l_{Join}$ behave neutrally. In the case of a slave thread, however, $\mathfrak{i}^l_{Fork}$ takes the (inherited) region stack of the master and assigns it to $\mathfrak{I}^l_i$. After that, the inherited stack contains not only the Enter events from its own location l but also the Enter events of the master at the moment when the slave was created. After parallel execution is finished, $\mathfrak{i}^l_{Join}$ reflects the slave's termination in that it assigns the empty set to $\mathfrak{I}^l_i$.

The programmer of an OpenMP application is able to control synchronization explicitly by calling OpenMP lock-synchronization functions. Among other mechanisms lock synchronization may have an important influence on the performance behavior of an application.

To track the ownership history of a lock object, it is useful to define an auxiliary state sequence that remembers its last change of ownership.

Definition 3.15 (Ownership Status). Let K be the set of all locks used at run time. The *ownership status* $\mathfrak{O}^k$ of a lock $k \in K$ is an auxiliary state sequence that contains the most recent event changing the ownership status of k. Its transition operator $\mathfrak{o}^k$ is defined by the following transition function:

$$\mathfrak{o}^k_{Sync} \;:\; \mathfrak{O}^k_i \;:=\; \begin{cases} \{e_i\} & \text{if} \quad e_i.lock = k \\ \mathfrak{O}^k_{i-1} & \text{else} \end{cases}$$

◇

This auxiliary state sequence is not intended to be used directly, instead it is used in the next subsection to define a pointer attribute linking the events that constitute a lock object's ownership history.

Pointer attributes

If a lock has finally been acquired after waiting a considerable period of time, it is interesting to know why the lock was unavailable. Also, if a lock has been released, it is useful to find out when it was acquired. Both questions can be answered by providing a pointer attribute that links a lock-ownership event, that is, a *Sync* event, to the preceding *Sync* event referring to the same lock object.

Definition 3.16 (lockptr). The $lockptr$ attribute is a pointer attribute for a Sync event $e_i \in E_{Sync}$ that points to the preceding *Sync* event operating on the same lock object:

$$e_i.lockptr := \begin{cases} e_j & \text{if} \quad \mathfrak{O}^{e_i.lock}_{i-1} = \{e_j\} \\ null & \text{else} \quad (\text{in this case is} \quad \mathfrak{O}^{e_i.lock}_{i-1} = \emptyset) \end{cases}$$

◇

It is clear that the $lockptr$ of an Rlock event always points to an Alock event and the $lockptr$ of an Alock event always points to an Rlock event. Only the first Alock event of a lock object points to $null$ because it represents the start of the ownership history. Figure 3.6 shows how to navigate along a lock object's ownership history using the $lockptr$ attribute. A lock object is acquired the first time by thread A using omp_set_lock. Because there is no ownership history prior to that event, the $lockptr$ points to $null$. Thread B acquires that lock in the same way after it was released by thread A. So the $lockptr$ of that second acquisition points to the first release and the $lockptr$ of the first release points to the first acquisition.

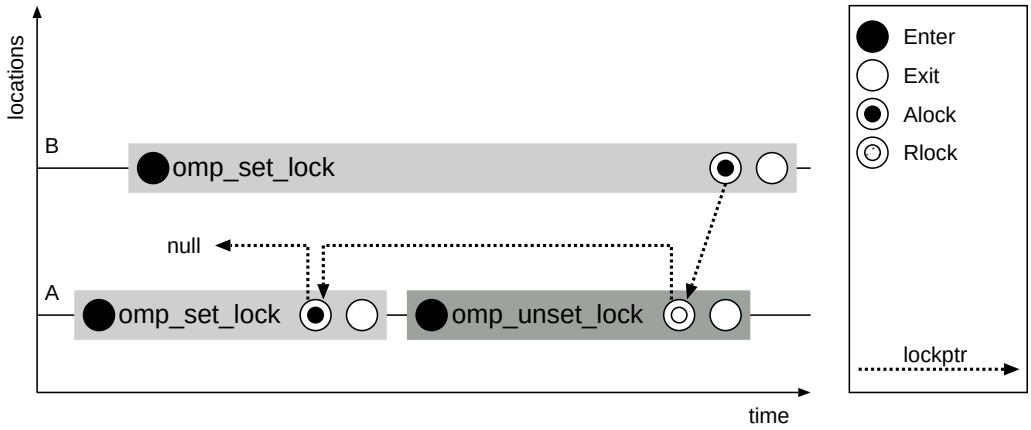

Figure 3.6: Navigating along the ownership history of a lock object using the $lockptr$ attribute. To keep the figure simple it does not show any other pointer attributes.

3.6 Model Enhancement: Hybrid Model

Coupled SMP systems represent a distributed-memory architecture whose nodes consist of shared-memory components. The central idea behind the hybrid programming model is to exploit this hybrid hardware architecture by combining MPI with OpenMP in the same application. In such a scenario, MPI is intended to perform the communication among distributed memories that constitute the nodes of the system and OpenMP is intended to share data within the shared memories of single SMP nodes. Of course, depending on the hardware and the algorithm there may be more than one MPI process per node, but in most cases it is only a single process.

Such a hybrid application runs multiple MPI processes, which may themselves consist of multiple threads, so the run-time structure of a hybrid application resembles the hierarchical hardware structure. Figure 3.7 illustrates the physical and logical hierarchies in a coupled SMP system. One SMP node with a physical shared memory and multiple CPUs may accommodate multiple processes, which provide a shared address space to one or more threads.

In the previous two sections the location of an event was always a logical one. In the case of MPI it was the process, in the case of OpenMP it was the thread. Now, the event location is a tuple consisting of a process and a thread.

Moreover, it may be useful to augment this tuple by adding a physical part containing information on the CPU, SMP node, or even machine if there are multiple machines involved in the same computation. For example, the MPI communication characteristics between two processes may depend on their physical location, that is, whether they reside on different

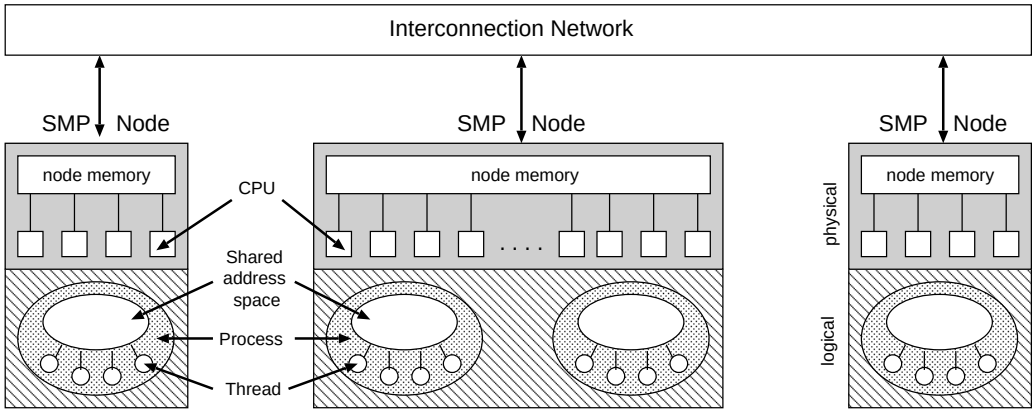

Figure 3.7: The physical and logical structure of coupled SMPs.

SMP nodes of the same machine or even on different machines.

However, whereas the machine and the SMP node are easily identifiable in most cases, it is very difficult to determine the CPU on which an event happens because most operating systems do not provide functions to ask for it and, in addition, a process or thread may switch rapidly among different CPUs on the same SMP node so that the result of a query may not reflect the CPU at the moment when the query is issued. For this reason, the location of an event is now described as a tuple (machine, SMP node, process, thread). The location coordinates are written in the order of increasing granularity because a machine may have multiple SMP nodes, which may accommodate multiple processes each running multiple threads.

Definition 3.17 (Location (hybrid)). The *location* $l \in L$ of an event occurring during execution of a hybrid application is a tuple (m, s, p, t). m denotes a machine (i.e., a coupled SMP system), s denotes an SMP node of that machine, p denotes a process, and t denotes a thread of that process. The different coordinates of a location $l \in L$ are referenced as $l.coordinate$ (e.g., $l.p$).

It is assumed that each process has a fixed SMP node $s = s(p)$ and machine $m = m(p)$, that is, a process may not migrate between SMP nodes or even machines.

Processes are uniquely identified by their rank $p \in P = \{0, \ldots, n_p - 1\}$ in MPI_COMM_WORLD, which is a global identifier across all physical locations. n_p is the total number of processes.

Threads are identified by their OpenMP thread number $t \in T(p) = \{0, \ldots, n_t(p) - 1\}$. The thread number is an identifier local to the process p. $n_t(p)$ is the maximum number of threads spawned by process p. The master thread of a process always has thread number zero.

A thread with thread number t that is created before and terminated after a parallel region is assumed to be the same location as a thread with the same thread number t that is

created before and terminated after another parallel region. Thus, the total number of locations owned by a process only depends on the maximum number of threads executed simultaneously as part of that process.

Thus, the set of locations is:

$$L := \{(m(p), s(p), p, t) | \quad p \in P, \quad t \in T(p)\}$$

◇

It follows that the total number of locations owned by a whole application is just the sum of the maximum thread numbers across all processes:

$$|L| = \sum_{p \in P} |T(p)| = \sum_{p \in P} n_t(p)$$

Definition 3.17 says nothing about machine and SMP-node identifiers because they are not used to define any abstractions. Also, they are not standardized and may vary from platform to platform.

The combination of MPI and OpenMP is possible in three different modes, which are described in order of increasing generality. The first mode allows MPI calls to be made only by the master thread. The second mode allows MPI calls to be made by an arbitrary thread, but in a serialized fashion, that is, only one thread at a time. The third mode imposes no restriction at all on the order and location of MPI calls.

The event model requires information on the source and destination location to be attached to the Send and Receive events of a point-to-point message. From an instrumenter's perspective, this is easy in the case of the first mode because this information is already supplied as an argument to point-to-point functions since the source or destination process always corresponds to one unique master thread. The second mode already requires complicated postprocessing of the event trace to match corresponding point-to-point events in order to calculate the correct source and destination locations. Also, the group associated with a communicator would no longer be a set of locations. Instead, it would be a set of processes, that is, a set of sets of locations (i.e., threads). Finally, the third mode, which allows the concurrent execution of MPI calls by multiple threads of the same team, makes it difficult to identify matching Send and Receive events because multiple threads may attempt to receive a message directed to a certain process. For this reason and due to the predominant practical importance of the first mode, the following discussion is restricted to the first mode, which allows only the master thread to invoke MPI operations.

The types in the hybrid event model are the union of the types from the separate models. The complete type hierarchy is depicted in Figure 3.8. Hatched boxes represent MPI-specific event types and spotted boxes represent OpenMP-specific event types. Attribute names that have not been defined so far will be introduced later in this section.

The remainder of this section describes how the models for single programming models need to be adapted to be suitable for the hybrid programming model. Moreover, an additional abstraction is introduced that allows the context of an event to be captured in a more convenient way than that provided by the abstractions developed so far.

3.6.1 MPI

The abstractions of the former model also remain valid in view of hybrid programming. The region stack, which is actually not related to a specific programming model, requires no changes. Also, the message queue and the collective-operation queue can remain unmodified, and the pointer attributes $enterptr$ and $sendptr$ do not need any changes either. Of course, the message queue needs to be considered only for pairs of source and destination locations that represent master threads, which have thread number zero (i.e., $(m, s, p, 0)$). As a consequence, a communicator's group is always a set of master-thread locations.

3.6.2 OpenMP

The abstractions defined for OpenMP require only minor modifications to take into account the fact that hybrid applications may have multiple master threads as a consequence of having multiple processes. So the previously defined constant $master$ is turned into a function that maps an arbitrary thread onto its master thread just by setting the thread number to zero and keeping the process number. In addition, the set $Team(p)$ denotes those locations that have the process number p.

Definition 3.18. Let $l \in L$ be a location with $l = (m, s, p, t)$ and $p \in P$ a process number:

$$\begin{aligned} master(l) &:= (m, s, p, 0) \\ Team(p) &:= \{l \in L | \quad l.p = p\} \end{aligned}$$

◇

Consequently, the state sequence $\mathfrak{P}$ for the capture of parallel-construct instances now needs to be defined separately for each process p because parallel constructs are collectively executed only by threads belonging to the same process. $\mathfrak{P}^p$ collects OMPCExit events occurring as part of process p, and removes them upon completion of the parallel-construct instance.

Definition 3.19 (Parallel-Construct Queue (hybrid)). The *parallel-construct queue* $\mathfrak{P}^p$ of a process p is a state sequence that collects the OMPCExit events of active OpenMP parallel-construct instances of the process p. Its transition operator $\mathfrak{p}^p$ is defined by the following transition functions:

$$\mathfrak{p}^p_{OMPCExit} : \mathfrak{P}^p_i := \mathfrak{P}^p_{i-1} \cup \{e_i\} \quad \text{if} \quad e_i.loc.p = p$$

$$\mathfrak{p}^p_{Event} : \mathfrak{P}^p_i := \begin{cases} \mathfrak{P}^p_{i-1} \setminus Inst & \text{if} \quad \forall l \in Team(p) : |haveloc(\mathfrak{P}^p_{i-1}, l)| \geq 1 \\ \mathfrak{P}^p_{i-1} & \text{else} \end{cases}$$

$$\text{where} \quad Inst := \bigcup_{l \in Team(p)} leastrcnt(haveloc(\mathfrak{P}^p_{i-1}, l))$$

◇

Also, the transition function $\mathfrak{i}^l_{Fork}$ of the inherited stack $\mathfrak{I}_l$, which is responsible for handing over the stack of the master thread, now refers to the master thread of the location it is associated with.

Definition 3.20 (Inherited Stack (hybrid)). The *inherited stack* $\mathfrak{I}^l$ of a location $l \in L$ is a state sequence that collects the Enter events of active region instances at location l. If l is a slave thread, $\mathfrak{I}^l$ inherits the stack of its master upon creation. The transition operator $\mathfrak{i}^l$ of $\mathfrak{I}^l$ is defined by the following transition functions:

$$\mathfrak{i}^l_{Enter} : \mathfrak{I}^l_i := \begin{cases} \mathfrak{I}^l_{i-1} \cup \{e_i\} & \text{if} \quad e_i.loc = l \\ \mathfrak{I}^l_{i-1} & \text{else} \end{cases}$$

$$\mathfrak{i}^l_{Exit} : \mathfrak{I}^l_i := \begin{cases} \mathfrak{I}^l_{i-1} \setminus mostrcnt(\mathfrak{I}^l_{i-1}) & \text{if} \quad e_i.loc = l \\ \mathfrak{I}^l_{i-1} & \text{else} \end{cases}$$

$$\mathfrak{i}^l_{Fork} : \mathfrak{I}^l_i := \begin{cases} \mathfrak{I}^{master(l)}_{i-1} & \text{if} \quad isslave(l) \\ \mathfrak{I}^l_{i-1} & \text{else} \end{cases}$$

$$\mathfrak{i}^l_{Join} : \mathfrak{I}^l_i := \begin{cases} \emptyset & \text{if} \quad isslave(l) \\ \mathfrak{I}^l_{i-1} & \text{else} \end{cases}$$

◇

Note that care should be taken when creating lock-object identifiers to be used as values for the $lock$ attribute. The current definition of $\mathfrak{O}^k$ expects that each lock object has an identifier that is also unique across different processes, so the instrumentation system must ensure this uniqueness also across process borders or the definitions of $\mathfrak{O}_k$ and $lockptr$ must be modified in a way that distinguishes among different processes.

3.6.3 Dynamic Call Path

Optimizing a parallel application requires knowledge of which parts of the program are responsible for inefficient behavior. However, identifying the weak parts means not only identifying code regions where inefficient behavior takes place, but also understanding the context in which it happens; that is, the purpose for which these region have been visited.

The (inherited) region stack provides exactly this context in that it gives the user the full path of regions that have been visited on the way to the current region. Although this region path may contain regions that cannot be entered by calling a function, this thesis refers to it as the *call path* because most readers are presumably more familiar with this designation.

However, the stack contains more information than necessary because in many cases the user is not interested in the time at which the individual predecessor regions have been entered. In addition, finding out whether two events have the same call path requires a tedious comparison of their respective region stacks.

The set of all call paths in a program forms a structure that is commonly called the *call graph*. The call graph created from considering potential call paths through static analysis is called the *static call graph*, whereas the call graph created from call paths that have really been executed during run time is called the *dynamic call graph*. Because event traces reflect only call paths visited during program execution, only the dynamic call graph is considered in the following.

The nodes of the call graph represent call paths visited during program execution and the directed edges represent transitions between call paths that have occurred during program execution while entering a new region, that is, while generating an Enter event. Before considering the dynamic call graph in more detail, the call path of an Enter event is defined and the notion of call-path equivalence between two Enter events is introduced.

Definition 3.21 (Call Path). The call path $cpath(e)$ visited by an Enter event $e \in E_{Enter}$ is a sequence of pairs (region, call site), which can be derived from the inherited stack of the event's location by extracting the corresponding attributes from the events of $\mathfrak{I}^{e.loc}(e)$ in the order of their occurrence in the event trace:

$$cpath(e) := (r_1, c_1) \circ \ldots \circ (r_n, c_n)$$

$$\begin{aligned} \text{where} \quad & \mathfrak{I}^{e.loc}(e) = \{e_{i_1}, \ldots, e_{i_n}\} && \wedge \\ & (\forall \quad 1 \leq j, k \leq n : \quad j < k \quad \Rightarrow \quad e_{i_j} < e_{i_k}) && \wedge \\ & (\forall \quad 1 \leq j \leq n : \quad e_{i_j}.reg = r_j \quad \wedge \quad e_{i_j}.csite = c_j) \end{aligned}$$

The symbol $\circ$ concatenates element pairs to form a sequence.

$\diamond$

Definition 3.22 (Call-Path Equivalence). Two Enter events $e, f \in E_{Enter}$ are call-path equivalent $e \equiv_{cp} f$ if and only if they have the same call path:

$$e \equiv_{cp} f \quad \Leftrightarrow \quad cpath(e) = cpath(f)$$

$\diamond$

The call-path equivalence identifies Enter events having the same call path. Since it is an equivalence relation, it partitions all Enter events into classes of call-path equivalent events. This suggests encoding the nodes N of the call graph, which are nothing but call paths, as representatives of such equivalence classes. One way to define a representative for each class is to take the first Enter event visiting the corresponding call path, that is, the least recent one in the class.

Definition 3.23 (Call-Path Representative). The representative $\overline{e}$ of the call-path–equivalence class an Enter event e belongs to is the least recent one in the class:

$$\overline{e} := leastrcnt(\{f \in E_{Enter} | \quad e \equiv_{cp} f\})$$

$\diamond$

Thus, the set of nodes N in the call graph can be written as the set of Enter events that represent themselves:

$$N := \{n \in E_{Enter} | \quad \overline{n} = n\} \tag{3.5}$$

There is an edge from a node $l \in N$ to a node $r \in N$ if and only if $cpath(l) \circ (r.reg, r.csite) = cpath(r)$, that is, if there exists an Enter event that moves from l to r. Consequently, there is an edge from l to r if and only if:

$$\exists e \in E_{Enter} : \quad \overline{e.enterptr} = l \quad \wedge \quad \overline{e} = r \tag{3.6}$$

The condition above requires that there is an Enter event executing a transition from the call path (i.e., node) represented by l to that represented by r. Obviously, if there is such an event e the condition is satisfied by e's representative $\overline{e} = r$ as well and, vice versa, because $\overline{e}$ moves between the same call paths as e does. So it is possible to rewrite Condition 3.6 as:

$$\overline{r.enterptr} = l \tag{3.7}$$

It follows that the set of edges can be derived solely from elements of N, that is, N encodes the whole call graph. Whereas the nodes are encoded as representatives, the edges are implicitly provided by the above relationship (3.7). To be able to associate a compound event with the call path where it happens, it seems reasonable to have a pointer attribute

$cnodeptr$ of an Enter event $e_i \in E_{Enter}$ that points to the corresponding call-graph node (i.e., its representative):

$$e_i.cnodeptr := \overline{e_i} \tag{3.8}$$

However, it is not obvious whether this (3.8) is a definition compliant with the definition template for pointer attributes (3.4). To give a compliant definition of $cnodeptr$, it is assumed to be defined for now and, on this basis, two auxiliary abstractions are defined that can be used to provide a valid definition for $cnodeptr$.

Definition 3.24 (cedgeptr). The $cedgeptr$ attribute is a pointer attribute for an Enter event $e_i \in E_{Enter}$ that points to the call-graph node (i.e., its representative) that has been left by e_i:

$$e_i.cedgeptr := \begin{cases} e_i.enterptr.cnodeptr = \overline{e_i.enterptr} & \text{if} \quad e_i.enterptr \neq null \\ null & \text{else} \end{cases}$$

◇

Note that $cedgeptr$ encapsulates the relationship that must exist between two nodes to have an edge connecting them. Although Definition 3.24 refers to a pointer attribute of e_i, it is compliant with the definition template for pointer attributes (3.4). This is because the expressions $e_i.enterptr$ can be replaced by a function of $\mathfrak{R}_{i-1}^{e_i.loc}$ (see Definition 3.10). Due to the inductive way of defining $e_i.cedgeptr$, $\mathfrak{R}_{i-1}^{e_i.loc}$ and all $\{e.cnodeptr | \quad e \in \mathfrak{R}_{i-1}^{e_i.loc}\}$ can be assumed to be already defined. Also, the definition of $e_i.cedgeptr$ satisfies the working-set requirement (Section 3.3.3) because to calculate $e_i.cedgeptr$ it is only necessary to access e_i and $\mathfrak{R}_{i-1}^{e_i.loc}$.

The second abstraction is an auxiliary state sequence $\mathfrak{D}$ that collects all call-path representatives and encodes the part of the dynamic call graph that has been visited so far. As already mentioned, the edges can be derived from the nodes, so it is sufficient to collect the nodes. However, the edge condition will contribute to the definition of $\mathfrak{D}$.

Definition 3.25 (Dynamic Call Graph). The *dynamic call graph* $\mathfrak{D}$ is a state sequence that collects all call-path representatives. Its transition operator $\mathfrak{d}$ is defined by the following transition function:

$$\mathfrak{d}_{Enter} \ : \ \mathfrak{D}_i \ := \begin{cases} \mathfrak{D}_{i-1} \cup \{e_i\} & \text{if} \quad \nexists \ \ l, r \in \mathfrak{D}_{i-1} : \\ & \qquad (\quad r.cedgeptr = l \quad \wedge \\ & \qquad \quad e_i.cedgeptr = l \quad \wedge \\ & \qquad \quad e_i.reg = r.reg \quad \wedge \\ & \qquad \quad e_i.csite = r.csite \quad) \\ \mathfrak{D}_{i-1} & \text{else} \end{cases}$$

◇

An Enter event is added to $\mathfrak{D}_{i-1}$ if there is no edge in the call graph as represented by $\mathfrak{D}_{i-1}$ that describes a transition between the call path left by e_i and the one visited by e_i. If there is no such edge contained in $\mathfrak{D}_{i-1}$ then a a new node with a new implicit edge is added in the form of e_i. If there is already such an edge, $\mathfrak{D}$ remains as it is when moving from $\mathfrak{D}_{i-1}$ to $\mathfrak{D}_i$.

The first part of the conjunction in $\mathfrak{d}_{Enter}$ requires l and r to be an edge (l, r) in the call graph. The second part ensures that the call path left by e_i is equal to the call path visited by l, whereas the last two parts ensure that the call path visited by e_i is equal to that visited by r. The full condition guarantees that the event added to $\mathfrak{D}_{i-1}$ is a call-path representative because it requires that the corresponding call path has never been seen before, which follows from the nonexistence of an edge leading to that call path.

$\mathfrak{D}_i$ is well defined because due to the inductive nature of defining $\mathfrak{D}_i$, $\mathfrak{D}_{i-1}$ and $e_i.cedgeptr$ can be assumed to be already defined. It is obvious that the definition of $\mathfrak{D}_i$ satisfies the working-set requirement. Note that, in contrast to the state sequences defined so far, $\mathfrak{D}$ never shrinks in size because the dynamic call graph never becomes smaller while program execution is proceeding. Now, it is easy to give a new definition of the $cnodeptr$ attribute (3.8) according to the template (3.4).

Definition 3.26 (cnodeptr). The $cnodeptr$ attribute is a pointer attribute for an Enter event $e_i \in E_{Enter}$ that points to the call-graph node (i.e., its representative) that has been visited by e_i:

$$e_i.cnodeptr := \begin{cases} r & \text{if } \exists\ l, r \in \mathfrak{D}_{i-1} : \\ & \quad (\ r.cedgeptr = l \ \wedge \\ & \quad\ \ e_i.cedgeptr = l \ \wedge \\ & \quad\ \ e_i.reg = r.reg \ \wedge \\ & \quad\ \ e_i.csite = r.csite\ ) \\ e_i & \text{else} \end{cases}$$

$\diamond$

The condition in Definition 3.26 is the negation of that used in Definition 3.25. It asks whether the call path visited by e_i has been seen before. If so, the attribute points to the corresponding representative $r \in \mathfrak{D}_{i-1}$. If not, it is a representative itself, in which case it must point to itself. Note that there is at most one $r \in \mathfrak{D}_{i-1}$ with those properties because each equivalence class has only one representative.

Similar to Definition 3.24, Definition 3.26 refers to a pointer attribute of e_i. However, by replacing $e_i.cedgeptr$ by the right-hand side of its definition and by expanding there the expression $e_i.enterptr$, which then appears as a function of $\mathfrak{R}_{i-1}^{e_i.loc}$, it becomes obvious that Definition 3.26 is compliant with the definition template for pointer attributes (3.4) and that it satisfies the working-set requirement.

Note that this view of the call graph does not include a special treatment of recursive programs. Each step in a recursion may create an additional node in the call graph instead of forming a cycle. For this reason, the call graph will form a tree if it has only one root path. This condition is usually satisfied in standard single-program multiple-data (SPMD) scenarios, where each instance of the program starts at the same main routine.

As will be demonstrated in Section 3.8.3, the $cnodeptr$ attribute provides a convenient means to associate a performance-relevant compound event with the corresponding node in the call graph and, thus, to quickly determine and compare the execution context of different compound events.

3.6.4 Summary

This subsection summarizes the enhanced model for hybrid applications including event types and constraints and gives an overview of all abstractions introduced so far. The model for hybrid applications comprises the union of all event types contained in the individual models for MPI and OpenMP. This union includes types referring to the control flow that denote the entry and the exit of a program region. In addition, MPI-specific event types include sending and receiving point-to-point messages as well as leaving collective operations; OpenMP-specific event types cover team creation and termination, the exit of parallel constructs, and lock acquisition and release. Figure 3.8 shows the complete event-type hierarchy. It also shows the pointer attributes in bold-face letters added to the basic attributes. Since the hybrid event model includes the execution of pure MPI and pure OpenMP applications as special cases, from now on everything will be expressed in terms of this hybrid event model. Those abstractions that have been introduced as auxiliary ones are necessary to define other abstractions but are not intended to be used in compound-event specifications.

Constraints. The entries and exits of region instances occurring on the same location must form a correct parenthesis expression, that is, a region is only allowed to be left after all enclosed regions have been left. In addition, the Send event of an MPI message must occur before its corresponding Recv event, and MPI collective operations must be executed in the same order by all members of the group associated with the communicator. Similar to MPI, OpenMP parallel constructs must be executed in the same order by all threads of the team. Finally, an Alock event must always occur before its corresponding Rlock event and two Alock events operating on the same lock are not allowed to follow each other immediately without an intermediate Rlock event that releases that lock before it can be acquired again.

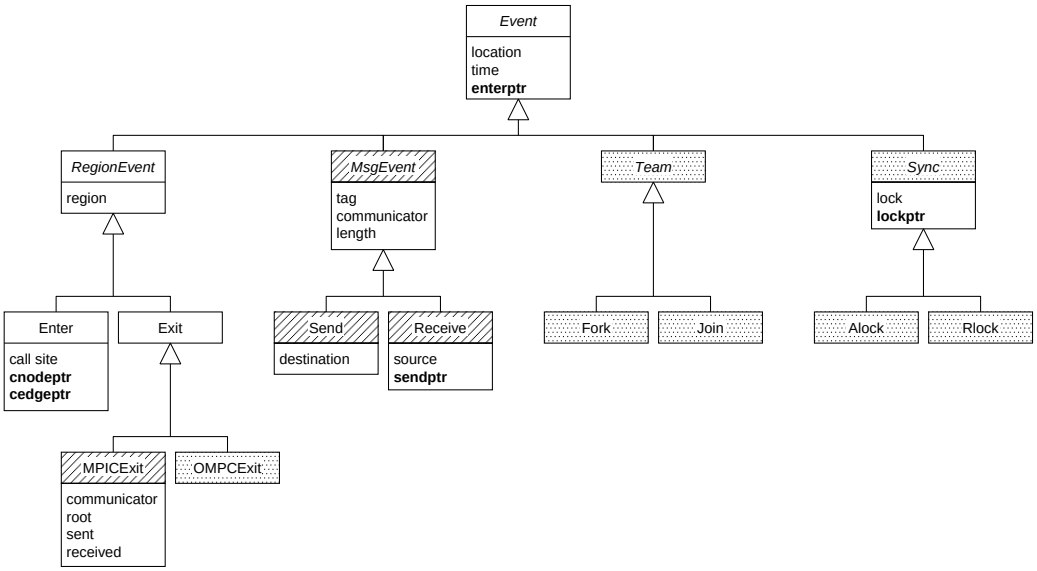

Figure 3.8: The type hierarchy for hybrid applications including the pointer attributes in bold-face letters. Hatched boxes represent MPI-specific event types and spotted boxes represent OpenMP-specific event types.

State sequences. The enhanced model provides state sequences to capture events that are still relevant to the context of a given event in contrast to events that are not relevant because they refer to already completed activities.

- $\mathfrak{R}^l$ (Definition 3.6): One region stack per location l that remembers all Enter events of active region instances at location l.
- $\mathfrak{I}^l$ (Definition 3.20): One inherited region stack per location l that remembers all Enter events of active region instances at location l and, in addition, for slave threads copies the (inherited) region stack of their master thread upon their creation.
- $\mathfrak{M}^{s,d}$ (Definition 3.7): One message queue per location pair source s and destination d that remembers all Send events of messages currently being transferred from s to d. According to the restriction that MPI statements are only allowed to be executed by the master thread, there are message queues only for source and destination locations that represent master threads (i.e., $(m, s, p, 0)$).
- $\mathfrak{C}^c$ (Definition 3.8): One state sequence per communicator c that collects all events belonging to the same instance of an MPI collective operation. If e is an event that completes such a collective operation, then the function $mpicoll(e)$ delivers all events belonging to that instance, otherwise it delivers the empty set.
- $\mathfrak{P}^p$ (Definition 3.19): One state sequence per team represented by a process p that

collects all events belonging to the same instance of an OpenMP parallel construct. If e is an event that completes such a collective operation, then the function $ompcoll(e)$ delivers all events belonging to that instance, otherwise it delivers the empty set.

- $\mathfrak{O}^k$ (Definition 3.15): One auxiliary state sequence per lock object k that starts as an empty set and after the first acquisition of that lock object always contains the last event changing the lock's ownership status.
- $\mathfrak{D}$ (Definition 3.25): One auxiliary state sequence that collects all call-path representatives and encodes the part of the dynamic call graph that has been visited so far.

Thus, the flat overall state Γ_i of an event e_i in the hybrid model is the union:

$$\Gamma_i = \bigcup_{l \in L} \mathfrak{R}_i^l \cup \bigcup_{l \in L} \mathfrak{I}_i^l \cup \bigcup_{s,d \in L} \mathfrak{M}_i^{s,d} \cup \bigcup_{c \in C} \mathfrak{C}_i^c \cup \bigcup_{p \in P} \mathfrak{P}_i^p \cup \bigcup_{k \in K} \mathfrak{O}_i^k \cup \mathfrak{D}_i$$

To allow the efficient computation of abstractions related to e_i, it is sufficient to provide fast access to the elements of $\Delta_i = \Gamma_{i-1} \cup \{e_i\}$ because others need not be accessed.

Pointer attributes. The enhanced model provides pointer attributes that link related events together to give better access to instances of higher-level concepts, such as region instances, messages, a lock's ownership history, and the call path.

- $enterptr$ (Definition 3.10): Each event provides an $enterptr$ attribute pointing to the Enter event of the currently active region instance at the location of that event. The $enterptr$ attributes of events occurring at top level points to $null$.
- $sendptr$ (Definition 3.11): Each Receive event provides a $sendptr$ attribute pointing to the corresponding Send event.
- $lockptr$ (Definition 3.16): Each *Sync* event provides a $lockptr$ attribute pointing to the last *Sync* event changing the ownership status of the same lock. If the *Sync* event is the first event operating on that lock, the $lockptr$ attribute points to $null$.
- $cedgeptr$ (Definition 3.24): Each Enter event provides an auxiliary $cedgeptr$ attribute pointing to the call-graph node (i.e., its representative Enter event) that has been left by that event. If the call-graph node visited by that event is an entry point to the call graph, the attribute points to $null$.
- $cnodeptr$ (Definition 3.26): Each Enter event provides a $cnodeptr$ attribute pointing to the call-graph node (i.e., the Enter event representing that node) that has been visited by that event.

3.7 Specifying Compound Events

In the preceding sections an event model has been defined that allows simple behavioral elements of a parallel application to be described. This section explains how to combine these elements to higher-level compound events.

A specification method for compound events representing performance properties of a parallel or distributed application should meet the following requirements:

1. It should be simple even in the case of complex compound events.
2. It should allow for an efficient implementation.

The first requirement demands the specification of the relationships among the constituent events of a compound event on a very high level of abstraction. The second requirement concerns the efficiency of possible search methods. This is especially important in view of the huge amount of data typically involved in event tracing. It seems that both requirements can be fulfilled on the basis of enhanced event models. First, a general scheme for specifying compound events is introduced and then it is explained how enhanced event models in conjunction with the scheme proposed here are able to meet the two requirements.

Definition 3.27 (Compound Event). A *compound event* $C \subseteq E$ is a subset of events of an event trace E. The elements of this subset are called the *constituents* of the compound event. The set of constituents can be divided into a set of not necessarily disjoint subsets, which are called the *fractions* of the compound event:

$$C := \bigcup_{i \in I} C_i, \quad I = \{1, \ldots, n\}$$

The fractions are connected by relationships that can be expressed using functional dependencies:

$$C_i := f_i(C'), \quad C' = \bigcup_{j \in J} C_j, \quad J \subset I$$

The resulting dependency graph must be acyclic. Furthermore, each C_i except for one fraction has at least one predecessor $C_{j \neq i}$ it depends on. In addition, there is one fraction that has no predecessor and that contains only a single event. This fraction is called the *root fraction* and the single event inside is called the *root event*.

The root event is characterized by a *root predicate* that can be used to decide whether an arbitrary event from the trace is a possible root event.

$\diamond$

A compound event C consists of a set of primitive events (i.e., its constituents). A constituent event is an instance of an event type defined in an enhanced event model. The fractions reflect the logical structure of the compound event. The relationships that connect the different fractions can be expressed using functional dependencies f_i which map a set of events onto another set of events from the trace. The functions may use the abstractions provided by an enhanced event model (i.e., state sequences and pointer attributes).

However, to be useful for the purpose of specifying compound events this scheme must meet several conditions. As already mentioned, it must ensure that the corresponding dependency graph is acyclic. Furthermore, each C_i except for one *root* fraction must have at least one predecessor $C_{j \neq i}$ that it depends on. So every C_i can be calculated from the root fraction by evaluating the functional dependencies. Of course, it is possible that an evaluation step fails, so each f_i can also be considered as a predicate imposing constraints on the structure of the compound event. As a final condition the root fraction is required to have exactly one element because in this way it can be easily characterized using a simple predicate. The root predicate can be applied to an arbitrary event in order to decide whether it is a possible root event.

Note that in many cases the fractions $C_i \subseteq C$ will consist of only a single event. Nevertheless, permitting multiple events to be members of such a subset is necessary, for example, to make complete states of a state sequence or MPI collective-operation instances part of a compound event.

Algorithm 3.1. *To locate all occurrences of a compound event C in an event trace E perform the following for all events $e \in E$:*

1. *Apply the root predicate to e.*
2. *If successful, instantiate all constituents that are reachable from the root event by evaluating the functional dependencies.*
3. *Instantiate all constituents that are reachable from the constituents already instantiated.*
4. *Repeat step 3 until all constituents are instantiated or an instantiation step fails. If all constituents have been instantiated, one instance of C has been found.*

◇

Note that to avoid two different root events leading to the instantiation of the same compound event it is necessary to either check for double instantiation or to ensure that each compound-event specification provides one unique root event.

Recall the two requirements from the beginning of this section. An enhanced event model provides abstractions that correspond to the vocabulary of the programming model used. By using these abstractions when defining the functions involved in the definition of a

whole compound event, it should be possible to produce a simple and understandable specification for most of the compound events representing typical performance properties. A demonstration of this will be given in Section 3.8.

The efficiency of an implementation of Algorithm 3.1, which is addressed by the second requirement, relies heavily on the mechanism used for accessing events. Consider the following typical scenario. In order to apply the algorithm for locating instances of a compound event, a search tool walks sequentially through the event trace. If an event fulfills the root predicate, the tool will start to evaluate the tree of functional dependencies used for defining the compound event. It will try to access other events from the event trace. As already mentioned, most of these events belong to the context (i.e., events from the overall state) of the root event or of events belonging to the recent past of the root event. All the tool is required to do is to track the overall state of the events it accesses during its sequential walk and to provide efficient buffered access to the working sets of a relatively small contiguous window of the event trace.

Of course, the scheme itself does not impose any restrictions on the complexity of the functions f_i, so these have to be defined carefully. Nevertheless, the design of the EXPERT performance tool presented in Chapter 4 suggests that in the context of performance analysis the complexity of the required functions is manageable.

3.8 Example Compound Events

This section shows that complex performance properties of parallel applications can be easily represented by applying the previously presented scheme. The properties are specified as compound events, thereby making use of the previously defined abstractions. Note that all examples require a performance-data granularity for their representation that is provided only by event traces. Although the performance properties are sorted according to the programming model to which they refer, the corresponding compound events are always based on the hybrid event model because the hybrid model contains the single programming models as special cases.

In the following compound-event examples, a symbol beginning with a lower-case letter denotes a single event, whereas a symbol beginning with an upper-case letter denotes a set of events.

3.8.1 MPI

The most important performance properties related to MPI appear in conjunction with blocking communication, which can cause significant waiting times. Note that non-blocking communication may be affected as well, since non-blocking communication may

be finished with a blocking operation, such as MPI_Wait. In particular, collective operations, such as MPI_Allreduce, frequently appear to be the source of performance problems because their inherent all-to-all semantics have a synchronizing effect that is difficult to avoid.

Example 3.1 (Late Sender). The first example describes the situation that occurs when an MPI_Recv operation is posted before the corresponding MPI_Send has been started (Figure 3.9). The receiver remains idle during the interval between the two calls instead of doing useful computation.

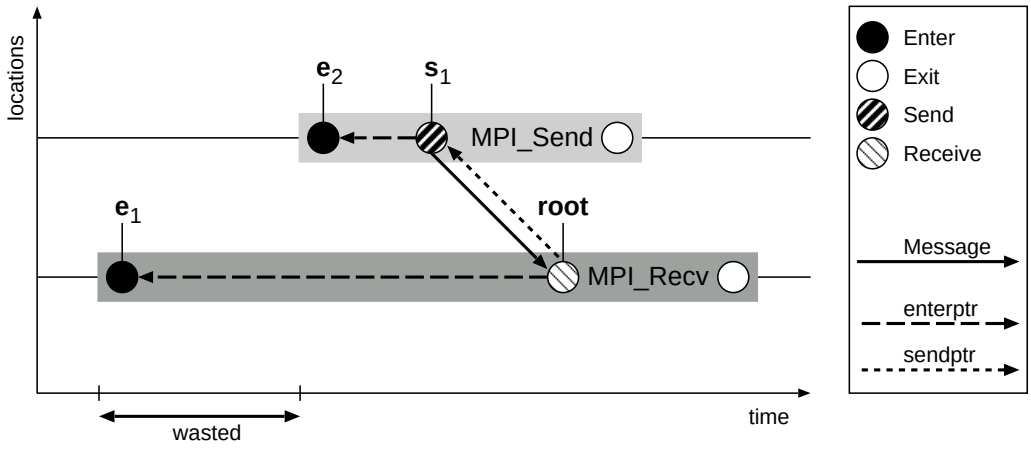

Figure 3.9: *Late Sender* compound event.

This compound event $\{\{root\}, \{s_1\}, \{e_1\}, \{e_2\}\}$ consists of four fractions, each containing only a single event. The root event ($root$) is just the event indicating the message arrival (i.e., an event of type Receive). Thus, we have the following root predicate:

$$type(root) = Receive$$

The other three events are the event of sending the message (s_1), the event of entering the MPI_Send region (e_2), and the event of entering the MPI_Recv region (e_1). They are defined as follows:

$$s_1 := root.sendptr$$

$$e_1 := \begin{cases} root.enterptr & \text{if } root.enterptr.reg = \text{MPI\_Recv} \\ fail & \text{else} \end{cases}$$

$$e_2 := \begin{cases} s_1.enterptr & \text{if } (s_1.enterptr.reg = \text{MPI\_Send} \wedge e_2.time > e_1.time) \\ fail & \text{else} \end{cases}$$

Applying Algorithm 3.1 to this compound-event specification would result in the following sequence of actions. When a potential candidate for the root event has been found by

evaluating the root predicate, that is to say, when an event of type Receive has been found, the algorithm traces back the $sendptr$ attribute of the root event to locate the corresponding Send event (s_1). Now the event of entering MPI_Recv (e_1) is determined by navigating along the $enterptr$ attribute of the root event. To ensure that this event really refers to a region instance of MPI_Recv, the reg attribute is checked. Event e_2 is instantiated in a similar manner, but here an additional constraint, which is essential for the whole compound event, must be taken into consideration. The MPI_Recv has to be called before the MPI_Send. So the two time stamps must be compared. After instantiation of all compound-event constituents, a tool can compute the amount of wasted time by subtracting the two time stamps:

$$wasted = e_2.time - e_1.time$$

The compound event described here is a frequently occurring situation of inefficient behavior, which can be observed for several real-world applications, as shown in Chapter 5.

○

Example 3.2 (Late Receiver). This compound event refers to the inverse case. The send operation MPI_Send blocks until the corresponding receive operation is called. This can happen for several reasons. Either the MPI implementation is working in synchronous mode by default, or the size of the message to be sent exceeds the available MPI-internal buffer space and the operation blocks until the data is transferred to the receiver. The behavior is similar to an MPI_Ssend waiting for message delivery. The situation is depicted in Figure 3.10. The definition is very close to the *Late Sender* compound event. In particular, the root predicate is identical so it is not shown again.

$$s_1 := root.sendptr$$

$$e_1 := \begin{cases} root.enterptr & \text{if } root.enterptr.reg = \text{MPI\_Recv} \\ fail & \text{else} \end{cases}$$

$$e_2 := \begin{cases} s.enterptr & \text{if } \left(\begin{array}{l} s_1.enterptr.reg = \text{MPI\_Send} \ \wedge \\ e_2.time < e_1.time \ \wedge \\ e_2 \in \mathfrak{R}^{s_1.loc}(e_1) \end{array} \right) \\ fail & \text{else} \end{cases}$$

An important difference to the previous compound event is the condition appearing in the definition of e_2. Of course, $e_2.time$ has to be less than $e_1.time$, since the receiver has to be later than the sender. In addition, the MPI_Send operation must not have finished before the MPI_Recv has been called. So it is necessary to look at the region stack of the location from where the message was sent and at the time just after the MPI_Recv call was posted ($\mathfrak{R}^{s_1.loc}(e_1)$). If e_2 is an element of this set, MPI_Send and MPI_Recv overlap in time.

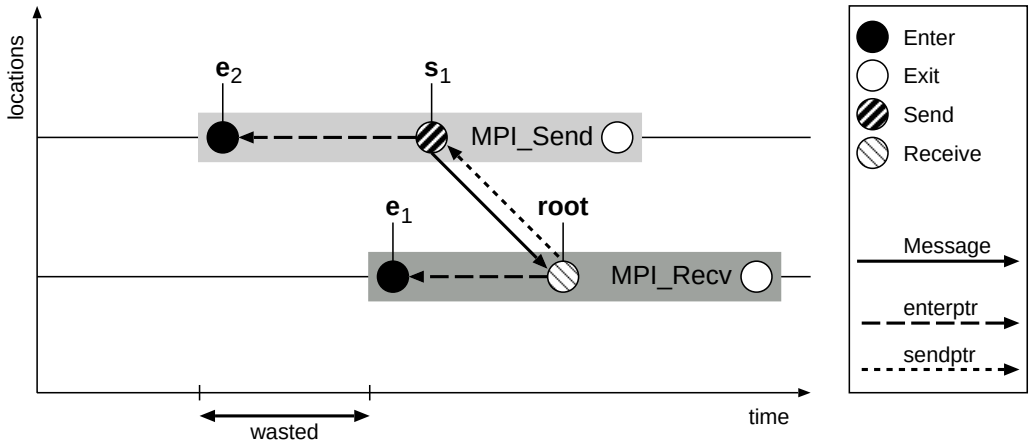

Figure 3.10: *Late Receiver* compound event.

It is clear that this criterion still does not prove waiting of MPI_Send due to lack of buffer space with maximum reliability. Nevertheless, it is a necessary condition and it is the strongest that can be proved based on the data available in a typical event trace. A detailed discussion of the performance problem related to this compound event can be found in [32].

This and the previous example are special cases of a larger class of similar compound events involving alternative MPI point-to-point communication functions.

○

Example 3.3 (Messages in Wrong Order). The third example is taken from the *Grindstone Test Suite for Parallel Performance Tools* [39] and highlights the problem of passing messages in the wrong order. This problem can arise if one process is expecting messages in a certain order, but another process is sending messages that are not in the expected order. In Figure 3.11 an extreme example is shown. In the left part of the picture, process 1 is processing incoming messages in the reverse sending order. Processing them in the order they were sent would not only speed up the program but would also require much less buffer space for storing unprocessed messages. This is shown in the right part of the picture.

This situation is modeled as a message that is sent later but received earlier than another message with the same sender and receiver. For this reason, the compound event $\{\{root\}, \{s_1\}\}$ consists of two fractions, each containing only a single event. The root event ($root$) is the message receipt, the other event (s_1) is the message dispatch. Again, the root predicate only requires the event type to be Receive. s_1 is defined as follows:

$$s_1 := \begin{cases} root.sendptr & \text{if } \exists e \in \mathfrak{M}^{root.src,root.loc}(root): \\ & \quad e.time < root.sendptr.time \\ fail & \text{else} \end{cases}$$

The condition on the right-hand side requires that there are older messages in the message

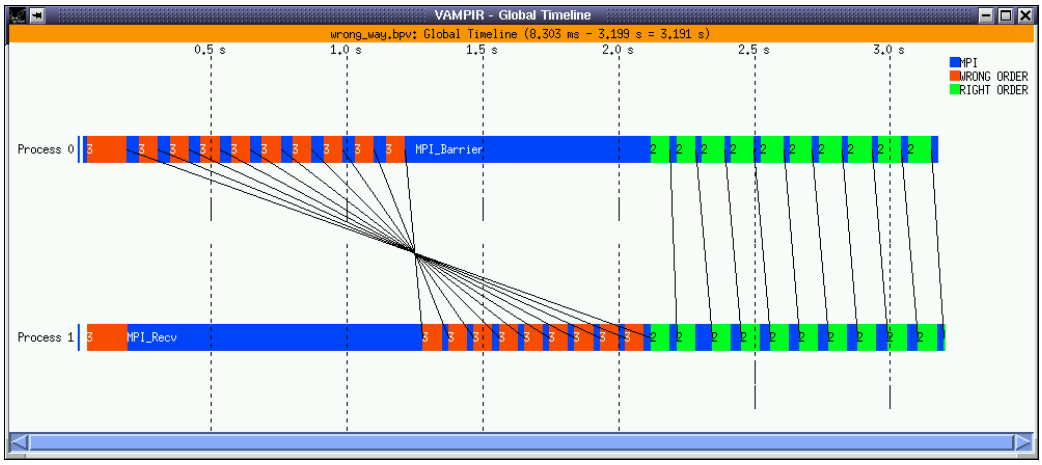

Figure 3.11: Passing messages in the wrong order.

queue for traffic between the source and the destination location of the message under consideration. So there have to be messages in transit that have not been received at the time the current message has been received.

○

Example 3.4 (Wait at N × N). This compound event involves an MPI collective operation and deals with a problem associated with n-to-n operations, such as MPI_Allreduce. Since each process involved in such an operation has to send to as well as to receive from every other process, no process can leave the operation until the last process has entered it. So there is an inherent synchronization that can introduce significant waiting times.

The compound event describes the frequently occurring situation of reaching an n-to-n operation at different points in time and thus introducing undesirable synchronization overhead. The starting point is now the last MPICExit event of an MPI_Allreduce operation instance. This is expressed by the following root predicate using the $mpicoll()$ function:

$$\begin{aligned} &type(root) = \text{MPICExit} \quad \wedge \\ &mpicoll(root) \neq \emptyset \quad \wedge \\ &root.reg = \text{MPI\_Allreduce} \end{aligned}$$

The compound event $\{\{root\}, E_1, E_2\}$ consists of three fractions: the root fraction ($\{root\}$), the set of MPICExit events belonging to the MPI_Allreduce instance (E_1), and the Enter events of that instance (E_2). E_1 is instantiated using the $mpicoll()$ function. Then E_2 is computed by tracing back the $enterptr$ attributes of E_1. For convenience, $E_1.attr$ is used as a short cut for $\{v| \quad \exists e \in E_1 : e.attr = v\}$.

$$\begin{aligned} E_1 &:= mpicoll(root) \\ E_2 &:= \begin{cases} E_1.enterptr & \text{if} \quad \exists e_i, e_j \in E_1.enterptr : \quad e_i.time \neq e_j.time \\ fail & \text{else} \end{cases} \end{aligned}$$

Of course, this compound event matches nearly every MPI_Allreduce instance because the operation is almost never entered at exactly the same time at different locations. However, instantiating this compound event allows the amount and location of the occurred synchronization overhead to be computed.

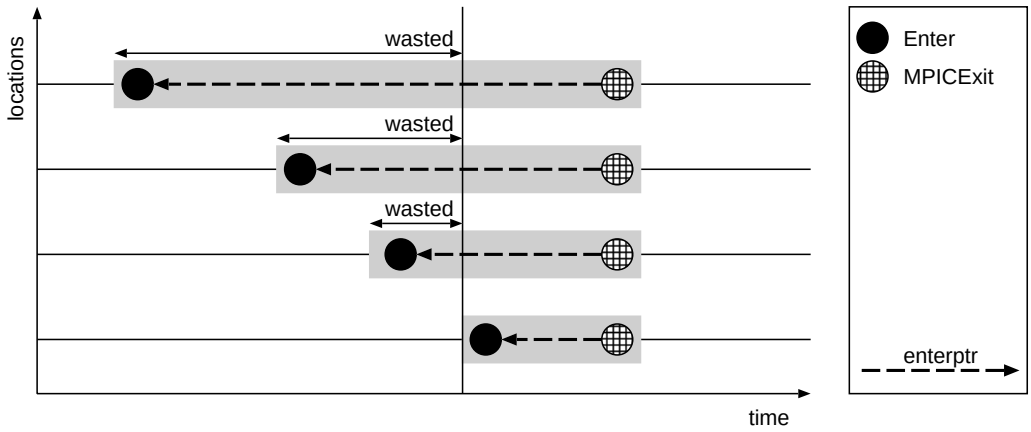

Figure 3.12: Synchronization overhead of n-to-n collective operations.

A good estimate of the synchronization overhead incurred during the execution of an n-to-n operation is the execution time of the operation until the last participant has joined in. Figure 3.12 shows the synchronization times for individual participants. The vertical line indicates the point in time when the last participant has started to execute the operation. Everything left of the line is estimated to be overhead. In terms of the compound-event specification, the total time wasted as a result of synchronization can be conveniently written as:

$$wasted = \sum_{e \in E_2} max(E_2.time) - e.time$$

○

3.8.2 OpenMP

In particular, OpenMP-related performance properties based on waiting times resulting from barrier or lock synchronization can be readily handled using the compound-event method.

Example 3.5 (Unbalanced Barrier). Similar to MPI applications, in OpenMP applications the threads of a team may reach a barrier instance at different points in time and thus introduce undesirable synchronization overhead. As mentioned earlier, the notion of collectively executed operations as incorporated in the model applies to both MPI and OpenMP in that it considers an instance of such an operation as a set of single region instances connected by constraints concerning the order of occurrence and the number of participating

locations. In both cases, a simple function delivers the set of Exit events belonging to a "collective" operation instance that has just been completed. In the case of MPI it was $mpicoll()$, in the case of OpenMP it was $ompcoll()$. However, OpenMP provides no way of identifying the region type solely based on the region name, so as opposed to MPI, the compound events for OpenMP must rely on the $regtype()$ function. This leads to a root predicate slightly different from Example 3.4:

$$\begin{aligned} &type(root) = \text{OMPCExit} && \wedge \\ &ompcoll(root) \neq \emptyset && \wedge \\ &regtype(root.reg) = \text{OpenMP Barrier} \end{aligned}$$

The rest of the compound event looks exactly like Example 3.4 and, therefore, it is not shown here. Detecting unbalanced OpenMP barriers may allow conclusions to be drawn about the efficiency of, for example, scheduling strategies applied in a parallel loop.

○

Example 3.6 (Lock Competition). This performance property deals with the situation that occurs when one thread tries to acquire a lock that is in the possession of another thread. That is, the thread trying to acquire the lock invokes omp_set_lock, before the current owner releases the lock using omp_unset_lock. Figure 3.13 illustrates this behavior in a time-line diagram.

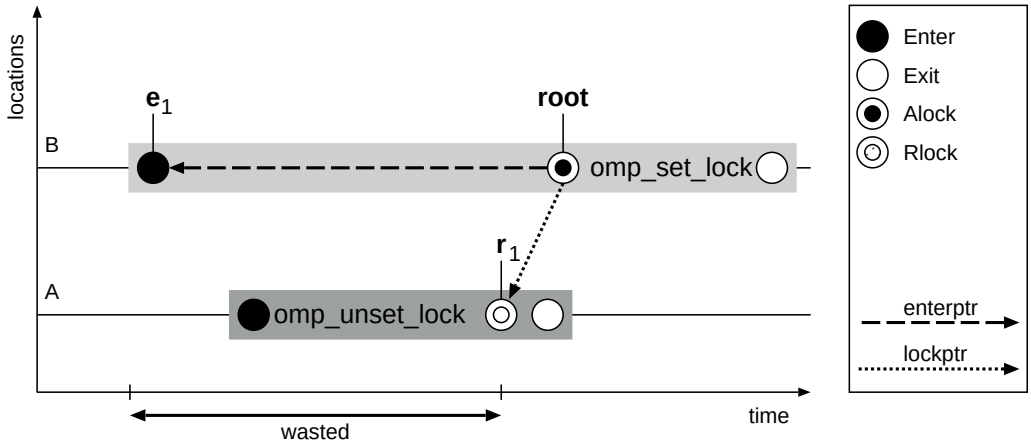

Figure 3.13: Waiting for an OpenMP lock.

The compound event describing this performance property consists of three single-event fractions $\{\{root\}, \{r_1\}, \{e_1\}\}$, which denote the moment of lock acquisition, the moment of the preceding lock release, and the moment of entering the OpenMP function omp_set_lock, respectively. The root condition requires the root event to be of the type Alock:

$$type(root) = \text{Alock}$$

By following the pointer attributes emanating from the root event, it is easy to locate the remaining constituents, as is shown in the figure. Since $root$ is the acquisition immediately

after it has been released by event r_1, $root.syncptr$ points to r_1. Moreover, $root.enterptr$ points to e_1 because it is the Enter event of the enclosing region instance. However, a complete characterization of this situation also imposes constraints on the temporal order of the constituents. That is to say, to be classified as inefficient behavior e_1 is required to take place prior to r_1 because otherwise the lock would already be available to thread B. It follows the formal compound-event specification:

$$
\begin{aligned}
r_1 &:= root.syncptr \\
e_1 &:= \begin{cases} root.enterptr & \text{if} \quad root.enterptr.time < r_1.time \\ fail & \text{else} \end{cases}
\end{aligned}
$$

The time lost due to this situation can be obtained by subtracting the time stamp of e_1 from that of r_1. In addition, the specification allows the location of thread A ($= r_1.loc$), which was the owner of the lock while thread B ($= root.loc$) was trying to acquire it, to be exactly determined and thus gives a better insight into the circumstances of the inefficient behavior considered here.

○

3.8.3 Call Paths

As mentioned in Section 3.6.3, it is useful to know in which call path the program is being executed when a compound event occurs. One way to obtain call path information is to simply ask for the inherited stack of a constituent event and to extract the region and call-site attributes. The drawback of this method is that it must be repeated for every compound-event instance. To avoid this, a performance tool could take advantage of the $cnodeptr$ attribute introduced in Section 3.6.3. Since the $cnodeptr$ encodes the call path associated with an Enter event by pointing to the call-path representative, the call-path extraction needs to be done only once for each representative. So the call-path extraction can be easily separated from determining the call path of a compound-event instance. The latter is simply done by remembering the $cnodeptr$ of one or more characteristic constituents or their corresponding Enter events, which are reachable via $enterptr$.

This can be exemplified using the *Late Sender* compound event from Example 3.1. The fraction of execution time that is wasted occurs during the MPI_Recv instance, whose Enter event is e_1. It follows that $p = e_1.cnodeptr$ encodes the corresponding call path, which can be obtained by querying $\mathfrak{I}^{p.loc}(p)$. The benefit of this method is that this needs only to be done once for all compound events whose call path is represented by p. Note that a compound event may also be associated with multiple call paths.

3.8.4 Compound-Event Instantiation and Constraints

The examples presented in the previous subsections reveal an interesting property of the description method used here. The constituent definitions of a compound event can be divided into two parts. The first part is responsible for locating, i.e., instantiating, the constituents of the compound event. The second part places additional constraints on the constituents that are essential for the performance property they describe. These considerations also apply to the root predicate. The root predicate can be divided into a condition requiring a certain type and optional constraints if necessary.

Writing the constraint part separately allows a clearer distinction among different concerns and improves the readability of the specification. Also, reuse of different specification parts might become easier. For example, Example 3.2 might be rewritten like this:

Root Type

Receive

Instantiation

$$\begin{aligned} s_1 &:= root.sendptr \\ e_1 &:= root.enterptr \\ e_2 &:= s_1.enterptr \end{aligned}$$

Constraint

$$\begin{aligned} root.enterptr.reg &= \text{MPI\_Recv} \quad \wedge \\ s_1.enterptr.reg &= \text{MPI\_Send} \quad \wedge \\ e_1.time &< e_2.time \quad \wedge \\ e_1 &\in \mathfrak{R}^{s_1.loc}(e_2) \end{aligned}$$

The separation of instantiation and constraints will be used for integrating compound events into the ASL specification language in Section 3.9.

3.9 Compound Events in ASL

The APART Specification Language ASL [21] developed by the APART working group on *Automatic Performance Analysis: Resources and Tools* is a novel approach to the formalization of performance properties and the associated performance-related data. ASL provides a formal notation for defining performance properties related to different programming models. By providing a general framework for the specification of performance properties ASL both encourages the collection of a multitude of different performance properties among users and tool builders and also provides a structure that can work as a basis for performance-tool design.

ASL allows performance-related data items to be referenced by means of an object-oriented specification model. ASL distinguishes between static data known at compile time (e.g., information on source-code entities) and dynamic data generated at run time (e.g., CPU-time summaries). In the ASL terminology, a performance property represents one aspect of performance behavior. To test whether such a property is present in an application, an associated condition must be evaluated based on the current performance data. The confidence of a property specifies the reliability of the test condition. If the condition is evaluated as true, the severity of a property indicates its relative importance with respect to other properties. Performance properties are specified in ASL using the *property* construct. Figure 3.14 shows the usage of the property construct to represent the overhead associated with barrier synchronization.

```
PROPERTY synchronization_costs (Region r,
                                Experiment e,
                                Region rank_basis)
{
LET
  float barrier_time = summary(r,e).sums.sync_time;
IN
  CONDITION:   barrier_time > 0;
  CONFIDENCE:  1;
  SEVERITY:    barrier_time/duration(rank_basis,e)
}
```

Figure 3.14: Usage of the ASL property construct.

The three parameters are the region `r` the property refers to, the experiment `e`, which delivers the actual performance data, and the region `rank_basis`, whose duration is the yardstick to which the overhead in region `r` is compared. The `LET` clause assigns the synchronization overhead to a local variable by accessing the object-oriented performance data of region `r` in experiment `e`. The `CONDITION` clause requires this overhead to be greater than zero, which is a reliable criterion, as indicated by the `CONFIDENCE` clause. Finally, the severity is defined in the `SEVERITY` clause as the fraction of the synchronization overhead compared to the duration of the rank basis.

Unfortunately, the initial ASL data model, as specified in [22], mainly concentrates on profiling data, i.e. summary information, and does not take advantage of the more detailed information contained in event traces. The fine-grained view of the execution behavior provided by event traces can be used to identify hidden idle times, to detect programming errors, or to trace back performance problems to source-code entities in a way not supported by profiling data. In particular, the notion of compound events indicating the existence of performance properties is not part of the initial ASL specification. But the very general design of ASL permits the easy integration of this approach into the language and data model, thereby requiring only minor extensions which are presented in the following

subsections. The extensions, which are reflected in the revised ASL version [21], can be divided into three parts. The first part deals with the ASL syntax, the second part describes how to specify abstractions, such as state sequences and pointer attributes, using the ASL specification model. The third part explains how to integrate the new language constructs with existing ones. Finally, a small example illustrates their effective usage.

3.9.1 Language Extensions

Expression Syntax

To be able to express state-transition operators, conditional expressions are required. For this reason, the grammar symbol *expr* is extended by adding a further alternative. In addition, a means to create a set from a list of single elements is provided. The empty set is created by substituting an empty list for the symbol *reference-list*. Finally, the `NULL` literal is introduced to indicate a void reference. The required grammar extensions are listed in Figure 3.15[3].

expr	**is**	[...]
	or	*cond-expr*
cond-expr	**is**	*if-part elif-part* $*$ [*else-part*]
if-part	**is**	`IF` '(' *bool-expr* ')' *expr*
elif-part	**is**	`ELIF` '(' *bool-expr* ')' *expr*
else-part	**is**	`ELSE` *expr*
set-expr	**is**	[...]
	or	'{' *reference-list* '}'
reference	**is**	[...]
	or	`NULL`

Figure 3.15: ASL expression-syntax extensions.

Compound-Event Specification

Compound events are specified in ASL using a new language construct *pattern*. Its name is motivated by thinking of compound events as event patterns. Its syntax is defined in Figure 3.16. Since ASL is intended to specify compound events rather than to implement efficient

[3]In the figure [...] is used as an abbreviation for the unchanged parts of the production rules as defined in [22]

matching algorithms, the *pattern* construct is designed according to the remarks in Section 3.8.4.

```
pattern          is  PATTERN p-name '(' arg-list ')' '{'
                     [LET
                         def *
                     IN ]
                         p-roottype
                         p-instantiation
                         p-constraint
                         p-export
                     '};'

arg              is  type ident
p-roottype       is  ROOTTYPE ':' ident ';'
p-instantiation  is  INSTANTIATION ':' const-def *
p-constraint     is  CONSTRAINT ':' bool-expr
p-export         is  EXPORT [m-name ]':' const-def *
```

Figure 3.16: ASL pattern-specification syntax.

It is possible to parameterize compound-event specifications by declaring formal parameters in the *arg-list*. These parameters, as well as the local definitions from the optional LET clause, can be used in the subsequent parts of the compound-event specification.

The ROOTTYPE clause contains the type of the root event. If it is necessary to allow the root event to have multiple types, a common base type can be used here. The compound-event fractions are defined in the INSTANTIATION clause as constants. Note that fractions consisting of more than one element have to be defined using a set type. It is possible to use conditional expressions here if a correct instantiation only can be guaranteed by evaluating a condition. If an instantiation step fails, the expression used in the constant definition should evaluate to NULL. A condition representing additional compound-event properties that are not needed for instantiation can be placed in the CONSTRAINT clause.

The dual use of the instantiation and the constraint clause therefore provides the opportunity to separate conditions necessary for locating the compound-event's constituents from those that only represent an additional characterization which is not needed to locate any constituents. Note that this separation simplifies the partial reuse of compound-event specifications.

The EXPORT clause defines attributes whose values are computed from the compound-event constituents. The attributes can be accessed through *match objects* of the pattern. Match objects represent compound-event instances and provide a way to access characteristic attributes of single instances. So the export clause implicitly defines a class to which the match objects will belong and which defines those attributes. If necessary the class can

be given a name *m-name*.

The root event as well as the complete event trace can be referenced in a compound-event definition by the two keywords `ROOT` and `TRACE`. In a potential implementation these keywords will be bound to the current root event and the event trace being investigated by the search algorithm. The pattern construct is a useful instrument to increase the property construct's expressiveness, as will be shown later.

3.9.2 Event Types and Abstractions in ASL

The ASL specification model allows the definition of classes and functions. The integration of pattern-based performance properties requires the definition of new classes to represent event types and new functions to represent abstractions, such as state sequences and pointer attributes. It is easy to specify a basic event model as a collection of ASL classes. As opposed to the basic event model, state sequences as well as pointer attributes need to be defined as functions because they are calculated from other events. However, doing this requires a way of representing the order of events in a trace. For this reason, another intrinsic ASL function is introduced that maps an event to its predecessor within the event trace it belongs to:

```
Event PRED(Event e);
```

This function does not have to be defined explicitly in ASL, since it is defined implicitly by the event order produced by common instrumentation systems. Using the new expression syntax it is easy to translate all previously defined state sequences and pointer attributes into ASL. Figure 3.17 gives an example of how to define a state sequence (region stack) in ASL. Figure 3.18 shows how to define a pointer attribute ($enterptr$) in ASL. The keyword `UNIQUE` is used to uniquely select one element from a set. This is possible since the definition forces the set to have exactly one element.

The other abstraction can be defined in a similar way. The purpose of these examples is to show that the ASL language constructs provide the capability to express the abstractions needed for event-model enhancement.

3.9.3 Pattern Matches

The ASL pattern construct is useful to specify two things. First, it specifies a compound event, that is, a set of events being connected by some relationships and fulfilling some constraints. This first aspect is collectively expressed by the root-type clause, the instantiation clause, and the constraint clause. Second, the pattern specifies a class *m-name* of match objects, which are computed from compound-event instances and used to represent

```
setof Enter Rs(Event e, Process p) =

  IF (e == NULL)
    {}
  ELIF (typeof(e) == Enter AND e.process_id == p)
    Rs(PRED(e), p) + { e }
  ELIF (typeof(e) == Exit AND e.process_id == p)
    Rs(PRED(e), p) - { f IN Rs(PRED(e), p) SUCH THAT
                         NEXISTS g IN Rs(PRED(e), p)
                          SUCH THAT g.timestamp > f.timestamp }
  ELSE
    Rs(PRED(e), p);
```

Figure 3.17: The `Rs()` function returning the region stack of a process.

```
Enter enterptr(Event e) =

  IF (Rs(PRED(e), p) == NULL)
    NULL
  ELSE
    UNIQUE({
             f IN Rs(PRED(e), e.process_id) SUCH THAT
               NEXISTS g IN Rs(PRED(e), e.process_id)
                 SUCH THAT g.timestamp > f.timestamp
           });
```

Figure 3.18: The `enterptr()` function returning the *enterptr* attribute.

performance-relevant metrics, such as idle times. This second aspect is expressed by the export clause.

This proposal defines an intrinsic ASL function to obtain the match objects computed from all compound-event instances occurring in an event trace:

```
setof m-name
PATTERN_MATCHES( p-name ( arg-list ), setof Event trace);
```

The function takes two arguments. The first argument is an ASL pattern provided with its own argument list according to its definition. The second argument is the event trace to be analyzed, which is represented by a set of events. When the function is invoked the `TRACE` keyword mentioned in the preceding section is bound to this set. The function returns the set of match objects corresponding to all compound-event instances according to the ASL pattern.

Note that defining more than one pattern will lead to an overloaded `PATTERN_MATCHES()` function whose return type depends on the first argument. This is a consequence of the flexibility introduced by the export clause. Another way would be to restrict pattern definitions to a fixed set of exported attributes. But this would also limit the usefulness of patterns in property definitions.

3.9.4 Example: Late Sender in ASL

This section contains an example of utilizing the ASL pattern construct to specify the *Late Sender* compound event described in Example 3.1. The ASL specification of this compound event is shown in Figure 3.19. It consists of four fractions `ROOT`, `s1`, `e1`, and `e2`, each containing only a single event. The root event or fraction is the event indicating the message arrival (i.e., an event of type `Receive`), which is expressed by the root-type clause.

The other three events are the event of sending the message (s_1), the event of entering the MPI_Send region (e_2), and the event of entering the MPI_Recv region (e_1). They are defined in the instantiation clause using pointer attributes (see also Example 3.1).

The first subproposition of the conjunction in the constraint clause requires the root event to occur when the process of the root event is being executed in region `r`. This is expressed by using the region stack `Rs` (Figure 3.17). Region `r` is supplied as a parameter of the pattern. Now the pattern is valid only for *Late Sender* instances occurring during execution of region `r`. The next two subpropositions require the region instances involved to be MPI_Recv and MPI_Send. The last subproposition describes the necessary temporal displacement between the two function calls. The export clause makes this displacement accessible through an attribute `idle_time`.

```
PATTERN LateSender(Region r)
{
ROOTTYPE: Receive;

INSTANTIATION:
  Send  s1 = sendptr(ROOT);
  Enter e1 = enterptr(ROOT);
  Enter e2 = enterptr(s1);
CONSTRAINT:
  EXISTS e IN Rs(ROOT, ROOT.process_id)
    SUCH THAT e.region == r              AND
  enterptr(ROOT).region  == MPI_Recv     AND
  enterptr(s1).region    == MPI_Send     AND
  e2.timestamp > e1.timestamp;
EXPORT:
  float idle_time = e2.timestamp - e1.timestamp;
}
```

Figure 3.19: *Late Sender* pattern specification in ASL.

```
PROPERTY late_sender(Region r,
                     Experiment e,
                     Region rank_basis)
{
LET
  float idle_time = SUM m.idle_time
    WHERE m IN PATTERN_MATCHES(LateSender(r), e.trace);
IN
  CONDITION: idle_time>0;
  CONFIDENCE: 1;
  SEVERITY: idle_time/duration(rank_basis, e);
}
```

Figure 3.20: *Late Sender* property using a pattern.

3.9.5 Using Patterns in Property Definitions

The purpose of patterns is to make the very detailed information contained in event traces available to property definitions. To meet that goal, a `late_sender` property is defined in Figure 3.20 using the pattern from Figure 3.19.

This property refers to a region `r` that creates *Late Sender* instances during its execution. The confidence is 1, since the criterion used here is safe. The severity corresponds to the time lost by the sum of all *Late Sender* instances. The time lost by individual *Late Sender*

instances is exported by the `idle_time` attribute of the pattern.

Event traces provide a very fine-grained view of the performance behavior of a parallel application. Based on this view, performance properties that cannot be represented by profiling data can now be specified in terms of compound events. For this reason, ASL has been equipped with the pattern construct, which allows the specification of complex performance properties by means of event traces.

3.10 Summary

The specification of compound events based on event-model enhancement provides a systematic approach to locating inefficiency situations of parallel applications in event traces. The central idea of model enhancement is to combine single events to abstractions that represent dynamic program entities, such as region stacks or messages. The strength of this approach lies in an understandable characterization of performance behavior using the vocabulary of the underlying programming model, which is represented by the abstractions defined during model enhancement.

The abstractions include state sequences, which represent aspects of a program's execution state as sets of events, and pointer attributes, which represent relationships between single events as pairs of related events. It has been shown that model enhancement provides a suitable means to describe complex situations of inefficient behavior in MPI and OpenMP applications. Also, it has been shown that it is even possible to combine the enhanced models for the two programming models into a hybrid model that is able to deal with the concurrent usage of MPI and OpenMP. The next chapter describes the design of an automatic performance tool according to the concept of compound-event specification based on event-model enhancement.

Chapter 4

Analysis of Performance Behavior

This chapter presents the design of an automatic performance tool EXPERT (Extensible Performance Tool) based on compound-event detection in event traces. After a brief outline of EXPERT's overall architecture, the chapter starts with a discussion of instrumentation and event-trace generation. Next, the analysis process is introduced as a transformation of trace data into a property-oriented performance space. After that, it is shown how a separation of compound-event specification from the actual analysis process leads to a modular and extensible tool architecture. A method of pinpointing performance problems and bottlenecks through visualization of that space is then presented, followed by a detailed description of the performance properties supported so far. Finally, the chapter discusses limitations of the current prototype and concludes with some ideas on further improvements.

4.1 Performance Behavior of Coupled SMPs

Combining multiple SMPs into one parallel computer offers the opportunity to build scalable high-performance architectures from standard server components, which need not be designed specifically for the scientific-computing market. However, this more economic way of producing high-performance computers comes at the price of a complex hierarchical architecture consisting of multiple shared-memory nodes distributed across an interconnection network.

Since the architecture exhibits a mix of distributed and shared memory, there are several possible ways of using such a machine. The traditional mapping of programming models and hardware architectures suggests message-passing and shared-memory programming as the models of choice. Most important among those are MPI for message-passing and OpenMP for shared-memory applications. Indeed, one approach to exploit the hybrid hardware design is the concurrent usage of MPI and OpenMP in the same application.

However, as a result of the increased complexity both in hard- and software, performance behavior tends to be more complex because communication among different control flows becomes more intricate. The increase in behavioral complexity creates a need for advanced performance tools that are custom-made to this class of computing environments. In particular, automatic tools are desired in view of the large amount of performance data often produced on such machines and the need to present performance results on a high level of abstraction to allow easy identification of performance problems.

4.2 Overall Architecture

This section contains a description of the different components of the EXPERT performance-analysis environment and explains how they are related to each other. The complete environment is depicted in Figure 4.1. The different tools are represented as boxes with rounded corners and their inputs and outputs are represented as sheets of paper with the upper-right corner turned down. The arrows illustrate the whole performance-analysis process from instrumentation to result presentation.

The EXPERT analysis process is composed of two parts: a semi-automatic multi-level instrumentation of the user application followed by an automatic analysis of the generated performance data. The first subprocess is called semi-automatic because it requires the user to slightly modify the make file. To begin the process, the user supplies the application's source code, written in either C, C++, or Fortran, to OPARI [57, 58], which performs automatic instrumentation of OpenMP constructs and redirection of OpenMP-library calls to instrumented wrapper functions on the source-code level. Instrumentation of user functions is done either on the source-code level using TAU [69, 70] or using a compiler-supplied profiling interface. Instrumentation for MPI events is accomplished with the PMPI [52] wrapper library, which generates MPI-specific events by intercepting calls to MPI functions. All the MPI, OpenMP, and user-function instrumentations call the EPILOG run-time library, which provides mechanisms for event-record buffering and trace-file creation. At the end of the instrumentation process the user has a fully instrumented executable.

Running this executable generates a trace file in the EPILOG format. After program termination, the trace file is fed into the EXPERT analyzer. The analyzer does not operate on the raw trace file, instead the analysis is carried out in terms of the enhanced event model defined in Chapter 3. For this purpose, EXPERT uses EARL [75], which is responsible for mapping the raw trace file onto the enhanced event model. EARL provides a high-level API to event traces and offers random access to events including pointer attributes and states from state sequences. Thus, the calculation of state sequences and pointer attributes is done by EARL. After analysis has been completed, the analyzer generates an analysis report, which serves as input for the EXPERT presenter, the component responsible for analysis-result presentation.

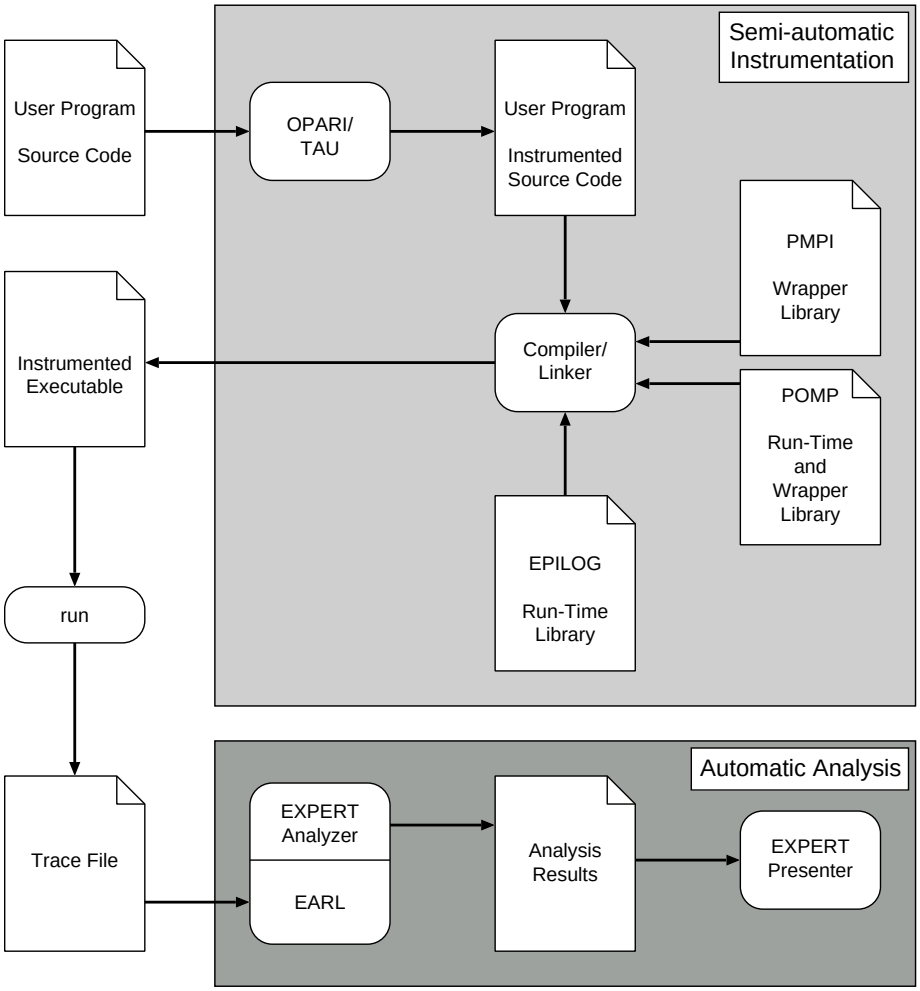

Figure 4.1: EXPERT overall architecture.

Currently, the software necessary to generate event traces has been successfully installed on two parallel computers with SMP nodes: the PC-based ZAMpano [27] and the HITACHI SR8000-F1 [49]. Instrumentation is done on ZAMpano and on the SR8000-F1 with the unpublished profiling interface of the PGI compiler [64] and of the proprietary HITACHI compiler [37], respectively. The analysis components of EXPERT run on Linux.

4.3 Event-Trace Generation

The analysis process relies on event traces as performance-data source because event traces preserve the temporal and spatial relationships among individual events. These relationships are necessary to detect the presence of many interesting performance properties in the target application. Event traces as generated by the instrumented application represent

the program execution on the level of the basic event model, as defined in Chapter 3.

4.3.1 Data Format

The event traces generated in this environment are compliant with the portable EPILOG (Event Processing, Investigating, and Logging) binary trace-data format. The EPILOG format was designed as part of this thesis to represent the basic event model from Section 3.6, that is all event types from the hybrid model but without any pointer attributes.

The EPILOG format is suitable to represent the executions of MPI, OpenMP, or hybrid parallel applications distributed across one or more (possibly large) coupled SMP systems. In addition to coupled SMPs, target systems also can be meta-computing environments as well as more traditional non-coupled or non-SMP systems. It maps events onto their location within the hierarchical hardware as well as onto their process and thread of execution, that is, an event location is described by a tuple (machine, SMP node, process, thread), as required by the model.

Furthermore, EPILOG supports storage of all necessary source-code information in terms of files, regions, and call sites. In addition to a name and a source-code location, which consists of a file name and a range of lines in that file, source-code regions can be distinguished by a region-type attribute that indicates whether a region is a function, a loop, or an OpenMP construct, and if so, which one. Also, each Enter event is able to carry call-site information including a file and a line from where the region was entered.

Although not used in the current prototype of EXPERT, the EPILOG format offers the option of recording hardware performance-counter values for each Enter and Exit event. The predefined semantics of possible performance-counters correspond to the counters provided by PCL [9] (Performance Counter Library). The counters covered by PCL comprise a broad range of common performance counters that provide information on events referring to the memory hierarchy, to instructions, to the status of functional units, and to ratios computed from a combination of multiple counters.

4.3.2 Instrumentation

The instrumentation of a program occurs on multiple levels. The source-to-source translator OPARI performs OpenMP directive and API call transformations on the source-code level to expose OpenMP parallel execution to the EPILOG run-time system. OPARI both inserts calls to the run-time system before and after directives and redirects API calls to wrapper functions generating events related to lock synchronization and API entry and exit.

OPARI supports the POMP [58] monitoring interface that can be implemented by different performance tools to support a variety of performance measurement tasks for OpenMP.

The POMP interface defines the points in the control flow of an OpenMP application that trigger a POMP interface call. The implementation of that interface must be provided by the respective performance tool itself. The POMP library included in EXPERT was the first implementation of the POMP monitoring interface. To be able to trace OpenMP applications, the implementation of EPILOG is thread safe, a necessary precondition not satisfied in many traditional tools.

Instrumentation targeting the interception of user functions occurs either on the source-code level using TAU or during compilation. The compiler, which must provide a profiling interface defining functions called upon function entry and exit as well as program start and termination, inserts calls to that interface into the object code. The interface is implemented by the EPILOG run-time system. However, currently neither TAU nor the PGI and HITACHI compilers support efficient call-site instrumentation, and hence the Enter events carry no call-site information. For this reason, a call path computed by EXPERT may actually represent a set of call paths differing only in call sites (e.g., line number of a function call).

Finally, the last level of instrumentation is performed during linkage of different libraries, which are actually combined into one single library. The PMPI wrapper library, which is based on the MPI standard profiling interface, intercepts MPI calls and redirects them to PMPI functions, while generating MPI-specific events before and after calling the PMPI function. As already mentioned, the POMP run-time system and wrapper library provides an implementation of the POMP interface and transforms calls inserted by OPARI into calls of the EPILOG run-time system to generate OpenMP-specific events. The EPILOG run-time system itself is responsible for event-record creation, buffering, merging of local traces, and post-mortem synchronization of local time stamps, as described in Section 4.3.3.

4.3.3 Clock Synchronization

Not all parallel computers with SMP nodes provide hardware clock synchronization among different SMP nodes. In these cases, their local clocks may vary in offset and drift at a given moment. The EPILOG run-time system addresses this with software synchronization to ensure the correct precedence order of distributed events. This is especially important in view of point-to-point communication to preserve the correct order of messages.

Instead of adjusting the local clocks during run time, EPILOG performs post-mortem synchronization of local time stamps when merging local event traces into a single global event trace. For this reason, EPILOG conducts run-time offset measurements based on the remote clock reading technique [15] once at program start and once at program end. Since the measurement module needs to call MPI functions, the measurements are taken immediately after MPI has been initialized and immediately before MPI is finalized. The synchronization occurs in an asymmetric fashion in that one local master clock provides

the time for the remaining local slave clocks. Each slave sends a request to the master at a time s_1, the master responds to the request by sending its current local time m, and the slave receives the response at a time s_2. The slave time corresponding to the master time m is estimated as:

$$s := s_1 + (s_2 - s_1)/2$$

The offset is computed as the difference $m - s$. To minimize the effect of asymmetric message delays, EPILOG applies a statistical approach that executes a series of message exchanges and takes that one with the minimum difference $s_2 - s_1$, which is assumed to have a minimal and therefore symmetric delay. After program termination, each slave has two pairs (s_s, o_s) and (s_e, o_e) which contain the local time together with the offset to the corresponding master time once for program start and once for program end. The post-mortem synchronization algorithm assumes that all clocks have a constant drift and that they can be described in terms of a linear function based on an initial offset and a constant decline. Based on this model each slave time stamp can be easily mapped onto the corresponding master time:

$$m(s) := s + \frac{(o_e - o_s)}{(s_e - s_s)} * (s - s_s) + o_s$$

To circumvent the effects of external NTP [56] synchronization of local system clocks, the measurements can refer to the cycle counter instead of the system clock, whose drift rate may be adjusted at regular intervals by NTP.

Of course, the assumption that the local clocks are paced with a different but constant drift is only an inaccurate approximation, since the drift may change as a result of temperature variation. Experiments [76] on ZAMpano suggest that the accuracy that can be achieved in this way allows the calculation of the correct precedence of message events for a program run time of at least a few minutes, that is, during this period the deviation lies below the network latency of $15\mu s$. While this proved to be sufficient for the applications presented in Chapter 5, in the absence of a global hardware clock a production tool should rely on alternative software solutions, such as the controlled logical clock [65].

4.4 Analysis Process

The actual trace analysis is done by the EXPERT analyzer. The analysis process is based on the notion of performance properties. A performance property characterizes one aspect of the performance behavior of a program and can be checked by a set of conditions. For every performance property a *severity* measure is provided, whose magnitude specifies the influence of a property on the performance behavior in relation to other properties. In addition, the conditions used to prove the existence of a performance property are associated

with a *confidence* value indicating their reliability. Note that a performance property does not necessarily denote negative, that is, inefficient behavior.

In the context of EXPERT, the presence of a performance property is checked by looking for occurrences of a compound event. The severity is the inefficiency-time fraction associated with that compound event, and the confidence is the reliability of the compound-event specification to match the desired behavior. In EXPERT, the confidence is used to inform the user about the reliability of the analysis process with respect to that property. It is given as a string value (e.g., *max* or *medium*).

The analysis process transforms low-level trace data into a multidimensional performance space consisting of three dimensions: performance property, call path, and location. Performance properties are specified as compound events in terms of the enhanced hybrid event model from Section 3.6. Instead of accessing the event trace directly, the EXPERT analyzer uses EARL to map the event trace onto the enhanced model. The analysis process is then executed with respect to the enhanced-model view. EXPERT determines for each performance property the time spent on a behavior related to that property. EXPERT measures the time separately for each location and call path and inserts the results into a three-dimensional data structure representing the performance space.

The user controls the analyzer, which is implemented in Python [7], either from the command line or using a GUI (*Graphical User Interface*) (Figure 4.2). The GUI allows the selection of particular performance properties for analysis while ignoring the rest.

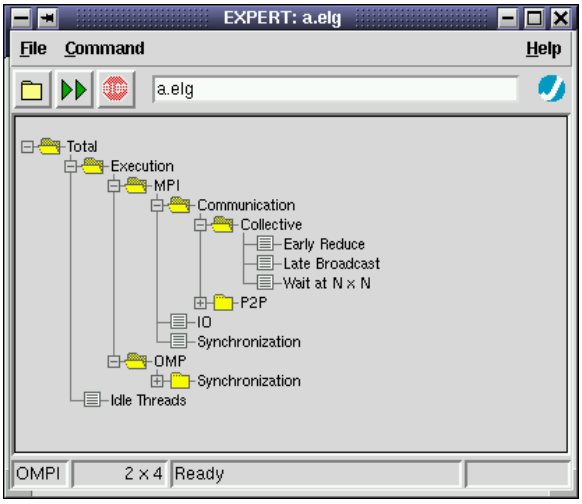

Figure 4.2: GUI of the EXPERT analyzer.

4.4.1 Representation of Performance Behavior

Performance behavior is represented in a three-dimensional performance space with the dimensions of performance property, call path, and location. The first dimension describes

the kind of behavior. The second dimension describes both its source-code location and the execution phase during which it occurs. Finally, the third dimension gives information on the distribution of performance losses across different processes or threads, which allows conclusions to be drawn, for example, on the workload balance.

The performance space only describes the structure in which performance behavior is represented. The actual performance behavior is a mapping that maps each point (property, call path, location) of that space onto a value indicating the extent to which a performance property is present with respect to a call path and a location. This mapping is called *severity* and expresses this extent in terms of the time spent on a particular property while the program was running in a particular call path and at a particular location.

4.4.2 Interval Sets

The run-time events of a parallel application occur on multiple time lines - one for each control flow. Hence, EXPERT describes the severity of a particular performance behavior in terms of wall-clock interval sets that may be distributed across different time lines.

However, as already mentioned, the term location denotes a control flow and not a CPU. To be able to compare wall-clock intervals across different locations in a reasonable fashion, EXPERT requires that different locations never run on the same CPU simultaneously, that is, processes or threads running on the same SMP node do not share a CPU. This requirement comes from thinking of a wall-clock interval used by one location as an interval of that location's resource usage, which should not overlap with another location's usage of the same resource. Therefore, all locations are regarded as being mapped to different CPUs at any time.

Since scheduling policies on most systems are aimed at balancing the work across different CPUs, the requirement can be assumed to be approximately fulfilled if an application does not run more threads than CPUs reserved for that application, that is, an application owns as many CPUs as it runs different control flows (i.e., locations). A more flexible solution could be achieved in the future by integrating events related to CPU acquisition and release to monitor the CPU usage more exactly.[1] The time model applied by EXPERT is based on the assumption of exclusive CPU reservation, that is, the application exclusively owns all CPUs from program start to program termination.

The severity mapping describes the performance behavior in terms of time spent on a particular behavior. The range of the severity is called the CPU-reservation time.

Definition 4.1 (CPU-Reservation Time). The CPU-reservation time A of an event trace $E = \{e_1, \ldots, e_{n_e}\}$ is the Cartesian product of the set L of locations used in E and the

[1] Instrumentation for these events may have to be placed in the operating system.

wall-clock interval from the very first to the very last event of E.

$$A := [e_1.time, e_{n_e}.time] \times L$$

An element of the CPU-reservation time $(t, l) \in A$ is a tuple consisting of a time and a location. ◇

Figure 4.3 shows the different time lines for a hybrid MPI/OpenMP application running two processes with four threads each. The figure shows one time line per location. The whole (spotted) rectangular area constitutes the CPU-reservation time. The dark-gray bars indicate the times when code is executed, whereas the light-gray bars indicate idle slave threads as a result of sequential execution. Note that the processes are launched at slightly different times by the parallel environment.

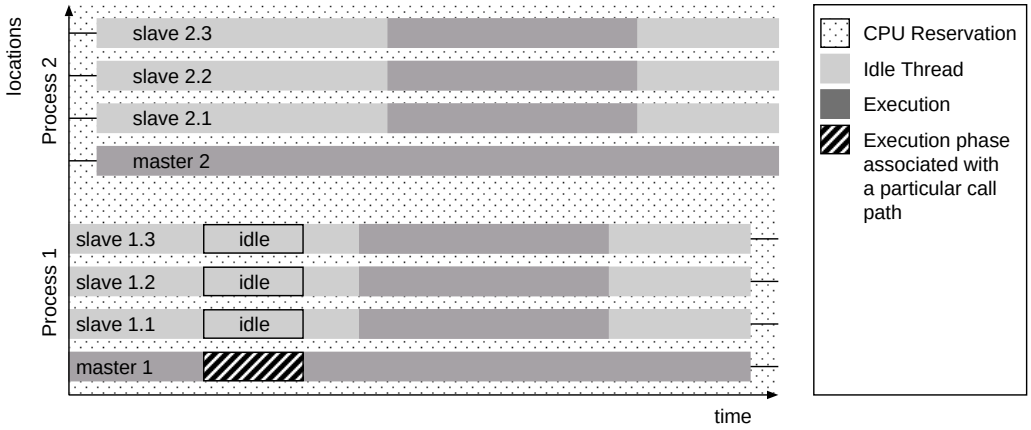

Figure 4.3: Time model of EXPERT.

The severity of a particular performance behavior is described in terms of interval sets spent on that behavior. Interval sets are sets of wall-clock intervals that may be associated with different locations because a behavior may take place at more than one location.

Definition 4.2 (Simple Interval). A *simple interval* $([t_1, t_2], l) \subseteq A$ is a closed interval of A that includes only elements from one location:

$$([t_1, t_2], l) := \{(t, l) \in A| \quad t_1 \leq t \leq t_2\}$$

◇

Definition 4.3 (Non-Overlapping Simple Intervals). Two simple intervals $s_1, s_2 \subseteq A$ are *non-overlapping*, $s_1 \bowtie s_2$, if and only if the interior of their intersection is empty:

$$s_1 \bowtie s_2 \quad \Leftrightarrow \quad \overbrace{s_1 \cup s_2}^{\cdot} = \emptyset$$

◇

A particular behavior may cover multiple simple intervals on different locations. For this reason, the notion of an *interval set* is introduced.

Definition 4.4 (Interval Set). An *interval set* D is the finite union of non-overlapping simple intervals $s_i \subseteq A$:

$$D = \bigcup_{i \in I_D} s_i, \qquad \forall i, j \in I_D : \quad i \neq j \quad \Rightarrow \quad s_i \bowtie s_j$$

I_D denotes the index set of all member intervals.

◇

Note that every union of simple intervals can be rewritten as a union of non-overlapping simple intervals.

Definition 4.5 (Non-Overlapping Interval Sets). Two interval sets $D_1, D_2 \subseteq A$ are *non-overlapping*, $D_1 \bowtie D_2$, if and only if the interior of their intersection is empty:

$$D_1 \bowtie D_2 \quad \Leftrightarrow \quad \overbrace{\overbrace{D_1 \cup D_2}^{\cdot}} = \emptyset$$

◇

Definition 4.6 (Sum of Interval Sets). The sum of two interval sets $D_1, D_2 \subseteq A$ is their union:

$$D_1 \oplus D_2 := D_1 \cup D_2$$

◇

Definition 4.7 (Difference of Interval Sets). The difference of two interval sets $D_1, D_2 \subseteq A$, $D_2 \subseteq D_1$, is the closure of their set difference:

$$D_1 \ominus D_2 := \overline{(D_1 \setminus D_2)}$$

◇

The user of a performance tool may wish to compare multiple interval sets and may therefore be interested in their size (i.e., their amount).

Definition 4.8 (Amount of a Simple Interval). The *amount* $|I|$ of a simple interval $I = ([t_1, t_2], l])$ is the difference of t_2 and t_1:

$$|I| := t_2 - t_1$$

◇

Definition 4.9 (Amount of an Interval Set). The *amount* $|D|$ of an interval set D is the sum of all of its member-interval amounts:

$$|D| := \sum_{i \in I_D} |s_i|$$

◇

Note that Definition 4.9 relies on the definition of interval sets as unions of non-overlapping simple intervals. It follows that the amount of the CPU-reservation time is:

$$|A| = (e_{n_e}.time - e_1.time) * |L|$$

The CPU-reservation time A can be divided based on the dynamic call graph. Each call path n (i.e., node in the call graph (3.5)) defines an interval set during which the program was running exactly in that call path and not in a successor call path (i.e., $\mathrm{cpath}(n) \circ \pi$). To also extend that mapping to the time during which OpenMP slave threads remained idle as a consequence of sequential execution, an interval $([t_1, t_2], slave)$ of idle slave threads (empty boxes in Figure 4.3) is associated with the same call path as the interval $([t_1, t_2], master)$ of the corresponding master thread (see the hatched box in Figure 4.3).

4.4.3 Performance Space

Performance behavior takes place in a property-oriented performance space. The performance space provides a coordinate system in which performance behavior can be represented. It is called "property oriented" because it contains the performance property (i.e., the kind of behavior) as a separate dimension, which allows different behavior to be accommodated in one representation.

Definition 4.10 (Performance Space). Let B be a set of performance properties, E an event trace, L the respective set of locations (Definition 3.17), and $N = N(E) := \{e \in E | \quad e.cnodeptr = e\}$ the set of call paths (i.e., call-path representatives) visited by events contained in E. The *performance space* $\mathcal{P} = \mathcal{P}(B, E)$ is the Cartesian product:

$$\mathcal{P} := B \times N \times L$$

◇

The first two of the dimensions in the performance space are arranged in a tree hierarchy: the performance properties in a hierarchy of general and more specific ones (Figure 4.8), the call paths in a prefix hierarchy. In addition, the locations can be extended to form a hierarchy similar to the property and call-path hierarchies. Each element $l \in L$ denotes a single control flow (i.e., a thread) and, therefore, all locations reside on the same hierarchy

level. However, each thread is associated with a process, an SMP node, and a machine (see Definition 3.17 and Figure 4.4), which constitute groups of locations on different hierarchy levels.

The current prototype of EXPERT supports only tree hierarchies with a single root node. That means there is exactly one root property, one root call path, and one root machine. Also, each node in a hierarchy has a unique parent so that the hierarchies can be conveniently displayed using standard tree browsers. For this reason, in the following, the call graph is referred to as the call tree. Support for multiple machines as part of a heterogeneous environment might be a useful extension in the future and is therefore already integrated in the event-location model (Definition 3.17).

Definition 4.11 (Tree Hierarchy). A *tree hierarchy* on a set H is a binary relation $<$ on H that satisfies the following properties:

$$\forall a \in H: \quad a \not< a \tag{4.1}$$

$$\forall a, b \in H: \quad a < b \quad \Rightarrow \quad b \not< a \tag{4.2}$$

$$\forall a, b, c \in H: \quad a < b \quad \wedge \quad b < c \quad \Rightarrow \quad a < c \tag{4.3}$$

$$\exists r \in H: \forall a \in H: \quad a \neq r \quad \Rightarrow \quad r < a \tag{4.4}$$

$$\forall a, b, c \in H: \quad b < a \quad \wedge \quad c < a \quad \Rightarrow \quad b < c \quad \vee \quad c < b \quad \vee \quad c = b \tag{4.5}$$

Let $s, p \in H$:

$$root(H) := r$$

$$ischild(s,p) := \begin{cases} true & \text{if} \quad p < s \quad \wedge \quad \nexists a \in H: \quad p < a \quad \wedge \quad a < s \\ false & \text{else} \end{cases}$$

$$children(p) := \{s \in H | \quad ischild(s,p)\}$$

◇

Thus, a tree hierarchy $<$ is a strict partial order (4.1 - 4.3) with one least element $r = root(H)$ (4.4). In addition, each element has a unique path to the root, which implies that the associated tree graph does not contain any cycles (4.5).

The set of performance properties is organized in a generalization-specialization hierarchy (i.e., general $<$ specific), which is depicted in Figure 4.8. The hierarchy has been explicitly established by specifying a parent for each of the performance properties except for the root property. Since it seems natural that a more specific behavior only takes place when a more general behavior takes place as well, the interval set associated with the more general behavior contains the interval set associated with the more specific behavior as

a subset. This relationship is a precondition for placement of a property as a child of another property. Note that in general, however, the subset relationship would allow the arrangement of properties in a fashion violating Condition (4.5).

Next, the set of call paths is organized in a prefix hierarchy. A call path $a \in N$ is smaller than a call path $d \in N$ (i.e., $a < d$) if and only if a is a true prefix of d:

$$a < d \quad \Leftrightarrow \quad \text{cpath}(a) \circ (r_1, c_1) \circ \ldots \circ (r_n, c_n) = \text{cpath}(d), \qquad n \geq 1 \tag{4.6}$$

In contrast to the latter two hierarchies, the location hierarchy exists only in an implicit manner, since all locations represent threads and thus belong to the same level. The other hierarchy levels (i.e., process, SMP node, and machine), as introduced in Definition 3.17, can only be derived by aggregation. To make the location hierarchy explicit, the set of locations L can be extended to represent the whole hierarchy.

Definition 4.12 (Hierarchical Location). A *hierarchical location* $\hat{l} \in \hat{L}$ is either a plain location $l \in L$ or it is an aggregate of plain locations.

$$\begin{array}{lllll} \hat{L} = & L & & \cup & \text{(i.e., threads)} \\ & \{(m,s,p) & \mid \exists (m,s,p,t) \in L\} & \cup & \text{(i.e., processes)} \\ & \{(m,s) & \mid \exists (m,s,p,t) \in L\} & \cup & \text{(i.e., SMP nodes)} \\ & \{(m) & \mid \exists (m,s,p,t) \in L\} & \cup & \text{(i.e., machine)} \end{array}$$

◇

$\hat{L}$ denotes the extended location set including the upper hierarchy levels, as depicted in Figure 4.4. Each node in the hierarchy corresponds to one element of $\hat{L}$. The upper levels of the hierarchy are aggregates of their children, they do not represent independent locations of single control flows.

Since EXPERT requires each hierarchy to have exactly one root element, the last part of the union in Definition 4.12 contains exactly one element. A formal characterization of the obvious tree hierarchy can, similar to call paths (4.6), be based on a prefix criterion. After extending the set of locations, it seems natural to also extend the performance space to cover the whole location hierarchy.

Definition 4.13 (Extended Performance Space). Let B be a set of performance properties, E an event trace, $\hat{L}$ the respective set of hierarchical locations (Definition 4.12), and $N = N(E) := \{e \in E | \quad e.cnodeptr = e\}$ the set of call paths (i.e., call-path representatives) visited by events contained in E. The *extended performance space* $\hat{\mathcal{P}} = \hat{\mathcal{P}}(B, E)$ is the Cartesian product:

$$\hat{\mathcal{P}} := B \times N \times \hat{L}$$

◇

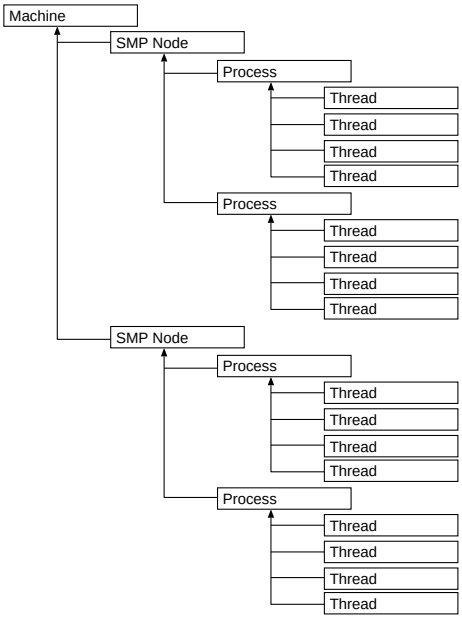

Figure 4.4: EXPERT's location hierarchy.

Severity

The EXPERT analyzer represents the performance behavior contained in an event trace as a mapping that maps each point in performance space onto an interval set within the CPU-reservation time.

Definition 4.14 (Severity). The *severity* of an event trace E with respect to a set of performance properties B is a mapping $sev()$ that maps each point in the performance space $\mathcal{P} = \mathcal{P}(B, E)$ onto an interval set within the CPU-reservation time A:

$$\begin{array}{rrcl} sev: & \mathcal{P} & \rightarrow & 2^A \\ & (b, n, l) & \mapsto & D \end{array}$$

(b, n, l) is mapped onto the interval set D spent on behavior associated with property b while the program was running in call path n at location l. D includes the time spent on behavior associated with more specific properties as a subset, and does not include any time that has been spent on call paths other than n:

$$\forall b_1, b_2 \in B: \quad b_1 < b_2 \quad \Rightarrow \quad sev(b_2, n, l) \subseteq sev(b_1, n_1, l) \tag{4.7}$$

$$\forall n_1, n_2 \in N: \quad n_1 \neq n_2 \quad \Rightarrow \quad sev(b, n_1, l) \bowtie sev(b, n_2, l) \tag{4.8}$$

Also, D contains only intervals occurring at location l:

$$sev(b, n, l) \subseteq ([e_1.time, e_{n_e}.time], l) \tag{4.9}$$

$\diamond$

Condition (4.8) implies that D, in particular, does not include any time that has been spent on call paths reached from n, that is, any call paths m with $n < m$. Also, the interval set associated with one point in the performance space contains only simple intervals from the same location l. However, interval sets covering multiple locations will be discussed when dealing with hierarchical locations.

Since the storage of interval sets covering a large number of member intervals has space requirements that come close to those of event traces, EXPERT computes only the amounts of these interval sets and not the interval sets themselves. However, as will be shown later, the restriction to amounts imposes constraints on the shape of these intervals.

Definition 4.15 (Extended Severity). The *extended severity* $\widehat{sev}()$ of an event trace E with respect to a set of performance properties B is the original severity function (i.e., $sev()$) with an extended domain covering the extended performance space $\hat{\mathcal{P}} = \hat{\mathcal{P}}(B, E)$:

$$\begin{array}{rccl} \widehat{sev}: & \widehat{\mathcal{P}} & \rightarrow & 2^A \\ & (b, n, \hat{l}) & \mapsto & \begin{cases} sev(b, n, \hat{l}) & \text{if} \quad \hat{l} \in L \\ \emptyset & else \end{cases} \end{array}$$

◇

Similar to the call-path hierarchy, the severity of an aggregate location is defined without covering any child nodes in the location hierarchy. However, since a location aggregate naturally does not have any severity not coming from any of its children, the extended severity assigns the empty set to all aggregates in the location hierarchy.

Inclusion and Exclusion

The extended severity function provides arguments with *inclusive* semantics and arguments with *exclusive* semantics. That is, the severity either covers an argument including all its children in the hierarchy or excluding all its children.

The property is interpreted using inclusive semantics (4.7), that is, the interval set associated with a property contains the interval sets of all its specializations as a subset. As opposed to properties, the call path and the location are interpreted using exclusive semantics. The severity of a call path n never includes a successor call path m with $n < m$ (4.8) and the location only delivers a non-empty interval if it is not an aggregate (Definition 4.15).

However, the user who is aware of the hierarchical organization of the performance space may wish to explore the severity of one hierarchy element in relation to other hierarchy elements in a more flexible way. For example, the user may wish to know to what extent the severity of a performance property is not contained in its specializations or may be interested in the sum of the severity of all threads belonging to a given process.

To answer these questions it is necessary to know the severity of a hierarchy element with the semantics of choice. That is, the severity function should be able to deliver both the inclusive and exclusive severity of a hierarchy element.

Definition 4.16 (Variable Severity). The *variable severity* $\widehat{sev}^{(x,y,z)}()$ of an event trace E with respect to a set of performance properties B is an extended severity mapping that interprets its kth argument either with exclusive or inclusive semantics depending on whether the tuple (x, y, z) carries an **e** or **i** in its kth position:

$$\begin{array}{rccc} \widehat{sev}^{(x,y,z)}: & \widehat{\mathcal{P}} & \rightarrow & 2^A \\ & (b,n,l) & \mapsto & D \end{array}$$

The variable severity is derived from the extended severity (Definition 4.15):

$$\widehat{sev}^{(\mathbf{e},y,z)}(b,n,l) := \widehat{sev}^{(\mathbf{i},y,z)}(b,n,l) \ominus \bigoplus_{c \in children(b)} \widehat{sev}^{(\mathbf{i},y,z)}(c,n,l) \tag{4.10}$$

$$\widehat{sev}^{(\mathbf{i},y,z)}(b,n,l) := \widehat{sev}^{(y,z)}(b,n,l)$$

$$\widehat{sev}^{(\mathbf{e},z)}(b,n,l) := \widehat{sev}^{(z)}(b,n,l)$$

$$\widehat{sev}^{(\mathbf{i},z)}(b,n,l) := \widehat{sev}^{(\mathbf{e},z)}(b,n,l) \oplus \bigoplus_{c \in children(n)} \widehat{sev}^{(\mathbf{i},z)}(b,c,l) \tag{4.11}$$

$$\widehat{sev}^{(\mathbf{e})}(b,n,l) := \widehat{sev}(b,n,l)$$

$$\widehat{sev}^{(\mathbf{i})}(b,n,l) := \widehat{sev}^{(\mathbf{e})}(b,n,l) \oplus \bigoplus_{c \in children(l)} \widehat{sev}^{(\mathbf{i})}(b,n,c) \tag{4.12}$$

$\diamond$

Note that the variable severity supports the characterization of performance problems and bottlenecks as subsets of the performance space by computing, for example, the severity of subtrees in the call-path hierarchy.

As already mentioned, to circumvent the storage of highly dispersed interval sets, EXPERT computes only $|sev(\mathcal{P})|$, extends it to $|\widehat{sev}(\widehat{\mathcal{P}})|$, and derives $|\widehat{sev}^{(x,y,z)}(\widehat{\mathcal{P}})|$ from it. However, computing $|\widehat{sev}^{xyz}(\widehat{\mathcal{P}})|$ requires computing the amount of interval-set expressions based on the amount of the operands involved. Unfortunately, this is only possible if the operands are non-overlapping in the case of an addition of interval sets or if the subtrahend is a subset of the minuend in the case of a subtraction of interval sets. Let $C, D \in 2^A$ be interval sets:

$$\begin{array}{lcl} |C \oplus D| = |C| + |D| & \Leftrightarrow & C \bowtie D \\ |C \ominus D| = |C| - |D| & \Leftrightarrow & D \subseteq C \end{array}$$

For this reason, it is necessary to postulate that the interval sets to be added in Equation (4.10), (4.11), and (4.12) are non-overlapping and that both operands of the interval-set subtraction in Equation (4.10) are connected by a subset relationship. Let b be a performance property, n a call path, and l a location:

$$\forall b_1, b_2 \in children(b) : b_1 \neq b_2 \Rightarrow \widehat{sev}^{(\mathbf{i},y,z)}(b_1, n, l) \bowtie \widehat{\mathrm{sev}}^{(\mathbf{i},y,z)}(b_2, n, l) \quad (4.13)$$

$$\forall n_1, n_2 \in children(n) : n_1 \neq n_2 \Rightarrow \widehat{sev}^{(\mathbf{i},z)}(b, n_1, l) \bowtie \widehat{\mathrm{sev}}^{(\mathbf{i},z)}(b, n_2, l) \quad (4.14)$$

$$\forall l_1, l_2 \in children(l) : l_1 \neq l_2 \Rightarrow \widehat{sev}^{(\mathbf{i})}(b, n, l_1) \bowtie \widehat{sev}^{(\mathbf{i})}(b, n, l_2) \quad (4.15)$$

$$\forall c \in children(b) : \widehat{sev}^{(\mathbf{i},y,z)}(c, n, l) \subseteq \widehat{sev}^{(\mathbf{i},y,z)}(b, n, l) \quad (4.16)$$

Conditions (4.14) and (4.15) are trivially satisfied as a consequence of (4.8) and (4.9). Similarly, Condition (4.16) follows from (4.7). Only Condition (4.13) must be ensured by carefully defining the performance properties such that the children of a property always have a non-overlapping severity. Note that this imposes a restriction on property coverage, which may be subject to improvement in a future version of EXPERT.

With Conditions (4.13 - 4.16), it is possible to compute $|\widehat{sev}^{(x,y,z)}(\widehat{\mathcal{P}}))|$ solely based on $|\widehat{sev}(\mathcal{P}))|$ and thus on $|sev(\mathcal{P}))|$:

$$\begin{aligned} |\widehat{sev}^{(\mathbf{e},y,z)}(b,n,l)| &= |\widehat{sev}^{(\mathbf{i},y,z)}(b,n,l)| - \sum_{c \in children(b)} |\widehat{sev}^{(\mathbf{i},y,z)}(c,n,l)| \\ |\widehat{sev}^{(\mathbf{i},y,z)}(b,n,l)| &= |\widehat{sev}^{(y,z)}(b,n,l)| \end{aligned}$$

$$\begin{aligned} |\widehat{sev}^{(\mathbf{e},z)}(b,n,l)| &= |\widehat{sev}^{(z)}(b,n,l)| \\ |\widehat{sev}^{(\mathbf{i},z)}(b,n,l)| &= |\widehat{sev}^{(\mathbf{e},z)}(b,n,l)| + \sum_{c \in children(n)} |\widehat{sev}^{(\mathbf{i},z)}(b,c,l)| \end{aligned}$$

$$\begin{aligned} |\widehat{sev}^{(\mathbf{e})}(b,n,l)| &= |\widehat{sev}(b,n,l)| \\ |\widehat{sev}^{(\mathbf{i})}(b,n,l)| &= |\widehat{sev}^{(\mathbf{e})}(b,n,l)| + \sum_{c \in children(l)} |\widehat{sev}^{(\mathbf{i})}(b,n,c)| \end{aligned}$$

Summary

EXPERT represents the performance behavior as the mapping $|\widehat{\mathrm{sev}}^{(x,y,z)}(\widehat{\mathcal{P}})|$ that maps a point in the performance space defined by the property, call path, and location coordinates, onto the amount of a CPU-reservation time interval set. The semantics of each coordinate may be either inclusive or exclusive depending on what the user would like to know.

The analysis process is an automatic transformation of an event trace into this representation and is performed in three steps. First, it computes the amount of the simple severity (Definition 4.14), extends it to include hierarchical locations (Definition 4.15), and finally adds inclusive and exclusive semantics for each coordinate of the performance space (Definition 4.16). Note that the structure of the performance space also depends on both the event trace and the set of performance properties, which is independent of the event trace, but which may be changed or extended, as will be explained later.

$$E, B \xrightarrow{(1)} |sev(\mathcal{P}))| \xrightarrow{(2)} |\widehat{sev}(\widehat{\mathcal{P}}))| \xrightarrow{(3)} |\widehat{sev}^{(x,y,z)}(\widehat{\mathcal{P}}))|$$

Note that most of the values obtained as a result of step (3) need not be precomputed by the analyzer. Instead, they can be computed on demand by the presenter.

4.4.4 EARL

EARL (Event Analysis and Recognition Language) is a class library [2] that offers a high-level interface to an EPILOG event trace. The interface provides random access to all events including the state sequences and pointer attributes defined in the enhanced event model (except for the auxiliary ones). To give access to state sequences and pointer attributes, EARL performs the calculations described in Chapter 3. In addition, EARL provides access methods to obtain static information on the event trace, such as the number and kind of event locations, and information on source-code entities, such as regions and files. Also, it gives the user the ability to traverse the dynamic call tree.

The initial prototype of EARL [75] developed earlier by the author provided only limited assistance in the analysis of MPI applications. The current version contained in EXPERT is the result of a substantial redesign and many enhancements, including support for MPI collective communication, OpenMP, hybrid programming, and a method of associating events with the dynamic call path.

EARL is implemented as a C++ class, whose interface is embedded in the Python [7] scripting language. The Python binding has been automatically generated using SWIG [6]. The class representing an event trace `EventTrace` allows random (read) access to events by supplying the event position as a parameter. The event position of an event e_i is just the index i. The access method `event()` returns for each event a Python dictionary consisting of a set of key-value pairs, where the keys represent the attribute names and the values represent the attribute contents. Thus, the attribute value can be obtained using the attribute name as the key (e.g., `e[loc]`). Whereas pointer attributes present themselves to the user the same way as regular attributes do, state sequences are implemented as methods of `EventTrace` that take an event position as an argument.

[2] Since it provides a notation to specify compound events, it has been named a language.

States sequences and pointer attributes provide events only as references expressed in terms of event positions. Thus, the state-sequence method `stack()` (i.e., the region stack) always returns a set of event positions and a pointer attribute `e[sendptr]` always contains a single event position as a value.

To provide efficient random access to events and, in particular, to support an efficient search for compound events by sequentially traversing the event trace from the beginning to the end, EARL uses two different buffer mechanisms: the history buffer and the bookmark buffer. However, before delving into the details of both mechanisms, the general event-access mechanism is explained.

When an event e_i is accessed, all its abstractions, that is, the overall state $\vec{\mathfrak{S}}_i$ and all pointer attributes $e_i.ptr$ need to be computed. Recall that the working-set requirement from Section 3.3.3 requires that each state $\mathfrak{S}_i$ and each pointer attribute $e_i.ptr$ can be computed solely on the basis of the overall state $\vec{\mathfrak{S}}_{i-1}$ and on e_i itself, that is, on the basis of the working set Δ_i. From the inductive definition of state sequences and pointer attributes it follows that to access an event e_i, EARL needs to take an overall state $\vec{\mathfrak{S}}_{k<i}$ it knows and all events from e_{k+1} to e_i to compute the abstractions related to e_i. In the worst case, $\vec{\mathfrak{S}}_k = \vec{\mathfrak{S}}_0 = \{\emptyset, \ldots, \emptyset\}$, that is, EARL does not know any overall state prior to e_i, which requires reading the event trace from the very beginning in order to compute e_i and its abstractions.

However, while traversing the trace file to compute all abstractions related to e_i, EARL computes $\vec{\mathfrak{S}}_j$ for each event e_j it reads on this way as a side effect. To utilize the work done for one event access, EARL is able to remember $\vec{\mathfrak{S}}_j$ at regular intervals together with the corresponding trace-file position and to store it in a buffer so that subsequent reading can start at the closest $\vec{\mathfrak{S}}_j$ in the buffer. In analogy to a bookmark used to remember a page in a book, this mechanism is called the bookmark buffer. Note that the distance between bookmarks, which can be changed by the user, must be chosen carefully due to potential memory requirements.

Whereas the bookmark buffer accelerates subsequent event accesses by avoiding the necessity of reading the trace file from the beginning, expensive file accesses may still occur. For this reason, the history buffer accelerates accesses to events and abstractions within the recent past (i.e., the history) of an event that has just been accessed. When the access to event e_i causes successive file accesses, EARL remembers a small window of events in conjunction with the overall state prior to the first event in the window. That is, after reaching e_i, EARL keeps in its history buffer $\vec{\mathfrak{S}}_{i-s-1}$ and $\{e_{i-s}, \ldots, e_i\}$, where $s+1$ is the size of the window. Thus, it is possible to get all events including states and pointer attributes from the history window without any file accesses.

To minimize the space requirements of both buffer mechanisms, EARL ensures that each event is stored only once even in the case of two stored $\vec{\mathfrak{S}}_i$ and $\vec{\mathfrak{S}}_j$ with $i \neq j$ and $\Gamma_i \cap \Gamma_j \neq \emptyset$. Also, to save memory the dynamic call graph is implemented slightly different from Section 3.6.3 in that an `EventTrace` object maintains only one call graph, which reflects

the most recent event that has been read, whereas other states can be obtained for arbitrary events. Recall that the state $\mathfrak{D}$ representing the call graph never shrinks in size. In addition, the call graph is internally represented as a tree, which decreases the time necessary to locate the successor node of the current node.

4.4.5 Pattern Classes

The analyzer's design follows a layered approach (Figure 4.5). Its architecture is based on the idea of separating the analysis process from the performance-property specifications. The analyzer operates on a set of property specifications, which adhere to a common interface that is independent of the actual property semantics. In addition, the design establishes a further layer by separating the property specifications from frequently used abstractions (i.e., state sequences and pointer attributes), which are accessible through the EARL class interface.

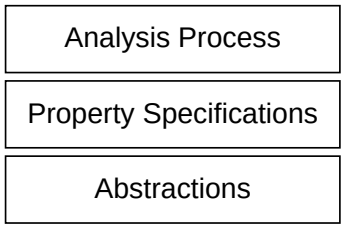

Figure 4.5: The layered design of the EXPERT analyzer.

Performance properties are specified as Python classes whose interface is defined in a common base class `Pattern` (Figure 4.6). Hence, as long as the classes fulfill the common interface, the analyzer is able to handle an arbitrary set of patterns. Pattern classes represent compound events to be matched against the event trace and are implemented using the EARL language.

The `parent()` method of the `Pattern` base class is intended to express the hierarchical organization of performance properties in that all descendant implementations should return their parent's name in the hierarchy, which is not necessarily the parent in the inheritance hierarchy. Whereas the property hierarchy is used to express an inclusion relationship with respect to the severity (4.7), the inheritance hierarchy is used to give all pattern classes a uniform interface. The `confidence()` method should return the confidence of the assumption made by a successful pattern match about the occurrence of a performance property. The default confidence is maximum confidence. Note that the confidence is always the same for a given pattern class and does not refer to the characteristics of a specific match. The `severity()` method is invoked after the analysis has been finished and should return the severity restricted to the property represented by that class, that is, a matrix representing $|sev(b, n, l)|$ for a variable call path $n \in N$ and a variable location $l \in L$ while the property b is fixed. Finally, the base class includes a method `configure()`

```
class Pattern:

    [...]

    # return name of the parent property
    def parent(self):
        return None

    # return the confidence
    def confidence(self):
        return 'max'

    # return the severity matrix for this property
    def severity(self):
        return self._severity

   # launch a configuration dialog
    def configure(self, parent):
        pass
```

Figure 4.6: Python definition of the base class `Pattern`.

to launch a configuration dialog for the input of pattern-specific parameters prior to event-trace analysis.

The analysis process follows an event-driven approach according to Algorithm 3.1. EXPERT walks sequentially through the event trace and for each single event invokes call-back methods of the pattern instances and supplies the event as an argument. A pattern can provide a different call-back method for each event type. Every time EXPERT encounters an event of type t, it invokes the call-back methods for type t of all pattern instances that provide one for type t.

A pattern class looking for occurrences of a compound event will provide at least one call-back method for the root event. Then it tries to instantiate that compound event starting from the root event, which is supplied as an argument. During this process the pattern instance may follow links (i.e., pointer attributes) or investigate states from state sequences. After completion of the analysis process, the analyzer knows $|sev(b, n, l)|$ for all combinations of a property b, a call path n, and a location l and writes it to a file, which is used as input for the EXPERT presenter.

Note that a pattern class may provide more than one call-back method, which allows a property implementation to be more flexible than is suggested by Definition 3.27. For example, a pattern class may collect additional state information beyond that provided by predefined state sequences.

Figure 4.7 exemplifies the concept of implementing performance properties as classes by means of the late-sender property (Example 3.1). The corresponding pattern class returns

```
class LateSender(Pattern):

    "Late Sender"

    def parent(self):
        return "P2P"

    def recv(self, root):
        e_1 = self._trace.event(root['enterptr'])
        if (self._trace.region(e_1['regid'])['name'] == "MPI_Recv"):
            s_1 = self._trace.event(root['sendptr'])
            e_2 = self._trace.event(s_1['enterptr'])
            if (self._trace.region(e_2['regid'])['name'] == "MPI_Send"):
                idle_time =  e_2['time'] - e_1['time']
                if idle_time > 0 :
                    locid  = e_2['locid']
                    cnode  = e_2['cnodeptr']
                    self._severity.add(cnode, locid, idle_time)
```

Figure 4.7: Python class definition of the *Late Sender* compound event.

`P2P` as its parent because the behavior specified by that class is a specialization of point-to-point communication.

Every time the EXPERT analyzer encounters a Receive event, the `recv()` method is invoked on the pattern instance and a dictionary containing the Receive event is passed as the `root` argument. The pattern first tries to locate the Enter event `e_1` of the enclosing region instance by following the `root[enterptr]` attribute. After verifying that this region instance is an MPI_Recv, the corresponding Send event is determined by tracing back the `root[sendptr]` attribute. Now, the pattern looks for the Enter event `e_2` of the region instance from which the message originated by following the `s_1[enterptr]` attribute. The analyzer then checks whether the region instance from where the message has been sent is an MPI_Send.

After locating all constituents, the chronological difference between the two Enter events `e_1` and `e_2` is computed. Since the MPI_Recv has to be posted earlier than the MPI_Send, the `idle_time` has to be greater than zero. If that is true, the measured idle time is added to the severity-matrix cell defined by the location and call path of `e_2` according to Section 3.8.3. After the analysis has been finished, each matrix cell contains the sum of all idle times introduced by the *Late Sender* situation.

4.4.6 Performance Properties

Figure 4.8 shows the hierarchy of predefined performance properties that are supported by the current prototype of EXPERT. The set should not be regarded as complete, but it is

representative in that it shows the usefulness and feasibility of the analysis method and the advantages of the tool architecture used to implement it.

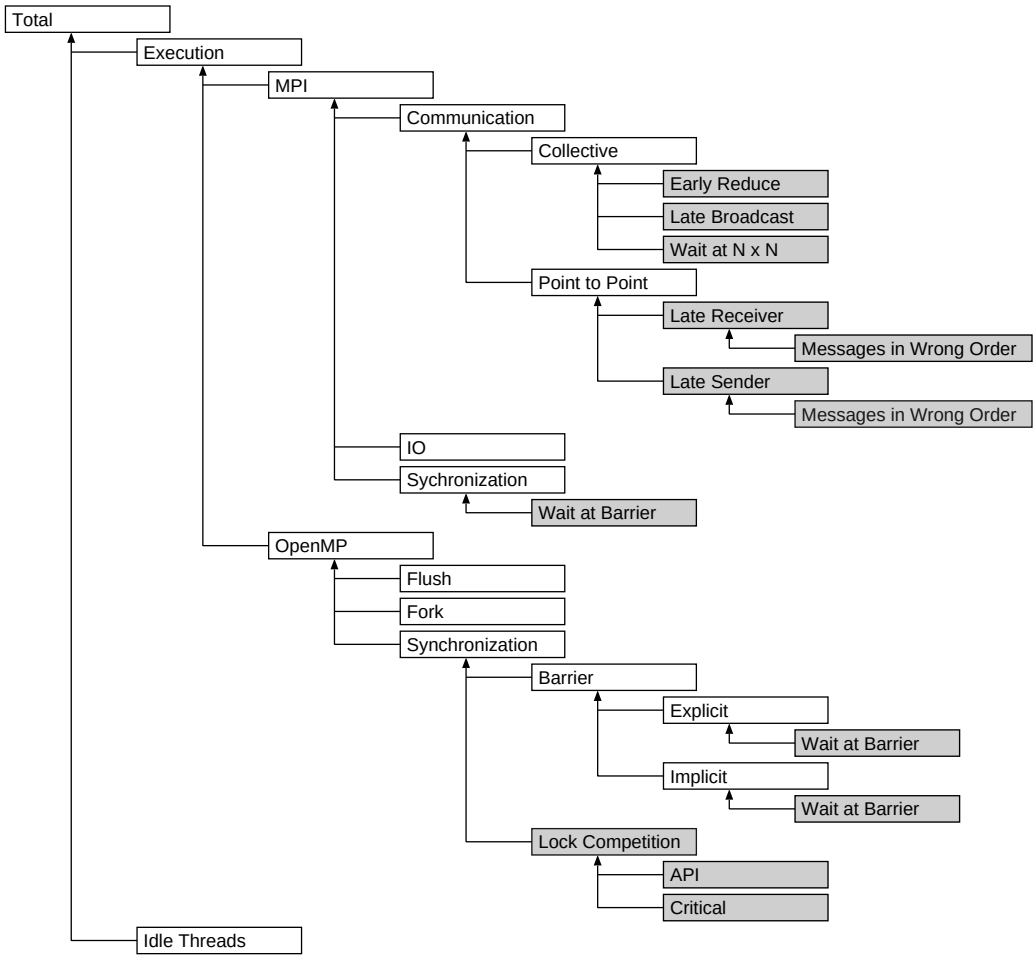

Figure 4.8: EXPERT's hierarchy of predefined properties.

The set of performance properties is split into two parts. The first part, which constitutes the upper layers of the hierarchy and which is indicated by white boxes, is mainly based on summary information involving, for example, the total execution times of special MPI routines, which could also be provided by a profiling tool. However, the second part, which constitutes the lower layers of the hierarchy and which is indicated by gray boxes, involves idle times that can only be determined by comparing the chronological relation between individual events. This is where the compound-event approach can demonstrate its full power. A major advantage of EXPERT lies in its ability to handle both groups of performance properties in one step. A detailed discussion of the most interesting situations of this second kind can be found in Section 3.8. Moreover, a way to extend the predefined set by adding custom-made property specifications is presented in Section 4.4.7. The following briefly explains the performance properties that are currently implemented in EXPERT.

Total. Time spent on program execution including the idle times of slave threads during OpenMP sequential execution. It corresponds to the intervals covered by the light-gray and dark-gray bars in Figure 4.3. When a master thread is executed in an interval $([t_1, t_2], master)$ during a sequential period, all its slaves are assumed to execute in $([t_1, t_2], slave)$ and to pass through the same call paths as the master does.

Execution. Time spent on program execution but without the idle times of slave threads during OpenMP sequential execution. It corresponds to the intervals covered by the dark-gray bars in Figure 4.3.

MPI. Time spent on MPI API calls.

Communication. Time spent on MPI API calls used for communication.

Collective. Time spent on collective communication.

Early Reduce. Collective communication operations that send data from all processes to one destination process (i.e., n-to-1) may suffer from waiting times if the destination process enters the operation earlier than its sending counterparts, that is, before any data could have been sent. The property refers to the time lost as a result of that situation.

Late Broadcast. Collective communication operations that send data from one source process to all processes (i.e., 1-to-n) may suffer from waiting times if destination processes enter the operation earlier than the source process, that is, before any data could have been sent. The property refers to the time lost as a result of that situation.

Wait at N $\times$ N. This property corresponds to the situation of Example 3.4 (p. 68). Collective communication operations that send data from all processes to all processes (i.e., n-to-n) exhibit an inherent synchronization among all participants, that is, no process can finish the operation until the last process has started. The time until all processes have entered the operation is measured and used to compute the severity.

Point to Point. Time spent on point-to-point communication.

Late Receiver. This property corresponds to the situation of Example 3.2 (p. 66). A send operation is blocked until the corresponding receive operation is called. This can happen for several reasons. Either the MPI implementation is working in synchronous mode by default or the size of the message to be sent exceeds the available MPI-internal buffer space and the operation is blocked until the data is transferred to the receiver.

Messages in Wrong Order (Late Receiver). A *Late Receiver* situation may be the result of messages that are sent in the wrong order. If a process sends messages to processes that are not ready to receive them, the sender's MPI-internal buffer may overflow so that from then on the process needs to send in synchronous mode causing a *Late*

Receiver situation. The detection of messages that have been sent in the wrong order is discussed in Example 3.3 (p. 67).

Late Sender. This property corresponds to the situation of Example 3.1 (p. 65). It refers to the time wasted when a call to a blocking receive operation (e.g, MPI_Recv or MPI_Wait) is posted before the corresponding send operation has been started.

Messages in Wrong Order (Late Sender). A *Late Sender* situation may be the result of messages that are received in the wrong order. If a process expects messages from one or more processes in a certain order while these processes are sending them in a different order, the receiver may need to wait longer for a message because this message may be sent later while messages sent earlier are ready to be received. The detection of messages that have been sent in the wrong order is discussed in Example 3.3 (p. 67).

IO (MPI). Time spent on MPI file IO.

Synchronization (MPI). Time spent on MPI barrier synchronization.

Wait at Barrier (MPI) This property is similar to the situation of Example 3.4 (p. 68). It covers the time spent on waiting in front of an MPI barrier. The time until all processes have entered the barrier is measured and used to compute the severity.

OpenMP. Time spent on the OpenMP run-time system.

Flush (OpenMP). Time spent on flush directives.

Fork (OpenMP). Time spent by the master thread on team creation.

Synchronization (OpenMP). Time spent on OpenMP barrier or lock synchronization. Lock synchronization may be accomplished using either API calls or critical sections.

Barrier (OpenMP). The time spent on implicit (compiler-generated) or explicit (user-specified) OpenMP barrier synchronization. As already mentioned, implicit barriers are treated similar to explicit ones. The instrumentation procedure replaces an implicit barrier with an explicit barrier enclosed by the parallel construct. This is done by adding a nowait clause and a barrier directive as the last statement of the parallel construct. In cases where the implicit barrier cannot be removed (i.e., parallel region), the explicit barrier is executed in front of the implicit barrier, which will be negligible because the team will already be synchronized when reaching it. The synthetic explicit barrier appears in the display as a special implicit barrier construct.

Explicit (OpenMP). Time spent on explicit OpenMP barriers.

Implicit (OpenMP). Time spent on implicit OpenMP barriers.

Wait at Barrier (Explicit). This property corresponds to the situation of Example 3.5 (p. 69). It covers the time spent on waiting in front of an explicit (user-specified) OpenMP barrier. The time until all processes have entered the barrier is measured and used to compute the severity.

Wait at Barrier (Implicit). This property corresponds to the situation of Example 3.5 (p. 69). It covers the time spent on waiting in front of an implicit (compiler-generated) OpenMP barrier. The time until all processes have entered the barrier is measured and used to compute the severity.

Lock Competition (OpenMP). This property corresponds to the situation of Example 3.6 (p. 70). It refers to the time a thread spent on waiting for a lock that had been previously acquired by another thread.

API (OpenMP). Lock competition caused by OpenMP API calls.

Critical (OpenMP). Lock competition caused by critical sections.

Idle Threads. Idle times caused by sequential execution before or after an OpenMP parallel region. It corresponds to the intervals covered by the light-gray bars in Figure 4.3.

4.4.7 Extensibility Mechanism

EXPERT provides a large set of built-in performance properties covering the most frequent inefficiency situations. But sometimes the user may wish to consider application-specific metrics, such as iterations or updates per second. In this case, the user can simply write another pattern class that implements a custom-made application-specific performance property. Of course, the new property must adhere to the common interface defined by base class `Pattern`. After that, the user can place it into the module where the other patterns are located.

At startup time, EXPERT dynamically queries the module's name space and looks through all of the module's pattern classes including the newly inserted ones, from which it is now able to build instances. The new patterns are integrated into the graphical user interface and can be used like the predefined ones. Note that this mechanism relies on the Python module concept, which allows a module's namespace to be searched at run time.

However, as already mentioned, the placement of new properties in the hierarchy or the modification of existing ones must satisfy the constraints from Section 4.4.3. That means, the severity of a property must always be a subset of the severity of its parent property (4.7) and the severity of sibling properties must always be non-overlapping (4.13), which may complicate the design of new properties in that new properties are limited in their potential coverage of CPU-reservation time.

4.5 Visualization of Performance Behavior

Pancake [63] regards human factors as the reasons for the limited acceptance of performance tools among program developers. Often tools are “hard to learn” and “too complex to use”. Influenced by the way developers approach the task of performance analysis, she identifies key requirements for performance tools that should be met to increase their usability: support for exploring the total performance space, support for comparing different aspects of program behavior, and support for navigating through complex source-code hierarchies. The design goal of the EXPERT presenter was the accommodation of all these three features in a simple but powerful display based on a uniform multi-dimensional hierarchical organization and ranking of different items using colors.

The user can interactively access each of the hierarchies constituting a dimension of the performance space using *weighted trees*. A weighted tree is a tree browser that labels each node with a weight. EXPERT uses the severity amount associated with that node as a weight. To simplify comparison of different weights, the weight is written as a percentage of the CPU-reservation time. The weight that is actually displayed depends on the state of the node, that is, whether it is expanded or collapsed. The weight of a collapsed node represents the whole subtree associated with that node, whereas the weight of an expanded node represents only the fraction that is not covered by its descendants because the weights of its descendants are now displayed separately. This allows the analysis of performance behavior on different levels of granularity.

For example, the call tree may have a node *main* with two children *foo* and *bar* (Figure 4.9). In the collapsed state, this node is labeled with the weight representing the time spent in the whole program. In the expanded state it displays only the fraction that is spent neither in *foo* nor in *bar*.

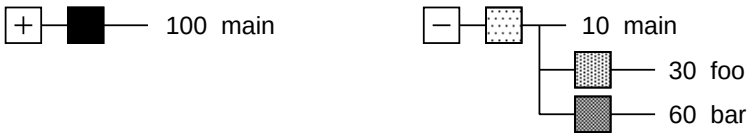

Figure 4.9: Weighted tree in collapsed and expanded state.

The weight is displayed simultaneously using both a numerical value as well as a colored icon. The color is taken from a spectrum ranging from blue to red representing the whole range of possible weights (i.e., 0 - 100 percent). To avoid an unnecessary distraction, insignificant values below a threshold of 0.5 percent are displayed in gray. Colors enable the easy identification of nodes of interest even in a large tree, whereas the numerical values enable the precise comparison of individual weights.

The complete performance-space display is depicted in Figure 5.3 (p. 119). The left tree

shows the performance-property hierarchy, the middle tree shows the call-path hierarchy, and the right tree shows the location hierarchy. The weighted trees of the different dimensions are interconnected so that the user can display the call tree with respect to a particular performance property and the location tree with respect to a particular node in the call tree. After selecting a property and a call path, the call tree refers to the selected property, and the location tree refers to both the selected property and the selected call path. Note that references to selected items also take into account the state (i.e., collapsed or expanded) of these items.

Table 4.1 summarizes the severity shown by a node in the performance-space display. A node d of one of the three trees either represents a performance property $b \in B$, a call path $n \in N$, or a location $\hat{l} \in \hat{L}$. The state of a node is given by a function $state(d) \in \{e, i\}$, which indicates whether a node is collapsed (i) or expanded (e). Note that a collapsed node corresponds to inclusive semantics (i) and an expanded node corresponds to exclusive semantics (e). In addition, b_{sel} denotes the selected property and n_{sel} the selected call path.

Table 4.1: Severity amounts shown in tree displays.

Displayed Node	Severity
$b \in B$	$\lvert\widehat{sev}^{(state(b)),i,i)}(b, root(N), root(\hat{L})\rvert$
$n \in N$	$\lvert\widehat{sev}^{(state(b_{sel}),state(n),i)}(b_{sel}, n, root(\hat{L})\rvert$
$\hat{l} \in \hat{L}$	$\lvert\widehat{sev}^{(state(b_{sel}),state(n_{sel}),state(l))}(b_{sel}, n_{sel}, \hat{l})\rvert$

In the default mode, the display represents the severity as a percentage of the total CPU-reservation time. This mode is called the *absolute mode* because all percentages refer to the same yardstick. However, applications that exhibit a large call tree and many locations may suffer from very small values in the call tree and, in particular, in the location tree, which may limit the display's scalability. For this reason, the presenter offers a *relative view mode*. In this relative view mode, a percentage shown in a tree always refers to the selection in the left neighbor tree. For example, the display in Figure 5.3 (p. 119) is in the absolute mode. Each value is a percentage of the total CPU-reservation time. In contrast, the display in Figure 5.1 (p. 116) is in the relative view mode. The 12.1 percent shown for process 0 in the location tree represents 12.1 percent of 2.4 percent (selected call path) of 2.0 percent (selected property) of the CPU-reservation time.

Weighted trees provide a uniform and very intuitive display for each of the analyzed dimensions. Once the user is familiar with this kind of display, it is possible to navigate across the performance space in a scalable but still accurate way along all its interconnected dimensions. First, the presenter allows exploration of the full performance space by showing the results of a multidimensional analysis in a multidimensional fashion using three interconnected tree browsers. Second, instead of confusing the user with differently styled views for different metrics, all performance properties are uniformly accommodated in the same display and thus provide the ability to easily compare the effects of different

kinds of performance behavior. In addition, since the user only needs to get accustomed to one way of presentation, the necessary learning efforts are small. Third, the ability to investigate the performance behavior of individual nodes in the call tree (i.e., call paths) either including or excluding their descendants allows the analysis of complex source-code hierarchies along the functional dependences of their elements.

4.6 Limitations

Despite its strengths, the approach taken in this thesis exhibits some limitations that result both from general limitations of event tracing on the one hand and from particular properties of the trace-analysis method proposed here on the other hand.

- The event-trace size, which may easily reach several millions of events or several hundreds of megabytes when dumped to a file, constitutes a severe obstacle to a ubiquitous application of all trace-based performance-analysis techniques. The difficulties of handling large traces result from their local buffer-memory requirements during generation, which may, in addition to competing for the target application's memory, cause significant perturbation when the buffer contents are written to a file as a result of buffer overflow. Also, global trace-file sizes may limit scalability in the case of massively parallel systems with thousands of processors.
- As a consequence of the enormous trace-file sizes, the analysis process performed by EXPERT may take several hours to complete. Although a processing time of several hours might be acceptable if it results in substantial performance improvements, to convince the user community a production tool should offer more convenience also with respect to speed. However, the current Python implementation still offers opportunities for optimization. Section 4.7 presents concepts that are aimed at optimizing the analysis process in terms of speed and maintenance.
- As already mentioned in Section 4.4.1, EXPERT does not compute the severity of a performance property as an interval set. Instead it computes only the amount by summing up the amounts of simple intervals. To ensure the correctness of computing the inclusive and exclusive severity, the severity intervals of siblings in the property hierarchy must be non-overlapping (4.13), which may limit the freedom of extending that hierarchy at least to some extent.

4.7 Advanced Techniques

This section presents concepts partially dealing with the previously mentioned limitations, which are too detailed and too specific to be mentioned in the last chapter, but still too early

in design and development to be considered finished.

4.7.1 Selective Tracing

Another option for speeding up the analysis process is selective tracing. The central idea of selective tracing is to reduce trace-file size and intrusion by recording only a selected subset of the performance space, which implies that the selection can be done along several dimensions, such as time, source code, or locations. The selection may be based both on omitting performance data showing only inconspicuous behavior or on avoiding repetition by restricting the measurement to a small but still representative subset of program execution. However, both methods require either some form of dynamic instrumentation or multiple experiments because neither suspicious program parts nor the occurrence of repetitive behavior are usually known prior to the first measurement.

A simple method of identifying performance-relevant program parts for the purpose of selective tracing is to generate a profile prior to event tracing. Then, tracing can be restricted to those program parts that show performance-relevant behavior in the profile. However, in some cases a call-graph–based profile, which can be obtained using a call-graph profiler, such as CATCH [17], might be necessary to also identify the context in which a certain function exhibits inefficient behavior.

Another strategy of dynamically moving from coarse-grained performance data with low space requirements for large portions of program behavior to fine-grained performance data for small suspicious parts of program behavior which have been identified based on the coarser data, was successfully applied by Miller et al. [55] in the Paradyn project, although it is not used there for event tracing. A recent addition to the Paradyn tool [12] was a call-graph–based search strategy that climbs down the call graph stepwise from callers to callees. A callee is chosen for instrumentation if the caller exceeds a certain threshold with respect to a certain performance metric.

A third approach motivated by the desire to restrict event traces to representative subsets of program execution is presented by Freitag et al. [28]. They try to exclude periodical repetitions of iterative patterns by applying a periodicity-detection algorithm to the stream of parallel function identifiers at run time. This seems to be promising in particular in view of the many iterative applications in computational science.

However, the approach of compound-event detection based on event-model enhancement imposes some consistency constraints on selective traces, which must be considered when opting for a selection method. An event trace that starts somewhere in the middle of program execution or that does not contain the events of certain execution phases may, for example, suffer from incomplete region instances and messages, that is, incomplete pairs of Enter-Exit and Send-Receive events. In addition, it may contain only fractions of MPI

collective-operation or OpenMP parallel-construct instances. All this may cause the abstractions defined in Chapter 3 to fail as a result of an incomplete event trace. In general, the necessary completeness constraints may vary depending on the set of abstractions required and on the kind of compound events to be detected.

A simple model of selective tracing in the context of model enhancement would require the selective trace to start with some form of check-point information and then continue with individual events. A check-point could contain the current call-graph node(s) and all pending communications. However, writing a check point would require tracking this information continuously during run time

DeRose and Wolf [17] propose a technique for tracking the call graph at run time with constant overhead based on binary instrumentation. They compute the static call graph in advance and for every control flow move a pointer from node to node as program execution proceeds. Each call site provides an index into an array of successor nodes so that the next node to be reached by that call site can be quickly determined based on that index.

A method of dealing with pending communications, that is, pending MPI point-to-point messages and collective operations, would be their avoidance by starting the selective trace only after finishing a (synchronizing) collective operation involving all processes since in most applications point-to-point communication is usually not interleaved with collective communication. This approach is especially well suited for selective traces covering single iterations of large loops because in many applications an iteration is finished with a global reduction operation or a barrier.

4.7.2 Publish and Subscribe

The current design of EXPERT implements performance properties as pattern classes. Each class is separately responsible for both compound-event instantiation and constraint verification. A sharing of functionality is restricted to exploiting the inheritance hierarchy, which is not identical to the specialization hierarchy and, thus, is of limited benefit for this purpose. This low level of cooperation among different pattern classes causes certain aspects of similar properties to be both specified and computed twice.

Another more effective way of sharing functionality among different properties would be to exploit the increasing specialization of compound events along a path in the property hierarchy. For example, the property *Message in Wrong Order* operates on *Late Sender* compound-event instances. Currently, the two properties are computed independently.

So instead of providing a call-back method only for primitive events a property (i.e., pattern object) could also “subscribe” to compound events that are “published” by other properties residing on a higher level in the hierarchy. The subscribers in turn could add some features to the compound event and republish it as a more specialized version of the one they

received as a subscriber. As long as a property may only subscribe to compound events published by more general properties it is ensured that there are no cyclic dependences. The compound events to be published could be specified as classes with specific access methods.

4.7.3 Generic Visualization

The strength of the visualization technique applied by EXPERT stems from its uniform treatment of all performance properties, which allows their accommodation in a single integrated view. This not only simplifies the tool usage but also gives an opportunity to correlate different aspects of performance behavior.

Besides improvements in the context of EXPERT, such as adding a source-code display to highlight code regions or integrating it with an event-trace browser to show representative compound-event instances, the underlying idea of presenting performance data in a multi-dimensional property-oriented performance space offers the opportunity of a much broader coverage of different performance data.

Since the representation of performance properties including their severity is independent of their semantics, a similar visualization could be used for a different performance tool relying on different performance metrics. For example, a profiler, such as CATCH [17], might collect cache events per call-graph node and location. Since cash events can be organized in a hierarchy, for example, based on access type (all, read, or write) or level in a multilevel cache, a multidimensional representation with the number of events indicated by color would be appropriate here as well. Even different dimensions, such as resource hierarchies as used in Paradyn [55], can conceivably visualized in that way.

A reasonable conclusion drawn from these considerations would be to establish the property-oriented performance space as an independent data model that can serve three different goals:

1. High-level data model of performance behavior
2. Portable data format
3. Generic presentation component

First, as a data model it can help to define relationships among different performance properties, such as specialization and generalization, or relationships among different locations, such as MPI process topologies. Note that it need not be restricted to hierarchical organizations. Second, as a portable data format it can be used to store both static performance metadata describing these relationships and dynamic performance data representing a particular experiment in the context of these relationships. In view of a frequently occurring hierarchical organization of performance entities, such a data format could be designed

as an XML [74] instance. Third, files in that data format can serve as input for a generic visualization component that is dynamically adapted based on the file's metadata.

4.8 Summary

The EXPERT performance tool analyzes the performance behavior of MPI, OpenMP, or hybrid applications by transforming event traces into a three-dimensional property-oriented performance space. The key idea behind the property-oriented performance space is the uniform treatment of all performance properties, allowing their convenient correlation along multiple dimensions using only a single integrated view.

A thread-safe multi-level instrumentation captures events related to ordinary user functions as well as events specifically related to MPI and OpenMP on the source-code, compiler, and linker level and merges them into a single event trace with global time stamps.

The analysis process tries to prove the presence of performance properties in the target application by looking for the existence of compound events in the event trace. Compound events are specified in terms of an enhanced event model, on which the actual analysis process takes place.

After analysis has been completed, all performance properties return their severity matrix representing their plane of the performance space. Then, the matrices of individual properties are combined into a three-dimensional data structure spawning the whole performance space. After adding hierarchical locations to the data structure, it is displayed using weighted trees. Each tree represents a dimension of the performance space and allows a scalable inspection of that dimension by showing the severity of a node in the tree either including all its children in a collapsed state or excluding all its children in an expanded state. In addition to a numeric value, the severity is also indicated by color to highlight extremes even in the case of large trees.

Chapter 5

Examples

The EXPERT performance-analysis environment has been tested for several real-world applications. The chapter demonstrates that the performance problems addressed by the present approach are of practical relevance and that they can be conveniently localized using the EXPERT presentation component. The test cases comprise two MPI applications, TRACE and CX3D, and two hybrid applications, REMO and SWEEP3D. The thesis considers one event trace per application.

All the experiments were conducted on ZAMpano [27], a parallel computer with eight SMP nodes, each having four Intel Pentium III Xeon (550 MHz) CPUs. CPU reservation was done such that one CPU per thread or single-threaded process was available to each application.

Table 5.1: Trace-file size and overhead.

	TRACE	CX3D	REMO	SWEEP3D
CPUs	16	8	16	16
Size (MB)	310	34	170	72
Execution time (sec)	58.9	139.8	37.2	16.5
Overhead (%)	4.2	0.1	9.7	6.0
Analysis time (h:m)	12 : 57	1 : 25	9 : 48	3 : 22

Table 5.1 summarizes trace-file size and overhead. The first row contains the program name, the second row shows the number of CPUs used, the third row lists the trace-file size, and the fourth row gives the execution time. To estimate the run-time overhead introduced by the instrumentation, the minimum execution time of a series of ten uninstrumented runs was compared to the minimum execution time of a series of ten instrumented runs. The result is listed in the fifth row. Finally, the last row shows the duration of the analysis process carried out on the test platform.

The large trace-files sizes obtained for only short execution times expose a limitation of the current approach. Selective tracing techniques, as discussed in Section 4.7.1, might help to reduce temporal event density while preserving relevant information. The inconveniently long analysis run times are not only a result of large trace-file sizes, but also a consequence of the prototype's early design stage. One opportunity for optimization is, for example, an improved information exchange among different performance properties during analysis, as outlined in Section 4.7.2. In addition, the re-implementation in a fast programming language, such as C++, might also contribute to better speed results. The overhead numbers presented in the table are satisfactory, only the instrumentation overhead of REMO reaches nearly ten percent. However, since the performance problem identified in REMO is large in relation to the overall execution time, the numbers presented here concerning this problem are still useful (Section 5.3).

5.1 TRACE

TRACE [26] simulates the subsurface water flow in variably saturated porous media. The parallelization is based on spatial decomposition and a parallelized CG algorithm. The application was executed using four SMP nodes with four processes per node (4 processes $\times$ 4 processes). MPI communication within SMP nodes was done via shared memory.

Using the performance-property view (Figure 5.1, left), it is easy to see that most of the

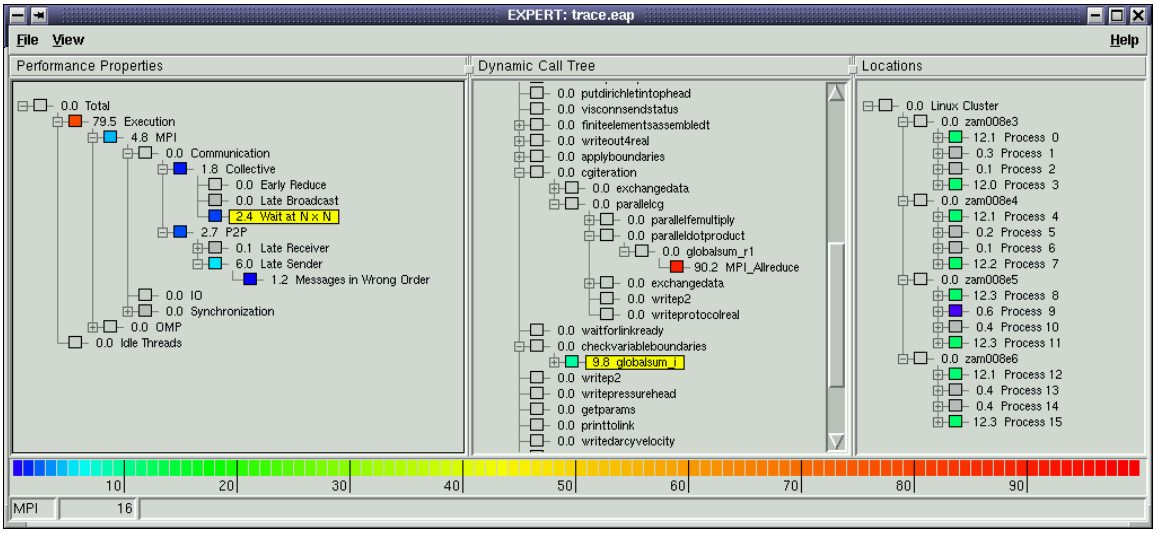

Figure 5.1: Display of performance behavior in EXPERT for TRACE in the relative view mode.

time used for communication routines was spent on waiting due to the situations *Late Sender* and *Wait at N × N*, which are described in Example 3.1 (p. 65) and 3.4 (p. 68).

Using the call-tree view (Figure 5.1, middle), one can quickly locate two call paths that are major sources of the previously identified performance problems. The call path mainly responsible for the property *Wait at N × N* is shown in the vertical middle of the call tree. The presenter display was switched to the relative view mode, that is, whereas the values and colors on the left are percentages of the total CPU reservation time, the percentages in the middle are fractions of the selection (node with framed label) on the left, and the percentages on the right are fractions of the selection in the middle. For example, the 9.8 percent shown for the selected call path in the middle is 9.8 percent of 2.4 percent of the total CPU reservation time.

The results of the analysis are listed in Table 5.2. While the top section of the table lists the two call paths, the bottom section contains the numerical results obtained for the whole program and these two call paths. The values in the bottom section represent percentages of the total CPU-reservation time. The first column refers to the whole program, whereas the second and third columns refer to the call paths listed above in the table. The first row corresponds to the time spent in MPI communication statements. For the two call paths this is just the time needed for completion of the specific MPI calls at their end. The second and third row correspond to the waiting times caused by the *Wait at N × N* and *Late Sender* situations.

The location view (Figure 5.1, right) shows the distribution of idle times introduced by *Wait at N × N* during execution of the call path selected in the middle tree of Figure 5.1, which is another call path responsible for that property. Obviously, the idle times expose an uneven but still symmetric distribution across the different processes. The "inner" processes of each SMP node exhibit significantly less waiting time than the "outer" ones. Figure 5.2 shows a VAMPIR [3] time-line diagram of TRACE when executing this call path. The time

Table 5.2: Performance problems found in TRACE in percentage of the total CPU-reservation time.

Call Paths			
(a)	`trace` → `cgiteration` → `parallelcg` → `parallelfemultiply` → `exchangedata` → `exchangebufferswf` → `mrecv` → `MPI_Recv`		
(b)	`trace` → `cgiteration` → `parallelcg` → `paralleldotproduct` → `globalsum_r1` → `MPI_Allreduce`		
Performance Property	**Whole Program**	**(a)**	**(b)**
Communication	14.3	7.8	3.0
Late Sender	7.3	5.8	
Wait at N × N	2.4		2.2

line presents a noticeable *Wait at N × N* instance. The distribution of the waiting times in MPI_Allreduce shown in the time line bears a clear resemblance to the distribution shown in the EXPERT result display (Figure 5.1, right) in that every second pair of processes suffers from significant waiting times.

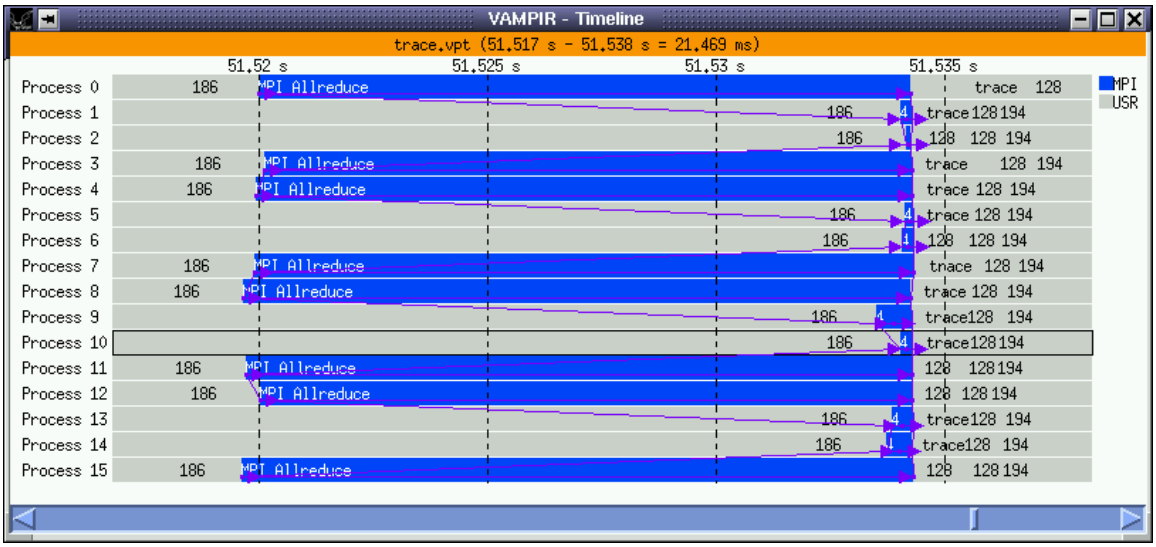

Figure 5.2: VAMPIR time-line diagram of TRACE.

5.2 CX3D

CX3D is an MPI application used to simulate *Czochralski* crystal growth [54], a method applied in the silicon-wafer production. The simulation covers the convection processes occurring in a rotating cylindrical crucible filled with liquid melt. The convection, which strongly influences the chemical and physical properties of the growing crystal, is described by a system of partial differential equations. The crucible is modeled as a three-dimensional cubic mesh with its round shape expressed by cyclic border conditions. The mesh is distributed across the available processes using a two-dimensional spatial decomposition. The application was executed on two SMP nodes with four processes per node. MPI communication within SMP nodes was done via shared memory.

Most of the execution time is spent in a routine called VELO, which is responsible for calculating the new velocity vectors. Communication is required when the computation involves mesh cells from the border of each processor's subdomain. The execution configuration of CX3D is determined by the number of processes that are assigned to each of

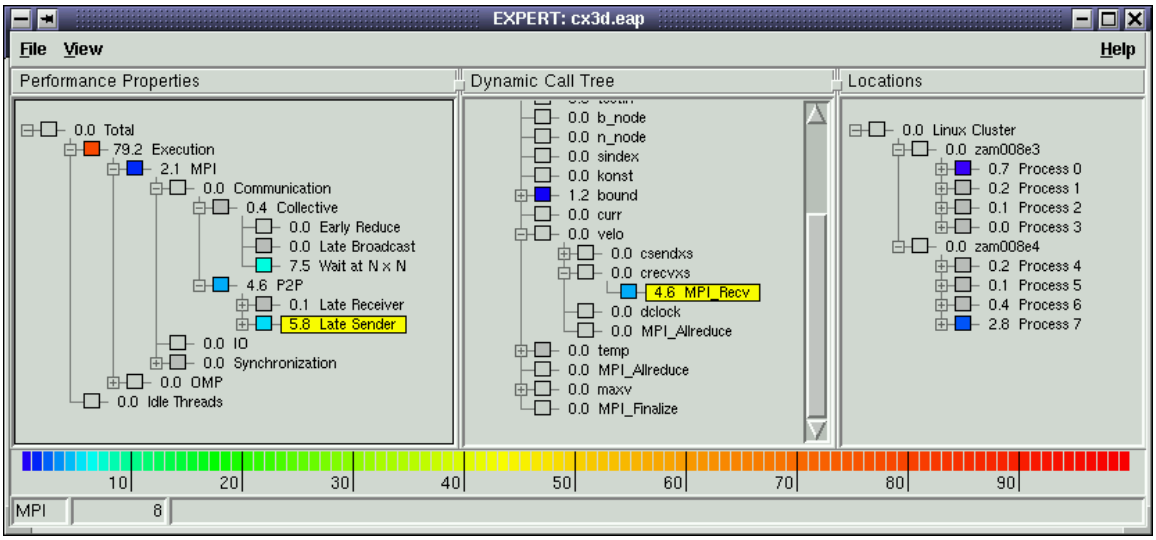

Figure 5.3: Display of performance behavior in EXPERT for CX3D in the absolute view mode.

the two decomposed dimensions. The experiment presented here was conducted with a decomposition configuration of 8×1 processes.

The results in Table 5.3 show that a significant amount of the communication time was introduced by *Late Sender* and *Wait at N × N*. Using the call-tree view (Figure 5.3, middle), it is easy to identify two call paths mainly responsible for these performance properties. Both call paths are executed as parts of VELO. They are listed in the top section of the table.

Using the location view (Figure 5.3, right), one can easily investigate the distribution of the identified performance problems across the processes and, in particular, look for sim-

Table 5.3: Performance problems found in CX3D in percentage of the total CPU-reservation time.

Call Paths			
(a)	`velo` → `crecvxs` → `MPI_Recv`		
(b)	`velo` → `MPI_Allreduce`		
Performance Property	**Whole Program**	**(a)**	**(b)**
Communication	18.4	7.1	6.9
Late Sender	5.8	4.6	
Wait at N × N	7.5		6.6

ilarities and correlations among the distributions of different properties. Figure 5.3 shows the distribution of the property *Late Sender* across the processes. It is obvious that most of the time associated with this property is caused by process 0 and 7.

The bar chart in Figure 5.4 compares the distribution of *Late Sender* in routine VELO to the distribution of other properties also available in the EXPERT presenter. *Execution (exclusive)* is the execution time of VELO that was not spent on MPI and, thus, roughly corresponds to the time spent solely on computation. *Communication* is the time spent on MPI communication statements. Since in VELO call path (a) is the only source of *Late Sender* and call path (b) is the only source of *Wait at N* $\times$ *N*, both properties in the bar chart refer to VELO as a whole as well as to the two call paths alone.

Apparently, the computation is unevenly distributed across the different processes, a situation that is commonly referred to as load imbalance. Moreover, it seems that there is a correlation between this load imbalance and the times spent on *Late Sender* and *Wait at N* $\times$ *N*. Every time the computation time is low, the times spent on both *Late Sender* and *Wait at N* $\times$ *N* are high. Notice that the difference between *Communication* and the sum of *Late Sender* and *Wait at N* $\times$ *N* is always very small compared to the computation time.

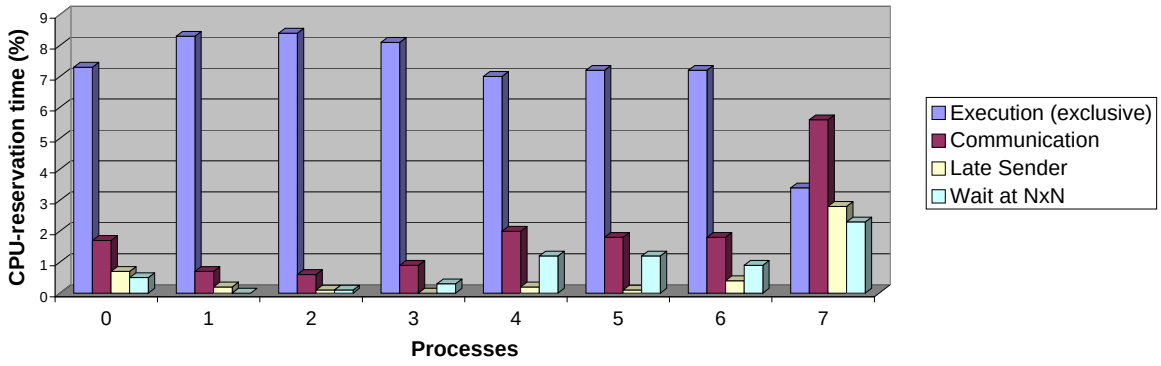

Figure 5.4: Distribution of performance properties in VELO across the processes.

A VAMPIR time-line diagram of CX3D when executing VELO is shown in Figure 5.5. The middle of the time line exhibits a noticeable *Late Sender* instance. Process 7 tries to receive a message from process 6 using MPI_Recv, but the message is sent long after process 7 has entered the receive operation. Some other but smaller instances follow shortly after this one. Finally, on the right part of the time line one can recognize a *Wait at N* $\times$ *N* instance across all processes. Note that the workload distribution across all processes for the section of the time line shown here corresponds to the observations made by EXPERT in that the fraction process 7 spent on computation is small compared to the other processes. This seems to be the reason that the MPI operations are entered earlier by process 7 and, thus, the reason for the inefficient behavior.

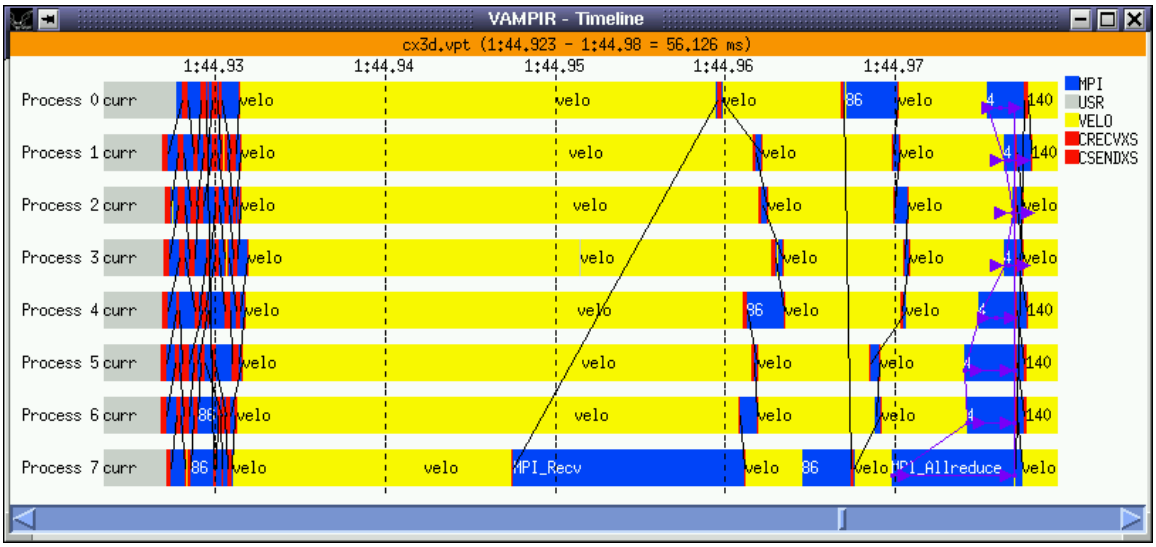

Figure 5.5: VAMPIR time-line diagram of CX3D.

5.3 REMO

REMO [18] is a weather forecast application of the DKRZ (Deutsches Klima Rechenzentrum). It implements a hydrostatic limited area model, which is based on the *Deutschland/Europa* weather forecast model of the German Meteorological Services (Deutscher Wetterdienst (DWD)). The thesis considers an early experimental MPI/OpenMP version of the production code. The application was executed on four nodes with one process per node and four threads per process (4 processes $\times$ 4 threads).

Figure 5.6 shows the result display of REMO in the absolute mode, that is, all values and colors represent percentages of the total CPU-reservation time. The property view indicates that one half of the total CPU-reservation time is idle time (i.e., *Idle Threads*) resulting from OpenMP sequential execution outside of parallel regions. Although during this period the idle threads actually do not execute any code, the time is mapped onto the call paths that have been executed by the master thread during this time. That is to say, for analysis and presentation purposes EXPERT assumes that outside parallel regions the slave threads "execute" the same code as their master thread. This method of call-path mapping helps to identify parts of the call tree that might be optimized in order to reduce the amount of sequential execution.

In the case of REMO, the EXPERT call-tree view (Figure 5.6, middle) allows the easy identification of two call paths as major sources of idle times. The location view (Figure 5.6, right) illustrates that this property only applies to slave threads. The analysis results are

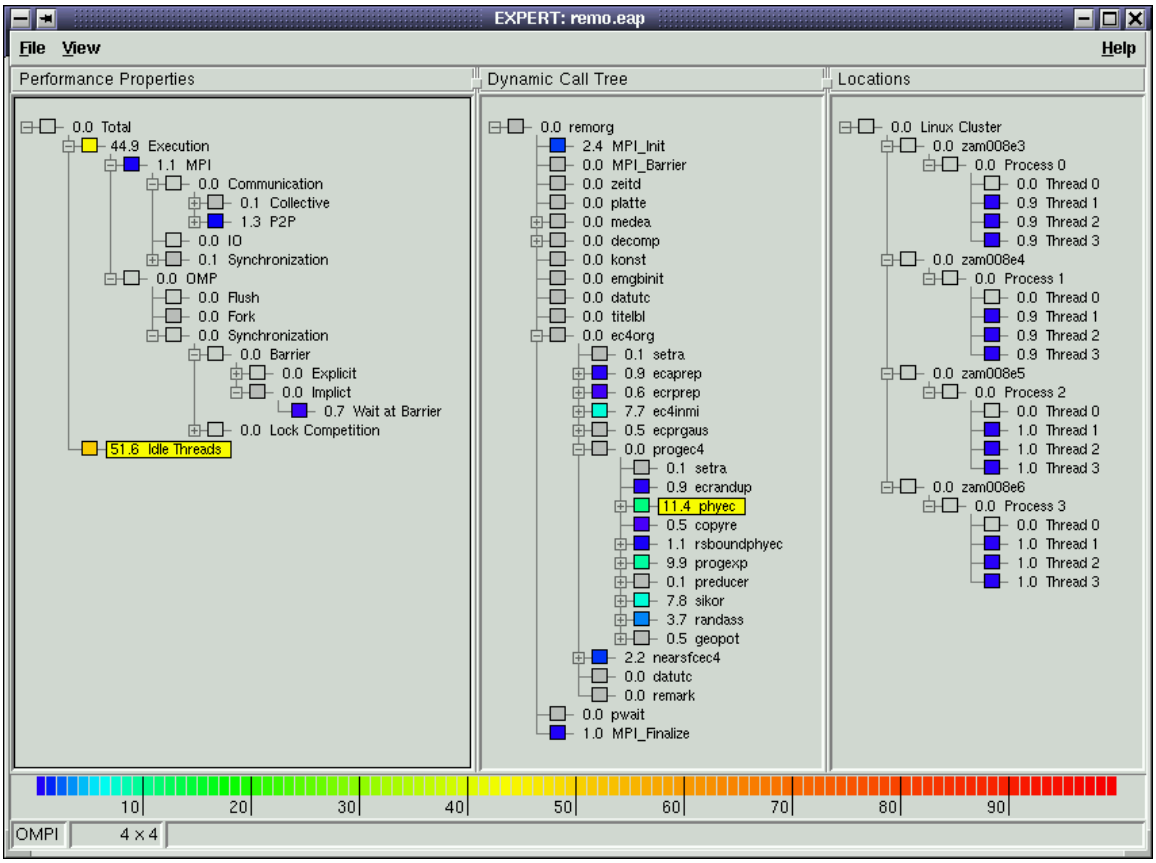

Figure 5.6: Display of performance behavior in EXPERT for REMO in the absolute view mode.

listed in Table 5.4. The values shown in the bottom section represent the severity of the property *Idle Threads* measured for the whole program and the two call paths. The values are percentages of the total CPU-reservation time lost as a result of this performance property. The two call paths are listed in the top section.

Table 5.4: Performance problems found in REMO in percentage of the total CPU-reservation time.

Call Paths			
(a)	`remo` → `remorg` → `ec4org` → `progec4` → `phyec`		
(b)	`remo` → `remorg` → `ec4org` → `progec4` → `progexp`		
Performance Property	**Whole Program**	**(a)**	**(b)**
Idle Threads	51.6	11.4	9.9

5.4 SWEEP3D

The benchmark code SWEEP3D [4] represents the core of a real ASCI application. It solves a 1-group time-independent discrete ordinates (Sn) 3D Cartesian (XYZ) geometry neutron transport problem. The thesis considers an early experimental MPI/OpenMP version of the original MPI version. While MPI is responsible for parallelism by domain decomposition, OpenMP is responsible for parallelism by multitasking.

The application was executed on four nodes with one process per node and four threads per process (4 processes $\times$ 4 threads). The performance behavior of SWEEP3D exhibits a weak point of hybrid programming, that is, a performance problem resulting from the combination of MPI and OpenMP. MPI calls made outside a parallel region prolong sequential execution and prevent available CPUs from being used by multiple threads. The results are shown in Table 5.5. The call path (a) shown in the table is responsible for most of the losses occurring due to the property *Idle Threads*. However, at the same time this call path exhibits a significant loss due to the property *Late Sender*. Note that *Late Sender* adds the times of the master threads, whereas *Idle Threads* adds the times of the slave threads (3 slaves per master). Taking this into account, reducing *Late Sender* by one percent would speed up the application by four percent. Obviously, one reason for the *Late Sender* problem at call path (a) is receiving messages in the reverse sending order (*Messages in Wrong Order*).

Moreover, a significant amount of time is spent on the implicit (i.e., compiler-generated) OpenMP barrier at the end of call path (b). Expanding the node of the property *Implicit Barrier* reveals that most of that time is lost due to the property *Wait at Barrier* (see also Example 3.5, p. 69). The property deals with the threads of a team reaching an implicit

Table 5.5: Performance problems found in SWEEP3D in percentage of the total CPU-reservation time.

Call Paths			
(a)	`seep3d` → `inner_auto` → `inner` → `sweep` → `recv_real` → `MPI_Recv`		
(b)	`driver` → `inner_auto` → `inner/sweep` → `!$omp parallel` → `!$omp do` → `!$omp ibarrier`		
Performance Property	**Whole Program**	**(a)**	**(b)**
Idle Threads	37.5	17.5	
Communication	6.5	5.8	
Late Sender	3.2	3.2	
Messages in Wrong Order	0.9	0.9	
Implicit Barrier (OpenMP)	4.3		3.3
Wait at Barrier (OpenMP, implicit)	2.8		2.6

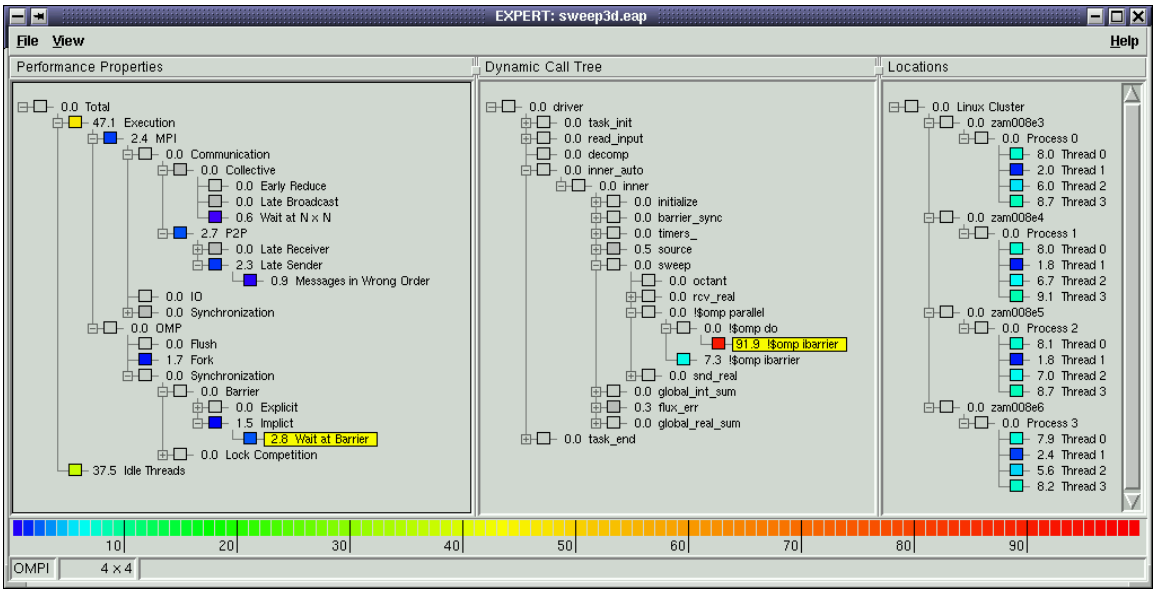

Figure 5.7: Display of performance behavior in EXPERT for SWEEP3D in the relative view mode.

barrier at different points in time so that threads arriving early have to wait for those which arrive later. The location view in Figure 5.7 shows an uneven distribution of these waiting times across the different threads. The display is in the relative view mode. Therefore, values and colors in the middle and left tree are scaled with respect to the selection in the right and middle tree, respectively.

The scheduling strategy applied by the enclosing parallel do loop was not specified in the source code. In this case, the compiler used for this experiment statically assigns a contiguous chunk of work (i.e., a contiguous section of the loop-index range) to each thread. If the loop-index range is not divisible by the number of threads or if the different chunks represent a different work load for another reason, the threads finish the loop at different points in time. To demonstrate EXPERT's capabilities in highlighting the effects of different scheduling strategies on the distribution of waiting times across different threads, the scheduling was changed to dynamic scheduling with a chunk size of one. This causes the program to dynamically assign one loop-index value to each thread every time a thread asks for new work. Figure 5.8 shows a result display for SWEEP3D with dynamic scheduling instead of static scheduling. The waiting time is now more uniformly distributed compared to the version with static scheduling.

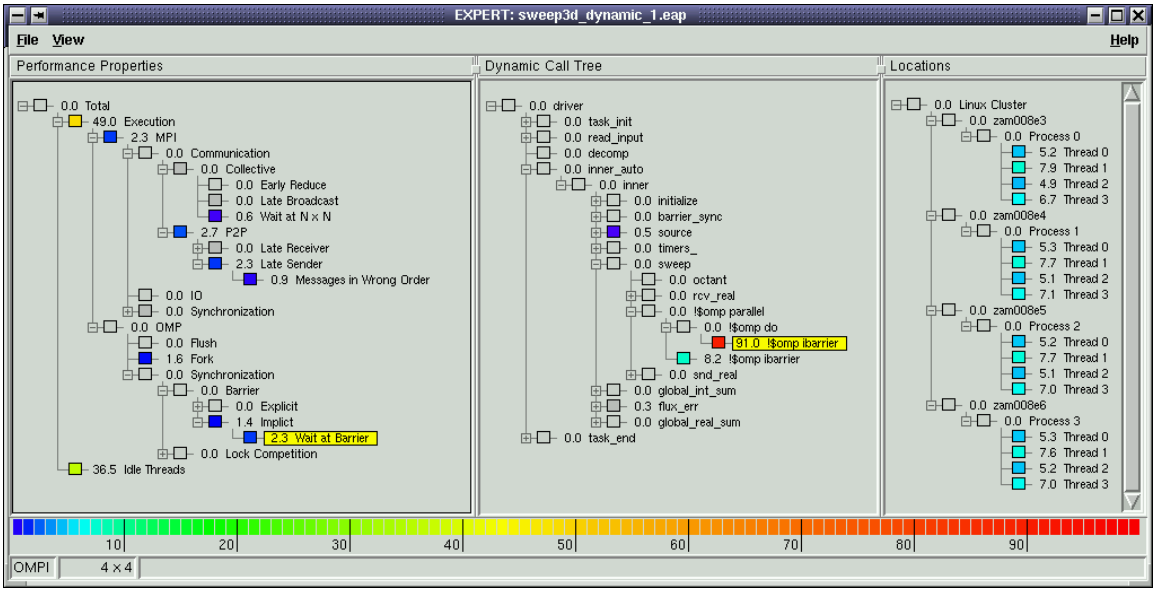

Figure 5.8: Display of performance behavior in EXPERT for SWEEP3D in the relative view mode. The figure highlights the distribution of idle times in front of the implicit barrier when applying dynamic loop scheduling.

Chapter 6

Related Work

The approach presented here is not the first work on automatic performance analysis. Related approaches - even if they did not serve as inspiration for the present approach - either offer a different solution or emphasize a different view of the problem.

Miller et al. [55, 72] developed automatic on-line performance analysis based on run-time instrumentation in the well-known Paradyn project. The Paradyn search process follows the W^3 Search Model *(why, where, when)*, which describes performance behavior along the dimensions: performance problem, program resource (i.e., focus), and time. Similar to EXPERT, the first two dimensions are organized in a specialization hierarchy. Performance problems are expressed in terms of a threshold and a metric. A metric usually refers to a counter and is represented either as a percentage (e.g., CPU time or blocking time), as a rate (e.g., operations per second), or as a plain value (e.g., number of active processors). Program resources include both hardware resources, such as processor nodes or disks, and software resources, such as procedures, message channels, or barrier instances. The time dimension tries to divide the program execution into phases with certain performance characteristics. A performance-problem hypothesis is regarded as proven if the target application exceeds the threshold associated with a metric for a certain amount of time. The search process starts at the top level of the *why* and *where* axis and performs a successive refinement both in terms of problem type and focus based on hypotheses that have already been proved. The main accomplishments of EXPERT in contrast to Paradyn is the description of performance problems in terms of complex event patterns instead of counter-based metrics. Also, the uniform mapping of performance behavior onto execution-time interval sets in conjunction with a formal characterization of specialization among performance problems allows the precise correlation of different behavior in a single integrated view.

The Autopilot [66] software infrastructure targets real-time adaptive control of resource interactions in parallel and distributed systems. Automatic behavioral classification of resource-request patterns based on data captured by distributed performance sensors and

user-written assertions precedes a fuzzy decision procedure, which relies on actuators to dynamically carry out changes in the current resource-management policy. Whereas this thesis's approach mainly concentrates on communication and synchronization, Autopilot is useful, in particular, to control the performance of parallel file IO.

Gerndt and Krumme [30] developed a rule-based approach to automatic performance analysis of programs on shared–virtual-memory environments, such as SVM Fortran [8]. The analysis process is specified as a rule base consisting of refinement rules and proof rules. Refinement rules consist of a coarse hypothesis and more precise hypotheses to be checked after the coarse hypothesis has been proved. Proof rules contain the declaration of performance information required to prove a hypothesis and predicates that represent the hypothesis's semantics. The approach of Gerndt and Krumme advocates a clear separation between the analysis process as represented by refinement rules and knowledge about potential performance problems as represented by the proof rules. The rationale behind the stepwise analysis process is to control the demand for finer performance data by evaluating predicates over coarser data and thus to reduce the total amount of data necessary to assess an application's performance.

Finnesse [59] is a prototype environment designed by Mukherjee et al. to support incremental parallelization of Fortran 77 programs for shared-memory architectures. The parallelization process is guided by an overhead-oriented interpretation of performance loss relative to the performance of a reference (serial) implementation. Automatic static analysis to calculate dependence information precedes the automatic collection and classification of empirical overhead data by conducting several experiments. Depending on the results, Finesse may recommend code transformations, whose effects can be assessed using a version-management mechanism.

Espinosa [19] implemented an automatic trace-analysis tool KAPPA-PI for evaluating the performance behavior of MPI and PVM message-passing programs. Here, behavior classification is carried out in two steps. First, a list of idle times is generated from the raw trace file using a simple metric. Then, based on this list, a recursive inference process continuously deduces new facts on an increasing level of abstraction. Finally, KAPPA-PI builds suggestions of possible improvements from the facts already proved on the one hand and from the results of source-code analysis on the other hand.

Vetter [73] performs automatic performance analysis of MPI point-to-point communication based on machine-learning techniques. He traces individual message-passing operations and then classifies each individual communication event using a decision tree. The decision tree has been previously trained by microbenchmarks that demonstrate both efficient as well as inefficient performance behavior. The ability to adapt to a special target system's configuration helps to increases the technique's predictive accuracy. In contrast to this approach, EXPERT draws conclusions from the temporal relationships of individual events in a platform-independent way, which does not require any training prior to analysis.

Helm and Malony propose the design of a novel performance-diagnosis system POIROT [36] based on heuristic classification, which means solving problems by matching them to previously stored solutions. The system consists of a problem solver and an environment interface. The latter tries to overcome the poor combination of automation and adaptability found in traditional approaches by separating diagnosis methods from the software that supports those methods. The problem solver selects and carries out performance-diagnosis actions. This process is supported by a knowledge base that provides both a method catalog and control knowledge. The method catalog is a library of performance diagnosis techniques, such as rules of hypotheses refinement, whereas the control knowledge specifies the general policy of the analysis process. However, recent research is moving away from matching problems towards matching of performance models with performance data.[1]

A novel approach to the formalization of performance properties and the associated performance-related data is the APART Specification Language (ASL) [22], which was developed by the APART working group on *Automatic Performance Analysis: Resources and Tools*. ASL provides a formal notation for defining performance properties related to different programming models. It allows performance-related data items to be referenced by means of an object-oriented data model. In the ASL terminology, a performance property represents one aspect of performance behavior. To test whether such a property is present in an application, an associated condition must be evaluated based on the current performance data. The notion of a performance property strongly influenced the work on EXPERT and motivated the notion of a property-oriented performance space. However, since the initial ASL data model mainly concentrated on profiling data (i.e., summary information) and did not take advantage of the more detailed information contained in event traces, the work on compound events done in this thesis stimulated the treatment of trace data within the ASL framework. Appropriate extensions have been proposed in Section 3.9 and are now part of the revised ASL specification [21].

Stimulated by ASL, JavaPSL [23] was designed by Fahringer et al. to specify performance properties based on the Java programming language. Like EXPERT, JavaPSL represents performance properties as abstract classes that can be implemented to provide an extensible set of performance properties to be used in a real tool. Whereas EXPERT uses Python to provide a uniform interface to performance properties, JavaPSL exploits mechanisms of the Java language, such as polymorphism, abstract classes, and reflection. As opposed to EXPERT, which concentrates on compound-event analysis and defines inter-property relationships based on a subset condition referring to the time spent on a specific behavior, JavaPSL emphasizes the definition of performance properties based on existing properties. Key ideas are the definition of abstract classes to isolate commonalities of the property implementation and the definition of metaproperties that depend on a whole set of existing performance properties. Common to both approaches is the integrated treatment of MPI, OpenMP, and hybrid programming.

[1] Allen Malony: personal communication

An alternative approach to describing complex event patterns was devised by Bates [5]. The proposed Event Definition Language (EDL) focuses on specifying incorrect behavior of distributed systems. It allows compound events to be defined in a declarative manner based on extended regular expressions, where primitive events are clustered to higher-level events using certain formation operators. Relational expressions over the attributes of the constituent events place additional constraints on valid event sequences obtained from the regular expression. Abstraction mechanisms allow the re-use of already defined compound events to form custom hierarchies of events. However, problems arise when trying to describe compound events that are associated with some kind of state, such as those representing performance problems in MPI and OpenMP applications.

Kranzlmüller [47] applies event-graph analysis in order to detect parallel-programming errors. An event graph is a finite set of events connected by a happened-before relation [48]. The happened-before relation is derived from either the sequential order of events generated by the same process or message communications among different processes. Complex erroneous behavior is expressed in terms of event patterns that are specified using a graphical tool named PatternTool [33]. Abstraction mechanisms are based on selection operations, relations, and macronodes. Selection operations identify groups of events with similar characteristics. Relations refer to the relative positions of events in the graph and allow the identification of predecessors and successors of an event. Finally, a macronode is a collection of possibly different event-graph patterns that allow arbitrary complex patterns to be constructed for any imaginable algorithm. As opposed to this approach, the compound-event specification used for EXPERT relies on complex relations based on state information that are suitable for expressing the inefficient behavior of parallel programs on the level of the underlying programming models.

Much work has been done on the visualization of performance data. Apart from standard displays of profiles and event traces, such as Apprentice [14] (Figure 2.3, p. 23) and VAMPIR [3] (Figure 2.2, p. 18), and call-graph–based profile displays, such as Xprofiler [42] (Figure 2.4, p. 24), which have all been described in Section 2.7, very sophisticated performance data displays tried to approach the problem of hiding tool complexity behind simple but still expressive presentation techniques. Solutions range from animated displays, such as those included in ParaGraph [35], to complete virtual reality environments that allow an immersive investigation of the performance space, such as Virtue [68]. However, the emphasis of EXPERT was not the invention of a new display in a technical sense. After all, the use of tree browsers is not revolutionary and even the coloring of nodes in the tree has been previously applied, for example, in the xlcb [13] corefile browser. However, EXPERT shows that an intuitive but still insightful perception of performance behavior can be achieved through uniformity and simplicity both in the logical model of the performance space as well as in its visual representation, which is realized just by coupling standard tree browsers.

The integration of MPI and OpenMP in a single tool has been addressed by other researchers

as well. Hoeflinger et al. [38] integrated the VAMPIR [3] event trace browser with the GuideView [45] OpenMP analyzer to build a new tool VGV for MPI/OpenMP applications. VGV provides a scalable time-line view of an event trace highlighting sections of multithreaded program execution. The user can select individual sections and analyze them using a graphical profile display. Although VGV is not an automatic tool in terms of automatic behavioral classification, it is listed here for its integrated treatment of both programming models. Similarly, the Paraver tool [20] provides trace visualization and quantitative trace analysis of hybrid applications but lacks support for automatic performance-problem detection.

Chapter 7

Summary and Conclusions

The structure of current parallel systems complicates their performance behavior in a way that limits the ability of program developers to reliably predict the performance implications of their design decisions. Complex interactions among multiple layers ranging from sophisticated processor architectures to elaborated communication middleware must be taken into account when "engineering" an application for high performance.

In this context, parallel computers with SMP nodes deserve major interest for two reasons. First, they combine the packaging efficiencies of shared-memory multiprocessors with the scaling advantages of distributed-memory architectures. The result is a computer architecture that can scale more cost-effectively in size. Second, this class of parallel computer architecture captures the two dominant architectures of shared memory and distributed memory as subsets. Its hybrid nature is reflected in different modes of parallel execution (i.e., shared-memory multithreading vs. distributed-memory message passing). As a consequence, performance optimization becomes more difficult and creates a need for advanced performance tools that are able to address this class of computing environments.

This thesis presents a novel approach to analyzing the performance behavior of parallel computers with SMP nodes. The approach is based on automatically transforming event traces of MPI, OpenMP, or hybrid applications onto a higher abstraction level that allows the program developer to identify complex situations of inefficient behavior and to quantify the extent to which they affect the overall performance.

The analysis of performance concentrates on a suboptimal usage of the parallel programming model. Inefficient performance behavior is specified in terms of compound events composed of simple events as contained in the trace file. To simplify their specification, a framework has been developed which offers two different kinds of abstraction that can be used to encapsulate complex programming-model–specific relationships. First, state sequences describe the execution state of an application and provide a convenient means to identify distributed activities, such as collective operation instances, by grouping all events

involved in such an activity. Second, pointer attributes connect single related events and allow the specification of compound events along a path of such events. The resulting specifications serve as input for an automatic analysis process that is responsible for detecting the corresponding compound events in event traces.

In spite of the fact that the framework restricts itself to events and sets of events as the only descriptive means, it is able to describe extraordinarily complex performance problems beyond the capabilities of simple counter-based metrics prevalent in traditional tools. Moreover, by referring only to platform-independent properties of the programming models, the approach is portable across multiple platforms.

The event traces are automatically transformed into a representation called the property-oriented performance space. It is based on the notion of a performance property, which describes a class of performance behavior and constitutes the first dimension of the three-dimensional performance space. The second dimension is the call path and describes both a performance property's source-code location and the execution phase during which it occurs. Finally, the third dimension gives information on the distribution of a performance property across different processes or threads, which allows conclusions to be drawn, for example, concerning the workload balance. A hierarchical organization of each dimension enables the representation of performance behavior on different levels of granularity and, in particular, pays attention to the hierarchical hardware and software structure of parallel computers with SMP nodes. Each point in the performance space is mapped onto the corresponding fraction of execution time, allowing the convenient correlation of different behavior along multiple dimensions using only a single view.

The EXPERT performance-tool demonstrates the usefulness of the approach taken in this thesis. Its multilayer architecture is based on the separation of the performance-property specifications from the actual analysis process. Every property can be accessed through a uniform interface, which allows the extension and customization of predefined properties to meet individual (e.g., application-specific) needs and additional properties to be automatically integrated in the overall representation of performance behavior. In addition, isolating frequently used abstractions (i.e., state sequences and pointer attributes) in a separate layer substantially simplifies the property specification. EXPERT has been successfully applied to several real-world applications.

The main accomplishments of this work are:

- A formal characterization of complex inefficient behavior in terms of compound events that can be automatically detected in event traces.
- Mechanisms that hide the complexity within compound event specifications and, thus, allow a simple description of complex inefficient behavior on a high level of abstraction.

- A specification of common performance problems related to MPI, OpenMP, and hybrid applications based on this method.
- A uniform multidimensional representation of performance behavior which provides the ability to conveniently correlate different behavior in a single integrated view.
- A modular tool architecture which allows the set of predefined performance properties to be extended by the experienced user to meet individual needs.

Future work should focus on extending the MPI event model to include the more recent features of MPI 2, such as parallel file IO, remote memory access, and dynamic process management. Also the remaining issues of OpenMP, such as nested parallelism, should be addressed to achieve a broader coverage of applications.

Moreover, the current set of compound events only refers to temporal relationships among their constituents. However, since various additional metrics, such as hardware performance counters, are easily available using performance-counter libraries, such as PCL [9], an integration of performance counters into the event model might be a promising enhancement in view of the increasingly complex memory hierarchies present in modern microprocessor architectures.

The degree of automation could also be increased by automatically searching the performance space and expanding nodes of interest in the tree display.

Finally, the current shape of the performance space does not cover all possible aspects of performance analysis. Continuing the considerations from Section 4.7.3, a future design might address missing aspects by adding new dimensions for:

- Time
- Application-level abstractions
- Multiple event traces

In particular, adaptive algorithms exhibit a strongly time-dependent performance behavior. This could be reflected in the performance space by partitioning the execution time into fractions associated with different execution phases or iterations of the main loop in the case of iterative applications. Then, it would be possible to show how performance behavior evolves over time.

In addition, instead of only computing the distribution of performance losses across the call tree or across different threads, it would be interesting to exploit the ideas of Shende [69] about application-level instrumentation by extending the current scheme of an event's location with respect to application-level abstractions, such as simulation subdomains.

A very important and challenging extension concerns comparative analysis of different experiments resulting from different execution configurations, from different input-data sets,

or from different program versions. This could provide insight into scalability, into data-dependent behavior, which might also influence the time-dependent behavior, and into the effect of optimizations. Here, the particular challenge lies in the varying structure of the remaining dimensions. For example, the call tree might change as a result of different input data, or the set of locations will become larger when increasing the number of processes. Future research on this specific problem might continue the work of Karanvanic and Miller [46].

Bibliography

[1] D. Amdahl. Validity of the single-processor approach to achieving large scale computer capabilities. In *Proc. of the AFIPS Conference*, pages 483–485, 1967.

[2] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. Data and Computation Transformations for Multiprocessors. In *Proc. of the fith ACM SIGPLAN Symposium on Principle and Practice of Parallel Programming (PPoPP '95)*, pages 134–143, Santa Barbara, 1995.

[3] A. Arnold, U. Detert, and W. E. Nagel. Performance Optimization of Parallel Programs: Tracing, Zooming, Understanding. In R. Winget and K. Winget, editors, *Proc. of Cray User Group Meeting*, pages 252–258, Denver, CO, March 1995.

[4] Accelerated Strategic Computing Initiative [ASCI]. The ASCI sweep3d Benchmark Code. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html.

[5] P. C. Bates. *Debugging Programs in a Distributed System Environment*. PhD thesis, University of Massachusetts, February 1986.

[6] D. M. Beazley. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *Proc. of the 4th Tcl/Tk Workshop*, Monterey, California, July 1996.

[7] D. M. Beazley. *Python Essential Reference*. New Riders, October 1999.

[8] R. Berrendorf, M. Gerndt, and A. Krumme. A Programming Environment for Parallel Computers with Global Address Space. In *Proc. of the Workshop on High-Level Programming Models and Supportive Environments (HIPS), in combination with IPPS*. IEEE, 1996.

[9] R. Berrendorf and H. Ziegler. PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors. Technical Report FZJ-ZAM-IB-9816, Forschungszentrum Jülich, October 1998.

[10] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison Wesley, October 1998.

[11] B. Buck and J. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[12] H. W. Cain, B. P. Miller, and B. J. N. Wylie. A Callgraph-Based Search Strategy for Automated Performance Diagnosis. In *Proc. of the 6th International Euro-Par Conference*, volume 1999 of *Lecture Notes in Computer Science*, Munich, Germany, August/September 2000. Springer.

[13] Parallel Tools Consortium. *xlcb Graphical Browser: User Manual*. Oregon State University. http://cs.oregonstate.edu/ pancake/ptools/lcb/xlcb.html.

[14] CRAY Research, Inc. *Introducing the MPP Apprentice Tool*, 1994. CRAY Manual IN-2511.

[15] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1998. Springer Verlag.

[16] L. A. DeRose, T. H. Hoover, and J. K. Hollingsworth. The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools. In *Proc. of the International Parallel and Distributed Pocessing Symposium*, April 2001.

[17] L. A. DeRose and F. Wolf. CATCH – A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications. In *Proc. of the 8th International Euro-Par Conference*, Lecture Notes in Computer Science, pages 167–176, Paderborn, Germany, August 2002.

[18] T. Diehl and V. Gülzow. Performance of the Parallelized Regional Climate Model REMO. In *Proc. of the Eighth ECMWF Workshop on the Use of Parallel Processors in Meterology*, pages 181–191, Reading, UK, November 1998. European Centre for Medium-Range Weather Forecasts.

[19] A. Espinosa. *Automatic Performance Analysis of Parallel Programs*. PhD thesis, Universitat Autònoma de Barcelona, September 2000.

[20] European Center for Parallelism of Barcelona. *Paraver - Parallel Program Visalization and Analysis tool*, October 2001. http://www.cepba.upc.es/paraver/manuals.htm.

[21] T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, and J. L. Träff. Knowledge Specification for Automatic Performance Analysis. Technical Report FZJ-ZAM-IB-2001-08, ESPRIT IV Working Group APART, Forschungszentrum Jülich, August 2001. Revised version.

[22] T. Fahringer, M. Gerndt, G. Riley, and J. L. Träff. Knowledge Specification for Automatic Performance Analysis. Technical Report FZJ-ZAM-IB-9918, ESPRIT IV Working Group APART, Forschungszentrum Jülich, November 1999.

[23] T. Fahringer and C. Seragiotto Júnior. Modelling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. In *Proc. of the Conference on Supercomputers (SC2001)*, Denver, Colorado, November 2001.

[24] J. Fenlason and R. Stallman. *GNU prof - The GNU Profiler*. Free Software Foundation, Inc., 1997. http://www.gnu.org/manual/gprof-2.9.1/gprof.html.

[25] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[26] Forschungszentrum Jülich. *Solute Transport in Heterogeneous Soil-Aquifer Systems*. http://www.kfa-juelich.de/icg/icg4/Groups/Pollutgeosys/trace_e.html.

[27] Forschungszentrum Jülich. *ZAMpano (ZAM Parallel Nodes)*. http://zampano.zam.kfa-juelich.de/.

[28] F. Freitag, J. Cubet, and J. Labarta. On the Scalability of Tracing Mechanisms. In *Proc. of the 8th International Euro-Par Conference*, Lecture Notes in Computer Science, pages 97–104, Paderborn, Germany, August 2002.

[29] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, MA, 1994.

[30] M. Gerndt, A. Krumme, and S. Özmen. Performance Analysis for SVM-Fortran with OPAL. In *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, pages 561–570, Athens, Georgia, 1995. IEEE.

[31] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. Society for Industrial and Applied Mathematics, 2001.

[32] W. Gropp, E. Lusk, and A. Skjellum. *USING MPI - Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[33] B. Gruber, G. Haring, D. Kranzlmüller, and J. Volkert. Parallel Programming with CAPSE - A Case Study. In *Proc. of the 4th Euromicro Workshop on Parallel and Distributed Programming*, pages 130–137, Braga, Portugal, 1996.

[34] J. Gustavson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

[35] M. T. Heath. Performance Visualization with ParaGraph. In *Proc. Second Workshop on Environments and Tools for Parallel Scientific Computing*, pages 221–230, Philadelphia, 1994.

[36] B. R. Helm and A. D. Malony. Automating Performance Diagnosis: a Theory and Architecture. In *International Workshop on Computer Performance Measurement and Analysis (PERMEAN '95)*, 1995.

[37] HITACHI. *HITACHI SR 8000 Compiler*. Technical Manual.

[38] J. Hoeflinger, B. Kuhn, W. Nagel, P. Petersen, H. Rajic, S. Shah, J. Vetter, M. Voss, and R. Woo. An Integrated Performance Visualizer for MPI/OpenMP Programs. In *Proc. of Workshop on OpenMP Applications and Tools (WOMPAT)*, West Lafayette, July 2001, USA.

[39] J. K. Hollingsworth and M. Steele. Grindstone: A Test Suite for Parallel Performance Tools. Computer Science Technical Report CS-TR-3703, University of Maryland, October 1996.

[40] F. Hoßfeld, K. Solchenbach, C. Bischof, and W. Nagel. Gekoppelte SMP-Syteme im wissenschaftlich-technischen Hochleistungsrechnen (GoSMP). Technical report, Forschungszentrum Jülich, Pallas GmbH, RWTH Aachen, Technische Universität Dresden, February 2000. Study on behalf of the BMBF.

[41] K. Hwang. *Advanced Computer Architecture*. McGraw-Hill, 1993.

[42] IBM Corporation. *IBM AIX Parallel Environment: Operation and Use*, 1996. IBM Corporation publication SH26-7231.

[43] IEEE. Standard 1596 - Scalable Coherent Interface, 1992.

[44] M Jovanovic and V. Milutinovic. An Overview of Reflective Memory Systems. *IEEE Concurrency*, 7(2):56–64, 1999.

[45] KAI Software (a division of Intel Americas). *Guide View Performance Analyzer*, 2001. http://www.intel.com/software/products/kappro/perfvis.htm.

[46] K. L. Karavanic and B. P. Miller. Experiment Management Support for Performance Tuning. In *Proc. of the Conference on Supercomputers (SC97)*, San Jose, California, November 1997.

[47] D. Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, Johennes Kepler Universität Linz, Department for Graphics and Parallel Processing, Austria, September 2000.

[48] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[49] Leibnitz Rechenzentrum der Bayerischen Akademie der Wissenschaften. *HITACHI SR 8000-F1*. http://www.lrz-muenchen.de/services/compute/hlr/#publish1.1.0.0.0.0.

[50] K. A. Lindlan, J. Cuny, A. D. Malony, B. Mohr, R. Rivenburgh, and C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *Proc. of the Conference on Supercomputers (SC2000)*, Dallas, Texas, November 2000.

[51] A. D. Malony. Tools for Parallel Computing: A Performance Evaluation Perspective. In Blasewic et al., editor, *Handbook on Parallel and Distributed Processing*, pages 342–363. 2000. Springer Verlag.

[52] Message Passing Interface Forum. MPI: A Message Passing Interface Standard, Juni 1995. http://www.mpi-forum.org.

[53] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, Juli 1997. http://www.mpi-forum.org.

[54] M. Mihelcic, H. Wenzl, and H. Wingerath. Flow in Czochralski Crystal Growth Melts. Technical Report Jül-2697, Forschungszentrum Jülich, December 1992.

[55] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvine, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, 1995.

[56] D. L. Mills. Network Time Protocol (Version 3). Network Working Group, March 1992. RFC 1305.

[57] B. Mohr. OPARI - OpenMP Pragma And Region Instrumentor. http://www.fz-juelich.de/zam/kojak/opari/. Documentation.

[58] B. Mohr, A. D. Malony, S. S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing*, 23:105–128, 2002.

[59] N. Mukherjee, G. D. Riley, and J. R. Gurd. FINESSE: A Prototype Feedback-guided Performance Enhancement System. In *PDP2000*, January 2000.

[60] Myricom, Inc. Homepage. http://www.myri.com/.

[61] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface - Version 2.0, November 2000. http://www.openmp.org.

[62] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface - Version 2.0, March 2002. http://www.openmp.org.

[63] C. M. Pancake. Applying Human Factors to the Design of Performance Tools. In *Proc. of the 5th International Euro-Par Conference*, number 1685 in LNCS, pages 44–60. Springer, August/September 1999.

[64] Portland Group Inc. *Product Documentation*. http://www.pgroup.com/docs.htm.

[65] R. Rabenseifner. *Die geregelte logische Uhr, eine globale Uhr für die tracebasierte Überwachung paralleler Anwendungen*. PhD thesis, Universität Stuttgart, March 2000.

[66] R. Ribler, H. Simitci, and D. A. Reed. The Autopilot Performance-Directed Adaptive Control System. *Future Generation Computer Systems: Special Issue on Performance Data Mining*, 18(1):175–187, 2001.

[67] G. D. Riley and J. R. Gurd. Requirements for Automatic Performance Analysis. Technical Report FZJ-ZAM-IB-9919, ESPRIT IV Working Group APART, Forschungszentrum Jülich, November 1999.

[68] E. Shaffer, S. Whitmore, B. Schaeffer, and D. A. Reed. Virtue: Immersive Performance Visualization of Parallel and Distributed Applications. *IEEE Computer*, pages 44–51, December 1999.

[69] S. S. Shende. *The Role of Instrumentation and Mapping in Performance Measurement*. PhD thesis, University of Oregon, August 2001.

[70] S. S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proc. of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145. ACM, August 1998.

[71] A. S. Tanebaum. *Computer Networks*. Prentice Hall, third edition, 1996.

[72] University of Wisconsin, Madison, Wisconsin. *Parady Parallel Performance Tools User's Guide*, January 2002. Release 3.3.

[73] J. Vetter. Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficencies. In *Proc. of the 14th International Conference on Supercomputing*, pages 245–254, Santa Fe, New Mexico, May 2000.

[74] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (2nd edition). http://www.w3.org/TR/2000/REC-xml-20001006, October 2000. W3C Recommendation.

[75] F. Wolf. EARL - Eine programmierbare Umgebung zur Bewertung paralleler Prozesse auf Message-Passing-Systemen. Master's thesis, RWTH Aachen, Forschungszentrum Jülich, Jül-Bericht 3551, June 1998.

[76] R. Zinetouline. Clock-Synchronisation auf dem ZAMpano Linux SMP-Cluster. In R. Esser and D. Mallmann, editors, *Beiträge zum Wissenschaftlichen Rechnen - Ergebnisse des Gaststudentenprogramms 2000 des John von Neumann-Instituts für Computing*, number FZJ-ZAM-IB-2000-15, pages 119–126. Forschungszentrum Jülich, November 2000.

Lebenslauf

8.7.1972	Geboren in München
1978-1982	Gemeinschaftsgrundschule Jülich Ost
1982-1991	Gymnasium Zitadelle Jülich - Abitur 1991
1991-1992	Zivildienst beim Malteser-Hilfsdienst Jülich
1992-1998	Studium der Informatik an der Rheinisch-Westfälischen Technischen Hochschule Aachen - Diplom 1998 Ausgezeichnet mit der *Springorum Denkmünze*
1998-1999	Stipendiat des Graduiertenkollegs *Infrastruktur für den elektronischen Markt* Technische Universität Darmstadt
seit 1999	Doktorand bei Prof. Dr. Friedel Hoßfeld am Zentralinstitut für Angewandte Mathematik Forschungszentrum Jülich
2001	Dreimonatiger Forschungsaufenthalt am IBM T. J. Watson Research Center Yorktown Heights, New York, USA

Already published:

Modern Methods and Algorithms of Quantum Chemistry - Proceedings
Johannes Grotendorst (Editor)
NIC Series Volume 1
Winterschool, 21 - 25 February 2000, Forschungszentrum Jülich
ISBN 3-00-005618-1, February 2000, 562 pages
out of print

Modern Methods and Algorithms of Quantum Chemistry - Poster Presentations
Johannes Grotendorst (Editor)
NIC Series Volume 2
Winterschool, 21 - 25 February 2000, Forschungszentrum Jülich
ISBN 3-00-005746-3, February 2000, 77 pages
out of print

Modern Methods and Algorithms of Quantum Chemistry - Proceedings, Second Edition
Johannes Grotendorst (Editor)
NIC Series Volume 3
Winterschool, 21 - 25 February 2000, Forschungszentrum Jülich
ISBN 3-00-005834-6, December 2000, 638 pages

Nichtlineare Analyse raum-zeitlicher Aspekte der hirnelektrischen Aktivität von Epilepsiepatienten
Jochen Arnold
NIC Series Volume 4
ISBN 3-00-006221-1, September 2000, 120 pages

Elektron-Elektron-Wechselwirkung in Halbleitern: Von hochkorrelierten kohärenten Anfangszuständen zu inkohärentem Transport
Reinhold Lövenich
NIC Series Volume 5
ISBN 3-00-006329-3, August 2000, 145 pages

Erkennung von Nichtlinearitäten und wechselseitigen Abhängigkeiten in Zeitreihen
Andreas Schmitz
NIC Series Volume 6
ISBN 3-00-007871-1, May 2001, 142 pages

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Kei Davis, Yannis Smaragdakis, Jörg Striegnitz (Editors)
NIC Series Volume 7
Workshop MPOOL, 18 May 2001, Budapest
ISBN 3-00-007968-8, June 2001, 160 pages

Europhysics Conference on Computational Physics - Book of Abstracts
Friedel Hossfeld, Kurt Binder (Editors)
NIC Series Volume 8
Conference, 5 - 8 September 2001, Aachen
ISBN 3-00-008236-0, September 2001, 500 pages

NIC Symposium 2001 - Proceedings
Horst Rollnik, Dietrich Wolf (Editors)
NIC Series Volume 9
Symposium, 5 - 6 December 2001, Forschungszentrum Jülich
ISBN 3-00-009055-X

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms - Lecture Notes
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
NIC Series Volume 10
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
ISBN 3-00-009057-6, February 2002, 548 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms- Poster Presentations
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
NIC Series Volume 11
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands

ISBN 3-00-009058-4, February 2002, 85 pages

Strongly Disordered Quantum Spin Systems in Low Dimensions: Numerical Study of Spin Chains, Spin Ladders and Two-Dimensional Systems
Yu-cheng Lin
NIC Series Volume 12
ISBN 3-00-009056-8, May 2002, 131 pages

Multiparadigm Programming with Object-Oriented Languages - Proceedings
Jörg Striegnitz, Kei Davis, Yannis Smaragdakis (Editors)
Workshop MPOOL 2002, 11 June 2002, Malaga
NIC Series Volume 13
ISBN 3-00-009099-1, June 2002, 133 pages

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms - Audio-Visual Lecture Notes
Johannes Grotendorst, Dominik Marx, Alejandro Muramatsu (Editors)
NIC Series Volume 14
Winter School, 25 February - 1 March 2002, Rolduc Conference Centre, Kerkrade, The Netherlands
ISBN 3-00-010000-8, November 2002, DVD

All volumes are available online at http://www.fz-juelich.de/nic-series/.