



5-2005

An Infrastructure for the Analysis of Communication Patterns in Virtual Topologies

Nikhil Bhatia

University of Tennessee - Knoxville

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes



Part of the [Computer Sciences Commons](#)

Recommended Citation

Bhatia, Nikhil, "An Infrastructure for the Analysis of Communication Patterns in Virtual Topologies. " Master's Thesis, University of Tennessee, 2005.
https://trace.tennessee.edu/utk_gradthes/590

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Nikhil Bhatia entitled "An Infrastructure for the Analysis of Communication Patterns in Virtual Topologies." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Felix Wolf, Major Professor

We have read this thesis and recommend its acceptance:

Jack Dongarra, Shirley Moore

Accepted for the Council:

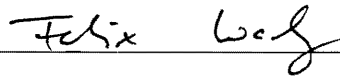
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

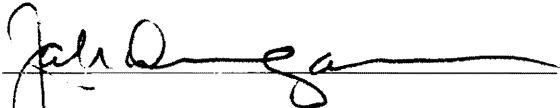
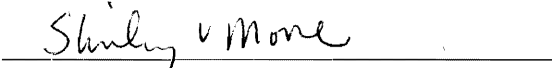
To the Graduate Council:

I am submitting herewith a thesis written by Nikhil Bhatia entitled "An Infrastructure for the Analysis of Communication Patterns in Virtual Topologies". I have examined the final paper copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.



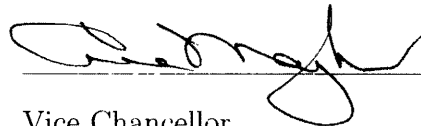
Felix Wolf, Major Professor

We have read this thesis
and recommend its acceptance:


Jack Dongarra
Shirley Moore

Shirley Moore

Accepted for the Council:



Vice Chancellor
and Dean of Graduate Studies

**An Infrastructure for the Analysis of
Communication Patterns in Virtual Topologies**

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Nikhil Bhatia

May 2005

Acknowledgments

This thesis was written at the Department of Computer Science of University of Tennessee, Knoxville. I would like to thank my advisor, Dr. Felix Wolf for giving me the opportunity to work with him and for his equally wise and generous guidance throughout the course of my MS. Also I would like to thank Dr. Shirley Moore for serving as the second referee. In addition, I would like to acknowledge Prof. Dr. Jack Dongarra, Director of the Innovative Computing Laboratory, for his continuous support of my thesis project.

It was great privilege to work with Fengguang Song, I would like to thank him for his constant help and creative influence.

It was a pleasure to carry out my thesis research as a part of the ICL working group KOJAK, whose members contributed many ideas and support and helped me complete this thesis.

Nikhil Bhatia

Abstract

The virtual topology of a parallel application is the neighborhood relationship between communicating processes developed due to specific communication patterns resulting from domain decomposition. We present an infrastructure that allows the usage of topological information for the performance analysis of a parallel application. For this purpose we have implemented an easy to use extension of the KOJAK performance analysis toolkit.

The KOJAK toolkit defines communication patterns for parallel applications which describe inefficient behavior. The performance analysis is carried out by calculating the effect of these inefficiency patterns on the application's performance. The distribution of these inefficiency patterns is studied across a three-dimensional performance space. The knowledge of virtual topology can be exploited to explain the occurrence of these inefficiency patterns in terms of higher-level events related to the parallel algorithm implemented in the application. Also, it can be used to visualize the relationships between pattern occurrences and the topological characteristics of the affected processes. To prove these principles, we have used our extensions to KOJAK to analyze two realistic MPI applications.

Contents

1	Introduction	1
1.1	Architectures of Parallel Computers	1
1.1.1	Distributed-Memory	2
1.1.2	Shared-Memory	2
1.1.3	Hybrid Systems	3
1.2	Programming Models	3
1.2.1	Message-Passing	4
1.2.2	Multi-Threaded	5
1.2.3	Hybrid Model	5
1.3	Performance Analysis and Tuning	5
1.3.1	Performance Data	6
1.3.2	Instrumentation	8
1.3.3	Performance Properties	9
1.4	Virtual Topologies	10
1.5	MPI Topology Support	10
1.5.1	Graph Topology	11
1.5.2	Cartesian Topology	11
1.6	Motivation	12
1.7	Contribution	13
1.8	Outline	14
2	KOJAK	15

2.1	Overall Architecture	15
2.2	Instrumentation and Trace Generation	18
2.2.1	Automated Multi-Level Instrumentation	18
2.2.2	EPILOG Trace Format	19
2.2.3	EPILOG Runtime System	21
2.3	Abstraction Layer	22
2.3.1	Introduction	22
2.3.2	Data Model	23
2.4	Analysis Layer	25
2.4.1	Introduction	25
2.4.2	Analysis Process	26
2.4.3	Hierarchical Organization of Performance Properties	26
2.5	Presentation Layer	28
2.5.1	Data Model	28
2.5.2	Display	30
3	Extensions	37
3.1	Overview	37
3.2	EPILOG	38
3.2.1	Trace Records	38
3.2.2	MPI Wrappers	39
3.2.3	API	40
3.3	EARL	41
3.3.1	Extensions to the class Cartesian	42
3.3.2	Extensions to the class EventTrace	43
3.3.3	Python Methods to Access Topology Information	44
3.4	EXPERT	44
3.5	CUBE	44
3.5.1	CUBE Data Format	44
3.5.2	Topology View	45

4	Examples	49
4.1	Sweep3D	50
4.1.1	Introduction	50
4.1.2	Domain Decomposition and Parallelism	50
4.1.3	Performance Problem in SWEEP3D	52
4.1.4	Topology Analysis with SWEEP3D	53
4.2	TRACE	57
4.2.1	Introduction	57
4.2.2	Performance Problems in TRACE	57
4.2.3	Topology Analysis with TRACE	57
5	Summary and Future Work	59
5.1	Summary	59
5.2	Future Work	60
	Bibliography	61
	Vita	67

List of Tables

2.1	EXPERT performance properties.	29
3.1	ELG_CART_TOPOLOGY definition record	39
3.2	ELG_CART_COORDS definition record	40
4.1	Machines used for experiments.	49

List of Figures

2.1	Overall architecture of the KOJAK system.	16
2.2	The EPILOG file structure.	20
2.3	The EPILOG record structure.	20
2.4	Hierarchy of EARL event types.	24
2.5	Hierarchy of EXPERT performance properties	27
2.6	Visualization of Performance Problems using CUBE.	31
2.7	<i>View</i> menu in the CUBE display.	33
2.8	<i>Region Profile</i> with <i>Relative Percentage</i> in the CUBE display.	34
3.1	Topology Display	46
4.1	Wavefront propagation of data in SWEEP3D	51
4.2	Late-sender pattern in MPI applications.	53
4.3	CUBE results for SWEEP3D	56
4.4	Distribution of late-sender wait states as a result of pipeline refill from North-West	56
4.5	Wait at N x N collective operation	58
4.6	Distribution of wait states caused by inherently synchronizing all-to-all operations in TRACE	58

Chapter 1

Introduction

Parallel computing is an essential paradigm to solve complex scientific problems. Parallel computing has transformed a number of science and engineering disciplines. The architectures for parallel computing have been evolving at a rapid rate. Unfortunately, parallel applications often do not exploit the peak performance of the underlying physical hardware. In parallel computing it is desirable to obtain performance closest to the peak performance of the underlying hardware. Thus, optimizing parallel application behavior is an integral part of the program development process. This optimization, however, is a complex process and requires the knowledge of the underlying architecture, the application's parallelization strategy, and the mapping of the application code and its programming model onto the architecture.

Thus, it becomes essential to innovate new methods of investigating the performance behavior of an application and finding the reasons for limited performance. In parallel applications, the communication between processes and threads has a major impact on their performance. Also, there are common performance patterns observed during this communication. Hence, it becomes imperative to investigate the correlation of communication patterns with the algorithmic details of parallel applications.

1.1 Architectures of Parallel Computers

Parallel computers are computers with multiple processors that are able to work jointly on one or more task at the same time. One common way to classify parallel computers is based on mem-

ory architecture. There are three major classes *distributed-memory*, *shared-memory*, and *hybrid systems*.

1.1.1 Distributed-Memory

Distributed-memory systems have many uniprocessor computers connected by a network. Every processor has a local memory which is often not accessible from another processor. The typical programming model used on such machines consists of separate processes on each computer communicating by sending messages (i.e., message passing).

The most successful commercial distributed-memory system is the IBM SP family. SP systems combine various versions of the successful RS6000 workstation and server nodes with different interconnects to provide a wide variety of parallel systems, from 8 processors to 8192-processor ASCI White system. Some distributed-memory systems have been built with special-purpose hardware that provides remote memory operations such as `put` and `get`. The most successful of these are the Cray T3D and T3E systems.

1.1.2 Shared-Memory

Shared-memory machines have many processors accessing one shared address space and controlled by one operating system image. Data is available to all the processors through the load and store instructions. 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. Memory in shared-address-space machines can be local when it is exclusive to one processor or it can be global when it is common to all processors. If in such machine, all the processors have a symmetric access to one shared-address space, it is referred to as a symmetric multiprocessor (SMP).

The most common problem in shared-memory machines is that of *cache coherence*. Each processor has its own cache and so its possible for a given cache line to be present in more than one cache. If such a line is altered in one cache, then both main memory and the other caches have an invalid version of that line. Ensuring that the memory system is cache coherent requires additional hardware and adds to the complexity of the system.

The complexity of providing cache coherency has led to two new ways of classifying shared-

memory systems. The first important class is called *uniform memory access* (UMA). In this design, each memory and cache is connected to all others. Each component observes any memory operation (such as load from a memory location) and ensures that cache coherence is maintained. The name UMA derives from the fact that the time to access a location from memory (local or global) is identical. Most of the shared-memory systems provided by Compaq, HP, IBM, Sun and SGI are UMA systems.

In non-uniform memory access (NUMA) design, the time taken to access certain memory words is longer than others. NUMA systems that are cache coherent are referred to as cache coherent NUMA (cc-NUMA) systems. Some cc-NUMA systems are the SGI Origin 3000 and the HP Superdome.

1.1.3 Hybrid Systems

There are various ways in which the two memory paradigms are combined. Some distributed-memory machines allow a processor to directly access a datum in a remote memory. These systems are referred to as *distributed-shared memory* systems. On these systems, the latency associated with a load varies with the distance to the remote memory. Cache coherency on DSM systems is a complex problem that is usually handled by a sophisticated network interface unit.

Also, some machines are distributed-memory systems in which each of the individual components is a symmetric multiprocessor rather than a single processor node. Such systems are referred to as *parallel computers with SMP nodes* or *SMP clusters*. This design permits high parallel efficiency within a multiprocessor node, while permitting systems to scale to hundreds or even thousands of processors.

1.2 Programming Models

Another consideration in forming a parallel program is which programming model to use. This decision will affect the programming language and the library for implementing the application. Two programming models were developed to support the two memory models of parallel machines (i.e., distributed-memory and shared-memory models).

The two common programming models used are the *message-passing* model and the *multi-threaded* model. The distributed-memory architectures use the message-passing model that consists

of separate processes on each computer communicating by sending messages. The shared-memory architectures use the multi-threaded model.

Although these two programming models are inspired by the corresponding parallel computer architectures, their use is not restricted. It is possible to implement the multi-threaded model on a distributed-memory computer, either through hardware (distributed-shared memory) or software systems that simulate DSMs (e.g., TreadMarks [9]). Also, message passing can be made to work with reasonable efficiency on a shared-memory system.

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

1.2.1 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 between 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 have to invoke an operation. This can be done using point-to-point communication, where one process sends a message to another process using send and receive operations.

The MPI (Message Passing Interface) communication library [12] defines a defacto standard for message passing and is available on most parallel computers. MPI supports all traditional message passing features, such as point-to-point communication and collective communication, advanced features, such as process topologies. The MPI 2 standard supports features that go beyond pure message passing such as parallel IO and one-sided communication.

More on MPI

MPI is not a new way of programming parallel computers. Rather, it is an attempt to collect the best features of many message-passing systems that have been developed over the years, improve them where appropriate, and standardize them.

MPI is a library for message-passing. It specifies the names, calling sequences, and results of subroutines to be called from Fortran programs, the functions to be called from C programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran C,

and C++ are compiled with ordinary compilers and linked with the MPI library.

The structure of MPI makes it straightforward to port it to existing codes and to write new ones without learning a new set of fundamental concepts.

1.2.2 Multi-Threaded

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

OpenMP (Open specifications for Multi Processing) [14] 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.

1.2.3 Hybrid Model

Coupled SMP systems can be programmed using 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 a 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 application is required.

1.3 Performance Analysis and Tuning

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 desired performance.

The basic performance tuning cycle consists of four steps:

- *Automatic or manual instrumentation.* During this step, measurement probes are inserted into the application code and system software. Measurement probes perform special tasks, such as measurement of hardware performance counters.
- *Execution of the instrumented application and collection of performance data.* Such executions record hardware and software metrics for offline analysis. The recorded data may include profiles, event traces, hardware counter values, and elapsed times.
- *Analysis of the captured performance data.* Manual or automatic analysis uses the recorded data and attempts to relate measurement data to hardware resources and application source code, identifying possible optimization points.
- *Modification of the application source code, recompilation with different optimization options, or modification of run-time parameters.* The goal of these modifications is to better match application behavior, hardware, and the parallel programming interfaces for higher performance.

Apart from offline analysis of an application, online analysis can also be performed. The executable of the parallel application can be instrumented at runtime to obtain the performance behavior of an application. For example, Paradyn [15], leverages a technique called dynamic instrumentation to obtain performance data from unmodified executables at runtime. The following subsections describe these aspects in more detail.

1.3.1 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 in the dynamic call graph are dynamic 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 type of raw observational data are *profiles* and *event traces*.

Profiles

Profiles map accumulated performance metrics (e.g., number of clock cycles, number of function calls, or number of cache misses). For example, a profile may contain the fraction of execution time spent in different functions of the program. 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. 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. 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.

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 a 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 and destination location and the message tag). In order to keep instrumentation simple, the information included in such a 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. The 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 instrumentation 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 timestamps. Systems that rely only on the local clocks have to adjust the timestamps with respect to chronological displacements and clock drifts.

For example, KOJAK performance analysis environment uses event tracing to capture the performance data of a parallel application. The events are written as trace records using a binary format trace file. This trace file is then analyzed offline to build a higher-level callpath profile which gives information about the application's performance.

1.3.2 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 [17] distinguishes three dimensions of classifying instrumentation and measurement:

1. *How* are performance measurements defined and instrumentation alternatives are chosen?
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 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 at run-time. This is also called *dynamic instrumentation*. It can be controlled by feedback, which provides an excellent way of reducing instrumentation.

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 trade-off between the demand for expressive performance data and the desire to avoid program perturbation.

1.3.3 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. [7] 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. For every performance property a *severity* measure is provided, whose magnitude specifies the importance of a 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 a bottleneck is not a performance problem, then the program's performance is considered to be acceptable and does not require any further tuning.

KOJAK [22] is a set of generic and interoperable tool components designed for the performance analysis of parallel applications. Their functionality addresses the entire analysis process including instrumentation, post-processing of performance data, and result presentation. Particular emphasis is put on automation techniques to transform the collected data into a high-level view of performance behavior. As an essential part of the software, KOJAK provides an integrated event-trace analysis environment for MPI and OpenMP applications. KOJAK's trace analysis layer represents performance properties as execution patterns indicating low performance and quantifies them ac-

ording to their severity. These patterns target problems resulting from inefficient communication and synchronization as well as from low CPU and memory performance.

1.4 Virtual Topologies

In many parallel applications, each process (or thread) communicates only with a limited number of other processes. For example, a simulation modeling the spread of pollutants in the atmosphere might decompose the entire simulation domain into smaller process and assign each of those to a single process. Each of these smaller domains is called a *subdomain*.

Given this distribution, a process would then communicate with processes owning sub domains adjacent to its own. The mapping of application domain onto processes and the neighborhood relationship resulting from this mapping is called *virtual topology*. The topological information can be used to map the processes onto the underlying physical topology of the parallel machine for better performance.

Virtual topologies can include processes or threads depending on the programming model being used. Often, the virtual topology influences the order in which certain computations are performed. For example, wavefront algorithms [1] propagate data along the diagonals of a multi-dimensional grid of processes.

In general a virtual topology is specified as a graph. Many applications use Cartesian topologies such as two- or three- dimensional grids. A virtual Cartesian topology is defined as an n -dimensional Cartesian grid. The Cartesian grid may have one or more processes in each dimension. The Cartesian grid may or may not be periodic in each dimension. The coordinates are specified as a vector of integers $0 - (n - 1)$ with n being the number of processes in the respective dimension. The order of the vector elements corresponds to the order of dimensions.

1.5 MPI Topology Support

A process group in MPI is a collection of n processes. Each process in the group is assigned a rank between 0 and $n-1$. In many parallel applications a linear ranking of processes does not adequately reflect the logical communication pattern of the processes (which is usually determined by the un-

derlying problem geometry and the numerical algorithm used). Often the processes are arranged in topological patterns such as two- or three-dimensional grids. As discussed in the previous section, this logical arrangement is known as a *virtual topology*

The MPI standard [12] offers a set of API functions to create and use virtual topologies. The virtual topologies in the MPI standard are referred to as *MPI process topologies*. The MPI process topology support may choose to efficiently map the virtual topology of the application onto the physical topology of the underlying hardware so that communication speeds between neighbors can be optimized.

MPI process topology support also provides a convenient naming scheme for the processes involved in communication, which enables the programmers to name these processes according to a convenient naming scheme. This enhances the readability and simplifies the development of MPI code. There are two types of MPI process topologies.

1.5.1 Graph Topology

MPI process topologies can be generally specified as a graph. Each process is represented by a node in the graph and the communication links between the processes are represented by the edges of the graph.

Some common graph topology functions provided by MPI are:

- `MPI_GRAPH_CREATE` makes a new MPI communicator to which the graph topology information is attached
- `MPI_GRAPHDIMS_GET` returns the number of nodes and edges in the graph

1.5.2 Cartesian Topology

Many MPI applications specify process topologies in terms of a Cartesian grid. Cartesian grids are special type of graphs. Each process in the topology is specified by a coordinate in the Cartesian grid. The Cartesian grid may be periodic in one or more dimensions to specify complex process topologies (e.g., a cylindrical vessel or a torus).

The Cartesian topology is specified by three parameters: the number of dimensions in the Cartesian grid, the number of processes in each dimension and the periodicity of the grid in each dimension.

sion.

Some commonly used Cartesian topology functions provided by MPI are:

- `MPI_CART_CREATE` makes a new MPI communicator to which the Cartesian topology information is attached
- `MPI_CART_COORDS` returns the coordinates of a certain process in the Cartesian grid
- `MPI_CART_RANK` returns the rank of a process which has been assigned to a given coordinate in the Cartesian grid

1.6 Motivation

Searching event traces of parallel applications for patterns is a successful method of automatically generating high-level feedback on an application's performance [22]. This is done by identifying wait states recognizable by temporal displacements between individual events across multiple processes or threads. For example, during message exchange between two MPI processes, the receiving process might enter the *receive* operation before the sending process enters the corresponding *send* operation and hence suffer from a wait state. This is a very common pattern observed in MPI applications and is known as the *Late Sender* pattern.

Topological knowledge can be used to identify and explain the occurrence of performance problems, especially as many algorithms are parametrized in terms of a virtual topology. By doing this, we can study the algorithmic details that correspond to the virtual topology of an application. With more information on the algorithmic details of an application, we may be able to explain the occurrence of certain wait states more clearly.

Topological information has been used earlier to highlight certain aspects of parallel performance. Ahn and Vetter [3] mapped counter data onto the virtual topology of the SWEEP3D ASCII benchmark to identify clusters of related behavior by statistical means.

Müllender [13] visualized different network topologies including four-dimensional hypercubes as well as upto three-dimensional grids and tori using a polygon-like vector representation and mapped certain communication parameters, such as the number of messages, onto their nodes to better observe communication activities in virtual shared memory systems.

Topological knowledge has also been used for semantic debugging of parallel applications. Huband and McDonald [10] describe a trace-based debugger called `DEPICT` that exploits topological information to identify processes with logically similar behavior in traces of MPI applications and to display semantic differences among these groups.

1.7 Contribution

The quality of performance data available has a great influence on the expressiveness of the performance problems that can be detected. Enriching the information contained in event traces with topological knowledge allows the occurrence of certain patterns to be explained in the context of parallelization strategy applied and, thus, significantly raises the abstraction level of the feedback returned.

This thesis presents a framework to map performance data onto the virtual topology of an application. For this purpose, an easy-to-use extension to the `KOJAK` toolkit has been developed. The extension to the instrumentation library provides a means to record the topological information as a part of an event trace. We also provide a means to record the topological information for those applications that don't use the MPI process topology support (e.g., OpenMP applications and many MPI applications). The extension to the analysis component within `KOJAK` provides an abstraction to retrieve and use topology information for performance analysis. The visualization component of `KOJAK` has been extended to visualize the mapping of performance data onto the virtual topology in a simple and comprehensive manner.

Using this extension, we have been able to enhance the quality of the performance analysis process in `KOJAK`. By mapping the performance data onto the virtual topology of the application we can accomplish the following tasks:

- Detect higher-level events related to the parallel algorithm, such as the change in the propagation direction in the wavefront scheme.
- Link the occurrence of patterns that represent undesired wait states to such algorithmic higher level events and, thus, distinguishing wait states by the circumstances causing them.
- Expose the correlation of wait states identified by our analysis with the topological character-

istics of affected processes by visually mapping their severity onto the virtual topology.

To study how the the virtual topology can be used to classify certain wait states, we applied our tool extension to two example MPI codes, the ASCI SWEEP3D benchmark [2] and an environmental science application called TRACE [8]. The results obtained by applying the tool extensions to the ASCI benchmark SWEEP3D gave the performance analysts a better understanding of the occurrence of certain wait states in relationship to the parallelization scheme used. The results obtained from TRACE helped the user identify semantically meaningful clusters of related behavior.

1.8 Outline

The thesis is structured in 5 chapters. Chapter 2 provides an overview of the KOJAK toolkit and its underlying approach of analyzing patterns in event traces. Chapter 3 describes the extensions which have been made to different components of KOJAK toolkit which provide a mechanism to collect topological information in the event trace and visualize performance data mapped onto the virtual topology. Chapter 4 demonstrates the usefulness of this approach in explaining the occurrence of specific patterns in two realistic applications. Chapter 5 summarizes the thesis research and comments on future work in this area.

Chapter 2

KOJAK

This chapter gives an overview of various layers of KOJAK that perform the necessary steps to do automatic performance analysis of parallel applications. KOJAK is a collaborative project between the Central Institute for Applied Mathematics at Forschungszentrum Jülich and the Innovative Computing Laboratory at the University of Tennessee.

2.1 Overall Architecture

KOJAK is a set of generic and interoperable tool components designed for the performance analysis of parallel applications. Their functionality addresses the entire analysis process including instrumentation, post processing of performance data, and result presentation. Particular emphasis is put on automation techniques to transform the collected data into a high-level view of performance behavior. An essential part of the software constitutes an integrated event-trace analysis environment for MPI and OpenMP applications.

Figure 2.1 shows the entire process of analyzing an application using KOJAK. First, the application has to be instrumented at source code or compiler level to get an instrumented executable. This executable is then linked with the EPILOG run-time system which enables the generation of the event trace. After this, the instrumented executable is executed on the given platform to generate the trace file.

The trace file is written in the EPILOG trace format [6], which provides event types covering MPI point-to-point and collective communication as well as OpenMP parallelism change. parallel

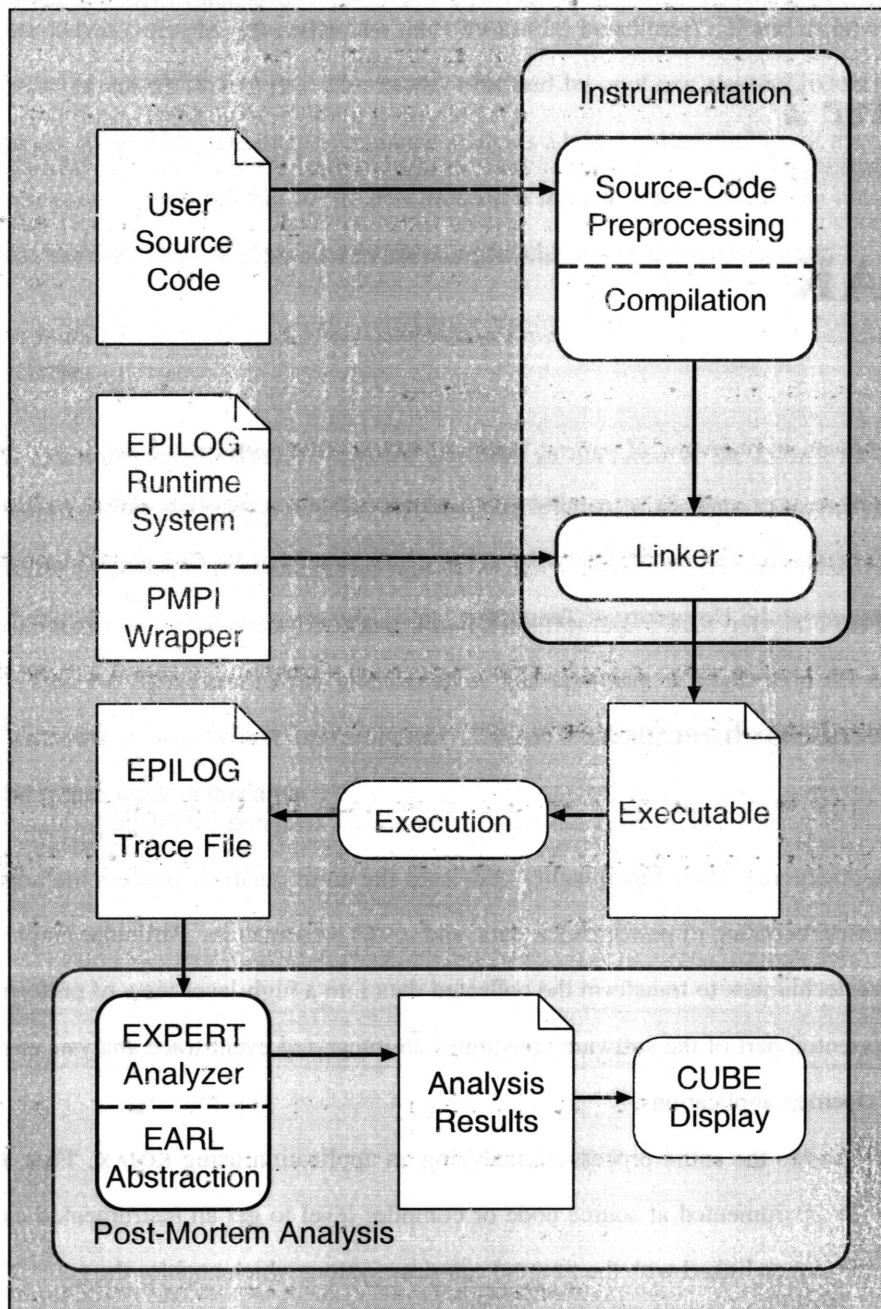


Figure 2.1: Overall architecture of the KOJAK system.

constructs, and synchronization. Also, the trace file may include data from hardware counters.

After program termination, the trace file is analyzed offline using EXPERT [23], which identifies execution patterns indicating low performance and quantifies them according to their severity. These patterns target problems resulting from inefficient communication and synchronization as well as from low CPU and memory performance. The analysis process automatically transforms the traces into a compact callpath profile that includes the time spent in different patterns.

To simplify the analysis, EXPERT accesses the trace through the EARL library interface [21], which provides random access to individual events and precalculated abstractions supporting the search process. EARL is well documented and can be used for a large variety of analysis tasks beyond the analysis performed by EXPERT. The major benefits of using EARL as an intermediate layer between the analysis and the event trace are reduced size and increased readability of the pattern specifications. In EXPERT, patterns are specified separately from the actual analysis process as C++/Python classes. Finally, the analysis results can be viewed in the CUBE performance browser [19].

KOJAK consists of four layers. Each of them performs different tasks at each step in the performance analysis process. These layers are:

- *The instrumentation and trace generation layer* instruments the source-code and generates event traces which can be later used for offline analysis. KOJAK uses the EPILOG (Event Processing, Investigating and Logging) runtime system to generate event traces in EPILOG binary trace format.
- *The abstraction layer* provides random access to the events in an event trace. KOJAK uses EARL (Event Analysis and Recognition Library) to access events from an event trace and thus, simplifies the specification of execution patterns representing performance problems.
- *The analysis layer* uses the analysis layer to convert the low-level trace file into a high-level performance profile. This layer also specifies performance properties described by execution patterns representing performance problems. In KOJAK, the EXPERT trace analyzer does an offline analysis of the event trace and converts it into a high-level callpath profile. It also specifies performance properties represented as hierarchical patterns that describe inefficient communication and low CPU utilization.

- *The presentation layer* reads the high-level performance profile provided by the analysis layer and provides a mechanism to view the effect of performance properties on the application's performance. In KOJAK, the CUBE performance browser is used to view the distribution of performance properties in a three-dimensional performance space.

The following sections describe these layers in more detail. Section 2.2 discusses the instrumentation and trace generation in KOJAK. Section 2.3 describes the abstraction layer in KOJAK, Section 2.4 describes the analysis layer in KOJAK, and section 2.5 describes the presentation layer in KOJAK.

2.2 Instrumentation and Trace Generation

Event tracing provides a very fine grained view of the performance behavior of parallel applications. In contrast to pure execution-time profiling, event tracing preserves the temporal and spatial order of individual events, which may indicate the presence of certain performance properties in an application. KOJAK stores the event traces generated at runtime in the EPILOG binary trace-data format [6].

2.2.1 Automated Multi-Level Instrumentation

Prior to trace generation, the application needs to be instrumented. Depending on the platform, this is done automatically using a combination of source-code preprocessing and compiler-based instrumentation. The various levels at which the application can be instrumented, depending on the choice of platform, are described below.

- *Source Code Level* : Source code instrumentation can be done using the TAU (Tuning and Analysis Utilities) profiling instrumentation [16]. For TAU instrumentation, macros must be added to the source code to identify routine transitions. It can be done automatically using the C++ instrumentor - `tau_instrumentor`, based on the Program Database Toolkit. PDT is used to parse the application and generate a program database file that contains program entities (such as routine locations). The `tau_instrumentor` uses this file and the source code to generate an instrumented version of the source code.

OpenMP applications can be instrumented using OPARI (OpenMP Pragma And Region Instrumentor). OPARI is a source-to-source translation tool that automatically inserts calls to the POMP runtime measurement library. This allows the collection of runtime performance data for Fortran, C or C++ OpenMP applications. POMP directives can also be used to manually instrument the user source-code of OpenMP and non-OpenMP applications.

- *Compiler Level* : Compiler level instrumentation can be performed by using profiling interfaces provided by certain compilers (e.g., PGI compilers on linux platforms).
- *Linker Level* : MPI functions can be instrumented at the linker level. A special library, the PMPI library [12], is used for this purpose. The PMPI library defines all MPI functions with a prefix *PMPI*. The tool developers who want to instrument MPI applications can write their interposition library which contains wrapper functions with prefix *MPI* that perform measurement and make calls to the corresponding PMPI routines. Finally, the MPI application can be linked with the interposition library, the PMPI library, and the MPI library. For example, EPILOG runtime system links MPI applications with the EPILOG interposition library, the PMPI library, and the MPI library.

Also, the application can be linked with hardware counter libraries (e.g., PAPI) to record hardware counter information in the event trace.

- *Binary Level* : Finally, certain platforms allow the usage of libraries that can automatically instrument the executable at various instrumentation points (e.g., function entry and exit). For example, on IBM PowerPC machines, DPCL (Dynamic Probe Class Library) [11] enables automatic instrumentation of the executable at various instrumentation points.

2.2.2 EPILOG Trace Format

The EPILOG (Event Processing, Investigating, and Logging) binary trace data format has been designed to provide a uniform data representation suitable for MPI, OpenMP, and hybrid applications. EPILOG maps events onto their location within the hierarchical hardware as well as to their process and thread of execution. It supports storage of all necessary source-code and call-site information, recording of performance metrics, such as hardware counters, and marking of collectively executed

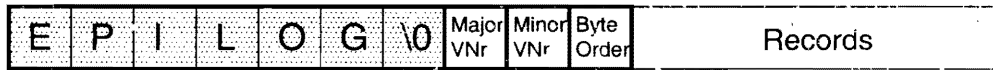


Figure 2.2: The EPILOG file structure.



Figure 2.3: The EPILOG record structure.

operations for both MPI and OpenMP.

File Structure

An EPILOG trace-data file consists of a header followed by a sequence of records. The header (Fig. 2.2) consists of the zero terminated string "EPILOG" followed by two bytes containing the major and minor EPILOG version number and another byte indicating the byte order of the current platform.

Each record (Fig. 2.3) consists of the record header followed by the record body. The header contains two bytes. The first byte contains the length of the record body in bytes without these two leading bytes. The second byte contains the record type.

EPILOG distinguishes between definition records and event records. Definition records define identifiers for objects to be referenced by event records. Event records can be kept small by referencing certain objects instead of specifying these objects as part of the event record. Such objects may be source-code regions or file names. Event records represent runtime events and always contain a location identifier as well as a timestamp.

Definition Records

Definition records deal with the following entities of a parallel application:

- Strings
- Locations
- Source-code entities such as regions and files
- Performance metrics

- MPI communicators.
- Virtual Topologies

Event Records

Event records describe the dynamic program behavior and reference objects that are defined in definition records. By letting event records store only references to those objects, trace file size can be reduced since an object, such as a region, is referenced many times. EPILOG provides records for the following kinds of events:

- Entering and leaving regions
- MPI point-to-point communication
- MPI collective communication
- OpenMP fork and join
- OpenMP parallel execution
- OpenMP lock synchronization
- Tracing events (i.e., events related to the tracing system)

2.2.3 EPILOG Runtime System

As a final step towards trace generation, the application is linked with the EPILOG runtime system, which includes a PMPI interposition library that intercepts MPI calls to perform measurements before and after each call. The EPILOG runtime system writes definition records for program and system resources and event records for dynamic runtime events occurring during one run of an MPI, OpenMP or a hybrid application into a trace file.

The EPILOG runtime system writes all trace records into a buffer to decrease the overhead of trace file generation. Once the buffer is filled, the runtime system dumps the buffer into the file and starts writing the buffer again. There are special tracing events that record the *buffer full* and *buffer empty* events. For more details please refer to the EPILOG specification [6].

In MPI applications, each process maintains a local event trace that contains the definition records related to the resources used by the process and the event records for the events occurred during the execution of that process. After program termination, these local event traces are merged together to form a single global event trace that contains an aggregate of all the definition records from the individual trace files and synchronized event records for the entire run of the application.

In OpenMP applications, threads have their local event traces which are merged together after program termination to form a global event trace. Hybrid applications have to go through a two-level merging of trace files. The first level merges per-thread local traces to form a per-process trace. The second level merges the local per-process traces to form a single global trace file.

2.3 Abstraction Layer

EARL is a high-level interface for accessing EPILOG event traces and can be used to write advanced trace-analysis software. EARL provides random access to single events and computes the execution state at the time of a given event as well as links between pairs of related events. EARL is implemented in C++ and offers both C++ and Python class interfaces.

2.3.1 Introduction

An event trace is a chronologically sorted sequence of runtime events recorded during program execution that can be used to analyze program behavior. In the KOJAK performance-analysis environment [22], event traces are used to identify patterns of inefficient execution.

EARL offers the following functionality:

- Random access to single events
- Access to the execution state at the time of a given event
- Links between pairs of related events
- Access to virtual topologies.
- Various statistical functions

EARL can be used for a large variety of trace-analysis tasks. The main purpose of EARL within KOJAK is to simplify the specification of execution patterns representing performance problems within the EXPERT analyzer [23] and, thus, to allow easy extension and customization of the pattern base used in the analysis process. The first prototype of EARL was completed in 1998 as part of a master's thesis [21].

2.3.2 Data Model

EARL (Event Analysis and Recognition Library) is based on a simple object-oriented data model whose simplicity is derived from the fact that all higher-level abstractions, such as execution states and links between related events, are expressed in terms of event sets or event references, thus never leaving the familiar notion of an event.

Abstractions

The central abstraction in EARL is an *event*. Every event has a type, a timestamp, and a location, which answers the questions what happened, when it happened, and where it happened, respectively. In addition, an event may provide type-specific attributes including links to related events.

The program resources represented in an event trace include files, regions, and call sites. The system resources associated with an event trace form a hierarchy consisting of machines, nodes, processes, and threads. *Machines* can be made up of multiple (potentially SMP) *nodes*. Each node can host multiple *processes*, which in turn can spawn multiple *threads*. This model mirrors one or more parallel computers with SMP nodes and can also accommodate more traditional non-SMP, single-SMP, or simple desktop architectures. An event *location* is a tuple consisting of a machine, a node, a process, and a thread. A location is basically a thread that includes information on the process, the node, and the machine it is associated with. A single-threaded process always has one explicit thread because in EARL the thread level is mandatory. Essentially the event location represents a thread in KOJAK. A system resource on the other hand can be a node, a process, or a thread.

There are other special types of resources included in an event trace like MPI *communicators* and *virtual topologies*. Also, some events may store the values of certain system *metrics*, such as the

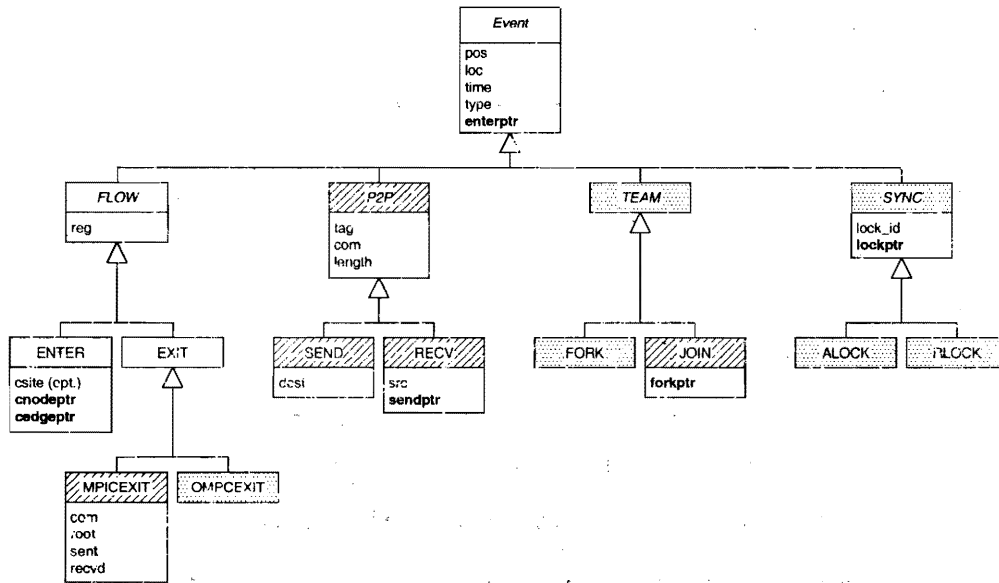


Figure 2.4: Hierarchy of EARL event types.

number of floating point operations executed. A metric may represent a count of event occurrences (e.g., from a hardware counter) across an interval, an occurrence rate measured across an interval, or the current value of a metric, such as the current memory utilization.

Event Model

The event model is defined by a hierarchy of abstract and concrete event types, which is shown in Figure 2.4 using UML notation [4]. Abstract event types do not appear in the event trace, they are used only to isolate commonalities in the model. In the figure, abstract event types have been distinguished by writing the type names in italics. The arrows illustrate an inheritance relationship with respect to the type attributes, that is, an event type inherits all attributes from its ancestors. Hatched boxes represent MPI-specific types, whereas spotted boxes represent OpenMP-specific types.

As shown in the figure, events are arranged in a multi-level hierarchy. For example, the events marking the exit of an MPI collective communication operation and the end of an OpenMP parallel construct are at the same level in the hierarchy and are children of the same parent that represents the region EXIT events. Also, MPI SEND and RECV events are children of the same parent that represents the MPI point-to-point communication events.

Higher-Level Abstractions

EARL provides the following two abstractions which are useful to easily identify related events.

- State sequences
- Pointer attributes

State sequences map individual events onto a set of events that represent one aspect of the parallel system's execution state at the moment when the event happens. This allows complex events to be described in the context of the execution state. For example, EARL maintains a region (call) stack for every location. The initial stack is empty. Whenever an ENTER event occurs, it is added to the stack and whenever an EXIT event occurs, the corresponding ENTER event is removed from the stack.

Pointer attributes connect two corresponding events with one another, so that one can define compound events along a path of corresponding events. For example, the attribute *sendptr* points from a RECV event to the corresponding SEND event.

2.4 Analysis Layer

EXPERT describes performance problems using a high level of abstraction in terms of common situations that result from an inefficient use of the underlying programming model(s).

2.4.1 Introduction

In EXPERT, the analysis is carried out along three interconnected dimensions: class of performance behavior (i.e., performance properties), position within the dynamic call-tree, and a location (e.g., node or process). Each dimension is arranged in a hierarchy, so that the user can view the behavior on varying levels of detail. The comprehensive behavioral classification used by EXPERT provides the ability to explain problems intelligibly in terms of common situations that result from non-optimal usage of the programming model to which they are related. In addition, it is possible to integrate application specific classifications by using appropriate extension mechanisms.

EXPERT uses EARL to access the events in an event trace. Its architecture is based on the idea of separating the analysis process from the specification of properties representing performance problems in parallel applications. In EXPERT, the performance properties are arranged in a hierarchical distribution of execution patterns representing inefficient behavior. The Python version is mainly used for prototyping.

2.4.2 Analysis Process

The performance properties are specified in the form of patterns. Patterns are C++/Python classes that are responsible for detecting compound events indicating inefficient behavior. They provide a common interface making them exchangeable from the perspective of the tool. The specifications use the abstractions provided by EARL and, for this reason, are very simple.

The analysis process follows an event-driven approach. EXPERT walks sequentially through the event trace and invokes call-back methods for each single event to pattern instances, supplying the event as an argument. A pattern can provide a different callback method for each event type. The call-back method itself then tries to locate a compound event representing an inefficiency, thereby following links (i.e., pointer attributes) emanating from the supplied event or investigating system states. This mechanism allows the simple specification of very complex performance relevant situations and an explanation of inefficiency that is very close to the terminology of the programming model.

2.4.3 Hierarchical Organization of Performance Properties

EXPERT organizes the performance properties in a hierarchy. The upper levels of the hierarchy (i.e., those that are closer to the root) correspond to more general behavior aspects such as time spent in MPI functions. The deeper levels correspond to more specific situations such as time lost due to blocking communication. Figure 2.5 shows the hierarchy of predefined performance properties that are supported by EXPERT.

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 used on summary information involving, for example the total execution times of special MPI routines, which could

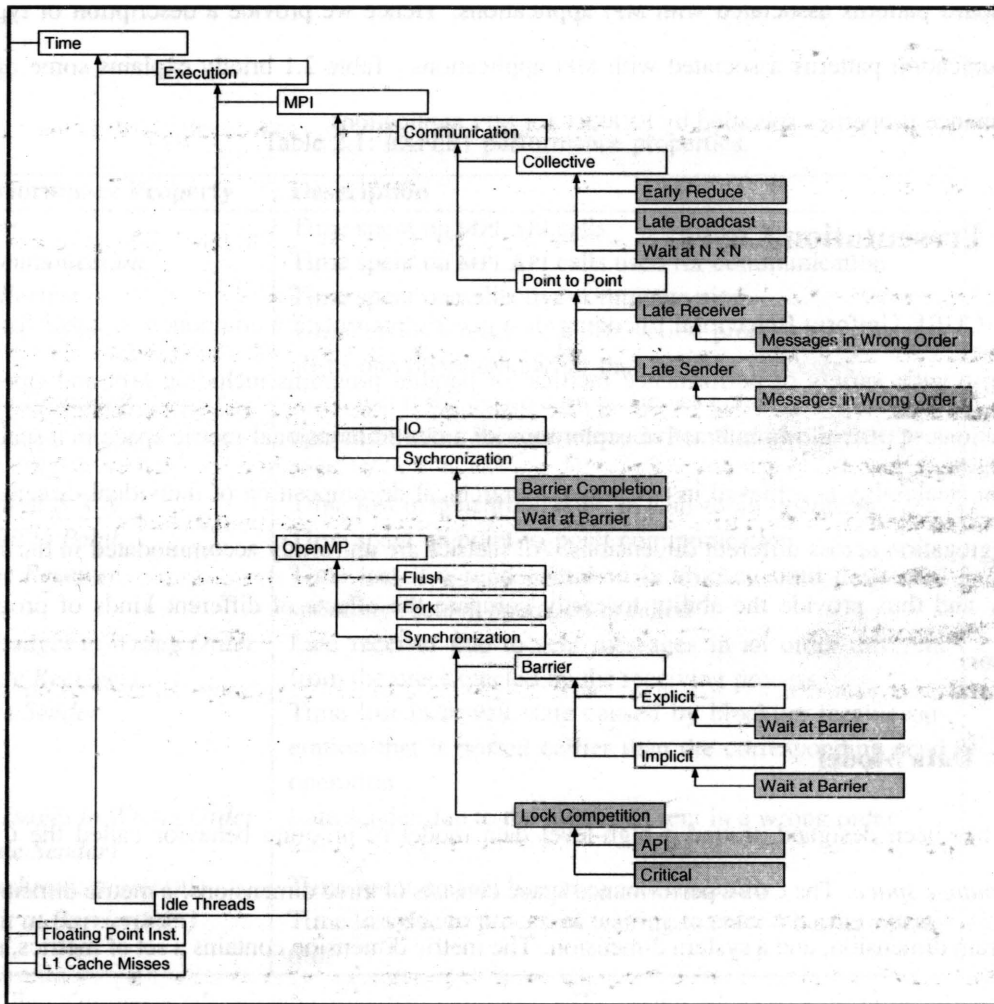


Figure 2.5: Hierarchy of EXPERT performance properties

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 information between individual events. We have studied two MPI applications to demonstrate the usefulness of this work. We have used the topological information to explain the occurrence of certain wait states in these applications. These wait states occur due to standard patterns associated with MPI applications. Hence we provide a description of typical communication patterns associated with MPI applications. Table 2.1 briefly explains some of the performance properties specified by EXPERT for MPI applications.

2.5 Presentation Layer

CUBE (CUBE Uniform Behavioral Encoding) is a generic presentation component suitable for displaying a wide variety of performance metrics for parallel programs including MPI and OpenMP applications. CUBE allows interactive exploration of a multidimensional metric space in a scalable fashion. Scalability is achieved in two ways: hierarchical decomposition of individual dimensions and aggregation across different dimensions. All metrics are uniformly accommodated in the same display and thus provide the ability to easily compare the effects of different kinds of program behavior.

2.5.1 Data Model

CUBE has been designed around a high-level data model of program behavior called the CUBE *performance space*. The CUBE performance space consists of three dimensions: a metric dimension, a program dimension, and a system dimension. The metric dimension contains a set of metrics, such as communication time or cache misses. The program dimension contains the program's call tree, which includes all the call paths onto which metric values can be mapped. The system dimension contains all the system resources of the program, which can be processes or threads depending on the parallel programming model. Each point (m, c, l) of the space can be mapped onto a number representing the actual measurement for metric m while the control flow l was executing call path c . This mapping is called the *severity* of the performance space.

Each dimension of the performance space is organized in a hierarchy. First, the metric dimen-

Table 2.1: EXPERT performance properties.

Performance Property	Description
<i>MPI</i>	Time spent on MPI API calls
<i>Communication</i>	Time spent on MPI API calls used for communication
<i>Collective</i>	Time spent on collective communication
<i>Early Reduce</i>	Time lost as a result of a destination process entering in a all-to-one operation earlier than sending processes
<i>Late Broadcast</i>	Time lost if the destination processes entering in one-to-all operation enter the operation earlier than the source process
<i>Wait at N X N</i>	Time lost in synchronization in a all-to-all operation
<i>Point to Point</i>	Time spent on point-to-point communication
<i>Late Receiver</i>	Time lost if a send operation is blocked until the corresponding receive operation is called
<i>Messages in Wrong Order (Late Receiver)</i>	Late receiver due to sent messages in an order different from the one expected by the receiving process
<i>Late Sender</i>	Time lost in a wait state caused by blocking receive operation that is posted earlier than the corresponding send operation
<i>Messages in Wrong Order (Late Sender)</i>	Late sender due to the messages sent in a wrong order
<i>Synchronization(MPI)</i>	Time spent on MPI barrier synchronization
<i>Wait at Barrier(MPI)</i>	Time lost due to processes waiting to enter a barrier operation

sion is organized in an inclusion hierarchy where a metric at a lower level is a subset of its parent, for example, communication time is below execution time. Second, the program dimension is organized in a call-tree hierarchy. Flat profiles can be represented as multiple trivial call trees consisting only of a single node. Finally, the system dimension is organized in a multi-level hierarchy consisting of the following levels: machine, SMP node, process, and thread.

2.5.2 Display

The CUBE display consists of three tree browsers, each of them representing a dimension of the performance space (Figure 2.6). The left tree displays the metric dimension, the middle tree displays the program dimension, and the right tree displays the system dimension. The nodes in the metric tree represent metrics. The nodes in the program dimension can have different semantics depending on the particular view that has been selected. In Figure 2.6, they represent call paths forming a call tree. The nodes in the system dimension represent machines, nodes, and processes from top to bottom.

Users can perform two types of actions: selecting a node or expanding/collapsing a node. At any time, there are two nodes selected, one in the metric tree and the other in the call tree. It is currently not possible to select a node in the system tree.

Each node is associated with a metric value, called the *severity*, which is displayed using a numerical value as well as a colored square. Colors enable the easy identification of nodes of interest even in a large tree, whereas the numerical values enable the precise comparison of individual values. The *severity* value of a metric describes the relative importance of the metric with respect to the other metrics used in the analysis process. When a node is in a collapsed state, it displays the inclusive numerical value of the severity of the pattern it represents. That is, it displays the aggregate sum of the exclusive severity for that node and the severities of all its children nodes. On the other hand, when the node is in an expanded state, it displays the exclusive value of the severity of the pattern it represents.

The color is taken from a spectrum ranging from blue to red representing the whole range of possible values. To avoid an unnecessary distraction, insignificant values close to zero are displayed in dark gray. Exact zero values just have the background color. Depending on the severity representation, the color legend shows a numeric scale mapping colors onto values.

The user can switch from the call-tree mode (default) to the module-profile mode or the region-profile mode. In this case, the user can switch from the call-tree mode (default) to the module-profile mode or the region-profile mode. In this case, the user can switch from the call-tree mode (default) to the module-profile mode or the region-profile mode. In this case, the user can switch from the call-tree mode (default) to the module-profile mode or the region-profile mode.

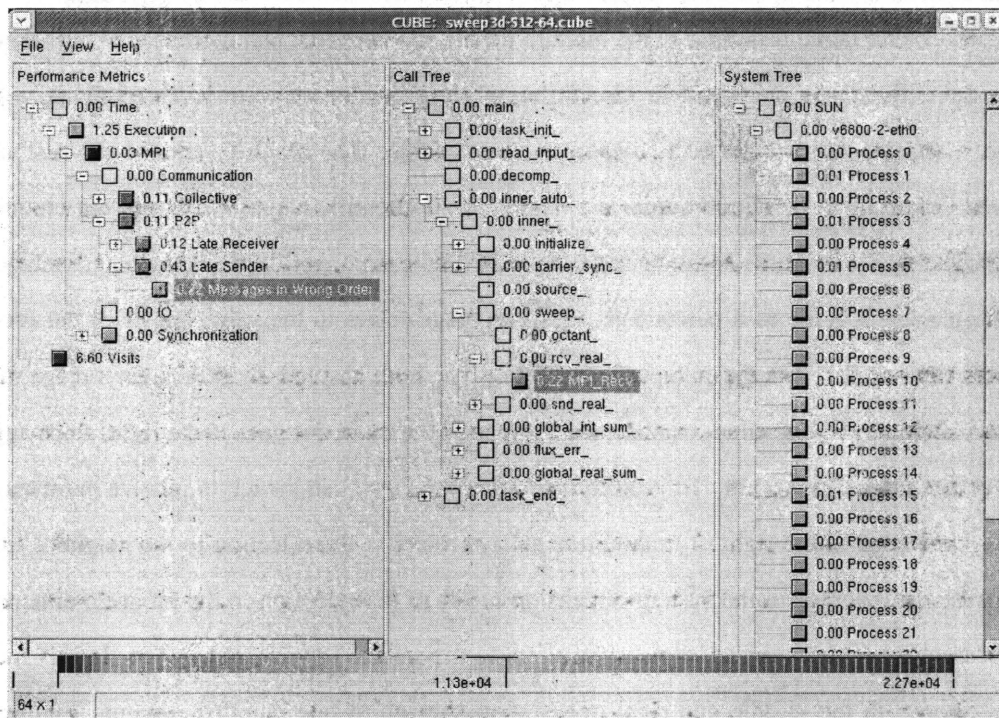


Figure 2.6: Visualization of Performance Problems using CUBE.

The view menu (Figure 2.7) in the display can be used to alter the way the program dimension is displayed, to change the number representation for the entire display, or to hide positive or negative values.

After opening a data set, the middle panel shows the call tree of the program. However, a user might wish to know which fraction of a metric can be attributed to a particular region regardless of from where it was called. In this case, the user can switch from the call-tree mode (default) to the module-profile mode or the region-profile mode. In the module-profile mode, the call-tree hierarchy is replaced with a source-code hierarchy consisting of three levels: module, region, and subregions. The region-profile mode is similar to the module-profile mode except that modules are not shown.

The severity can be displayed in four different ways: as an *absolute value* (default), a *percentage*, a *relative percentage*, or a *comparative percentage*. The absolute value is the real value measured. In absolute mode, all values are displayed in scientific notation. To prevent cluttering the display, only the mantissa is shown at the nodes with the exponent displayed at the color legend. When displaying a value as a percentage, the percentage refers to the value shown at the root of the metric tree when it is in the collapsed state. However, both absolute mode and percentage mode have the disadvantage that values can become very small the more one goes to the right, since aggregation occurs from right to left. To avoid this problem, the user can switch to relative percentages. Then, a percentage in the right or middle tree always refers to the selection in the neighbor to the left, that is, a percentage in the system dimension refers to the selection in the program dimension and a percentage in the program dimension refers to the selected metric dimension. In this mode, the percentages in the middle and right tree always sum up to one hundred percent. Figure 2.8 shows a region profile with relative percentages. Furthermore, to facilitate the comparison of different experiments, users can choose the comparative percentage mode to display percentages relative to another data set. The comparative percentage mode is basically like the normal percentage mode except that the value equal to 100% is determined by another data set.

If one or more virtual topologies have been defined in the CUBE file, the *Topology* menu item is enabled. Otherwise it is disabled. After selecting *Topology*, the topology-selection dialog pops up if the CUBE file has multiple topologies. Through this dialog, users can choose a specific topology to be displayed in a topology window. Each topology is displayed in a separate window. The topology display is described in more detail in Section 3.5.2. For further information about using the CUBE

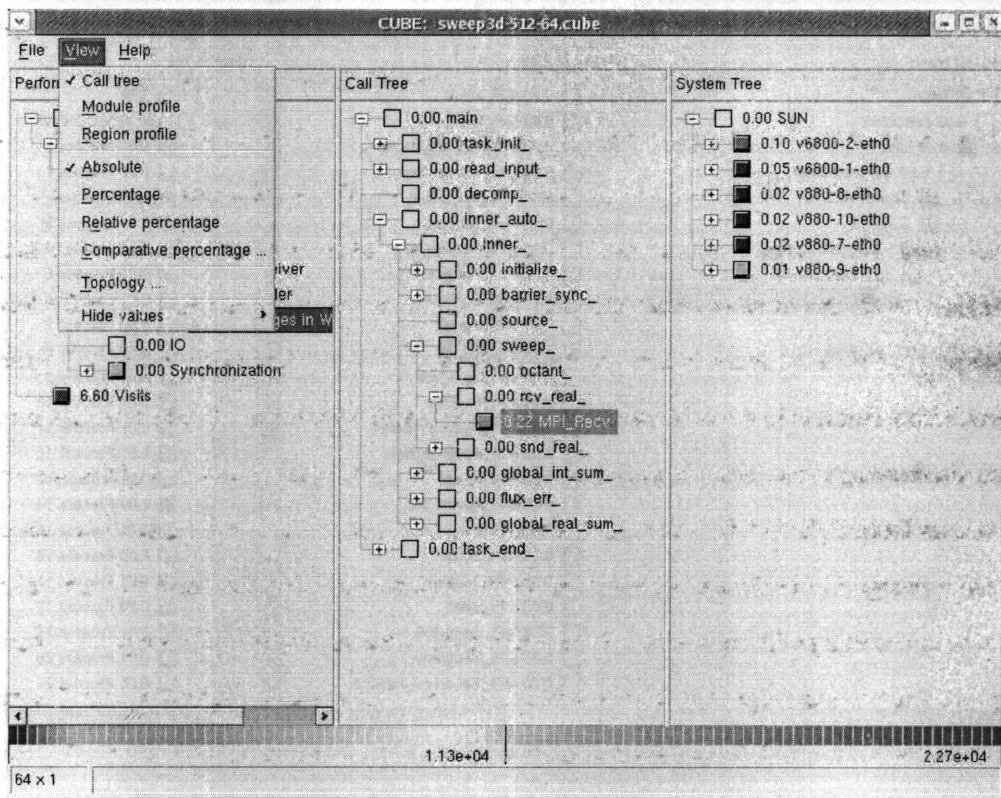


Figure 2.7: View menu in the CUBE display.

The view menu (Figure 2.7) in the display can be used to alter the way the program information is displayed, to change the number representation for the entire display, or to hide positive or negative values.

After opening a data set, the middle panel shows the call tree of the program. However, a user might wish to know which fraction of a metric can be attributed to a particular region regardless of how often it was called. In this case, the user can switch from the call-tree mode (default) to the module-profile mode or the region-profile mode. In the module-profile mode, the call-tree hierarchy

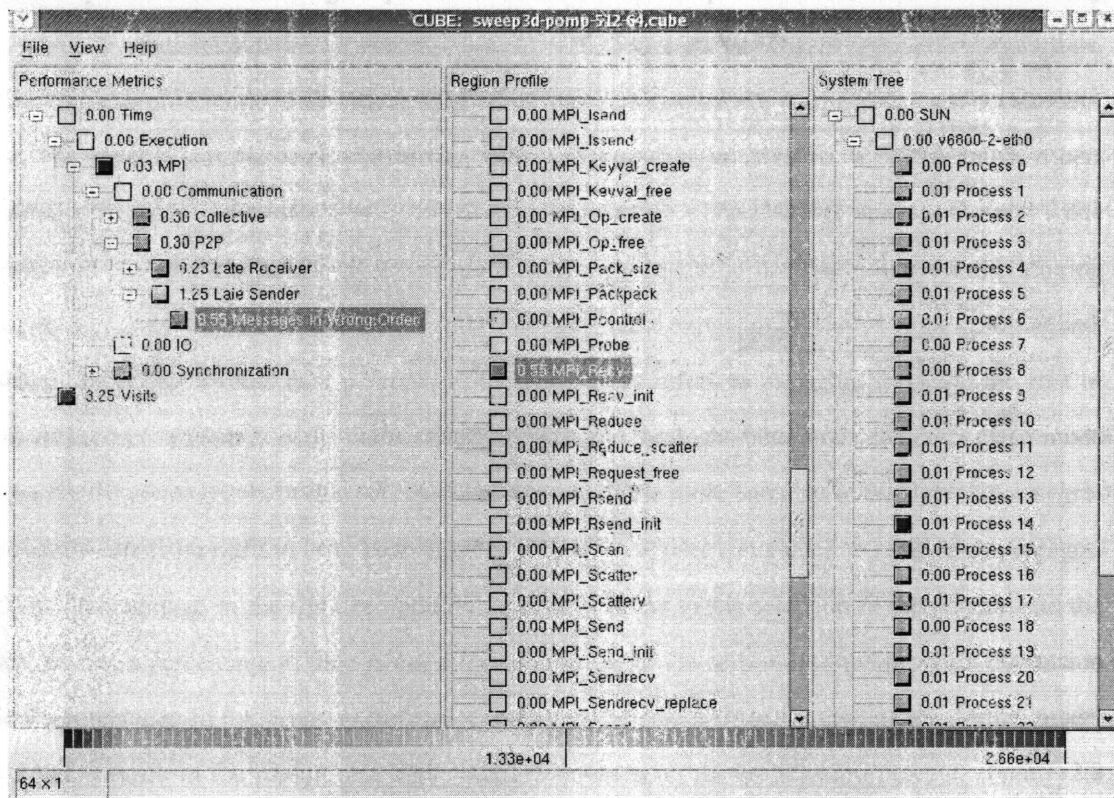


Figure 2.8: Region Profile with Relative Percentage in the CUBE display.

shows a region profile with relative percentages. Furthermore, to facilitate the comparison of different data sets, the comparative percentage mode is basically like the normal percentage mode except that the value equal to 100% is determined by another dataset.

If one or more virtual topologies have been defined in the CUBE file, the *Topology* menu item is enabled. Otherwise it is disabled. After selecting *Topology*, the topology-selection dialog pops up if the cube file has multiple topologies. Through this dialog, users can choose a specific topology to be displayed in a topology window. Each topology is displayed in a separate window. The topology display is described in more detail in Section 3.5.2. For further information about using the CUBE

display please refer to [18].

Chapter 3

Extensions

3.1 Overview

To make the analysis process topology-aware, various extensions have been made to the KOJAK toolkit. The EPILOG trace format has been extended to provide definition records to include the topology-specific information in the trace file. The EPILOG run-time system has been extended to automatically record MPI process topology information in applications that utilize the MPI process topology support. Also, an instrumentation API has been provided for C/C++ and Fortran applications to setup and use virtual topologies. This extends the usage of virtual topologies for performance analysis to applications that do not utilize the MPI topology support (e.g., OpenMP applications and many MPI applications). EARL has been extended to access topology information from the event trace and provide an abstraction to map the performance data onto the virtual topology of the application. The EXPERT analyzer has been extended to transfer the topology information from the event trace to the CUBE data format which can be viewed using the CUBE performance browser. The CUBE data format has been extended to record topology information, and the CUBE GUI has been extended to incorporate a new window to visualize the distribution of performance data across the virtual topology in a user-friendly way.

To keep the extension simple, we restricted ourselves to Cartesian topologies as a common case found in many of today's parallel applications.

The following sections describe the extensions made to all the layers of KOJAK to use topological information for the performance analysis of parallel applications. Section 3.2 describes the

extensions made to EPILOG. Section 3.3 describes the extensions made to EARL. Section 3.4 describes the extensions made to EXPERT. Finally, section 3.5 describes the extensions made to CUBE.

3.2 EPILOG

Two new definition records have been added to the EPILOG binary trace format to define Cartesian topology-specific information. Provision has been made to automatically record the topology information of applications using the MPI topology support. An instrumentation API has been provided for those applications that do not utilize MPI process topology support.

3.2.1 Trace Records

The EPILOG trace format consists of definition records that define various system and program resources available to the application in the current programming environment. Two definition records have been added to the trace format to define the Cartesian topologies. One record type to define the general layout of a Cartesian topology and one to map a system resource (e.g., a process or a node) onto a particular position within a previously defined topology. The semantics of the topology can be arbitrary and the records can be used to declare a virtual or a physical topology.

The two definition records added are as follows:

- ELG_CART_TOPOLOGY
- ELG_CART_COORDS

ELG_CART_TOPOLOGY

This record defines an identifier `topid` for a Cartesian topology. The record also defines an identifier `cid` of the MPI communicator representing the topology if it was created using MPI. It is set to a special value when MPI process topology support is not used. `topid` can be used to uniquely identify a topology in this case. The record specifies the number of dimensions `ndims` of the Cartesian grid and includes a vector `dimv[]` of size `ndims` containing the number of system resources in each dimension and a vector `periodv[]` of size `ndims` specifying whether the grid is periodic

Table 3.1: ELG_CART_TOPOLOGY definition record

Data Type	Attribute	Description
elg_ui4	topid	Cartesian topology identifier
elg_ui4	cid	Communicator identifier
elg_ui1	ndims	Number of dimensions in the Cartesian grid
elg_ui4	dimv[ndims]	Number of locations in each dimension
elg_ui1	periodv[ndims]	Periodicity of the grid in each dimension

in each dimension or not. To specify the periodicity of the grid in a particular dimension, each entry of `periodv` must carry one of the two symbolic constants:

- ELG.TRUE
- ELG.FALSE

Table 3.1 summarizes the various fields of this definition record.

ELG_CART_COORDS

This record specifies the coordinates of a system resource in a Cartesian topology. It contains the topology identifier `topid` of the Cartesian topology to which the coordinates refer, a location identifier `lid` for each system resource, the number of grid dimensions `ndims`, and a vector `coordv` of size `ndims` containing the coordinates of the system resource.

The Table 3.2 explains the various fields of this definition record.

3.2.2 MPI Wrappers

As discussed in the previous chapter, EPILOG uses the PMPI library to instrument MPI applications. The PMPI library generates MPI-specific events by intercepting calls to MPI functions. These events then call the EPILOG run-time library, which provides methods for buffering and writing the definition and event records to the trace file.

The PMPI wrapper function for the `MPI_CART_CREATE` routine uses the function parameters to this MPI function to write the `ELG_CART_TOPOLOGY` definition record. After processing the

Table 3.2: ELG_CART_COORDS definition record

Data Type	Attribute	Description
elg_ui4	topid	Cartesian topology identifier
elg_ui4	lid	Location identifier
elg_ui1	ndims	Number of dimensions in the Cartesian grid
elg_ui4	coordv[ndims]	Coordinates of the system resource

topology outline, the wrapper requests the coordinates of the calling process from the MPI runtime system and writes the corresponding ELG_CART_COORDS definition record.

3.2.3 API

Most parallel applications, rarely use MPI process topology support. Also, there is no special support for virtual topologies in OpenMP applications. For these reasons, EPILOG provides a C/C++ and Fortran API to write the topology specific definition records. The API consists of two functions that allow the definition of upto three dimensional Cartesian topologies. Using this API is fairly simple and requires only minimal effort.

The two functions in C/C++ start with a prefix `elg` whereas in Fortran these functions start with a prefix `elgf`. The functions are implemented in C and the Fortran functions are wrappers around their C implementations.

The two functions in the API are explained below.

- `elg(f)_cart_create(size0, size1, size2, period0, period1, period2)` : This function allows the user to setup a Cartesian topology. The number of dimensions of the Cartesian grid can be at most three. $size_X$ is an integer describing the number of system resources in dimension X . $period_X$ is the integer describing the periodicity in dimension X . Its value is zero if the Cartesian grid is non-periodic in that dimension. This function is usually called once. If more than one system resource calls this function for a given topology then the multiple calls will be redundant. The merge component ensures that only one definition record per topology is written to the global trace file.
- `elg(f)_cart_coords(coord0, coord1, coord2)` : This function allows the calling system

resource to define its coordinates in a previously defined Cartesian grid. $coord_X$ is an integer describing the coordinate of a location in dimension X . The range of $coord_X$ is $[0, size_X - 1]$ where, $size_X$ is the number of system resources in dimension X . This function must be called exactly once by every system resource that is a part of the Cartesian topology.

The following example defines a three-dimensional $4 \times 4 \times 4$ topology that is periodic in the first but not in the remaining two dimensions.

```
if (rank .eq. 0) then
  call elgf_cart_create(4,4,4,1,0,0)
endif
call elgf_cart_coords(x,y,z)
```

Every process executing these lines assigns itself coordinates defined through the variables x , y , z , containing values between 0 and 3.

3.3 EARL

The EARL abstraction layer has been extended to access the topological information from the event trace and to create an abstraction that maps the performance data onto the Cartesian topology of the application.

The event trace contains timestamped events that describe the dynamic program behavior. EARL can access information about the type, time and location of occurrence of an event. This information can be used to collect the performance data of the application which in turn can be used to simplify the specification of execution patterns representing performance problems in the EXPERT analyzer.

With topological knowledge at our disposal, we can map the performance data onto the virtual topology of the application. This information can be utilized to identify higher-level algorithmic events related to the parallelization scheme applied in the parallel algorithm. Given this knowledge, occurrences of certain wait states can be explained more clearly.

The classes *Cartesian* and *EventTrace* were extended to provide member functions to access topological information from an event trace. Also, member functions were provided that can convert the exact system resource where a event occurred to its corresponding coordinate in the Cartesian

grid. This abstraction helped map the occurrence of every event to a coordinate in the Cartesian grid and thus, map the performance data onto the topology of the application. Following extensions were made to the EARL library:

- Extensions to the class `Cartesian`, and
- Extensions to the class `EventTrace`

3.3.1 Extensions to the class `Cartesian`

The Cartesian topology is a special type of resource in the application and there are two types of definition records in the trace file to define this resource. The primary purpose of EARL is to access and process EPILOG event traces and thus, to simplify the specification of execution patterns representing performance problems within the EXPERT analyzer. Therefore, the class `Cartesian` has been added to EARL to access topology specific information from the event trace.

This class contains data that records the topology information and provides member functions to access this information through an object of this class. Following is a list of these member functions:

- `long get_id()`: Returns the unique Cartesian topology identifier.
- `Communicator* get_com()`: Returns the MPI communicator that represents the processes defining this topology. It is set to `NULL` when MPI process topologies are not used.
- `long get_ndims()`: Returns the number of dimensions in the grid.
- `void get_dimv(std::vector<long>& out)`: Returns in `out` the number of locations in each grid dimension. Note that the size of `out` is equal to `ndims`.
- `void get_periodv(std::vector<bool>& out)`: Returns in `out` the periodicity in each grid dimension. Note that the size of `out` is equal to `ndims`. A boolean value of `true` indicates that the dimension is periodic.

The member functions to conveniently convert system resources to coordinates in the Cartesian grid and vice-a-versa are:

- `void get_coords(std::vector<long>& out, Location* loc)`: Returns in `out` the coordinates of the location `loc` in the Cartesian grid. Note that the size of `out` is equal to `ndims`.
- `Location* get_loc(std::vector<long>& coordv)`: Returns the location corresponding to the coordinates represented by `coordv` in the Cartesian grid.

3.3.2 Extensions to the class `EventTrace`

This class primarily provides random access to all events in the trace file including the execution state at the time of a given event. This class also provides information on program and system resources involved in the program execution. The event trace can contain more than one Cartesian topology. Hence, a couple of routines were added here to access multiple Cartesian topologies in the event trace. These routines are the following:

- `long get_ncarts()`: Returns the total number of Cartesian topologies.
- `Cartesian* get_cart(long cart_id)`: Returns the Cartesian topology with identifier `cart_id`.

Two member functions have been added to directly access system resource from the coordinates and vice-a-versa.

- `void get_coords(std::vector<long>& out, long cart_id, long loc_id)`: Returns in `out` the coordinates of the location with identifier `loc_id` in the Cartesian topology with identifier `cart_id`. The coordinates are specified by the vector `out` in the order of dimensions (i.e., first dimension first, etc.).
- `Location* get_loc(std::vector<long>& in, long cart_id)`: Returns the location at given coordinates in the Cartesian topology with identifier `cart_id`. The coordinates are specified by the vector `in` in the order of dimensions (i.e., first dimension first, etc.).

3.3.3 Python Methods to Access Topology Information

The Python API is a wrapper around the C++ API that has been generated using SWIG [20]. The main advantage of the Python interface is that it enables rapid prototyping as well as interactive programming. The existing typemaps have been extended to also make the Python interface topology-aware.

3.4 EXPERT

The EXPERT analyzer uses the EARL library interface to access the event trace. It then utilizes the low-level trace-record information to create a high-level callpath profile representing the performance of an application. This profile is based on the specification of hierarchical patterns representing performance properties of an application.

The EXPERT analyzer has been extended in two ways. First, its been enabled to read the topological information from the event trace and record it in the high-level model.

Secondly, new patterns have been added to analyze a specific algorithm using topological information onto which the performance data have been mapped. These patterns will be discussed in the next chapter when we describe the results of our analysis while investigating wavefront processes in the ASCI benchmark SWEEP3D.

3.5 CUBE

The CUBE data format has been extended to include the topological information. A topology view has been added to the CUBE GUI. This extension of the GUI can be used to view the distribution of performance data across the Cartesian virtual topology of the application.

3.5.1 CUBE Data Format

New XML elements have been added to the CUBE data format to represent the topological information in the CUBE file.

The `<topologies>` element marks the beginning of topology-specific information in the CUBE file. The `<cart>` element defines a one, two, or three-dimensional Cartesian topology. The `<dim>` element defines the total number of system resources in each dimension. The `<coord>`

element maps each system resource to its coordinate in the Cartesian grid. The following example shows a typical representation of a three-dimensional Cartesian topology which is non-periodic in all three dimensions and has two processes in each dimension

```
<topologies>
  <cart ndims="3">
    <dim size="2" periodic="FALSE"/>
    <dim size="2" periodic="FALSE"/>
    <dim size="2" periodic="FALSE"/>
    <coord locId="0">0 0 0</coord>
    <coord locId="1">0 0 1</coord>
    <coord locId="2">0 1 0</coord>
    <coord locId="3">0 1 1</coord>
    <coord locId="4">1 0 0</coord>
    <coord locId="5">1 0 1</coord>
    <coord locId="6">1 1 0</coord>
    <coord locId="7">1 1 1</coord>
  </cart>
</topologies>
```

3.5.2 Topology View

If one or more virtual topologies have been defined in the CUBE file, the *Topology* menu item is enabled. Otherwise it is disabled. After selecting *Topology*, the Cartesian-selection dialog pops up if the CUBE file has multiple topologies. Through this dialog, users can choose a specific topology to be displayed in the topology view. Each topology can be displayed in a separate view.

If the CUBE file contains topological information, the distribution of the performance metric across the topology can be examined using the CUBE topology view. The CUBE topology view shows performance data mapped onto the Cartesian topology of the application. The corresponding grid is specified by two parameters: the number of dimensions and the size of each dimension

Figure 3.1 show the menu bar and the actual Cartesian grid. The Cartesian grid is presented by

statement reads each system resource and returns the following:

shows a 3D plot representation of a three-dimensional, periodic topology which is now ready to be used in the Python interface. The existing topologies have been extended to also make the Python interface more user-friendly.

3.4 EXPERT

The EXPERT analysis uses the same interface as the other analyses. The EXPERT analysis uses the same interface as the other analyses.

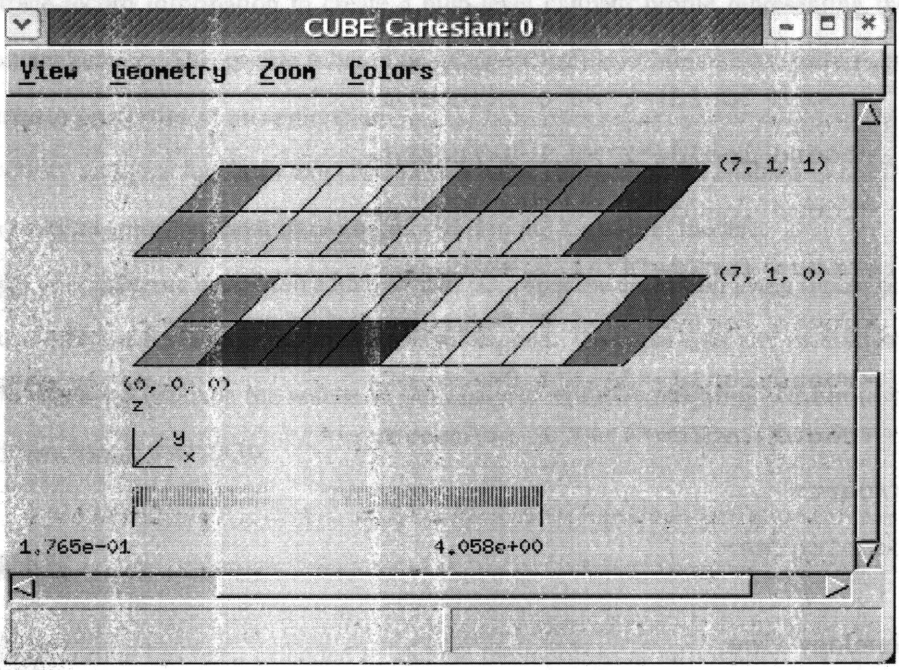


Figure 3.1: Topology Display

3.5.1 CUBE Data Format

to be displayed in the topology view. Each topology can be displayed in a separate view.

with the cube. The user can obtain a separate view of the topology. The user can also obtain a separate view of the topology. The user can also obtain a separate view of the topology.

planes stacked on top of each other in a three dimensional projection. The number of planes depends on the size of the *Z* dimension. Each plane is divided into squares. The number of squares depends on the dimension size. Each square represents a system resource (e.g a process) of the application and has a coordinate associated with it.

The grid displays the severity of the selected metric in the selected call path for each system resource participating in the application's topology. The severity is represented as a color. A system resource might not be a part of the application's virtual topology or may have a zero value for a metric.

Menu Bar

The menu bar consists of four menus: a view menu, a geometry menu, a zoom menu and, a colors menu.

View: The view menu can be used to choose one of the three possible orientations of the grid.

The coordinate axes at the bottom of the picture indicate the direction of the *X*, *Y* and *Z* dimensions in the three-dimensional space. In the case of one- or two- dimensional grids, users are provided with only one orientation of the grid.

Geometry: Due to varying dimension sizes, planes in the grid might overlap with each other and the size of the squares might be too small to recognize their color. This may pose a problem for the user to view the topology information effectively. The geometry menu circumvents this problem by providing options to scale the picture in various ways. The *Angle* option helps the user to adjust the skew of the three-dimensional projection. The *Plane Distance* option helps to adjust the inter-plane distance. The *Plane Length* option helps users scale the edge length of each plane.

Zoom: The zoom menu can be used to zoom-in or zoom-out on the grid.

Colors: The colors menu can be used to modify the text color and the background color of the topology display. Finally, there are two resolution modes to choose from. The *Low Resolution* mode assigns colors to the squares according to the severity values shown in the system dimension (Figure 2.8, rightmost tree browser). The relation between colors and the

corresponding values have been described in section 2.5.2. Often, these values have small variations from each other and do not help user to study the relative distribution of severities across the grid. As described in the last chapter, the CUBE color spectrum ranges from blue to red representing the whole range of possible values. To exploit the entire spectrum of available colors and to enable the user to study the relative distribution of severities, a *High Resolution* mode is provided. This mode highlights the minute differences between severity values of the system resources. Severity values of zero are assigned the background color of the display. This mode has its own color legend showing the minimum and maximum values for the selected severities across the grid. These values can be absolute values, percentages, or relative percentages depending on the CUBE view mode.

Chapter 4

Examples

We have designed an infrastructure that enables the identification of higher-level algorithmic events related to the parallelization scheme applied in a parallel algorithm. We believe that this infrastructure can be used to study the relation between certain inefficient patterns and these higher-level algorithmic events. Also, this infrastructure can be used to identify clusters of system resources that show semantically similar behavior due to their position in the virtual topology of the application. This infrastructure has been developed by extending the KOJAK performance analysis toolkit as described in the previous chapter.

To study how virtual topology can be used to accomplish the above mentioned goals, we have applied our tool extensions to two example MPI codes, the ASCI SWEEP3D benchmark [2] and an environmental science application called TRACE [8].

This chapter focuses on the results derived from these experiments and thus, shows proof of concept that mapping performance data onto the virtual topology of a parallel application can help better understand the performance behavior of these applications.

All the experiments were conducted on different parallel computers. Table 4.1 summarizes

Table 4.1: Machines used for experiments.

Name	Location	CPU description	OS
Jump	FZJ, Germany	IBM Power4+	AIX
Galaxy	Houston, USA	UltraSparcIII 750 MHz	SunSolaris9
y Beowulf	Houston, USA	Intel PentiumIII Xeon 550MHz	Linux
Copper	UIUC, USA	IBM Power4	AIX

the machines used for our experiments. Various event traces were collected for both applications with different configurations of the Cartesian grids used to define the virtual topology of these applications.

4.1 Sweep3D

The first example is the ASCI benchmark SWEEP3D. This example shows how the topology-specific information in the event trace can be utilized to identify higher-level algorithmic events in an application. Also, it demonstrates the way in which these higher-level events can be used to explain a specific performance problem in SWEEP3D. For our analysis, we extended the hierarchical patterns described in the EXPERT analyzer to incorporate four new sub-patterns of the late-sender pattern. This extension was necessary to explain the relation of late-sender wait states to the specific algorithm used in SWEEP3D.

4.1.1 Introduction

The benchmark code SWEEP3D is an MPI program performing the core computation of a real ASCI application. It solves a 1-group time-independent discrete ordinates (Sn) 3D Cartesian geometry neutron transport problem by calculating the flux of neutrons through each cell of a three-dimensional grid (i, j, k) along several possible directions (angles) of travel. The angles are split into eight octants, each corresponding to one of the eight directed diagonals of the grid.

4.1.2 Domain Decomposition and Parallelism

SWEEP3D exploits parallelism via a wavefront process. First, it maps the (i, j) planes of the three-dimensional domain onto a two-dimensional grid of processes. Thus, SWEEP3D has a two-dimensional Cartesian virtual topology. SWEEP3D uses a wavefront algorithm to do its computation. The eight octants of the three-dimensional cubic domain result in wavefronts originating from the four corners of the two-dimensional Cartesian grid. That is, the direction of the wavefront, at any given time, depends on the octant being processed at that time.

To improve parallel efficiency, blocks of work are pipelined through the domain. The parallel computation follows a pipelined wavefront process that propagates data along diagonal lines through

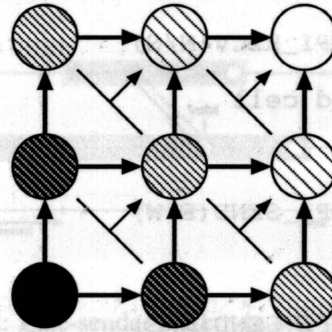


Figure 4.1: Wavefront propagation of data in SWEEP3D

the grid. Responsible for the wavefront computation in the code is a subroutine called `sweep()`, which initiates wavefronts from all four corners of the two-dimensional grid of processes. The wavefronts are pipelined to enable multiple wavefronts to follow each other along the same direction simultaneously.

Figure 4.1 shows the data dependence graph for a 3×3 array. This figure illustrates the propagation of wavefronts initiated at the South-West (i.e., bottom-left) corner of the two-dimensional grid. In this example, each process is data-dependent on its Western and Southern neighbors. The long, bold arrows symbolize data dependencies. The processes that are to the North-West and South-East (i.e., diagonally aligned with a process) of a process are algorithmically independent with respect to that process. In the figure, diagonal lines cut through algorithmically independent processes. The diagonal arrows toward North-East (i.e., top-right) represent the computation as it progresses in the form of wavefronts from the lower-left to the upper-right corner. Thus, the parallelization in SWEEP3D is based on concurrency among algorithmically independent processes and pipelining among algorithmically dependent processes.

The basic code structure of routine `sweep()` in pseudo-code notation is as follows:

```
DO octants
  DO angles in octant
    DO k planes
      ! block i-inflows
      IF neighbor(E/W) MPI_RECV(E/W)
```

```

! block j-inflows
IF neighbor(N/S) MPI_RECV(N/S)
    ... compute grid cell ...
! block i-outflows
IF neighbor(E/W) MPI_SEND(E/W)
! block j-outflows
IF neighbor(N/S) MPI_SEND(N/S)
END DO k planes
END DO angles in octant
END DO octants

```

It can be seen from the pseudo-code that in the innermost loop, each process executes an `MPI_RECV()` to get data from the neighbors it algorithmically depends on. Then, the process performs the required computation and sends the result to the neighbors which depend on it. This is done by the two `MPI_SEND()` calls at the end of the loop.

4.1.3 Performance Problem in SWEEP3D

SWEEP3D suffers from wait states due to the late-sender pattern. This pattern is explained in the Figure 4.2. This pattern is caused due to two communicating processes where one process sends a message to another process. However, the receiver process might enter the receive operation earlier than the corresponding sender process enters the send operation. Therefore, it has to wait until the sender actually sends the message. This is an undesirable wait state in which the receiver waits for the sender to send the message without doing anything useful.

In SWEEP3D this pattern occurs frequently because of the parallelization scheme applied. It can be seen from the `sweep()` pseudo-code that each process has to execute two `MPI_RECV()` calls to get data from the processes on which it depends for its data. The direction of the wavefront changes depending on the octant being processed. When the direction of the pipeline changes (e.g., from North-East to South-West), a pipeline refill takes place starting from a different corner. During the pipeline refill, every process except the process which initiates the wavefront, has to wait for data from its algorithmically dependent neighbors before it can start its computation. Thus, these

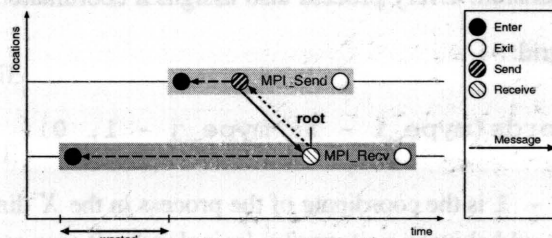


Figure 4.2: Late-sender pattern in MPI applications.

processes will suffer from late-sender waiting times. The process at the corner reached by the wavefront last incurs most of the waiting times, whereas the process where the wavefront originates incurs none. The change in the pipeline direction can be a major contributor to the late-sender patterns observed in SWEEP3D.

4.1.4 Topology Analysis with SWEEP3D

To perform topology analysis with SWEEP3D, a three-dimensional problem domain of size $512 \times 512 \times 150$ was chosen. This three-dimensional domain was decomposed into a two-dimensional grid of processes. The size of the grid was 8×8 (i.e., 64 processes). The instrumentation of the user functions was done fully automatically using the platform compiler's profiling interface.

Setting up a two-dimensional Cartesian topology

SWEEP3D does not utilize the functions provided by MPI topology support to setup and use virtual topologies. Therefore, we used the Fortran routines of our instrumentation API (explained in the previous chapter) to define a two-dimensional Cartesian topology with 8 processes in each dimension. The domain decomposition is done in the file `decomp.f`. We inserted calls to the topology API in `decomp.f`.

The following Fortran call sets up the two-dimensional Cartesian topology which is non-periodic in all dimensions:

```
call elgf_cart_create(npe_i, npe_j, 0, 0, 0, 0)
```

The variable `npe_i` is the number of processes in the X dimension and `npe_j` is the number

of processes in the Y dimension. Every process also assigns a coordinate to itself according to its position in the Cartesian grid:

```
call elgf_cart_coords(mypr_i - 1, mypr_j - 1, 0)
```

The variable `mypr_i - 1` is the coordinate of the process in the X dimension and `mypr_j - 1` is its coordinate in the Y dimension. The coordinates are calculated in `decomp.f` depending on the MPI rank of the calling process and the total number of processes in the X dimension.

After instrumenting all user functions of SWEEP3D fully automatically, using the compiler specified profiling interface, it was executed with 64 processes on a Solaris Cluster equipped with UltraSPARC-III 750 MHz processors. The execution yielded a trace file enriched with topological information. This trace file was analyzed offline using the EXPERT analyzer.

EXPERT Analysis

The EXPERT analyzer specifies execution patterns that symbolize performance problems in parallel applications. Processes in SWEEP3D incur wait states due to the late-sender pattern which can be attributed to the pipelined wavefront algorithm. The direction changes of wavefronts that are initiated at the four corners of the two-dimensional Cartesian grid are higher-level events related to the parallelization scheme of this algorithm.

Now, according to our discussion above, SWEEP3D incurs waiting times due to the pipeline direction change of wavefronts originating from the four corners of the Cartesian grid. A significant percentage of the waiting times incurred due to the late-sender pattern in SWEEP3D can be attributed to these waiting times. Thus, four new sub-patterns of the late-sender pattern were added to the EXPERT hierarchy of performance patterns.

The EXPERT pattern hierarchy can be easily extended to include new patterns because of its flexible publish and register scheme. In this scheme, pattern classes can publish detected pattern instances and new pattern classes can register for instances detected by others. For example, the late-sender pattern class can publish the detected pattern instances. The new sub-pattern classes can then register for the published late-sender pattern instances. The new sub-patterns in SWEEP3D are named as follows:

1. Wavefront from NW

2. Wavefront from SW
3. Wavefront from NE
4. Wavefront from SE

Each sub-pattern class uses the topological information provided by EARL to track the pipeline direction. If there is a change in the pipeline direction simultaneously with a late-sender instance, the instance becomes also an instance of that sub-pattern. For example, if the pattern *Wavefront from NW* finds out that the direction of the pipeline changed from some other direction to North-West, then that instance of the late-sender becomes an instance of the sub-pattern *Wavefront from NW* as well. To identify the pipeline direction change, EXPERT maintains a FIFO queue for each process which records the directions of the most recent messages received by the process. Finally, EXPERT records the wait times in the high-level callpath profile which can be viewed using the CUBE performance browser.

CUBE Display

The high-level callpath profile provided by EXPERT can be viewed with the CUBE performance browser. Figure 4.3 shows the percentages of late-sender instances caused by the four new sub-patterns. The new patterns appear in the metric tree on the left underneath the late-sender pattern and are labeled with the percentage of execution time spent in wait states caused by them. The total time spent in wait states, which can be obtained by collapsing the late-sender node, was 25.4%. *Late Sender* instances observed simultaneously with a pipeline direction change account for about a little less than 60% of the overall late-sender time. The time measured for individual directions vary between 6.0% from North-West and 1.7% from pipeline refill from North-East.

Figure 4.4 shows the new topology view rendering the distribution of the late-sender times for pipeline refill from North-West. The high-resolution mode shows the relative distribution of severities of the selected property with respect to each other. As discussed above, the corner reached by the wavefront last incurs most of the waiting times, whereas the origin of the wavefront incurs none.

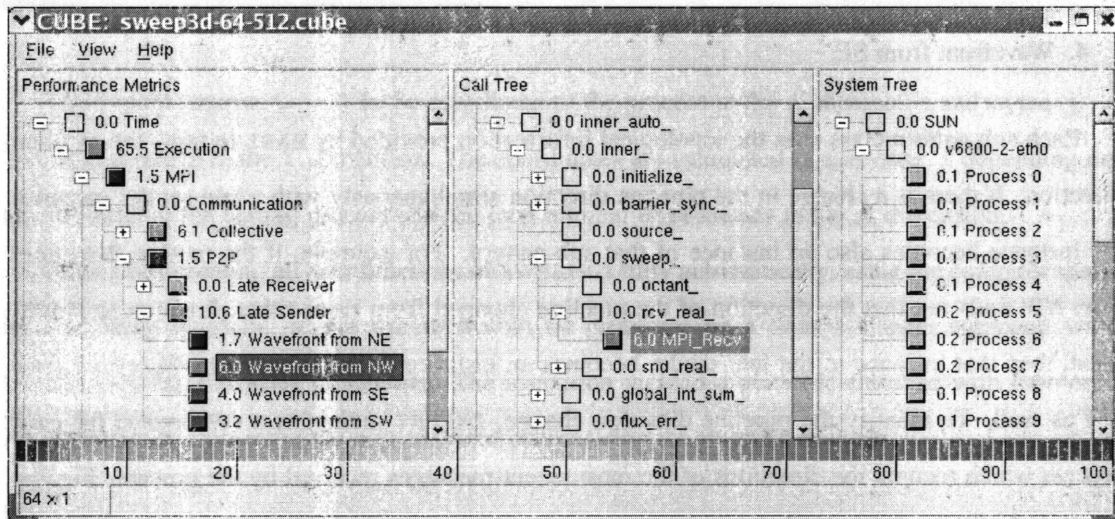


Figure 4.3: CUBE results for SWEEP3D

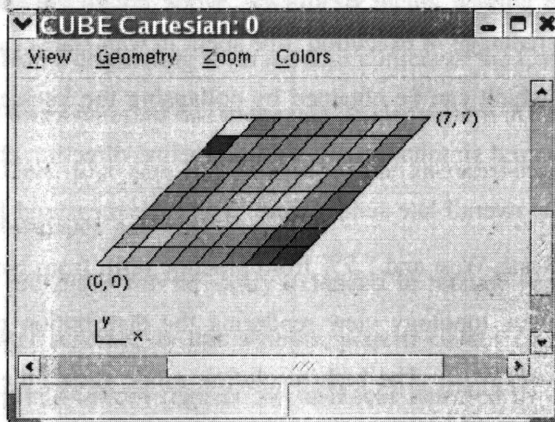


Figure 4.4: Distribution of late-sender wait states as a result of pipeline refill from North-West

4.2 TRACE

The second example highlights how visually mapping the results of our pattern analysis onto the virtual topology can help the user identify semantically meaningful clusters of related behavior.

4.2.1 Introduction

TRACE [8] simulates the subsurface water flow in variably saturated porous media. It solves the generalized Richards equation in three spatial dimensions. The parallelization is based on a parallelized CG algorithm, which divides the grid into overlapping subgrids and communicates via MPI. The main computation is done in a subroutine called `parallelcg()`.

We executed the application with 64 processes on a IBM cluster with 41 Power4+ 1.7GHz 32-way nodes. The resulting topology is a three-dimensional Cartesian $16 \times 2 \times 2$ grid (Figure 4.6).

4.2.2 Performance Problems in TRACE

TRACE suffers from wait states caused by inherently synchronizing all-to-all operations that occur when some processes enter the operation earlier than others. The pattern describing this situation is among the standard patterns included in the EXPERT analyzer. The pattern is illustrated in Figure 4.5. The top three processes enter the synchronizing operation before the last process. Thus, the top three processes suffer from waiting times until the last process has reached the operation.

4.2.3 Topology Analysis with TRACE

Most of the computation and MPI communication in TRACE takes place in the routine `parallelcg()`. Figure 4.6 shows the distribution of wait states in `parallelcg()` caused by inherently synchronizing all-to-all operations. The figure exhibits clusters of increased waiting times at the corners of the three-dimensional grid. These processes, due to their exposed location are assumed to have different computation as well as communication requirements. Without topological knowledge the affected processes would appear as arbitrary processes and the user would be unaware of the correlation between their particular role in the topology and the occurrence of specific inefficiencies.

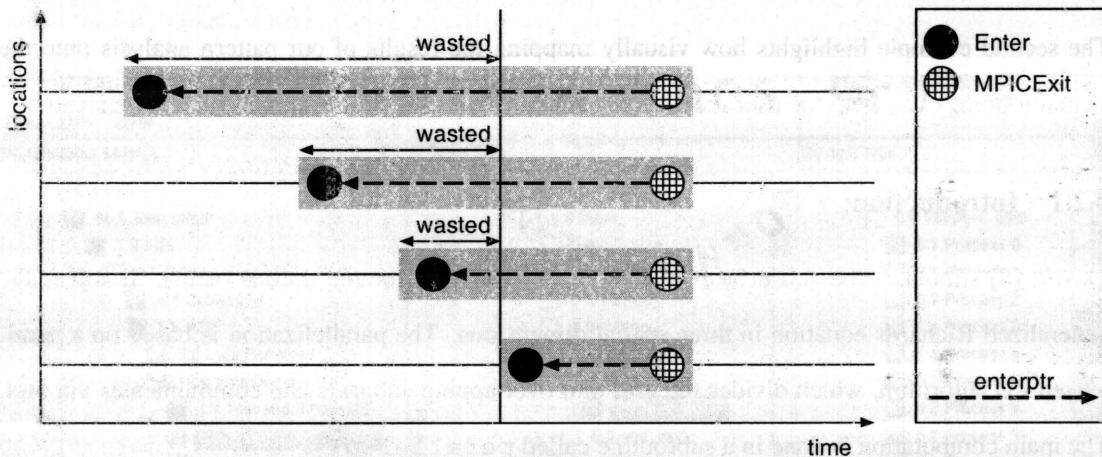


Figure 4.5: Wait at N x N collective operation

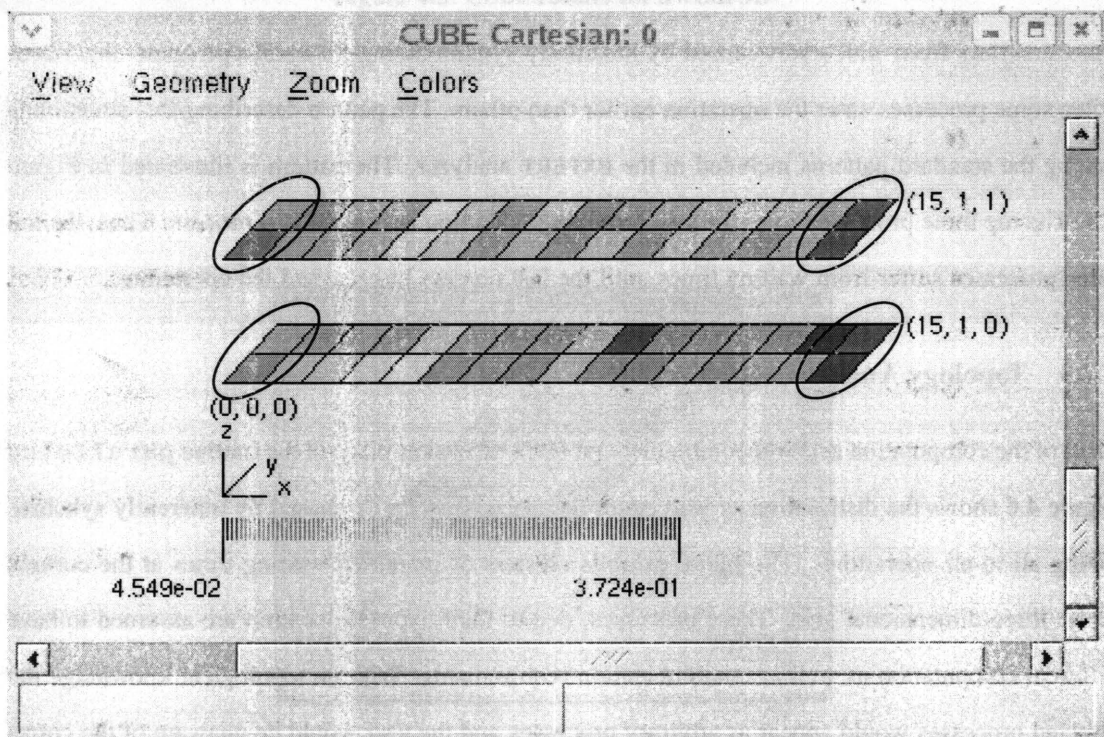


Figure 4.6: Distribution of wait states caused by inherently synchronizing all-to-all operations in TRACE

Chapter 5

Summary and Future Work

This chapter summarizes our effort to demonstrate the importance of mapping the performance data onto the topology of the application for the analysis of certain communication patterns. We also present future work in this area that will generalize this technique for the purpose of performance analysis of parallel applications.

5.1 Summary

The main focus of this thesis has been the development of an infrastructure to use topological information for performance analysis of parallel applications. The infrastructure has been build as an extension to the KOJAK performance analysis environment.

We extended the KOJAK toolkit at various levels, each of which performs necessary tasks at different stages in the performance analysis process. The EPILOG binary trace format has been extended to record topological information in the event trace. The EPILOG runtime system has been extended to automatically record MPI topology information. An instrumentation API has been provided to manually instrument the source code to record the topology-specific information in the event trace when MPI topology support is not used. The abstraction library, EARL, has been extended to access topological information from the event trace and provide an abstraction to conveniently access this information. The EXPERT analyzer has been extended to identify and pass topological information to the high-level callpath profile produced as output. Finally, the presentation layer, CUBE, has been made topology aware and the CUBE GUI has been extended to visualize

the distribution of performance data across the virtual topology of the application.

We demonstrated the feasibility of our work by performance analysis of two realistic MPI applications based on our concept. The first one is the ASCI benchmark, SWEEP3D, and the second is an environmental science application, TRACE, provided by Forschungszentrum Jülich, Germany. Using SWEEP3D's pipelined wavefront algorithm as an example, we demonstrated that with topological knowledge, EXPERT is now able to identify the direction of messages in the virtual topology of SWEEP3D. This information was then used to identify higher-level events related to the parallelization scheme used in SWEEP3D and the correlation of these higher-level events with wait states identified by KOJAK's pattern analysis. This correlation allowed us to reintroduce a time dimension into an otherwise timeless data model of analysis results by letting pattern specifications refer to distinct algorithm-specific execution phases. Using TRACE as our example, we further showed that visually mapping wait states identified by KOJAK's pattern analysis onto the topology enables the correlation of these wait states with topological characteristics of the affected processes.

5.2 Future Work

Future work will address the extension of our infrastructure to generalize the idea of performance analysis of parallel applications by utilizing their topological information. Presently, our work is restricted to Cartesian topologies. We intend to provide support for general graph topologies. Also, we have not used the concept of periodicity of a Cartesian grid in our work. We intend to support this concept and hence, support more complex Cartesian topologies (e.g., a hypercube or a cylinder).

Future work will also address the understanding of operations of wavefront processes in more detail by studying the overlap between pipelines coming from different directions. We also intend to extend the scope of the underlying principles to other algorithms, such as parallel multi-frontal methods [5].

Bibliography

Bibliography

- [1] O. Lubeck A. Hoisie and H. Wasserman. Performance Analysis of Wavefront Algorithms on Very Large Scale Distributed Systems. *Lecture Notes in Control and Information Sciences*, 249:171, 1999.
- [2] Accelerated Strategic Computing Initiative (ASCI). *The ASCI sweep3d Benchmark Code*. http://www.llnl.gov/asci_benchmarks/.
- [3] D. H. Ahn and J. S. Vetter. Scalable Analysis Techniques for Microprocessor Performance Counter Metrics. In *Proc. of the Conference on Supercomputers (SC2002)*, Baltimore, November 2002.
- [4] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide* Addison Wesley, October 1998.
- [5] Iain S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3:193–204, 1986.
- [6] B. Mohr F. Wolf. EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, May 2004.
- [7] T. Fahringer, M. Gerndt, B. Mohr, G. Riley, J. L. Träff, and F. Wolf. Knowledge Specification for Automatic Performance Analysis. Technical Report FZJ-ZAM-IB-2001-08, ESPRIT IV Working Group APART, Forschungszentrum Jülich, August 2001. Revised version.
- [8] Forschungszentrum Jülich. *Solute Transport in Heterogeneous Soil-Aquifer Systems*. http://www.kfa-juelich.de/icg/icg4/Groups/Pollutgeosys/trace_e.html.

- [9] Lei Huang, Barbara Chapman, and Ricky Kendall. Executing openmp on distributed memory systems via global arrays. *Journal of Parallel Computing*, 2004 to appear.
- [10] S. Huband and C. McDonald. A Preliminary Topological Debugger for MPI Programs. In R. Buyya, G. Mohay, and P. Roe, editors, *Proc. of the First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 422–429. IEEE Computer Society, 2001.
- [11] IBM. *Dynamic Probe Class Library*. <http://dpci.sourceforge.net/>.
- [12] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. <http://www.mpi-forum.org>.
- [13] C. Müllender. Visualisierung der Speicheraktivitäten, von parallelen Programmen in Systemen mit virtuell gemeinsamen Speicher. Master's thesis, RWTH Aachen, Forschungszentrum Jülich, May 1994.
- [14] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface - Version 2.0*, November 2000. <http://www.openmp.org>.
- [15] The Paradyn Project. *Homepage*, February 2004. <http://www.cs.wisc.edu/~paradyn/>.
- [16] 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.
- [17] S. S. Shende. *The Role of Instrumentation and Mapping in Performance Measurement*. PhD thesis, University of Oregon, August 2001.
- [18] F. Song and F. Wolf. CUBE User Manual. Technical Report ICL-UT-04-01. University of Tennessee, Innovative Computing Laboratory, Knoxville, TN, February 2004.
- [19] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An Algebra for Cross-Experiment Performance Analysis. In *Proc. of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.
- [20] SWIG. Simplified Wrapper Interface Generator. <http://www.swig.org/>.

- [21] F. Wolf. EARL - Eine programmierbare Umgebung zur Bewertung paralleler Prozesse auf Message-Passing-Systemen. Master's thesis, RWTH Aachen, Forschungszentrum Jülich. Jülicher Bericht 3551, June 1998.
- [22] F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003. Special Issue “Evolutions in parallel distributed and network-based processing”.
- [23] F. Wolf, B. Mohr, J. Dongarra, and S. Moore. Efficient Pattern Search in Large Traces through Successive Refinement. In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, Pisa, Italy, August - September 2004.

Vita

Nikhil Bhatia was born in Delhi, India, on September 16, 1980, the son of Rajan Bhatia and Kamla Bhatia. After graduating in 1998 from Ramjas School Pusa Road, New Delhi, India, he attended the Mumbai University where he received a Bachelor of Engineering degree from the Computer Engineering department in 2002.

After his undergraduate studies, Nikhil attended the University of Tennessee, Knoxville, Tennessee, where he received a Master of Science degree in 2005 from the Computer Science department. During his stint at the University of Tennessee, Nikhil worked as a Graduate Research Assistant in the Innovative Computing Laboratory.

Nikhil is currently working in the Computer Science and Mathematics division of the Oak Ridge National Laboratory as a junior member of the Future Technologies group. In the future, Nikhil hopes to develop systems software for next generation parallel computers which would help parallel applications to achieve optimal performance on these sophisticated machines.