Scalable On-line Automated Performance Diagnosis

BY

Philip Charles Roth

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin–Madison

2005

Scalable On-line Automated Performance Diagnosis

Philip Charles Roth

Under the supervision of Professor Barton P. Miller

at the University of Wisconsin–Madison

Performance tools are critical for the effective use of large-scale computing resources, but existing tools have failed to address three problems that limit their scalability: managing a large volume of performance data, communicating between a large number of distributed components, and presenting performance data and analysis results for a large number of application components. We present a four-part approach enabling tools to overcome these scalability barriers.

First, we introduce Multicast/Reduction Overlay Networks (MRONs) and MRNet, our MRON implementation. An MRNet-based tool interposes a hierarchy of processes between the tool's front-end and its back-ends that can be used to distribute and parallelize tool activities. Using MRNet in simple tests and in the Paradyn performance tool, average message latency and throughput, start-up latency, and performance data processing throughput scaled linearly with no sign of resource saturation for up to 512 tool back-ends.

Next, we present the Distributed Performance Consultant, an approach for finding performance problems in applications with a large number of processes, and the Sub-Graph Folding Algorithm, a technique for producing scalable graph-based displays of bottleneck search results. The Distributed Performance Consultant includes a distributed bottleneck search strategy and a way to monitor and control the cost of a search's instrumentation. The Sub-Graph Folding Algorithm combines sub-graphs of a search results graph that show similar qualitative behavior into a composite sub-graph. Using an approach that combines these two synergistic parts, we performed bottleneck searches on up to 1024 application processes with no sign of resource saturation. For 1024 application processes, the Sub-Graph Folding Algorithm converted a 30309-node search results graph into a 44-node graph with a single composite sub-graph.

Finally, we introduce Deep Start, an approach for finding performance problems in applications with a large number of functions. Deep Start uses sampling to augment an instrumentation-based bottleneck search strategy. Deep Start finished finding bottlenecks 10% to 61% faster than a search strategy guided only by the application's call graph. Also, Deep Start often found bottlenecks hidden from the call graph search strategy.

Acknowledgments

First and foremost, thanks to God for providing the opportunity to perform this work and the strength to see it through to the end.

Thanks to Bart Miller, my advisor, for his guidance, encouragement, and support. I cannot put into words my appreciation for his willingness to be flexible when I found myself in a tough situation. Without a well-timed "I have no doubt" when things looked bleakest, I would not have had the courage to finish this work.

Thanks to the other members of my committee: Miron Livny, Remzi Arpaci-Dusseau, Marvin Solomon, and Gregory Moses. I appreciate the time they spent to evaluate my research, and their insightful comments and questions about the work.

Thanks to the members of the Paradyn research group, past and present, for fruitful discussions and for developing and maintaining the Paradyn performance tool. Special thanks to Dorian Arnold for great work on MRNet and to Dorian, Vic Zandy, Karen Karavanic, Will Benton, and Brian Wylie for their friendship and willingness to listen.

Thanks to Scott Futral, Bronis de Supinski, Chris Chambreau, John Gyllenhaal, Barbara Herron, and Charlie Hargreaves for their help with the computing resources at Lawrence Livermore National Laboratory.

Thanks to Jeffrey Vetter for his support and sponsorship, both at Lawrence Livermore National Laboratory and now at Oak Ridge National Laboratory. His understanding and encouragement have been a substantial help as I finished this research.

Thanks to Daniel Reed, my Masters advisor at the University of Illinois at Urbana-Champaign, for his words of encouragement even though I was no longer his student.

Last, thanks to my family for their support and love. Though they did not understand why this took so long and was so difficult, I appreciate their willingness to look past bad moods, absences, and distracted presences. Most special thanks to my wife Traci and to Sarah and Elijah, for they bore the brunt of it all. I dedicate this dissertation to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Performance tools are critical for the effective use of large-scale computing resources, but existing tools have failed to address several problems that limit their scalability. This dissertation describes several novel techniques for surmounting these performance tool scalability barriers.

With the increasing availability of large parallel systems [1,60,81,101], large clusters of commodity systems [13,63,87,99], and the Grid [30,31,33], many real-world high-performance applications are also large-scale applications. They create thousands of processes, are built from hundreds of thousands of lines of code, and run for days, weeks, or sometimes longer. However, large-scale applications usually make poor use of the systems that run them. Applications that achieve more than a small fraction of their system's theoretical peak performance are uncommon. By design, performance monitoring and diagnosis tools are intended to expose the reasons why applications fail to achieve the desired level of performance. However, existing performance tools have been ineffective for improving the performance of large-scale applications because they fail to overcome three barriers to performance tool scalability: how to manage a large volume of performance data, how to communicate efficiently between a large number of distributed tool components, and how to make scalable presentations of performance data and analysis results.

This dissertation describes the design and implementation of several novel techniques that enable tools to overcome these three performance tool scalability barriers. To address the problems of managing a large volume of performance data and communicating efficiently between a large number of distributed tool components, we designed and implemented MRNet [89], a software-based infrastructure that provides scalable multicast and data aggregation functionality for all types of parallel tools. Also for the problem of managing a large volume of performance data, we designed and implemented the Distributed Performance Consultant that automatically finds and explains performance problems in large-scale applications. The Distributed Performance Consultant uses *search*, a well-established technique for methodically examining a solution space. Because centralized performance data and tool control processing limit tool scalability as the number of application processes grows, our search strategy distributes sub-searches to reduce centralization of tool activities. The Distributed Performance Consultant also includes a new model for the cost of instrumentation in parallel and distributed applications, and a new strategy for scheduling instrumentation requests during a distributed performance problem search. To complement the Distributed Performance Consultant, we developed a strategy called Deep

Start [90] for efficiently finding performance problems in applications with large, complex executables. Finally, to address the problem of making scalable presentations of performance analysis results, we designed and evaluated the Sub-Graph Folding Algorithm, a visualization technique for making scalable displays of the results of our automated performance problem searches.

Because of the importance of parallel performance tools for the effective use of large-scale computing resources, our research focuses on overcoming barriers to the scalability of parallel performance tools. However, our techniques may be generalized (with some adaptation, depending on the technique) to improve the scalability of a broader class of parallel tools. For example, MRNet is designed to support scalable communication and data processing in all types of parallel tools. It provides a rich set of built-in data aggregation operations and is easily extended to use tool-specific aggregations. The Distributed Performance Consultant, Deep Start, and Sub-Graph Folding Algorithm also incorporate techniques that can be generalized, though generalization is likely to require more substantial modification to these techniques than is necessary with MRNet.

1.1 Parallel Performance Tool Scalability Barriers

Like any parallel tool, a parallel performance tool uses a system of distributed components to implement its functionality. A parallel performance tool uses its distributed component system to implement functionality like data collection and application process control. A tool function may be a *tool scalability barrier* if its cost in terms of computation, communication, or storage is larger than the under-

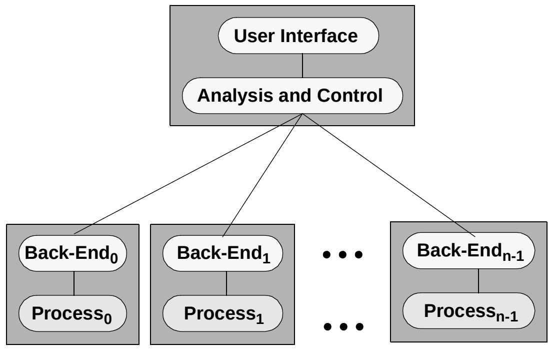


Figure 1.1 The components of a typical parallel tool. Shaded boxes represent potential machine boundaries.

lying system can support. The presentation of performance data and analysis results may also be a tool scalability barrier if characteristics such as its complexity or abstraction level are appropriate for small-scale computation but prevent users from obtaining useful information for large-scale computation.

Regardless of the functionality it provides, a parallel tool implements its functionality using a system of distributed components. The components of a typical parallel tool system are shown in Figure 1.1. Many parallel tools follow this organization, including the TotalView [25], PRISM [94,95], and p2d2 [51] parallel debuggers, the SCALEA [103], TAU [92], and Paradyn [74] parallel performance tools, and the Autopilot [85] computational steering system. Data collection and process control occur in the tool's *back-end* components (often called *tool daemons*) running on the nodes of a parallel or distributed system. Data analysis and global tool control (i.e., control of the entire tool system) are usually implemented in a component distinct from the tool back-ends, although some low-level analysis may take place in the tool back-ends. The user interacts with the tool's user interface component. Often the user interface, data analysis, and global tool control activities are provided by the same component. In this case, this combined component is called the tool's *front-end*.

All tool functionality comes at a cost. These costs can be divided into several categories:

- **Computation.** Tools incur a computation cost whenever some processor executes code to implement tool functionality. The most obvious computation cost is for data analysis, but a tool also pays a computation cost for other activities like data collection and implementing a user interface.
- **Communication.** Tools incur a communication cost whenever they transfer data between tool components. For example, tools incur a communication cost if they transfer data from their back-ends for analysis, either because the analysis is centralized in the tool's front-end or because the analysis occurs in tool components running on a different system than the one on which the data was collected. Tools with a parallelized analysis incur a communication cost for exchanging data between analysis components. There is also a communication cost for transferring control requests and responses within the tool system. Also, there is a communication cost to transfer analysis results to the user interface component if the analysis and user interface activities are not implemented in the same tool component.
- **Storage.** Tools that do not analyze data as it is collected must store the data for off-line, or post mortem, analysis. The cost of storing this data has a *space* component and may also have a *time* component. The storage cost's space com-

ponent is paid whenever data is moved between collection and analysis. If a tool stores the collected data in a database or dedicated storage server, it also pays a storage cost in terms of the time required to transfer the data to and from the storage server. In contrast to off-line tools, *on-line* tools that analyze the data as it is being collected do not incur a storage cost between data collection and analysis.

In this dissertation, we describe several new techniques for surmounting tool scalability barriers in parallel performance tools. We divide these scalability barriers into three categories: management of performance data, collective communication within a tool system, and the presentation of performance data and analysis results. Although we focused on these scalability barriers in the context of parallel performance tools, the techniques we propose may be adapted to address scalability problems in other types of parallel tools.

The first barrier to performance tool scalability is how to manage a large volume of performance data. A large number of processes, complex executables, and high processor speeds tend to increase performance data volume. Less obvious factors may also increase data volume, such as the complexity of the hardware, operating system, and software infrastructure. There is an inherent tension between collecting data with sufficient detail and keeping data collection, storage, and processing overheads to tolerable levels. Approaches for controlling performance data volume typically have used one or more of the following techniques:

- collecting data with a high level of detail from all nodes for the duration of an application's execution and then reducing it [54,88,107];
- collecting data with limited detail from all nodes throughout the execution [4,38,57,108];
- collecting high-detail data only for a limited subset of the nodes (often called *representative nodes*) [85,88]; or
- collecting high-detail data for limited intervals during an application's execution [49,74].

The first two techniques are inherently non-scalable; the volume of data they produce is proportional to the number of nodes used to execute the application and the length of the application execution. In contrast, the last two techniques hold promise for controlling performance data volume, and we use them in combination in the Distributed Performance Consultant. Nevertheless, existing policies for determining when and where to collect data have not been successful in limiting performance data volume to manageable levels with large-scale applications.

Inefficient collective communication is a barrier to the scalability of all parallel programs, including performance tools. Like all parallel programs, the components of a parallel tool communicate with each other using *point-to-point* or *collective* communication operations. Point-to-point communication involves one source component and one destination component. In contrast, if the operation involves more than one source component or more than one destination component it is a collective (or *group*) communication operation. Many important parallel tool activities are implemented using collective communication. For example, a tool's front-end may send a request to all of its back-ends asking them to spawn an application's processes. In response, each back-end indicates to the front-end whether it spawned the application process(es) successfully. Because hardware support for collective communication is rare, collective communication operations are usually implemented using a sequence of point-to-point communication operations. In a tool with the typical parallel tool organization (Figure 1.1) where the back-ends do not communicate directly with each other, the point-to-point operations that constitute the collective operation are serialized at the tool's front-end. This serialization makes the tool front-end a bottleneck as the number of tool back-ends is increased, limiting the overall scalability of the tool.

Research on improving the scalability of software-based collective communication has focused on reducing the effect of serialization by parallelizing a collective operation's point-to-point communication [8,9,27,55,66,69,97] or implementing collective operations at the network layer [21]. However, existing collective communication infrastructures have proven unsuitable for building scalable, high-performance parallel tools because they exhibit limited communication throughput [8,9,27], provide a limited interface for interacting with the communication sub-system [8,66], rely on kernel or network layer features that are not generally available [69,97], or lack usage models [71] or available implementations [72] that support running tools and parallel applications together.

The third barrier to performance tool scalability is the presentation of performance data and analysis results from large, complex applications. Visualization has been effective for presenting the performance of sequential and small-scale parallel applications [44,67,70,74,76,80,83,111,112]. Most of these tools feature displays that show the behavior of individual application elements (e.g., processes or functions). Displays like ParaGraph's space-time display [44] are intuitive and provide detailed information about an application's communication behavior. Unfortunately, displays that present data from individual application elements are inherently non-scalable. Although researchers have proposed techniques for constructing scalable visualizations [18,56,98], none has gained widespread acceptance. Adaptations of existing small-scale displays using scrollable views are insufficient because they limit the user to a local view of application behavior. The most promising techniques are those that categorize application elements by their behavior and represent each category with a single graphical element [18]. However, existing research has not produced scalable, automated visualization techniques for this categorization step.

1.2 On-line Automated Performance Tools

On-line automated performance tools are uniquely suited for large-scale performance tuning. Many performance tools support measurement, simple data analysis, and visualization [4,38,44,83]. These tools can be used to diagnose application performance problems, but they require significant user involvement. To use these tools effectively, the user must have a good understanding of the interactions between the application, the operating system(s), and the hardware. Unfortunately, obtaining this understanding is difficult with large-scale applications because the interactions are complex and it is difficult to cull the interesting data from the large performance data volume.

In contrast, automated performance tools [24,35,45,74,77,78,79,111] diagnose performance problems with minimal user involvement. The user need not be a performance tuning expert to use the tool effectively because the expertise is built into the tool. Automated performance tools are especially desirable for large-scale performance tuning for two reasons. First, they relieve the user from the difficult task of identifying which data is important. Because the data volume generated by performance monitoring of large-scale applications can be massive, these tools' ability to automatically find the interesting data is extremely valuable. Second, they relieve the user from having to understand the complex interactions between application, operating system, and hardware. Automated performance diagnosis tools try to analyze these complex interactions with minimal user involvement. On-line automated performance tools augment these benefits because they can adapt their activity as the application runs in response to the application's behavior. Whereas other tools require multiple runs to obtain useful results, this adaptability allows on-line automated tools to adapt the data being collected to obtain useful results with only a single application run [74]. Also, because they collect only the data they need when they need it, they can obtain the same results using a smaller performance data volume than other types of tools. Because we believe on-line automated tools are the most feasible approach for effective performance tuning as scale increases, we chose an on-line automated performance tool (Paradyn [74]) as the context of our research.

1.3 Contributions

This dissertation describes several contributions that allow on-line automated performance tools to overcome the scalability barriers presented in Section 1.1:

- MRNet, a novel software-based multicast and data aggregation infrastructure for parallel tools;
- The Distributed Performance Consultant, a new approach for automatically finding performance problems in applications running on a large number of hosts;
- Deep Start, a new approach for automatically finding performance problems in applications with large, complex executables; and
- The Sub-Graph Folding Algorithm, a new technique for making scalable, graphical presentations of the results of search-based performance diagnosis strategies.

In this section, we give an overview of each of these research contributions. Subsequent chapters describe each contribution in detail.

The first contribution of our research is the design and evaluation of MRNet, a novel software-based infrastructure providing scalable multicast and data aggregation functionality for parallel tools. This infrastructure may be used to reduce the cost of many important activities in all types of parallel tools, including performance tools. A fundamental part of this contribution is the exploration of a wide range of uses of the multicast/aggregation idiom for improving the scalability of important tool activities. For example, we used this idiom to improve the scalability of bottleneck search control in the Distributed Performance Consultant, reduction of bottleneck search results in the Sub-Graph Folding Algorithm, performance data aggregation, and a collection of activities typically performed during start-up of a parallel performance tool. We describe MRNet and our scalable performance data aggregation and tool start-up techniques in Chapter 4.

The second contribution of our research is the design and evaluation of the Distributed Performance Consultant, a new approach for automatically finding performance problems in applications with a large number of processes. Our approach encompasses a distributed automated performance problem search strategy, a model for instrumentation cost in parallel and distributed applications, and a policy for efficiently scheduling data collection instrumentation.

The first component of the Distributed Performance Consultant is a distributed automated performance problem search strategy. This search strategy has two complementary parts. The first part deals with *local* application behavior, i.e., the behavior of specific application processes. The second part deals with *global* application behavior, i.e., the behavior across all application processes. For examining local behavior, our strategy distributes sub-searches: whenever the search examines a specific process' behavior, our strategy delegates control to a search agent running on that process' host. For examining global application behavior, the strategy uses MRNet for efficient aggregation of performance data collected from all application processes. The Distributed Performance Consultant also uses MRNet for overall tool control during the search.

The second component of the Distributed Performance Consultant is a new model for tracking the cost of instrumentation for parallel and distributed computation. Knowledge of instrumentation cost is useful for controlling [47] or compensating for [68,109] the effects of instrumentation. Existing models of instrumentation cost are not suitable for use in a scalable distributed on-line tool that uses complex instrumentation like the Distributed Performance Consultant either because they assume instrumentation has a constant cost or because they track instrumentation with an aggregated value that loses the information about the cost of instrumentation in each individual application process. Our model does not assume simple instrumentation and tracks overall instrumentation cost as a vector of instrumentation costs, where each item corresponds to a single process' instrumentation cost.

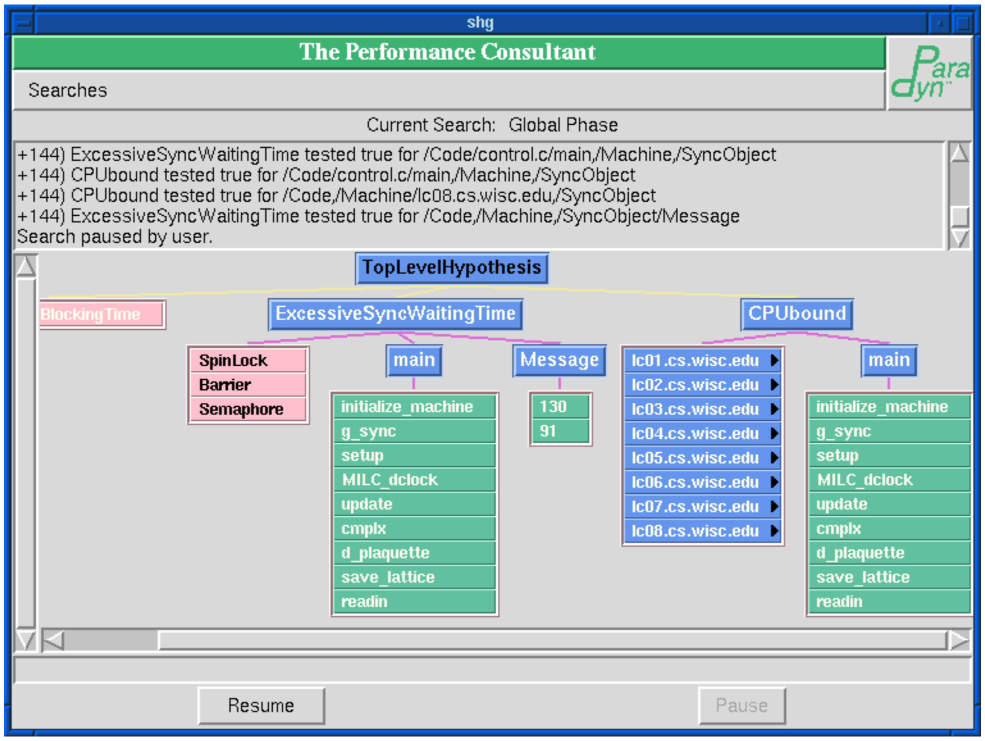
The third component of the Distributed Performance Consultant is a new policy for efficiently scheduling requests for local and global instrumentation. Distributing local sub-searches introduces a problem for managing instrumentation requests not faced in existing performance tools. In our distributed performance diagnosis strategy, local sub-searches could occur concurrently with the global search. Therefore, the strategy must be able to schedule requests efficiently for both local and global data collection instrumentation. However, global instrumentation must be inserted into all application processes to be useful. With the Distributed Performance Consultant, we introduce a new instrumentation scheduling policy that schedules instrumentation requests fairly, avoids starvation of both global and local instrumentation requests, and generates an efficient search.

Distributing local application behavior searches yields the expected scalability benefit, but at a cost in terms of tool complexity for instrumentation cost tracking and instrumentation scheduling. As a natural step from a partially-distributed search strategy that combines explicit examination of global application behavior with distributed investigation of local application behavior, we also introduce a truly distributed search strategy that leverages our Sub-Graph Folding Algorithm to approximate the results of a global application behavior search while avoiding the additional complexities to instrumentation cost tracking and instrumentation scheduling.

The techniques incorporated in the Distributed Performance Consultant can be generalized for use in a broader class of tools. The Distributed Performance Consultant incorporates an on-line, distributed search-based algorithm. The techniques used to distribute and control the search are applicable to both on-line or off-line search-based tools. Also, because we believe any tool that uses instrumentation should monitor and control the impact of its instrumentation, our instrumentation cost model and scheduling policy are applicable to any parallel tool that uses instrumentation.

We describe the Distributed Performance Consultant's distributed search strategy, parallel application instrumentation cost model, and instrumentation scheduling policy in Chapter 5.

The third contribution of our research is Deep Start, a new technique for efficiently examining the behavior of large, complex application executables. Deep Start uses stack sampling to augment an instrumentation-based automated search for performance bottlenecks. This hybrid strategy locates performance



Current Search: Global Phase

```
+144) ExcessiveSyncWaitingTime tested true for /Code/control.c/main,/Machine,/SyncObject
+144) CPUbound tested true for /Code/control.c/main,/Machine,/SyncObject
+144) CPUbound tested true for /Code,/Machine/lc08.cs.wisc.edu,/SyncObject
+144) ExcessiveSyncWaitingTime tested true for /Code,/Machine,/SyncObject/Message
Search paused by user.
```

Figure 1.2 A Search History Graph display.

problems more quickly and finds problems hidden from a more straightforward search strategy. Deep Start uses stack samples collected as a by-product of normal search instrumentation to find *deep starters*, functions that are likely to be application bottlenecks. Deep starters are examined early during a search to improve the likelihood of finding performance problems quickly. The Deep Start approach of combining search and sampling is applicable to other tools that rely on search alone; the type of data being sampled is likely to be tool-specific. Deep Start is described in Chapter 6.

The final contribution of our research is a new technique for making scalable, graphical presentations of the results of search-based performance diagnosis strategies like the Distributed Performance Consultant. For sequential and small-scale parallel applications, the results of a performance diagnosis search are presented effectively using a *search history graph* as shown in Figure 1.2. A search history graph is a directed acyclic graph that represents the decisions made during a search as nodes and edges. These graphs become hopelessly cluttered for applications with more than a few tens of processes. Our new technique, called the Sub-Graph Folding Algorithm (SGFA), significantly reduces the complexity of a search history graph. Under the hypothesis that the processes of a large-scale application will fall into a small number of behavioral categories, SGFA combines sub-graphs showing similar qualitative process behavior into a single composite sub-graph. SGFA builds on previous work (e.g., [56,94,95]) that combines graph nodes that are similar in some way to reduce graph complexity, but SGFA is unique in applying the approach on the basis of qualitative analysis results. The SGFA "folding" technique can be applied to displays of qualitative data beyond the search history graph, including textual displays. SGFA is described in Chapter 7.

1.4 Dissertation Organization

This dissertation contains seven chapters. Chapter 2 compares the research described in this dissertation to related research in areas such as overlay networks, automated performance tools, and scalable visualization techniques. Because Paradyn's Performance Consultant was the context for our research, we briefly review this automated performance diagnosis component in Chapter 3. The next four chapters detail our research on scalable on-line automated performance diagnosis. Chapter 4 presents MRNet, our multicast and data aggregation infrastructure for scalable parallel tools. As an example of MRNet's use in a real-world parallel tool, this chapter also describes our MRNet-based techniques for global performance data aggregation and scalable tool start-up. In Chapters 5 and 6, we present techniques for automatically finding application performance problems in large-scale applications. Chapter 5 describes the Distributed Performance Consultant, our strategy for automatically finding performance problems in applications with a large number of processes. Chapter 6 describes Deep Start, our strategy for automatically finding performance problems in complex applications with a large number of functions. In Chapter 7, we present the Sub-Graph Folding Algorithm, our approach for producing scalable, graphical displays of the results from our scalable performance diagnosis strategies. We summarize our research on scalable on-line automated performance diagnosis and outline possible directions for future research in Chapter 8.

Chapter 2

Related Work

This chapter describes previous work that is closely related to our research. First, we survey existing work in collective communication infrastructure and parallel algorithms related to our scalable tool infrastructure and scalable performance diagnosis approach (Section 2.1). Next, we examine existing work in on-line automated performance diagnosis and tuning (Section 2.2). We then examine sampling as a technique for obtaining performance data (Section 2.3). Next, we relate our approach for managing software-based instrumentation to existing work in instrumentation cost models and workload scheduling policies (Section 2.4). Finally, we examine previous work on scalable performance visualization techniques (Section 2.5). We conclude the chapter with a summary of the related work (Section 2.6).

2.1 Collective Communication

Collective communication is communication between a group of processes. Collective communication operations like “broadcast” and “gather” and data reductions like “global sum” are the building blocks for all but the most trivial parallel algorithms. Given their low-level nature and widespread use, the performance of collective communication operations has a significant impact on the scalability of parallel programs, including performance tools. Our research into scalable multicast and data reduction techniques is related to existing work on parallel infrastructure and tools that use hierarchical overlay networks for collective communication, and on parallel algorithms for data aggregation and data volume reduction.

2.1.1 Collective Communication Using Hierarchical Overlay Networks

A collection of connected processes is often called an *overlay network* because it defines a logical network that overlays a physical network. Hierarchical overlay networks with data aggregation and multicast capabilities have been used in collective communication infrastructures and distributed cluster monitoring tools.

Lilith [27] and Ygdrasil [8] are multicast and data aggregation infrastructures for parallel tools. Although both use a hierarchy of processes for scalability, they differ in their communication model and tool architecture. In Lilith’s communication model, synchronous waves of messages are sent to or from the tool’s front-end at the root of the process tree [26]. Lilith’s architecture allows tool back-end code to run within each process of the Lilith process network. Ygdrasil generalizes the multicast/reduction capabilities of the Ladebug [9] parallel debugger. It is best suited to the synchronous request/response communication model used by tools like parallel debuggers. Unlike Lilith, Ygdrasil allows tool back-end code only at the leaves of its process tree. The remaining processes in the process tree perform multicast and reduction operations on data transferred between tool front-end and back-ends. Both Lilith and Ygdrasil are implemented in Java to leverage that language's natural ability to load code dynamically.

MRNet differs from Lilith and Ygdrasil in its communication model, tool architecture, and software engineering trade-offs. Unlike Lilith's communication model of synchronous message waves and Ygdrasil's synchronous request/response communication model, MRNet's communication model supports multiple simultaneous asynchronous collective communication operations. MRNet-based tools have an architecture similar to that of Ygdrasil-based tools, with tool back-end code running only at the leaves of its process network. Whereas Lilith and Ygdrasil use Java for ease of extensibility, MRNet trades ease of extensibility for the higher potential data throughput made possible by explicit memory management.

Most research in software-based collective communication has focused on providing multicast and data aggregation support for applications. The Message Passing Interface [71,72] standards include several collective communication operations, such as broadcast, scatter, and gather. Although some MPI implementations use serialized point-to-point operations to implement these collective operations, others provide optimized implementations that use an overlay network. For example, MagPIe [55] provides MPI collective communication primitives optimized for applications run in a geographically-distributed environment like the Grid. MagPIe uses a process tree consisting of a flat, single-level tree at the root for efficient communication across a WAN, followed by a binary tree for efficient communication within the local network. The ACCT [105] system automatically tunes its MPI collective communication algorithms based on modelling and experimental results, tailoring the algorithms to the system on which the MPI application runs. Because optimized MPI implementations are not universally available, we cannot depend on the availability of a high-throughput MPI layer for efficient collective communication in parallel tools.

The data reductions supported by MPI are more restrictive than those supported by MRNet. MPI only supports ordinal processing of the input arrays (i.e., they apply the reduction operation to the first element of each input array, then the second element, and so on.) In contrast, MRNet's filters can operate on as much (or as little) of each of the input arrays as desired, allowing MRNet data aggregations to perform sophisticated data aggregations like the timestamp-based data aggregation described in Chapter 4. Furthermore, a tool's use of MPI may conflict with MPI use in the monitored application. For example, in a common tool start-up scenario a process manager creates tool back-end processes, which then create application processes. The tool back-end processes are supposed to be transparent to the process manager, but may not be if they are also MPI processes. The dynamic process and communicator creation functions specified by the MPI-2 standard [72] may alleviate this problem, but support for these functions is lacking in existing MPI implementations. MRNet does not use MPI for collective communication, so it is safe to use in tools that monitor MPI applications or even as the foundation for an MPI implementation's multicast and data reduction functionality.

In addition to the parallel tool and application infrastructure projects mentioned previously, a few cluster monitoring tools such as Ganglia [69] and Supermon [97] use a hierarchical overlay network with data aggregation functionality for scalability.

Ganglia [69] is a widely-used distributed monitoring system for clusters and computational Grids consisting of federations of clusters. Ganglia can monitor system-level and application-level metrics, with built-in support for common system-level metrics like CPU utilization, number of processors, and memory usage. Ganglia uses a hierarchical organization for scalability. Above the level of individual clusters, Ganglia uses an overlay network of processes connected by point-to-point TCP connections for data transfer and aggregation. The topology of this daemon hierarchy is specified statically, but fail-over connections can be specified for fault tolerance. Within each cluster, Ganglia uses IP multicast functionality [21] to implement a listen/announce protocol for maintaining cluster membership and for distributing monitored data to all cluster nodes for fault tolerance within the cluster.

Like Ganglia, Supermon is a cluster monitoring system that can use a hierarchical overlay network for aggregating data. Within each cluster node, a loadable kernel module extracts system-level configuration and performance data and presents it via a file in the `procfs` filesystem to a user-level monitor program. A supermon (for “super monitor”) is connected with TCP point-to-point connections to the monitor on one or more cluster nodes. For scalability, supermons can be connected to other supermons to form a hierarchy.

Although Ganglia, Supermon, and MRNet each use a hierarchical overlay network for collective communication, their use of this network differs in significant ways that make their approaches less suitable than MRNet as a foundation for parallel tools. Ganglia, Supermon, and MRNet-based tools each use an overlay network for efficiently gathering data from the tool's back-ends to its front-end, but neither Ganglia nor Supermon uses the network for scalable multicast. Also, data aggregation support is more flexible with MRNet than with Ganglia and Supermon. Ganglia and Supermon's only data aggregation is the concatenation operation. In contrast, MRNet provides several built-in data aggregations for common operations, and allows a tool to dynamically add tool-specific data aggregations to be run in the overlay network processes. Finally, Ganglia, Supermon, and MRNet differ in their support for high-throughput collective communication. Ganglia seems to be targeted to data collection intervals on the order of several seconds or more. On the other hand, Supermon and MRNet share the goal of high-throughput data collection, though a throughput comparison based on published results is not possible due to differences in the data aggregation used and uncertainty about the amount of data contained in each Supermon sample.

2.1.2 Parallel Data Aggregation and Volume Reduction Algorithms

As part of our evaluation of the MRNet infrastructure, we modified Paradyn to use MRNet for tool start-up and performance data aggregation. This work is related to previous research on parallel algorithms for data aggregation and data volume reduction.

The first aspect of our MRNet evaluation in a real-world setting involved its use for performance data aggregation. In addition to performance tools, data aggregation has also been studied in the context of parallel databases and ad-hoc wireless sensor networks that provide a database-like query interface. Graefe [39] presents basic techniques for computing aggregates in a parallel database. He also suggests guidelines for efficient data aggregation, such as to compute aggregates at the lowest-possible system level to avoid unnecessary data movement. Shatdal and Naughton [91] present an adaptive algorithm that dynamically selects between two data aggregation approaches based on query characteristics including the type of aggregation being performed and the layout of the data being aggregated. Gray et al [40] suggest ways for efficiently implementing their “data cube” aggregation operator in a parallel database. Although this parallel database research provides useful guidance for implementing these more conventional data aggregations, it does not cover some of the more unusual (and important) aggregation operations needed by many performance tools such as the time-aligned performance data aggregation and clock skew detection reductions described in Section 4.3. Also, this research does not address the use of a hierarchy of processes for scalable data aggregation like that used by Lilith, Ygdrasil, TAG (described below), and MRNet.

TAG [66] is a wireless sensor network that acts like a parallel database. It provides a SQL-based interface for expressing data aggregation queries and a relational database model for representing aggregation results collected from wireless sensor networks. Like MRNet, TAG supports multiple simultaneous aggregation operations and supports streams of aggregated data in response to an aggregation request. TAG uses a SQL/relational interface, in contrast to our RPC-style interface. Also, TAG organizes its sensors with an ad-hoc routing tree, whereas MRNet's network configuration is specified a priori using a configuration file. Like MPI data reductions, TAG only supports ordinal data aggregation; MRNet's filters can align and aggregate data based on other criteria such as its timestamp.

The second aspect of our MRNet evaluation in a real-world setting involved its use for performance data volume reduction. One of the techniques used in our MRNet-based tool start-up research is based on an approach used in ETRUSCA [88], our earlier system for reducing performance data volume, and further refined by Reed et al [84]. ETRUSCA (for Event Trace Reduction Using Statistical Cluster Analysis) processes event traces from parallel applications to compute a small number of performance metrics for each process. Periodically, ETRUSCA categorizes processes so that those with similar metric values are in the same category. Once the processes are categorized, ETRUSCA chooses process from each category that best represents the category and filters the traces to keep only the representatives' event trace records. Although this technique is effective for reducing event trace data volume, its use of centralized processing to find process categories and to select category representatives limits its overall scalability. In our scalable tool start-up approach, we overcame this scalability barrier by using MRNet-based data reductions to categorize the processes and to select category representatives.

2.2 On-line Automated Performance Diagnosis And Tuning

On-line performance tools are those that perform the bulk of their data analysis while the monitored application is still running. Automated performance diagnosis tools are those that find and explain application performance problems with minimal user involvement. Tools that combine the two approaches, on-line automated performance diagnosis tools, are especially desirable for tuning large-scale applications because:

- they automatically cull the interesting data from a potentially massive volume of performance data, relieving the user from the challenging task of analyzing the complex interactions between application, system software, and hardware; and
- they avoid the need to store the potentially massive volume of performance data between its collection and analysis.

However, an on-line automated performance diagnosis tool contends with the application for computing resources. Therefore, to avoid significantly altering the application's behavior when it is monitored by an on-line automated performance diagnosis tool, the tool must use efficient methods for collecting performance data, for transferring the data to the tool's data analysis component(s), and for analyzing the data. Because of their desirability for tuning large-scale applications, on-line automated performance diagnosis tools are an active research area. In this section, we survey existing work in on-line automated performance diagnosis tools. Most of the projects mentioned here are affiliated with the APART working group [5], a working group funded by the European Union for discussing tools and techniques for automating performance diagnosis.

Perhaps due to the combination of challenges in implementing a low-impact on-line tool and a useful automated performance diagnosis tool, few on-line automated performance diagnosis and tuning tools exist despite their desirability for large-scale performance tuning. Paradyn [15,74] is the seminal instance of this class of tools, but recent projects Peridot [36] and SCALEA [103,104] also provide on-line automated performance diagnosis functionality. Because on-line automated computational steering toolkits and tools like Falcon [42], Autopilot [85], and Active Harmony [100] share several characteristics with on-line automated performance diagnosis tools, we include them in our survey. Finally, the Monitoring, Analysis, and Tuning Environment (MATE) [78] augments on-line automated performance diagnosis with dynamic optimization based on the results of the performance diagnosis, exceeding the advantages of both automated performance diagnosis and computational steering.

Most of the automated performance diagnosis tools described in this section find application performance problems using search, a technique described in introductory artificial intelligence texts (e.g., Rich and Knight [86]) for finding a goal state in a problem state space. One heuristic for reducing the time required for a search through a problem state space is to start the search as close as possible to the goal state. We adapted this idea for Deep Start, our technique for automatically finding performance bottlenecks in applications with a large number of functions. Deep Start uses stack sample data to select deep starters that are close to the goal states in our problem domain: the bottlenecks of the application being examined. Like the usual situation for an artificial intelligence problem search, one of our goals for Deep Start is to reduce the time required to find solutions (i.e., application bottlenecks). However, in contrast to the usual artificial intelligence search that stops when the first solution is found, Deep Start attempts to find as many bottlenecks as possible.

Paradyn [15,74] is an on-line performance diagnosis tool. Paradyn's Performance Consultant uses an automated, instrumentation-based search to find application performance bottlenecks. The Performance Consultant's search determines both *why* and *where* an application is experiencing a performance problem. Paradyn uses *dynamic instrumentation* [49] to control performance data volume and *time histograms* to support long-running applications. Dynamic instrumentation is a technology for inserting, modifying and removing instrumentation as an application runs. Using dynamic instrumentation, Paradyn collects only the data it needs for performance monitoring and automated performance diagnosis. Although dynamic instrumentation is used to generate performance data, Paradyn's daemons using sampling to control the volume of data delivered to the tool's front-end. Paradyn stores sampled data in time histograms, fixed-size data structures that each hold the performance data for a single performance metric. A decreasing sampling rate is used to keep time histograms from overflowing with long-running applications. Time histograms were originally developed for Paradyn's progenitor IPS-2 [48,75]. Because the Performance Consultant provides the context for our work in scalable automated performance diagnosis and scalable data visualization, we provide a more detailed review of this tool in Chapter 3.

Peridot [36] is an on-line automated performance diagnosis system whose design shares several characteristics with our approach to scalable on-line automated performance diagnosis. Like our approach, it uses an agent running on each node of the system to examine the behavior of processes running on that node, with a global diagnosis agent to examine global application behavior. Peridot uses selective instrumentation, instrumentation that is inserted at compile-time but can be enabled or disabled while the application runs. Peridot's strategy for finding application performance problems is based on the APART Specification Language [28], a language for formally expressing an application's performance properties.

SCALEA [103] is an on-line automated performance diagnosis tool for use with MPI, OpenMP, and HPF programs and the Vienna Fortran Compiler [10]. Like Paradyn's Performance Consultant, SCALEA's performance diagnosis strategy performs an instrumentation-based, on-line search for why and where an application is experiencing performance problems. SCALEA's search is guided by an extensive hierarchy of *performance overheads*, or reasons why an application may be exhibiting a performance degradation. SCALEA's search strategy determines where an application is experiencing a performance overhead by dividing the application's source into code regions. The tool instruments code regions at compile time, with instrumentation requests expressed in a formal, text-based language called the Instrumentation Request Language (IRL). To support the refinement of code regions, and to avoid the need to instrument code regions to collect data about all possible performance overheads, SCALEA uses an experiment management component that allows it to perform its automated performance diagnosis search across multiple application runs. SCALEA uses a novel language called the Instrumentation Request Language (IRL) for expressing requests to its instrumentation service. Recently, SCALEA functionality has been re-packaged in a Grid monitoring system called SCALEA-G [104]. SCALEA-G is based on the Grid Monitoring Architecture [102] and implemented as a collection of services conforming to the Open Grid Services Architecture [37]. In this form, the tool can monitor both Grid infrastructure and Grid applications.

On-line automated computational steering toolkits and tools like Falcon [42] and Autopilot [85] share several characteristics with on-line automated performance diagnosis tools. Both may dynamically control an application based on the performance data they collect: on-line performance diagnosis tools, to adapt the data they are collecting in order to manage the performance data volume; computational steering systems, to modify application variables in order to change the workings of the application for improved performance. Both must analyze the performance data quickly enough to make effective on-line control decisions.

Falcon and Autopilot are steering toolkits for distributed computation. They use a distributed system of *sensors* to collect data about an application's behavior and *actuators* to make modifications to application variables to change that behavior. They require the applications to include sensors and actuators at build time so that they are available when the application runs for monitoring and controlling the application's behavior. That is, the applications must be designed to be steerable, although the literature about Autopilot (the newer of the two systems) indicates the use of dynamic instrumentation for inserting sensors and actuators at run time is a possible direction for future work. Autopilot also provides a distributed directory service for sensors and actuators so Autopilot clients can discover and attach to a steerable application dynamically, and a decision engine based on fuzzy logic to assist clients in decision-making.

Active Harmony [100] extends the approach of these computational steering toolkits to include a component that automatically tunes an application's performance by manipulating the values of steerable parameters in search of settings that give good performance. Like Falcon and Autopilot, Active Harmony requires the application and the libraries it uses to be modified to expose steerable parameters. At run-time, Active Harmony searches through the space defined by the set of steerable parameters by applying a collection of parameter values, observing the performance with those values, and then modifying the values based on the observed performance. Unlike Autopilot and Falcon, Active Harmony does not provide explicit support for distributed applications.

MATE [78] augments on-line performance diagnosis with a dynamic performance tuning capability like that of computation steering. Unlike traditional computational steering, MATE does not require the application to include and expose monitoring points and tuning controls (e.g. steerable parameters, sensors, and actuators), but can make use of such information if it is available. Without such information, MATE operates in *automatic* tuning mode, using only built-in knowledge of the tuning and monitoring points available in the operating system and the libraries used by the application. Although this approach is desirable because it completely relieves the burden from the user, in practice better results were possible in *cooperative* tuning mode, i.e. when the user provided some insight into the workings of the application. Like Paradyn, MATE uses dynamic instrumentation to collect the data needed by its automated performance diagnosis component. MATE also uses dynamic instrumentation to optimize the application at run time, for example by changing the values of global variables or replacing one function with another.

The projects described in this section have placed varying degrees of emphasis on the issue of scalability. Most of the projects use dynamic instrumentation or some form of dynamically-enabled compile-time instrumentation. Although the nominal reason for using such instrumentation is not always scalability, its effect is to control the volume of performance data collected when monitoring the application's behavior. Outside their instrumentation approach, Active Harmony and MATE do not explicitly address the issue of scalability (though MATE suggests distributed performance analysis and tuning as future work). Paradyn's time histograms address temporal scalability by fixing the memory requirement needed to store performance data regardless of the application's run time; none of the other projects explicitly addresses temporal scalabilty. For scalability as the number of application processes increases, Peridot uses a distributed analysis approach with local and global diagnosis agents like that of our performance diagnosis strategy; Falcon and Autopilot support distributed decision-making with a similar organization. However, the Peridot literature lacks detail about how it addresses the issues of distributed control of its automated performance diagnosis strategy, and the two computational steering toolkits provide only distributed decision-making mechanism by design. Also, unlike our scalable performance diagnosis strategy, none of these projects incorporates a hierarchical process organization for scalable processing of global application performance data. Finally, none of these projects addresses the problem of scalable visualization of performance diagnosis results, the issue we address with our Sub-Graph Folding Algorithm.

2.3 Sampling-Based Performance Tools

Many performance tools use sampling as their primary method to obtain application performance data. Of these tools, most sample the system's performance counter to build a profile of the application's execution. Most UNIX distributions include the prof and gprof [38] profiling tools for performing flat and call graph-based profiles, respectively. Quartz [4] profiles parallel applications running on shared memory multiprocessors. The Digital Continuous Profiling Interface [3], or DCPI, uses program counter sampling to obtain low-level information about instructions executing on in-order Alpha [17] processors. Recognizing the limitations of the DCPI approach for out-of-order processors, ProfileMe [20] uses hardware support to obtain accurate instruction profile information on out-of-order Alphas.

Although most profilers sample only a program's performance counter, at least one profiler samples the program’s call stack instead. CPPROF [43] uses call stack samples to build a “Call Path Refinement Profile” that represents the time spent to execute all call sequences observed during a program’s run. Using a powerful language for describing various types of call sequences, CPPROF’s interactive query component can be used to search manually through the profiling data to pinpoint the call sequences that are the program’s performance bottlenecks.

Like CPPROF, our Deep Start automated performance diagnosis strategy samples the entire execution stack. However, Deep Start does not use a dedicated sampling activity to gather samples; rather, it acquires samples opportunistically from its instrumentation-based automated search activity. When Paradyn inserts or removes dynamic instrumentation during a bottleneck search, its daemons walk the program’s execution stack to ensure the changes to the program’s text are safe. Deep Start uses these stack walks as samples. Unlike CPPROF and the other sampling-based tools, Deep Start does not necessarily acquire samples on a regular basis, so the profile that Deep Start builds from its samples may be less accurate than that obtained by the other sampling-based tools. However, because Deep Start’s profile is used as a secondary source of qualitative behavioral information rather than the primary source of quantitative performance information, this potential lack of accuracy is acceptable in Deep Start. Furthermore, Deep Start does not replace the instrumentation-based search with sampling because sampling is not appropriate for obtaining certain types of performance information such as wall clock time and inclusive CPU utilization (i.e., the CPU utilization of a function and all of its direct and indirect callees).

2.4 Dynamic Instrumentation Management

The Distributed Performance Consultant introduces a new approach for managing dynamic instrumentation, consisting of a model for the instrumentation cost and a policy for scheduling instrumentation requests. In this section, we present previous work related to our instrumentation cost model and instrumentation scheduling policy.

Software-based performance monitoring systems affect the behavior of the application being studied. Because these effects are usually negative, they are often called “costs” of the monitoring system. For instrumentation-based monitoring systems, there are three approaches for dealing with instrumentation cost: ignore it, compensate for it, or control it. Although widely used due to its simplicity, the approach of ignoring the cost of instrumentation is difficult to justify. The approaches of compensating for and controlling instrumentation cost each require the monitoring system to incorporate a model of the cost of instrumentation.

The cost of software-based performance measurement has many dimensions, including the time required to execute instrumentation code, the memory required for temporary storage of performance data, the effect of instrumentation code and performance data on the system’s caches, and the differences between the uninstrumented and instrumented program’s ordering of events. Trace-based performance measurement systems also include a cost for storing performance data to trace files. On-line performance measurement systems may have a cost for communicating performance data among tool components.

Because the effects of instrumentation on an application are so complex, no existing model captures all dimensions of instrumentation cost. Existing models have focused on the dimensions that are easiest to measure or approximate. Malony and Reed [68] and Yan and Listgarten [109] propose cost models for trace-based instrumentation systems that try to compensate for the effects of instrumentation in the event traces. Malony and Reed use a single constant to model the cost of each instrumentation point, chosen as the average execution time of a benchmark program's instrumentation points. Yan and Listgarten extend this model by placing instrumentation points into a small number of categories, and assigning a constant instrumentation cost for each category. Unlike these models for trace-based instrumentation systems, Hollingsworth and Miller [47] present a simple model for predicting and measuring the cost of more general instrumentation in an on-line performance tool. Paradyn's Performance Consultant [74] uses their model in a feedback system that controls instrumentation cost during automated performance bottleneck searches.

None of these instrumentation cost models is well suited for our scalable automated performance diagnosis strategy. Because our performance diagnosis strategy sometimes uses instrumentation that is more complex than the writing of a fixed-size trace record, the Reed and Malony and Yan and Listgarten instrumentation cost models are not appropriate for the Distributed Performance Consultant. The Hollingsworth and Miller model is better than these other models for representing the cost of instrumentation with a wide range of complexity, but it is also not well-suited for use in the Distributed Performance Consultant because it aggregates the instrumentation cost for parallel computation into a single value. To control the effects of instrumentation requested by search agents running on each node of a parallel system, the Distributed Performance Consultant must be able to monitor instrumentation cost at the granularity of individual nodes, but this information is lost in Hollingsworth and Miller's aggregated instrumentation cost value. In short, there is a need for a rigorous model of instrumentation cost in parallel computation that can be used in a scalable, distributed tool.

In addition to the Distributed Performance Consultant's need for a new instrumentation cost model, there is a need for a new policy for scheduling dynamic instrumentation. The instrumentation scheduling problem facing the Distributed Performance Consultant is isomorphic to the problem of scheduling a mixed sequential/parallel workload on a parallel system, where local instrumentation requests correspond to sequential jobs and global instrumentation corresponds to parallel jobs. While there is no existing work on distributed instrumentation cost models, previous research has been done on parallel scheduling algorithms. This existing work has investigated scheduling policies for this type of mixed sequential/parallel workload, including studies by Leutenegger and Vernon [62] for shared-memory multiprocessors, Arpaci et al [6] for a network of workstations, and Arpaci-Dusseau [7] for avoiding the scalability problems of traditional gang scheduling systems with implicit coscheduling.

The Distributed Performance Consultant's local/global instrumentation request scheduling policy draws upon this previous research to solve a different and more restricted problem. In the Distributed Performance Consultant, gang scheduling global instrumentation requests is a requirement, not just a performance benefit. Also, whereas the mixed workload job scheduling problem usually considers parallel jobs that have a different number of processes than the number of available processors, our problem is limited to instrumentation requests for a single node of the parallel system or for all nodes in the system. In addition, although preemptive scheduling of instrumentation requests is possible, our initial investigation into scalable automated performance diagnosis uses a simpler, non-preemptive instrumentation scheduling policy.

2.5 Scalable Data Visualization

A well-designed data visualization can be easier to understand than a textual presentation of the same data, making data visualization a popular approach for understanding the behavior of a parallel computation. Insightful performance data visualizations are especially important for understanding the behavior of large-scale computations, where the volume of performance data and the complexity of the interaction between application software, system software, and the hardware can make it challenging to identify and understand interesting application behavior. The most common technique for visualizing the behavior of a parallel computation is to represent each entity of the computation (e.g., each process) with its own graphical element in an aggregate display (e.g., an individual bar within a bar chart containing multiple bars). This technique is popular because it explicitly represents the computation's parallelism in the visualization, but it does not scale. With this technique, the number of graphical elements in the visualization is the same as the number of entities in the computation. However, the number of graphical elements that can be visualized is limited by the display technology, limiting the size of the computation whose data can be visualized using this technique. Research into scalable visualization techniques is an area of ongoing research. In this section, we survey existing techniques for scalable data visualization and relate them to the Sub-Graph Folding Algorithm (SGFA), our approach for visualizing the results of an automated performance bottleneck search.

Couch [18] proposes an abstract technique for constructing scalable performance visualizations that groups processes into categories and displays only per-category statistics. Both Couch's Seeplex tool and our SGFA put this approach into practice. However, Seeplex requires significant user involvement to manually categorize the data, whereas SGFA categorizes processes without user interaction. Also, Seeplex categorizes quantitative data, whereas SGFA categorizes processes using qualitative data describing the results of a performance bottleneck search.

Kimelman et al [56] describe a technique for reducing the complexity of dynamic graph-based displays like search history graphs (see Figure 1.2). In their approach, the user specifies criteria to select nodes from the complete graph. The user then "disposes" of the selected nodes by placing them in the background visually, removing them from the graph, or grouping them into a new meta-node. Their approach re-applies the user's node disposal preferences as the graph changes. Because their technique produces graphs that are less complex than the

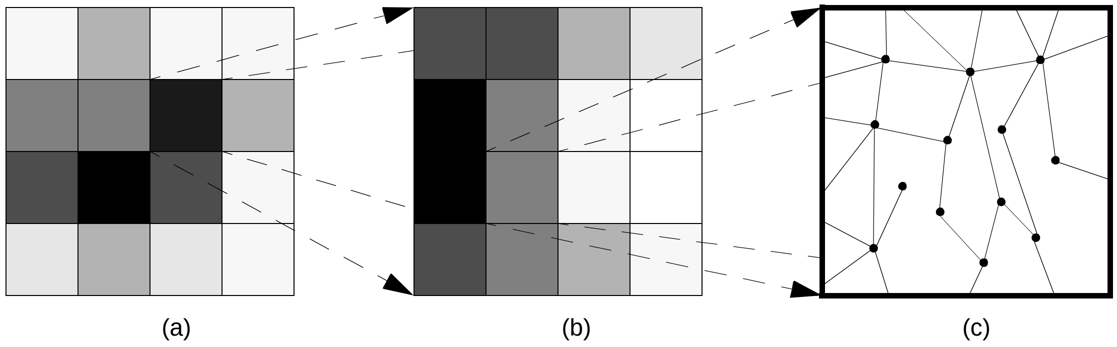


Figure 2.1 A semantic zooming display of a graph. Shades of gray are used to indicate the number of nodes in the sub-graph associated with each block. When zoomed out to show the entire graph (a), the display provides a coarse indication of the node distribution in the graph. Zooming in (b) provides more detail. When zoomed in as far as possible, the display shows the graph's individual nodes and edges (c).

original graph, their technique may be used to improve the scalability of graph-based displays. SGFA is similar to their technique because it combines several nodes from the full search history graph into a single node in the folded sub-graph to reduce graph complexity. However, their technique requires the user to define the node selection and disposal rules manually, whereas SGFA operates automatically. On the other hand, their technique is more generally applicable than SGFA because it does not require the graph to have sub-graphs with a sensible categorization. Whereas SGFA takes advantage of the regular structure present in Paradyn's qualitative performance data, their technique makes no assumptions about the underlying data.

Stasko and Muthukumarasamy [98] propose the *semantic zooming* technique for constructing scalable graph-based displays (see Figure 2.1). Unlike the zooming functionality found in many graphical tools, semantic zooming does not imply strict magnification and demagnification as the user zooms the display into and out of the data. Instead, when zoomed out of a graph-based display like the search history graph display, semantic zooming represents the graph as a matrix of blocks where each block represents a sub-graph of the underlying graph. For each block, the characteristics of the sub-graph associated with the block are indicated by the graphical characteristics of the block, such as its color and texture. For example, in Figure 2.1 the blocks that represent sub-graphs with many nodes have a darker color than those representing sub-graphs with few nodes. Figure 2.1a shows the display when zoomed out to show the entire graph. At this zoom level, the display provides a coarse idea of the distribution of nodes in the graph. To zoom in to the next level of detail, the user selects a single block from the display and the display adjusts so that the sub-graph associated with the selected block is represented by all the blocks in the display (Figure 2.1b). When zoomed in far enough, the display shows the individual nodes and edges of the underlying graph (Figure 2.1c). Although a rigorous user interface study is needed to decide conclusively, we believe many users will prefer an SGFA-based search history graph display over a semantic zooming display because the SGFA display always retains the familiar graph-like characteristics (i.e., nodes and edges) of the underlying graph, whereas the semantic zooming display does not.

SGFA's folding operation is similar to that used by the PRISM parallel debugger [94] and Karavanic's structural merge operator [53]. PRISM uses a *Where Tree* display (see Figure 2.2) to show the call stacks from multiple SPMD application processes in a single, scalable representation. In this display, each node represents a function call in one or more processes. Nodes from different process' call stacks are grouped together into a single tree node if they call the

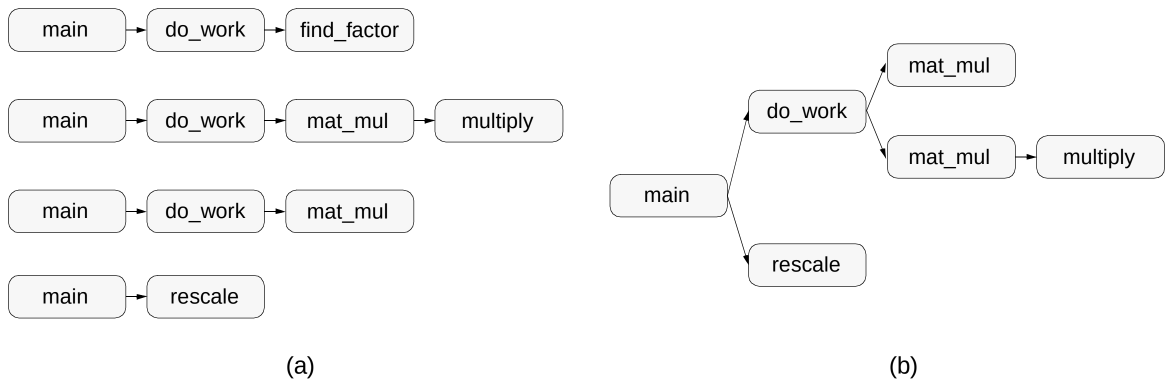


Figure 2.2 Example of folding in the PRISM debugger's Where Tree. Four call stacks (a) are folded into a single tree (b).

same function at the same call site. Because each application process has the same program entry point (e.g., a C program's `main` function), the node that represents the program entry function is the root of the Where Tree. Identical call sequences from this entry point function are represented as common node sequences in the tree. Karavanic's structural merge operator also forms a composite tree by folding equivalent graph nodes together. The structural merge operator combines two trees of program resources called EventMaps. Each EventMap represents the program resources involved in a single application run. The structural merge operator combines equivalent nodes from the two EventMaps to form a composite tree containing the program resources that were present in both application runs. Like the PRISM folding operation and Karavanic's structural merge operator, SGFA combines equivalent nodes from source graphs into a composite graph. In contrast to the PRISM folding operation, SGFA folds trees rather than linear sequences of stack frames. Also, SGFA considers node types and truth values in addition to node names when determining which nodes can be folded together, whereas PRISM only considers the function name. SGFA differs from Karavanic's structural merge operator in that the structural merge operator requires the user to provide mappings that indicate equivalent nodes, whereas SGFA's built-in node equivalence rules allow SGFA to operate without user intervention.

2.6 Summary

Our research is related to previous work in several areas. The MRNet scalable multicast and data reduction infrastructure builds on previous work in collective communication infrastructure but provides flexibility in high-throughput data reductions that is not found in the existing work. Our scalable tool start-up techniques and performance data aggregation approach leverage MRNet's flexibility and performance to greatly improve tool scalability. Our scalable tool start-up techniques also apply the general approach of categorizing application processes based on some measurable characteristic and then operating only with a representative or summary of each category. The Distributed Performance Consultant builds on previous work in automated performance diagnosis tools, but addresses scalability issues of performance data management and tool control that are not addressed by existing on-line automated performance diagnosis tools. The Deep Start hybrid strategy for automatically finding application performance problems augments instrumentation with sampling, a novel approach for automated performance tools. Finally, our Sub-Graph Folding Algorithm applies a process categorization technique like that used by our scalable tool start-up strategy to the domain of scalable visualization. Its graph folding operation is similar but somewhat more complex than the graph folding or merging operations found in existing tools.

Chapter 3

The Performance Consultant

The Performance Consultant is the automated performance diagnosis component of the Paradyn performance tool. Because the Performance Consultant provides the context for our research, this chapter briefly reviews the Performance Consultant to provide background for subsequent chapters. More complete descriptions of Paradyn and the Performance Consultant can be found in earlier publications [15,47,48,49,50,74].

The Performance Consultant automatically finds application bottlenecks by performing experiments that determine whether the application is exhibiting performance problems and, if so, where the problems are occurring. Initially, the experiments test the behavior of the application as a whole. If the Performance Consultant discovers global performance bottlenecks (by comparing observed performance against user-configurable thresholds), it performs new experiments that are more and more specific to pinpoint the nature and location of each bottleneck. Because dynamic instrumentation [49] is used to collect performance data,

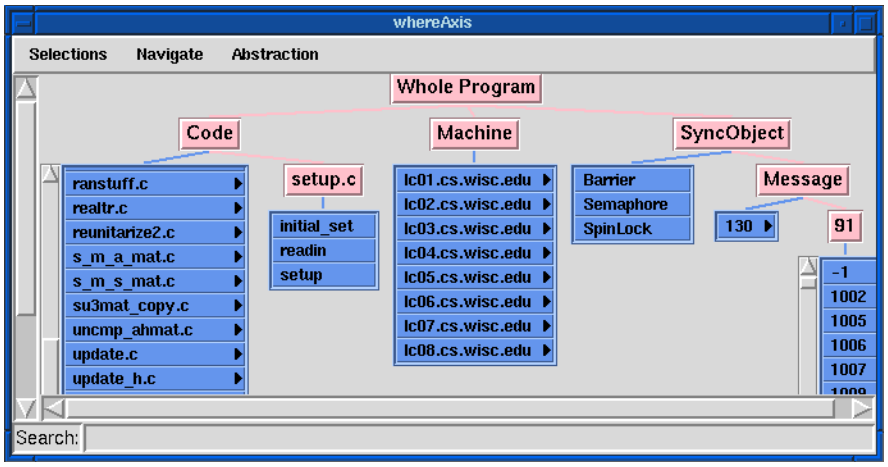


Figure 3.1 Paradyn organizes the application resources into hierarchies.

the Performance Consultant collects only the data it needs to test its experiments and can remove the instrumentation when it is no longer needed.

The Performance Consultant involves a small number of key concepts:

- **The Hypothesis.** An experiment's *hypothesis* is a reason why an application may be performing poorly. For example, "too much time spent blocked for I/O" is one of the hypotheses used by the Performance Consultant. An experiment's hypothesis determines the instrumentation that is used to collect performance data for the experiment. Each experiment tests a single hypothesis.

- **The Resource.** An application's *resources* are the entities that comprise the application, both statically and as it runs. The functions that constitute the application's executable code are resources, as are its processes and the hosts on which they are running. The synchronization objects used by the application such as message tags, barriers, spin locks, and semaphores are also program resources. Paradyn organizes resources into hierarchies by type (see Figure 3.1). Each resource represents all resources organized beneath it within its hierarchy. For example, the `Code` resource represents all functions in the application's executable(s).

- **The Focus.** A *focus* is a tuple containing one resource from each resource hierarchy. A focus provides a concise name for a potentially large set of resources. For example, the focus

  ```
  </Code/setup.c,/Machine/lc01.cs.wisc.edu,/SyncObject>
  ```

  denotes all functions from the source file `setup.c`, but only if they are executing in processes running on the host named `lc01.cs.wisc.edu`. This focus specifies the top level SyncObject resource, so it does not constrain the set of resources it names to any synchronization object or type of synchronization object. Each experiment reflects both a hypothesis and a focus.

- **Experiment activation.** An experiment is *activated* when the Performance Consultant requests that instrumentation code be inserted into the application to collect the performance data needed to evaluate whether the experiment is true (i.e., its hypothesis is true at its focus).

- **Search refinement.** If an experiment is true, the Performance Consultant uses a technique called *refinement* to continue its bottleneck search by creating one or more new experiments that are more specific than the original experiment. New experiments can be defined by refining either the experiment's hypothesis or its focus. However, because the Performance Consultant uses a small number of predefined hypotheses, search refinement usually involves the refinement of an experiment's focus. To refine a focus, the Performance Consultant replaces one of the resources in the focus with another of the same type. If the resource being replaced is a Code resource (i.e., a resource representing a function), the new foci are generated by replacing it with the resources representing its callees in the application's call graph. If the resource being replaced is not a Code resource, the replacement resources are the children of the original resource from the Paradyn resource hierarchy. Refinement allows the Performance Consultant to perform a step-by-step narrowing of its search to examine a more specific hypothesis or set of resources when it determines an experiment is true.

- **The search path.** A *search path* is a sequence of experiments related by refinement. The Performance Consultant terminates a search path when it cannot refine the last experiment on a path. Because the refinement of an experiment may result in the creation of more than one new experiment, the Performance Consultant may have many search paths that are active simultaneously (i.e., are not terminated). Thus, the Performance Consultant may have a large number of experiments under consideration at any given time. To keep the effects of instrumentation to tolerable levels, the Performance Consultant throttles experiment activation to keep the instrumentation's cost below a user-configurable threshold.
- **The search history graph.** A *search history graph* records the cumulative refinements of a search. Paradyn's Search History Graph display (see Figure 1.2) represents this graph visually. This display is dynamic; nodes are added to the graph as the search is refined. Node label and background colors indicate whether the experiment is active and its truth state, respectively.

Also, by clicking on graph nodes users can obtain detailed information about each experiment including its full hypothesis and focus names, the current value of the performance metric it uses to determine its truth state, and when it started and stopped collecting performance data.

Chapter 4

Scalable Multicast/Reduction Overlay Networks

Two of the tool scalability barriers presented in Chapter 1, the management of large-volume performance data flows and communication between a large number of distributed tool components, are encompassed by the same high-level problem: the problem of *monitoring and controlling a large number of distributed components*. For the large-scale parallel computing systems of today and the near future, "a large number" means thousands or even tens of thousands of components, but the need for tools that can monitor and control hundreds of thousands of components looms on the horizon.

There are several high-level requirements that must be satisfied by any solution to the large-scale monitoring and controlling problem. A solution must:

- Be applicable to on-line tools, because of their desirability for large-scale performance diagnosis and tuning (see Chapter 1);
- Enable a tool to process monitoring data as quickly as necessary to support the tool's purpose. (With the first requirement, this implies that a solution

must enable tools to process monitoring data at the same rate as it is being generated to support on-line tools that adapt its behavior based on its collected data);

- Enable a tool's front-end to issue control requests scalably to any subset of its back-ends;
- Enable any subset of a tool's back-ends to deliver responses scalably to its front-end; and
- Provide a scalable approach for starting and connecting tool processes.

To address the large-scale monitoring and controlling problem, we propose the use of scalable overlay networks that provide multicast and data reduction services. We call these networks Multicast/Reduction Overlay Networks, or MRONs. An MRON-based tool interposes a hierarchical network of processes between its front-end and back-ends as shown in Figure 4.1, unlike a typical parallel tool where each back-end is connected directly to the tool's front-end (see Figure 1.1). A tool can use an MRON to parallelize and distribute important tool activities, yielding a twofold benefit: parallelization reduces the latency of each activity and distribution reduces the tool front-end's computation and communication load.

This chapter presents our research into using MRONs to address the large-scale monitoring and controlling problem. Section 4.1 describes our MRON approach and discusses how it addresses the large-scale monitoring and controlling problem by presenting the design and implementation of MRNet, our initial implementation of the MRON concept. Section 4.2 discusses issues involved in choosing the process layout, or connection topology and process placement strat-

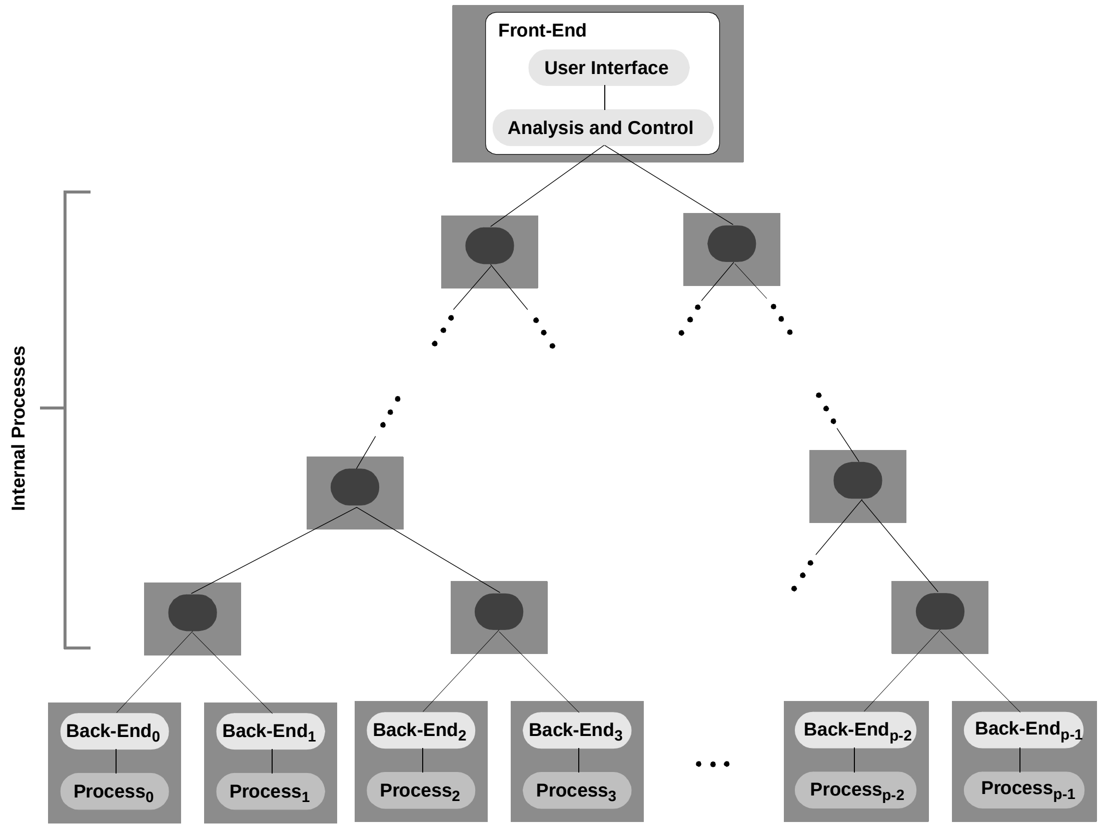


Figure 4.1 The components of an MRNet-based parallel tool. Shaded boxes represent potential host boundaries. Dark ellipses represent internal processes of the overlay network. Any tree topology is possible between front-end and back-ends; a binary tree topology is shown. Compare this organization with that of a typical parallel tool (Figure 1.1).

egy, of MRNet-based tools. Section 4.3 presents our experience when applying MRNet to the Paradyn parallel performance tool, including its use for non-conventional collective communication operations like partitioning tool back-ends into equivalence classes and finding the skew between the clock's of the tool front-end and each back-end process. Section 4.4 presents the results of our evaluation of MRNet's low-level behavior and its behavior when integrated into Paradyn. Section 4.5 concludes the chapter with a summary of our MRON research and a discussion of potential future work.

Our MRON research makes several research contributions:

- A list of requirements for solving the large-scale monitoring and controlling problem. This list, presented above, builds on our delineation of the tool scalability problem space into three scalability barriers (see Chapter 1);
- An MRON design. MRNet's design embodies our approach to implementing MRONs. This design incorporates techniques for scalable communication and data processing that enables tools to address the large-scale monitoring and controlling problem;
- Evaluation of MRNet's behavior and performance. Our MRNet evaluation confirms the scalability of our MRON approach's low-level collective communication operations;
- Exploration of a wide range of data reductions. As part of our work in integrating MRNet into Paradyn, we explored a wider range of non-conventional data reductions than had been investigated previously.

4.1 MRNet Design and Implementation

Our approach to building MRONs is embodied in the design and implementation of tool infrastructure called MRNet. MRNet-based tools are organized as a tree of processes, with the tool's front-end at the root and its back-ends at the leaves of the tree (Figure 4.1). The processes between the root and leaves are called *internal processes*. MRNet does not dictate the process layout, or connection topology and process placement strategy, of tool and MRNet internal processes. Instead, tools use a simple text-based language called the MRNet

Configuration Language, or MCL, to specify the desired layout. Tools can use common topologies like k-ary and k-nomial trees or custom topologies tailored to the system(s) running the tool. For instance, a tool may use a custom layout so that the overlay network topology reflects the physical topology of the underlying system's interconnect.

Using MRNet, tools can efficiently multicast data and compute averages, sums, and other more complex data aggregations. Tools send data across MRNet on logical data channels called *streams*. A stream transfers data between the tool's front-end and some collection of its back-ends. Data travelling toward the tool's front-end is said to be flowing *upstream*; *downstream*-flowing data is travelling toward the tool back-ends. Regardless of its direction, data transferred on a stream can be manipulated using *filters* running throughout the overlay network. Using filters, MRNet can manipulate data sent over a stream in parallel to aggregate data efficiently.

The hierarchical communication structure used in our approach to MRONs has been studied previously in a variety of contexts (e.g., [8,27,55,66,69,97,106]). However, several features make our approach better-suited as a general facility for building scalable parallel tools:

- **Flexible organization.** Tools can use any tree organization for the MRNet process layout, allowing tools to customize the process layout for the system running the tool.
- **Scalable, flexible data aggregation.** MRNet filters are the mechanism for efficient computation of averages, sums, concatenation, and other conven-

tional data reductions. In our MRON approach, custom filters can be added to MRNet to perform tool-specific aggregation operations. For example, when we integrated MRNet into Paradyn (see Section 4.3), we used a custom histogram filter to partition the application processes into equivalence classes based on characteristics of each process such as its collection of functions. We also used a custom filter to implement a scalable algorithm for detecting the clock skew between the tool's front-end and each Paradyn daemon. No other overlay network system has explored such a rich variety of aggregation operations.

- **High-bandwidth communication.** Data is transferred within MRNet in an efficient, packed binary representation. Copy-free data paths are used when possible for low-cost transfer of data through the overlay network.
- **Scalable multicast.** When sending control requests to tool back-ends, serialization limits the scalability of existing tools as the number of back-ends increases. MRNet's efficient message multicast enables low-cost communication of control requests from the tool front-end to its back-ends.
- **Multiple concurrent data channels.** MRNet supports multiple concurrent data streams. Multicast and data reduction takes place within the context of a stream, so more than one multicast or data reduction operation can be active simultaneously.

MRNet enables tools to address the problem of monitoring and controlling a large number of distributed components. MRNet's data aggregation facility enables scalable processing of *global* application data, data that describes the behavior of all application processes as a whole. MRNet-based tools can off-load their global data processing onto the processes of the overlay network, reducing the processing and communication load on the tool's front-end. In addition, MRNet's multicast and data aggregation capabilities enable scalable communication of requests and responses between the tool's front-end and its back-ends.

4.1.1 MRNet Abstractions

A tool's front-end and back-ends interact with MRNet using a small number of abstractions. MRNet implements the abstractions in a C++ API. All classes and type definitions in the API are contained within a C++ namespace called "MRN." MRNet's abstractions are:

- **Network:** For the tool's front-end and back-end processes, the `Network` object abstracts the MRON. The tool's front-end uses its `Network` object to configure the MRON, for example to create streams. In MRNet, tool back-ends do not create streams. Instead, back-ends become aware of new streams when they first receive a message sent along the new stream. To support this model, the back-end's `Network` object provides a method for receiving a message without a priori specification of the message's stream.
- **EndPoint:** When creating a stream, the tool's front-end must indicate which back-ends will communicate using the stream. In MRNet, each tool back-end is represented using an `EndPoint` object. Because communication between tool back-ends is not supported in our MRON communication model, tool back-ends do not use `EndPoint` objects.

- **Communicator:** A `Communicator` object holds a collection of one or more `EndPoint` objects. `Communicators` provide a concise way to specify which tool back-ends are involved in MRNet collective communication operations, providing a similar function to communicators in MPI [71,72].
- **Stream:** A `Stream` object implements the MRNet concept of a logical data channel. In MRNet, each `Stream` is associated with exactly one `Communicator` that specifies which back-ends can transfer data across the `Stream`. To give a tool a means to differentiate data intended for different purposes but sent to the same collection of back-ends, a `Communicator` object can be associated with several `Stream` objects. For example, in MRNet there is one broadcast `Communicator` that specifies all back-ends. Any collective communication operation that involves the front-end and all back-ends can share the broadcast `Communicator`. A tool's front-end creates MRNet streams using a method provided by its `Network` object, but tool back-ends do not create `Stream` objects directly. Instead, a back-end obtains `Stream` objects via a data receive operation provided by the back-end's `Network` object. This operation allows a back-end to receive data without specifying the stream on which the data is sent; a `Stream` object representing the data's stream is provided via an output parameter of the receive operation.
- **Filter:** When a tool front-end creates a `Stream` object, it provides the IDs of the synchronization and transformation filters to be used by the stream.

4.1.2 MRNet Data Transfer

With any communication infrastructure, there are two high-level aspects to data transfer: the data transfer operations provided in the infrastructure's API and how the infrastructure transfers data internally. We describe these two aspects in this section.

Several styles of data transfer operations have been used in the APIs of existing communication infrastructures. The styles differ in whether programs explicitly manage a data buffer and whether data is added to and removed from the buffer using a single API operation or a sequence of operations. For example, in the PVM [34] communication model, programs explicitly manage a data buffer when transferring multiple data items; the buffer is packed and unpacked using a sequence of operations, one per data item. In contrast, a program that uses an XDR [65] library to send data over a TCP/IP socket sends multiple data items using a sequence of operations, but can rely on the XDR library to buffer the data until the sender calls an XDR "end of record" function.

An example of MRNet's API is shown in Figure 4.2. The tool front-end begins by creating its `Network` object to instantiate the MRON (Figure 4.2a, line 1), providing the name of a configuration file containing the desired MRNet process layout. Next, the front-end obtains the broadcast `Communicator` from its `Network` and uses it to build a `Stream` for communicating with all of the tool's back-ends (line 2). The `Stream` is configured to use a synchronization filter that implements the "wait for all" synchronization policy (see Section 4.1.4) and a "floating point maximum" transformation filter on upstream-flowing data, and no transforma-

```
    front_end( const char* config_file_path, int nvalues, float* values )
    {
       float result;
       void* buf = NULL;

1      Network* mrn = new Network( config_file_path );
2      Communicator* bc_comm = mrn->get_broadcast_communicator();
3      Stream* stream = new Stream( bc_comm, SYNC_WAIT_FOR_ALL,
                                                TRANS_FP_MAX,
                                                TRANS_NULL );

4      stream->send( VALUE_TAG, “%af”, values, nvalues );
5      stream->recv( buf );
6      stream->unpack( “%f”, &result );
    }
```

(a)

```
    back_end( const char* host, unsigned short port )
    {
       Stream* stream = NULL;
       float* values = NULL;
       unsigned int nvalues;
       void* buf = NULL;
       int tag;

1      Network* mrn = new Network( host, port );

2      mrn->recv( &tag, &stream, &buf );
3      stream->unpack( “%af”, &values, &nvalues );

4      float result = do_something( values, nvalues );

5      stream->send( “%f”, result );
    }
```

(b)

Figure 4.2 Example tool code using the MRNet API. Example code is shown for both the tool’s front-end (a) and its back-end (b). Error checking code and a statement allowing use of the MRN namespace without explicit scoping are omitted for the sake of readability.

tion filter on downstream-flowing data. Next, the front-end sends an array of floating point values on the `Stream` (line 4). MRNet’s data transfer operations are similar to the C standard library’s `printf` and `scanf` functions, including the use of a format string to specify the type(s) of the data being transferred.

Unlike `printf` and `scanf` format strings, MRNet format strings also support notation for specifying arrays of simple types. For example, the format string `"%d %f %aud"` indicates data consisting of an integer, a floating point number, and an array of unsigned integers. Although it is not specified in the format string, MRNet requires the number of elements in an array to be specified when it is sent, and provides the number of elements with the array when it is received. In the example code, the send operation takes a format string, a tag, and the array to be transferred. Next, the front-end receives (line 5) and unpacks (line 6) a response from the `Stream`. When the data is received, the front-end obtains the tag value and an opaque data buffer containing the transferred data. The front-end calls an unpacking operation with a format string (perhaps chosen based on the tag value received with the data buffer) to extract the response from the buffer. Separating data receipt and unpacking enables a receiver to use event-driven control flow. For example, the receiver can employ an event loop that receives the next available data buffer without a priori specification of the data's sender or its type(s). Once the data buffer and tag are received, the receiver can take the appropriate action to extract the data items from the buffer. In the example, the response received by the front-end is the maximum value of the back-end's responses.

The code of the tool's back-end (Figure 4.2b) is symmetric to the front-end code. The back-end first connects to MRNet by creating its `Network` object (line 1), specifying the process to which it must connect. Next, it receives the array of floating point values from the front-end on a `Stream` (line 2) and unpacks the array elements and length (line 3). The back-end performs some unspecified operation on the data to produce a floating point value (line 4) and sends the value to the tool front-end on its `Stream` (line 5).

Internally, MRNet transfers data across its overlay network in the form of messages containing one or more packets. Each packet holds a collection of typed data and type information is transferred with a packet as it is transferred across the overlay network. For efficient transfer between processes, packets keep data in packed binary form. Data is packed into and unpacked from these buffers using functions that provide the XDR interface but use a "receiver makes right" approach like that of PBIO [23] when transferring data between heterogeneous hosts. For efficient delivery of multi-packet messages, MRNet uses system functions that transfer multiple non-contiguous memory buffers with a single system call (e.g., `readv` and `writev` on UNIX systems) if such functions are available.

4.1.3 MRNet Filters

In our MRON design, filters manipulate data as it is transferred through the overlay network. MRNet uses two types of filters: *synchronization filters* that synchronize data arriving asynchronously at MRNet processes, and *transformation filters* that manipulate data, for example to compute its sum or average. In our approach, filters are implemented as C++ functions that take a collection of input packets and produce a collection of output packets. Filters can be stateful, and custom filters can be dynamically loaded into MRNet to perform tool-specific synchronization or aggregation operations.

MRNet uses synchronization filters to synchronize data that arrives asynchronously to an MRNet process. Synchronization filters enable an MRNet process to ensure that it has a collection of packets synchronized according to some synchronization policy before it manipulates the data in the packets. We have identified three synchronization policies that we feel are generally useful:

- *Wait for all.* Under this policy, the filter collects input packets until it has at least one from each child connection involved in the filter's stream. The collection of packets produced by this filter includes exactly one packet per child connection.
- *Wait for all with timeout.* This policy is the same as the *Wait for all* policy, except that it produces packets if a user-configured timer expires before a packet has been collected from each child connection. The collection of packets produced by this filter includes at most one packet per child connection; if the packets are produced in response to the expiration of the timer, packets may be missing from some of the child connections.
- *Do not wait.* Under this policy, the synchronization filter produces packets as soon as they are received.

Transformation filters produce a collection of output packets in response to a (potentially synchronized) collection of input packets. Transformation filters can implement simple data aggregation operations like summation and averaging, or more complex operations like the computation of a sliding window average. For example, when applying MRNet to Paradyn, we used transformation filters for unconventional operations like a histogram filter that partitions application processes into equivalence classes based on the characteristics of each process.

To support complex synchronization or transformation behavior, MRNet filters can retain state. In synchronization filters, this state can be used to hold packets that are not yet ready to be passed along to the stream's transformation filter. In transformation filters, this filter state may be used to support complex aggregation operations such as sliding window averages. Filter state is kept on a per-filter, per-stream basis, so stateful filters need not contain complex logic to manage filter state for a dynamic collection of streams.

No collection of synchronization and transformation filters is appropriate for all tools, so our MRON design supports the use of custom filters. In general, there are two approaches to incorporating custom filters into an implementation of our MRON design: custom filters may be incorporated at compile time or at run-time (i.e., dynamically). The compile-time approach is simpler to implement, but it produces a tool-specific executable file for internal processes. On the other hand, a dynamic approach allows the internal process executable to be tool-independent. Also, the dynamic approach provides flexibility in how custom filter code is incorporated into internal processes. For example, the custom filter code may be distributed from the tool's front-end to each internal process, or each internal process may load the custom filter code from the file system available to it.

MRNet supports dynamic loading of custom filters into internal processes. In its initial implementation, MRNet requires the custom filter code to be in a shared object located within a file system accessible to the tool front-end and each internal process (e.g., a file system shared across all hosts that run MRNet internal processes). Dynamic loading of tool-specific filters is initiated by the tool's front-end, but carried out by the tool front-end and all internal processes. To load a tool-specific filter, the front-end calls the `Network` object's `load_filterFunc` method with the filter function's name and the pathname of the shared object that contains the function. The front-end delivers a request containing this information to all MRNet internal processes. Upon receiving the request, each process uses system functions to load the shared object and to obtain the address of the filter function (e.g., `dlopen` and `dlsym` on UNIX) in the process' address space. If successful, the `load_filterFunc` method returns an identifier representing the newly-loaded filter. This identifier can then be used in subsequent `Stream` creation.

4.1.4 Data Handling in MRNet Processes

MRNet's scalability is derived from the parallelism made possible by its hierarchical process network. A tool's front-end and internal processes perform most of the MRON-related data handling. These processes execute the filters that manipulate the data sent across MRNet streams, and these processes route downstream-flowing data so that it eventually reaches the back-ends associated with the stream.

The tool front-end and internal processes handle data differently depending on whether it is flowing upstream or downstream. Transformation filters can be applied to manipulate the data flowing in either direction, but synchronization filters are needed only for data flowing upstream because only upstream-flowing data arrives on multiple input connections. The actions taken when processing upstream-flowing data are illustrated in Figure 4.3. When a message arrives (Figure 4.3a), its individual packets are extracted (Figure 4.3b) and passed into the synchronization filter associated with the message's stream (Figure 4.3c). If the addition of a packet causes the synchronization filter's synchronization criteria to be met, it passes its collection of synchronized packets to the transformation filter (Figure 4.3d). The transformation filter applies its transformation operation and produces a collection of output packets (Figure 4.3e). In the example, the transformation filter has reduced the four input packets to one output packet. Any packets produced by the transformation filter are added to the outgoing message. Once all incoming packets have been handled, the outgoing message is either queued for receipt by the tool (if the message is being handled in the tool's front-end) or is sent to the process' parent process (Figure 4.3f).

Synchronization filters are not needed for data flowing downstream, but downstream-flowing data must be routed to the correct outgoing downstream connections to reach the appropriate back-ends. The actions taken for downstream-flowing data are illustrated in Figure 4.4. When a message arrives its packets are extracted (Figure 4.4a), as with upstream-flowing data. The packets are then passed to the transformation filter associated with the message's stream (Figure 4.4b). If the transformation filter produces any packets, they are added to the outgoing message. Once all input packets have been handled, the MRON process determines the outgoing connections that lead to the back-ends associated with the stream, and sends the outgoing message on each of those connections

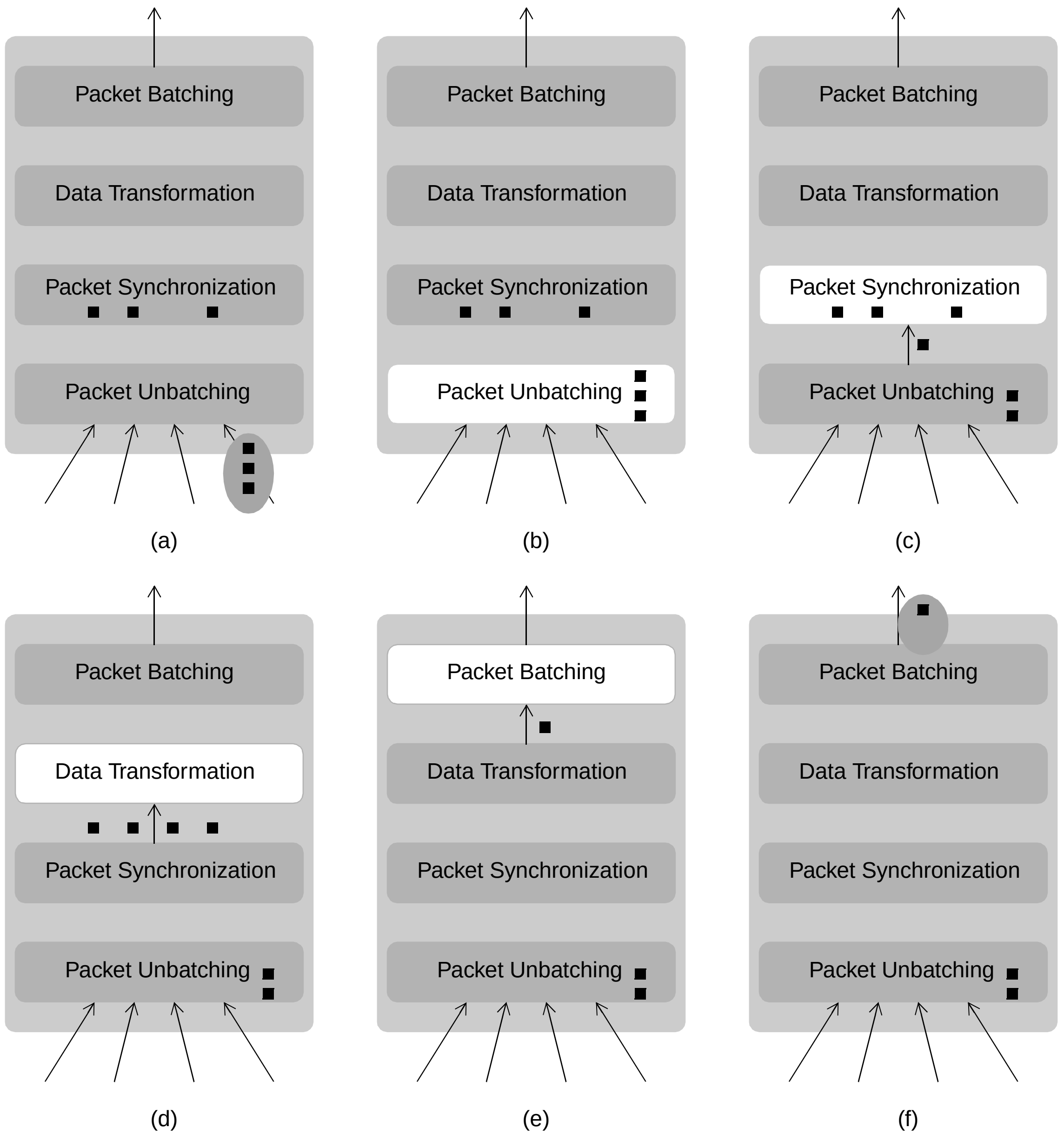


Figure 4.3 Handling of upstream-flowing data in an MRNet process. .A message with three packets (represented by a gray oval containing three black squares) arrives to an MRON process (a). The packets are extracted from the message, and passed one by one to the synchronization filter associated with the message's stream (b). If the addition of the new packet causes the synchronization filter's synchronization criteria to be met, a synchronized collection of packets is passed along to the stream's upstream transformation filter (c). The transformation filter produces a collection of output packets (e). The output packet(s) are batched into a message for later delivery to the process' parent (f).

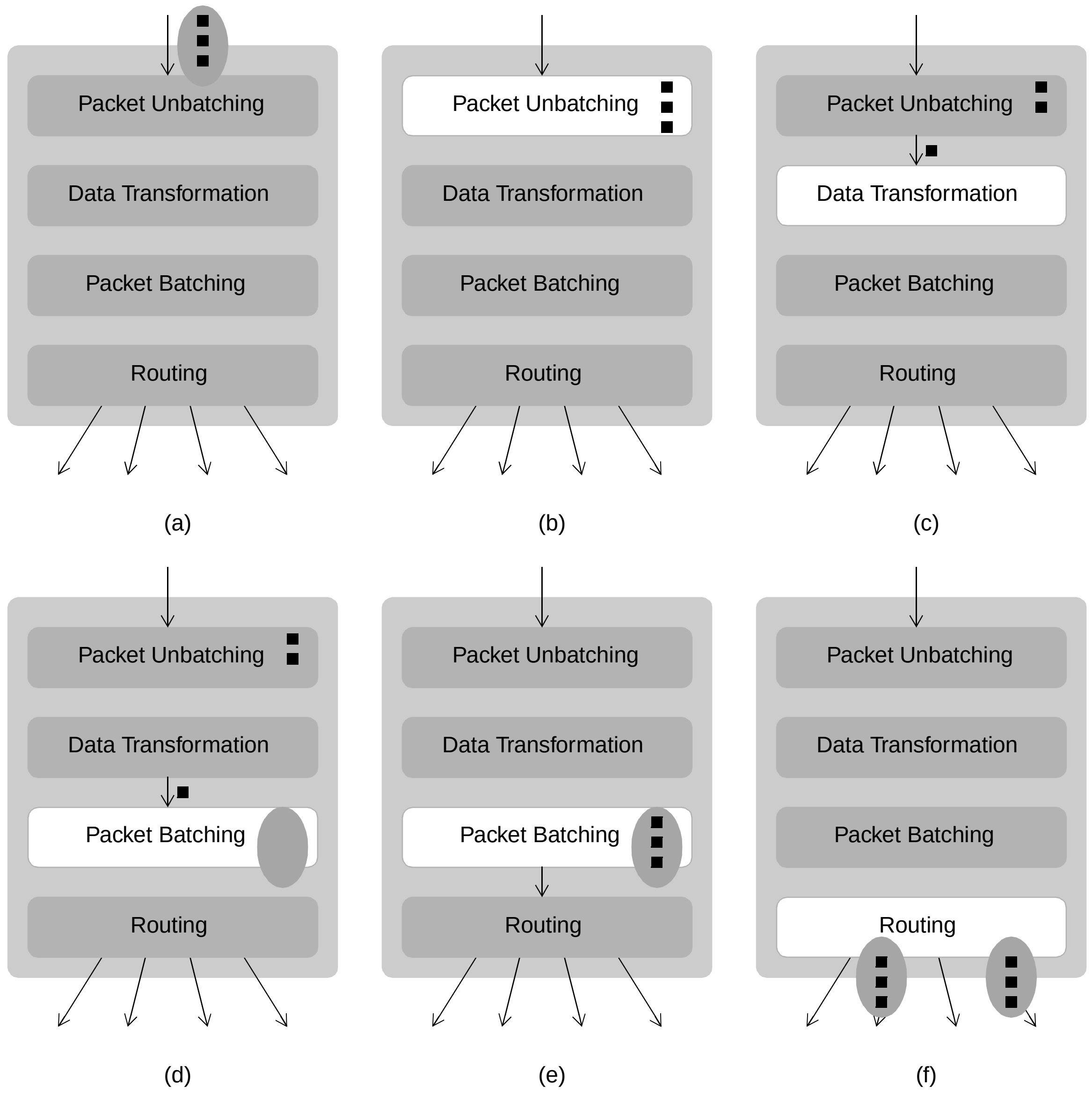


Figure 4.4 Handling of downstream-flowing data in an MRNet process. .A message with three packets (represented by a gray oval containing three black squares) arrives to an MRON process (a). The packets are extracted from the message (b), and passed one by one to the downstream transformation filter associated with the message’s stream (c). The transformation filter produces output packets that are batched into an output message (d). Once all input packets have been processed and batched into the output message (e), the output message is routed on the appropriate child connections to reach the back-ends associated with the stream (f).

(Figure 4.4c). As a performance optimization, if a stream does not use a transformation filter for downstream-flowing data, the packets of a downstream-flowing message sent on the stream need not be extracted before it is sent on outgoing connections.

Within the tool front-end and each internal process, MRNet implements the data handling approach described above using `StreamManager` objects. There is one `StreamManager` object per `Stream`, and the `StreamManager` object associated with a given stream manages all data processing for messages sent on that stream. When a message arrives at the tool front-end or one of MRNet's internal processes, the process first determines the stream to which the message belongs and delivers the message to that stream's `StreamManager` object. The `StreamManager` object extracts the message's packets, delivers each packet in turn to the stream's filters, collects any output packets into an outgoing message, and delivers the outgoing message on the appropriate outgoing connection.

4.1.5 Multithreading in MRNet Processes

Parallel systems whose nodes contain multiple processors are becoming commonplace. On such systems, MRNet processes can manage their connections using multiple threads for efficiency. Currently, MRNet creates one thread per input connection. Each thread performs a read operation on its input connection, blocking until data is available. Once data is available on the connection, the data is read and processed within the context of that thread. On hosts with multiple processors, the use of multiple threads allows MRNet to process data sent along different streams simultaneously.

To avoid problems caused by conflicting thread packages and to avoid introducing a thread package into single-threaded tools, parent and child processes can also operate in single-threaded mode. In single-threaded mode, MRNet polls all of its input connections for available data and blocks if none is available. However, in situations such as when a single-threaded tool must service both a connection to a user interface server and to MRNet, it is undesirable for MRNet to block if no data is available. In such situations, a useful approach for managing control in the tool front-end is to have the front-end poll for available input outside of MRNet, but to include the MRNet connections when polling. When input is available on one of the MRNet connections, the front-end gives control to MRNet to consume and process the data. To facilitate this approach, the `Network` abstraction provides a method for obtaining the local MRNet connections.

4.1.6 Instantiation of MRNet-Based Tools

Monitoring and controlling large-scale parallel computation involves two large-scale programs: a parallel application and a parallel tool. Applications typically use a parallel communication facility like MPI [71,72] or PVM [34] to communicate between processes. Tool processes monitor and control application processes, and communicate between themselves (e.g., to transfer monitoring data and control messages). Although some parallel communication facilities like PVM allow a tool to share the application's communication infrastructure, the model used by common communication facilities like MPI-1 [71] assumes that all processes participating in the parallel computation are application processes. From the perspective of a communication facility like MPI-1, a tool is a separate parallel program whose interaction with the application is outside the scope of the communication facility.

Large-scale computing environments usually provide a process management system, such as a batch queueing system, to start parallel applications. When starting application processes, a process manager provides the processes with the information they need to participate in the parallel computation. For example, IBM's Parallel Operating Environment (POE) [52] uses environment variables to pass information to the MPI run-time library in each application process, such as its position within the application's global MPI communicator.

Existing process management systems are not generally tool-aware. They reflect the assumption that all processes created to start a job are part of the same parallel application, so they provide no support for starting an application and a tool to monitor and control the application as part of the same job. To work around this limitation, parallel tools often request the process manager to start the tool's back-end processes, which then start the application processes. The tool back-end process is transparent to the application process, which behaves as if it were started directly by the process management system. Using this work-around, each application process obtains from the process management system the information it needs to participate in the parallel computation, yet is created under the control of a tool back-end process.

Although it is often desirable or even necessary to use a process manager to start tool back-ends, the lack of tool awareness in existing process managers makes them unsuitable for launching other types of tool processes such as MRNet's internal process tree. As noted above, existing process managers expect that all processes created to start a job are part of the parallel application. This assumption is reflected in the information they provide to the processes they create. If the process manager starts tool processes in addition to tool back-ends, the information provided to the tool back-ends (hence, to the application processes) is incorrect. For example, POE communicates the size of an application's global MPI communicator to the MPI library in each application process using an environment variable. If POE creates both tool back-end and internal processes as part of the same request, POE sets the environment variable to the total number of processes being created, not the number of application processes. Using one job to launch tool back-ends and another job to launch internal processes may work in some environments, but it is not a general solution because some environments do not guarantee that the two jobs will be scheduled together. In short, until process management systems become tool-aware, another approach to creating internal processes is needed.

In MRNet, we use a recursive technique with a remote shell utility to create and connect internal processes (see Figure 4.5). In environments without a process management system, the technique can be extended to create tool back-end processes in addition to internal processes. Beginning at the tool's front-end with configuration information specifying the topology and process placement of the entire tool process network, the root of each process tree (Figure 4.5a) creates the

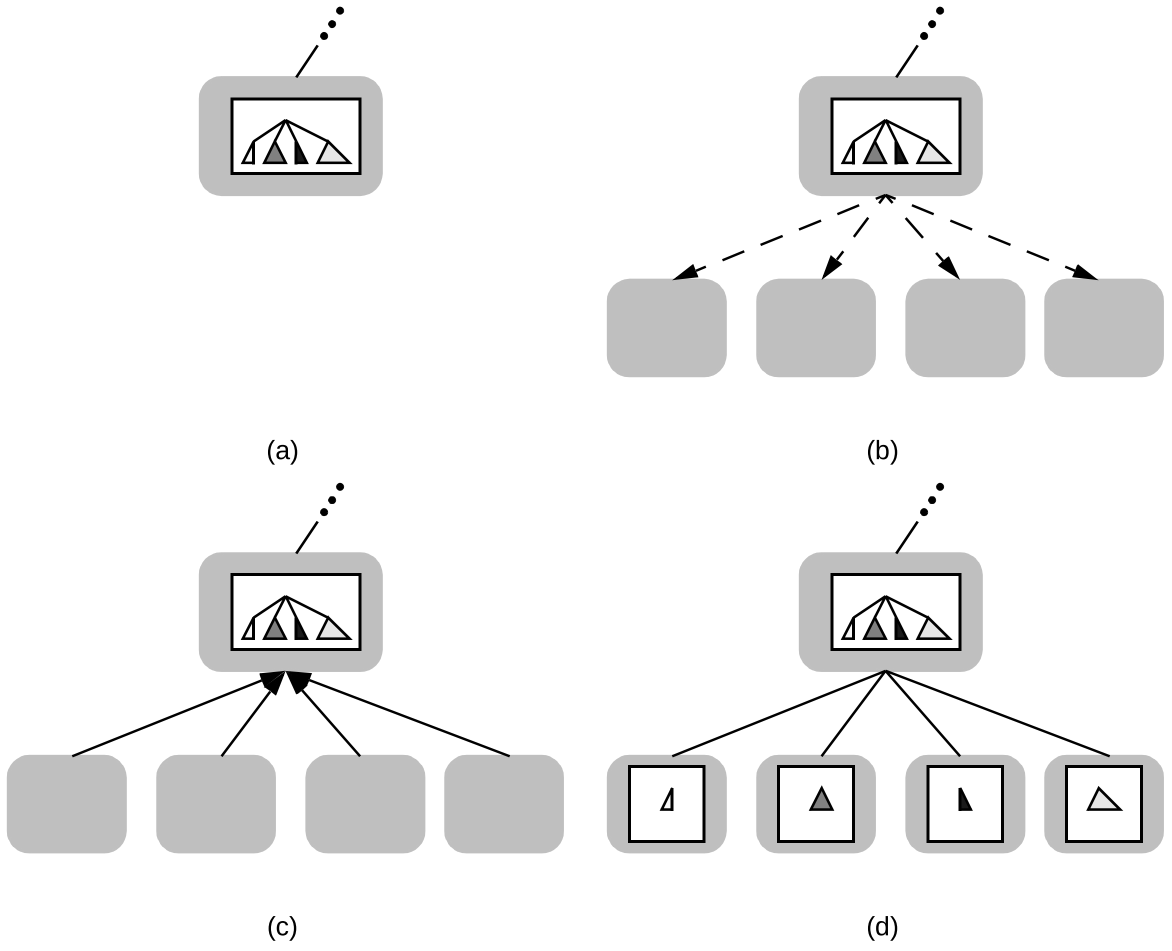


Figure 4.5 Recursive instantiation of MRNet's internal process network. Beginning with a process at the root of a sub-tree of the process network (a), the root process creates its directly-connected children sequentially (b). Each child connects back to the root process (c), and receives a description of its own sub-tree (d). Independent of its parent and its siblings, each child then uses the same approach to instantiate its sub-tree.

processes that are its directly-connected children (Figure 4.5b). The remote shell utility is used to create processes that are not located on the same host as the root process. Once started, each child process connects to its parent in the tree using information provided to it when it is created such as its parent's host name and the port number of a socket waiting for connections (Figure 4.5c). When a child process establishes a connection to its parent, the parent delivers the portion of the MRNet configuration that specifies the sub-tree rooted at the child, if any (Figure 4.5d). The child then instantiates its own sub-tree using the same approach, independent of further activity by parent and siblings. The recursion stops at processes whose only children are tool back-ends. With our recursive approach, different branches of the internal process tree can be instantiated in parallel for scalable instantiation of MRNet's process tree.

As described, our recursive approach creates and connects only internal processes. In environments where a process management system must be used to create the tool back-ends, MRNet returns control to the tool's front-end after creating the internal processes along with information its back-ends need to connect to the leaves of the tree of internal processes. Then, the front-end issues a request to the process management system to create the tool's back-end processes, providing them the information they need to connect to the leaves of the internal process tree. Using the connection information provided by the front-end via an information channel like the environment or a shared file system, each back-end process connects to the appropriate internal process. In environments without a suitable process management system, our recursive approach for instantiating the tree of internal processes can be extended to create tool back-end processes. In this approach, each leaf of the internal process tree uses the remote shell utility to create the tool back-end processes that are its children, providing each back-end process with the information needed to connect back to the creating process. With both approaches, when all back-ends have connected to their appropriate internal process, the tool's instantiation is complete.

4.2 MRNet Process Layout

Tool requirements and system capabilities vary, so no single process layout is appropriate for all tools. Hence, an MRON implementation should allow a tool to tailor its process layout to the tool's computation and communication requirements and to the capabilities of the system running the tool. We briefly discuss several process layout issues to provide guidance for constructing high-performance internal process network layouts. A full investigation of internal process network layout issues is left for future work.

When choosing the process layout for an MRON-based tool, there are two key issues to consider: whether the internal processes are co-located with the application processes under study, and how the internal processes are connected. Our primary measures of a layout's quality are its:

- latency for a single broadcast operation, measured from initiation by the front-end to the last receipt by a back-end;
- latency for a single data aggregation operation, measured from initiation by the back-ends to receipt by the front-end;
- throughput for streams of broadcasts and data aggregations; and
- CPU utilization of MRON processes.

The first issue to consider when choosing an MRON process layout is whether to co-locate internal processes and application processes on the same nodes. While some literature on broadcast/reduction networks assumes that internal processes will be co-located with application processes [8,27,66], we believe this approach has serious flaws in practice. First, the internal processes would contend with application processes for CPU and network resources, perhaps seriously impacting the application's performance. Second, differing loads across MRON internal processes could create an imbalance among the application processes, skewing their performance. Because a parallel program's speed is often limited by its slowest process, this performance skew would increase the tool's impact on the application. As a result, we recommend that an MRON's internal processes be located on resources distinct from those running the application processes to achieve more predictable and understandable application behavior.

The second issue to consider when choosing an MRON process layout is the topology of internal processes. Both balanced and unbalanced tree topologies have attractive properties for MRON layouts. The literature on parallel collective communication algorithms argues for unbalanced tree topologies in many situations. For example, Bernaschi and Iannello [12] show that the optimal communication tree for broadcast is somewhere between a single-level flat tree and a binomial tree, depending on the latency for transferring messages between processes and the minimum interval between message send operations in a process. Similarly, optimal algorithms for several broadcast and data aggregation problems evaluated under the LogP [19] and LogGP [2] models use unbalanced communication trees. However, this literature assumes all processes involved in the operation are data sources (for reductions) or sinks (for broadcasts). In contrast, MRON internal processes are neither data sources nor sinks. For reduction operations, MRON leaf processes are the only data sources and the root is the only data sink; for multicast operations, the root is the only data source and the leaf

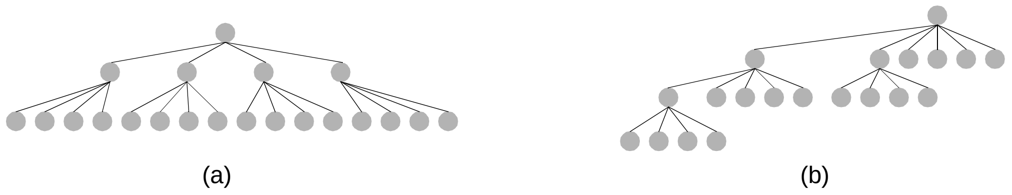


Figure 4.6 **Balanced and unbalanced MRON internal process topologies with the same number of back-ends.** The latency of a single broadcast or aggregation operation might be better with the unbalanced (b) topology, but the balanced topology (a) has better throughput for pipelined operations.

processes are the only data sinks.

Balanced tree topologies provide several attractive advantages over unbalanced tree topologies for our work. Their regularity makes them more predictable and easier to analyze when choosing the most appropriate size and shape for the MRON internal process tree. Although the latency of individual collective communication operations may be worse with balanced trees than unbalanced trees, they can provide better throughput for pipelined collective communication operations. For example, consider the tree topologies shown in Figure 4.6 connecting a tool front-end to sixteen tool back-ends. Assuming a LogP model with a minimum gap g between successive send operations in a process, an overhead o for each send and receive, and a message transfer latency L, the time required to complete a broadcast operation to all sixteen back-ends using the balanced tree topology shown in Figure 4.6a is $8g+4o+2L$, but the tool can start a new broadcast each $4g$ cycles. A comparable unbalanced tree topology reaching sixteen back-ends is shown in Figure 4.6b. This topology is constructed from a binomial tree with four nodes providing low-latency broadcast to each binomial tree node, with four tool back-ends attached to each binomial tree node. Depending on the relative values of g, o, and L, a single broadcast operation using this topology may complete before the balanced tree's broadcast, but a tool using this topology needs at least $6g$ cycles between each broadcast operation due to the larger fan-out at the tree's root. Furthermore, if the tool supports six-way fan-out as is being used at the root of the unbalanced tree topology, then it could use a balanced topology with a six-way fan-out throughout the tree to reach far more than sixteen tool back-ends. Therefore, we chose to use balanced tree topologies in our evaluation of the prototype MRON implementation, leaving investigation into optimal MRON communication topologies for future work. Furthermore, we used multi-level topologies with moderate (four- or eight-way) fan-out at each internal process because the ability of each internal process to keep up with its upward and downward data flows is limited.

4.3 A Real-World Tool Example

To evaluate MRNet's usefulness as infrastructure for scalable tools, we integrated MRNet into Paradyn, an existing parallel performance tool. There are two main ways that Paradyn can use MRNet: to simplify the complex interactions between front-end and tool daemons during tool and job start-up, and to off-load the performance data processing tasks from the Paradyn front-end. We describe how we integrated MRNet into Paradyn, providing a quantitative evaluation of MRNet within Paradyn in Section 4.4.2.

4.3.1 Scalable Tool Start-Up

Tools such as debuggers and performance tools may transfer large amounts of data during tool start-up when they create or attach to an application's processes. For example, a debugger that sets breakpoints by function name might deliver the names and addresses of all functions to the tool's user interface. In parallel tools with the typical process organization depicted in Figure 1.1, the front-end becomes a bottleneck when connected to a large number of application processes. Besides reducing tool interactivity, the start-up latency caused by this bottleneck may create problems for parallel runtime systems that fail if the application processes are not created in a timely fashion. Our modified version of Paradyn uses both built-in and custom MRNet aggregation filters for all activities involving the tool's daemons (i.e., its back-ends) during the tool start-up phase, including:

- reporting data about Paradyn daemons to the front-end;
- distributing data about known performance data metrics to all daemons;
- detecting clock skew between the front-end process and each daemon process; and
- reporting data about application processes to the front-end.

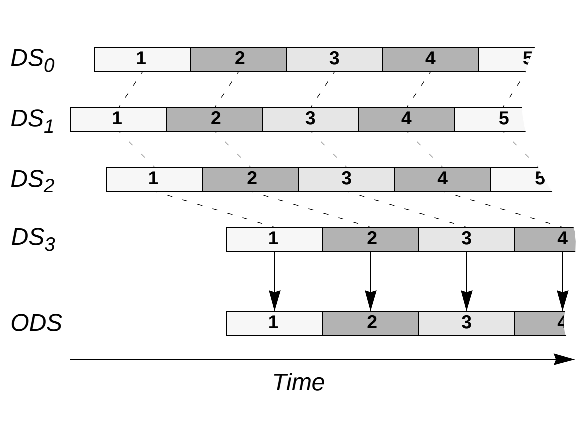
Although most of these activities manipulate Paradyn-specific data, the MRNet-based techniques they use are applicable to many activities commonly performed by parallel tools.

During Paradyn start-up, most of the data transferred within the tool system can be placed into two categories: information about the application processes, and information about instrumentation. Data in the first category is transferred from tool back-ends to the tool front-end. At start-up, the Paradyn back-ends examine application processes to identify the relevant parts of the program, such as modules, functions, and process IDs. Such items are called *resources* in Paradyn terminology. Once the application resources are identified, they are reported to the front-end along with statically-determined call-graphs for all application processes. In contrast to the application process information, data in the second category is transferred from the tool's front-end to its back-ends. This data consists of a collection of performance metric definitions that specify how to instrument processes to collect performance data.

Paradyn uses MRNet in two ways to reduce the cost of reporting data from daemons to the front-end. The method used depends on whether the data is likely to be the same across a significant number of processes (e.g., function names and their addresses) or is likely to be different across processes (e.g., process IDs and host names). If the data is likely to be the same across a significant number of processes, then most of the data transferred during tool start-up is redundant (especially if the application processes are created from a small number of executables and run on a collection of homogeneous hosts). To report this data, each Paradyn daemon first computes a summary of the data (i.e., a checksum). Next, the daemons write the checksums to an MRNet stream created to use a custom binning filter. This filter partitions the daemons into equivalence classes based on their checksum values. When the front-end receives the final set of equivalence classes, it requests complete function resource information only for a representative process from each class. Unlike function names, data like process identifiers and host names are likely to be different across hosts. Nevertheless, Paradyn also uses MRNet for reporting this data. Paradyn uses a parallel concatenation aggregation to construct larger resource report messages that are more efficiently delivered by the underlying communication subsystem than many small resource report messages.

Paradyn uses MRNet to deliver configuration data efficiently from the front-end to all back-ends. In Paradyn, metric definitions describing how to instrument processes to collect metric performance data are provided to the front end in a configuration file written in the Paradyn Metric Definition Language [50]. The front-end uses simple broadcast operations to deliver these metric definitions to all tool back-ends.

Clock skew detection is the only start-up activity that does not fall neatly into the two communication paradigms mentioned earlier. The MRNet-based clock skew detection scheme occurs in two phases. The first phase consists of repeated broadcast/reduction pairs on a special stream reserved for finding clock "local" clock skew between each process and the downstream processes to which it is directly connected (i.e., its children in the MRNet process tree). The second phase consists of a single broadcast to all daemons requesting them to initiate the collection of skew results. Each daemon initializes its "cumulative skew" value to zero, and passes it upstream into the MRNet network. When an MRNet internal process receives a cumulative skew value from one of its downstream connections, it adds its observed local clock skew value for that connection to the cumulative value, thereby computing the skew of its clock with each daemon reachable along that connection. By induction, when the algorithm finishes the Paradyn front-end holds the skews between its clock and the clocks of each tool back-end.



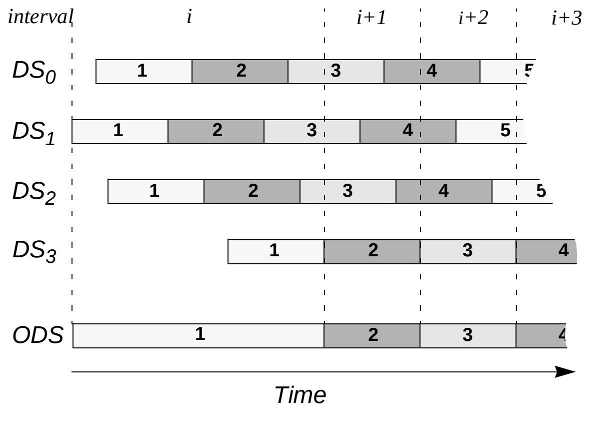


Figure 4.7 Performance data aggregation using ordinal aggregation (a) and time-aligned aggregation (b). In both examples, four sample data streams $DS_{0..3}$ are being aggregated into one output sample stream ODS. Ordinal aggregation aggregates the first sample from each stream, then the second, and so on. Time-aligned aggregation considers the samples' start and end times to aggregate data taken from the same interval during the program's execution.

4.3.2 Distributed Performance Data Aggregation

Like many parallel performance tools, Paradyn aggregates performance data collected by its back-ends to examine an application's global behavior. For each global performance measure being monitored, each Paradyn back-end produces a sequence of data samples representing the measure's value for the processes and threads that it controls. For example, to obtain a sequence of samples representing an application's overall CPU utilization, each Paradyn back-end collects a sequence of CPU utilization samples for its processes, and the Paradyn front-end aggregates corresponding samples across all sequences into a single global sample sequence. Ordinal aggregation is a common technique for constructing a global sample sequence; that is, aggregating the first sample from each sequence, then the second, and so on as shown in Figure 4.7a. The Paradyn design recognizes that its back-ends collect data asynchronously, so ordinal aggregation may combine samples representing different intervals of the application's execution. As a result, Paradyn represents a data sample as *{v,i}*, where *v* is the sample's value and *i* is the time interval to which the value applies. The interval's start and end timestamps are set by the back-ends when the sample is collected. Paradyn's performance data aggregation takes into account each sample's time interval as well as its value, so that aggregation is done with values over comparable time intervals as illustrated in Figure 4.7b.

Without MRNet, Paradyn aggregates data samples entirely within its front-end. The computation and communication cost of aggregation causes the front-end to become a scalability barrier when Paradyn monitors global performance measures on a large number of nodes. Using MRNet, Paradyn distributes its aggregation activity to filters running throughout the MRNet network, reducing its front-end data processing load. Paradyn's distributed data aggregation scheme uses a custom Performance Data Aggregation filter within each MRNet internal process that aligns data samples from all its inputs and then reduces them to form a single output sample. Collectively, these filters produce a single aggregated sample for the tool's front-end.

Paradyn's Performance Data Aggregation filter collects data samples on all of its inputs, aligns the data samples, and then reduces them. To determine how to align the samples and when to deliver the aligned samples to the aggregation filter, the filter maintains the notion of an output sample interval. This interval defines the start and end times for the aligned data samples, and therefore the

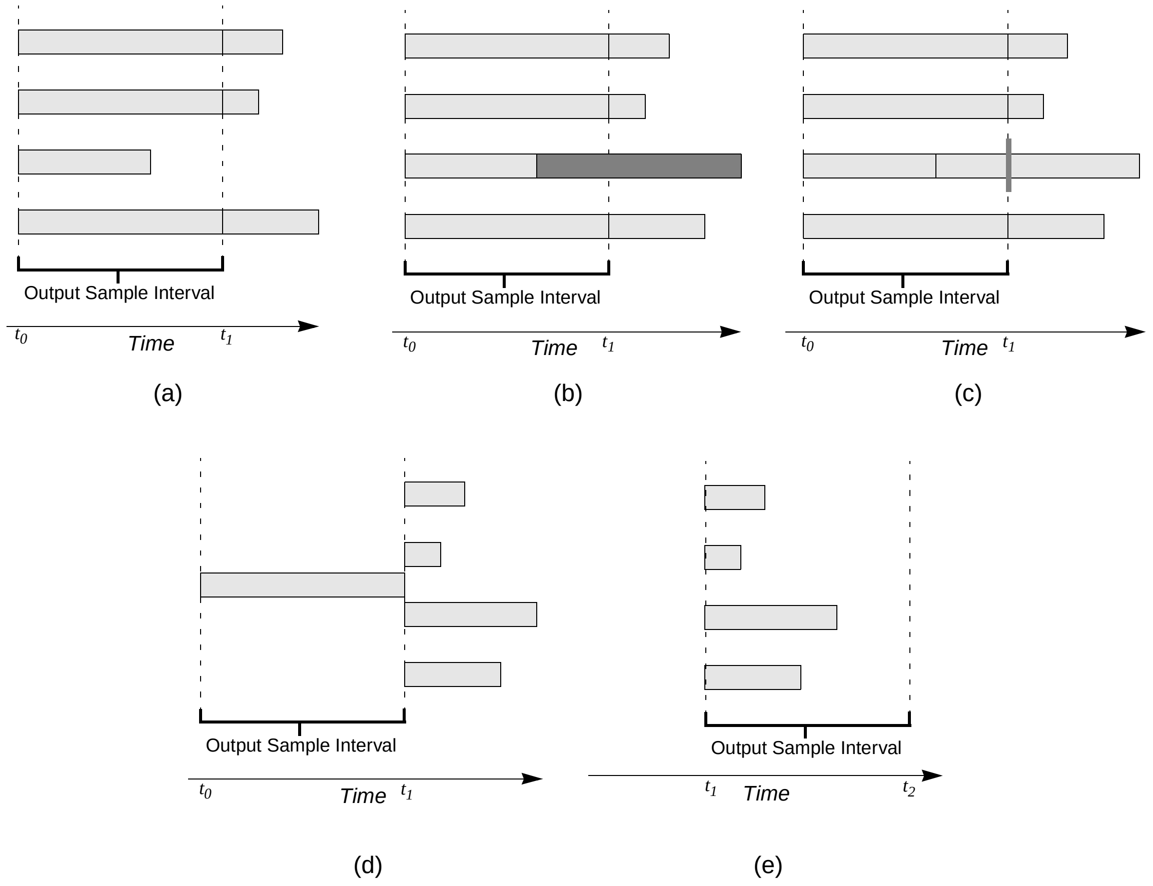


Figure 4.8 Distributed data aggregation of four data streams using Paradyn's custom MRNet filter. The initial situation (a) where one of the data streams does not have a complete output interval of data. When a sample arrives, it is placed on a queue associated with its input connection (b). If the sample's interval overlaps the current output sample interval, it is split to attribute the overlap to the output sample interval (c). If the newly-arrived sample completes the data for the output sample interval, the samples are reduced (d), and the output sample interval is advanced (e).

start and end time for the aggregated output sample.

Consider the example illustrated in Figure 4.8, showing the Performance Data Aggregation filter in an internal process with four input connections. Samples have already arrived for some of the input connections (Figure 4.8a). When a sample S arrives on an input connection, the filter places it on a queue associated with that input connection (Figure 4.8b). The filter then checks to see whether the interval of the newly-arrived sample overlaps with the current output sample interval. If so, it attributes a percentage of *S*'s value to the input connection's current output sample, leaving the remainder in *S* and adjusting its interval start time to remove the overlap (Figure 4.8c). Note that because the sample's value is attributed proportionally to the current output interval, and the remainder used in the next output sample interval, there is no lost performance data due to round-off issues. If *S*'s arrival caused the current output sample interval to be full (i.e., to have sample data from all input connections over all input connections), the filter reduces the aligned samples (Figure 4.8d) and advances its output sample interval (Figure 4.8e). The output sample uses the same interval as the aligned input samples.

Paradyn's MRNet-based performance data aggregation scheme exhibits a common trade-off between centralized and distributed algorithms. The centralized aggregation scheme has complete knowledge of all of the samples to be aggregated, so it only considers each sample once when finding the aggregated sample's start and end times. On the other hand, the distributed scheme performs multiple alignments throughout the network, leading to more overall work in the tool system. Nevertheless, because distributed scheme does these alignments in parallel and reduces the computation cost for data aggregation in the tool's front-end, the MRNet-based distributed scheme exhibits better overall scalability than the centralized scheme.

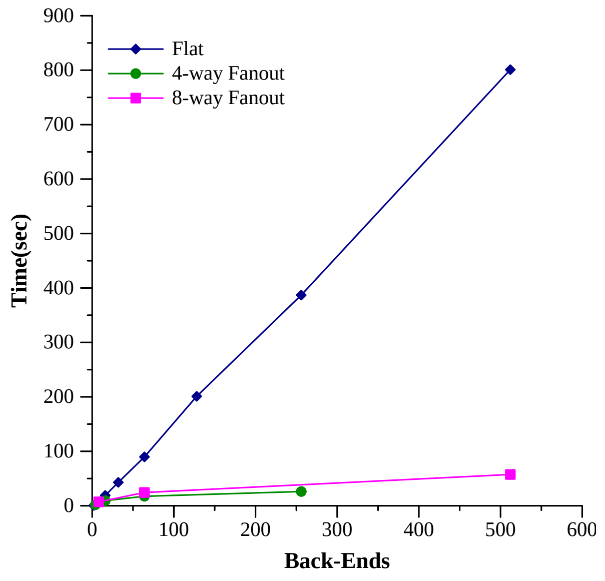
4.4 Evaluation

To evaluate MRNet, we measured its performance alone within a test harness and then integrated with Paradyn. Our micro-benchmark experiments with the test harness tool measured MRNet's start-up latency, the round-trip latency of a single broadcast followed by a reduction, and MRNet's reduction throughput using several process tree topologies. Our Paradyn experiments compared the performance of both start-up and performance data aggregation activities with and without MRNet. Our experiments were run on the ASCI Blue Pacific system [59] at Lawrence Livermore National Laboratory. At the time the experiments were performed, Blue Pacific contained 280 nodes (256 compute nodes) connected by an IBM SP switch interconnect. Each node has four 333 MHz PowerPC 604e processors, 1.5 GB RAM, and runs AIX 5.1 with Parallel System Support Programs version 3.4. Our results showed that MRNet significantly improves the scalability of key activities in parallel performance and system administration tools.

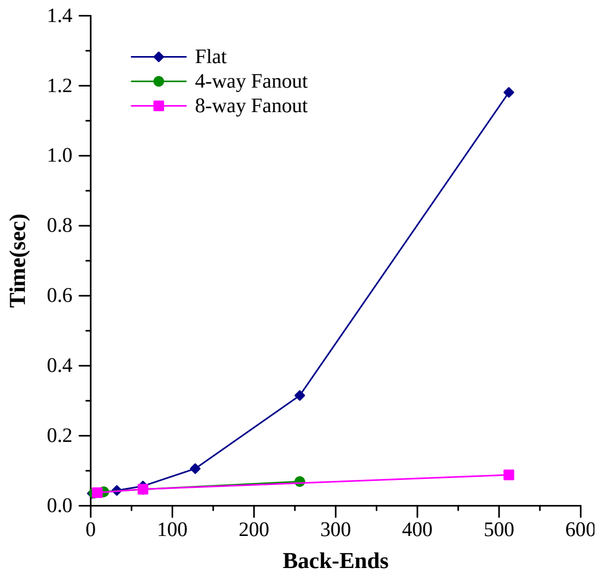
4.4.1 Micro-benchmark Results

We began by measuring the low-level performance of MRNet within a minimal test harness. For each run of our test harness tool, we requested an appropriately-sized partition from the Blue Pacific batch scheduling system. Once we were given our partition, we determined the partition nodes' host names and used an automatic configuration generator program to build an MRNet configuration file with the desired topology within the partition. We then executed the tool's front-end program, passing the configuration file's name as an argument. During each run of the test harness, we measured three MRNet performance characteristics: the latency to instantiate the MRNet network, the latency of a broadcast operation followed by a data reduction, and the MRNet's throughput during a sequence of data reductions. The results of these experiments are shown in Figure 4.9.

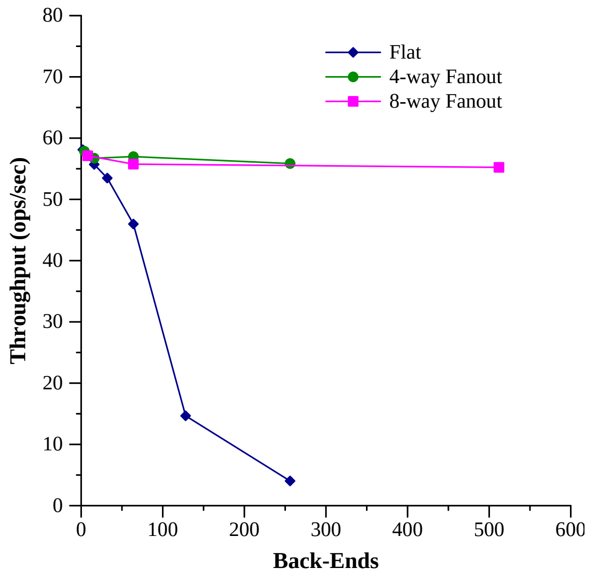
Our micro-benchmark measurements confirm the necessity of infrastructure like MRNet for building scalable parallel tools. Using a flat, single-level topology (which closely approximates the architecture of many parallel tools), instantiation latency grows quickly as the number of tool back-ends increases due to the serialization of the process creation operations. The instantiation latency grows quite slowly when using MRNet with fully-populated balanced tree topologies with four- and eight-way fan-outs because MRNet creates the process tree in parallel. The round-trip latency and data reduction throughput measurements also show the benefits of MRNet to parallel tools. In the flat topology, each broadcast or reduce is implemented using serialized point-to-point message transfers. Although each message transfer is less time-consuming than the `rsh` used to create processes during tool instantiation, the effect of serialization is similar: the latency grows rapidly as the number of back-ends increases. Also, the tool front-end is involved in every message transfer, so it cannot start a subsequent reduction before the previous operation completes. Multi-level MRNet process configurations allow MRNet to perform point-to-point message transfers in parallel. Furthermore, the moderate fan-outs at each MRNet process allows data reductions to be pipelined as they pass through the network, keeping reduction throughput high as application size increases. The trends in MRNet's micro-



(a)



(b)



(c)

Figure 4.9 MRNet micro-benchmark experiment results. Tool instantiation latency (a), round-trip latency of a single broadcast followed by a single reduction (b), and data reduction throughput (c) using single- and multi-level MRNet topologies. Compared to the “flat” (i.e., single-level) topology commonly found in parallel tools, multi-level MRNet topologies exhibited dramatically better scalability and overall performance, showing the necessity of multi-level process networks like MRNet for building scalable parallel tools.

benchmark scalability studies were expected; previous tool infrastructures using a hierarchy of processes such as the Ladebug parallel debugger [9] and Lilith [27] show similar scalability trends.

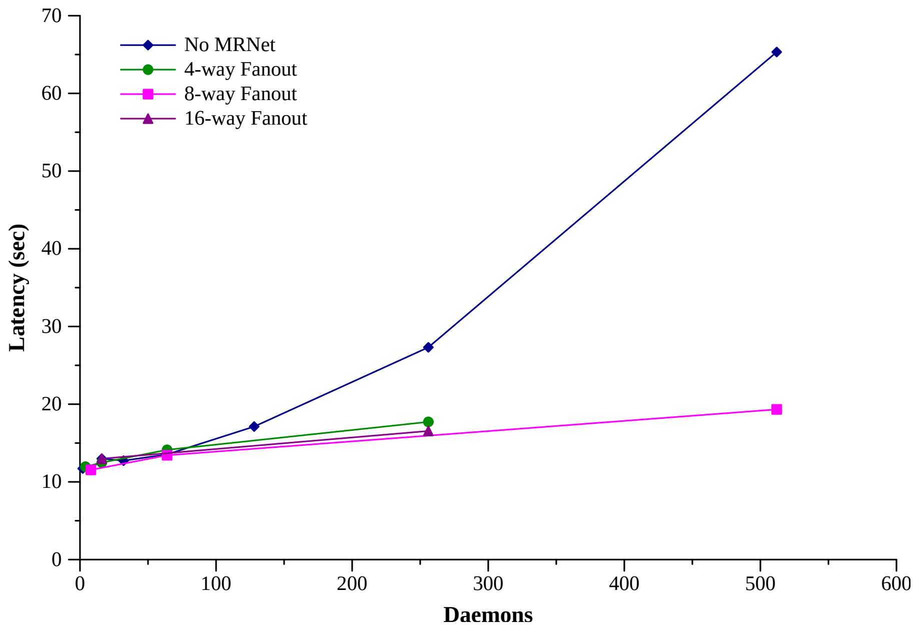
4.4.2 Integrated Performance Results

To evaluate MRNet's real-world performance, we modified the Paradyn parallel performance tool to use MRNet as described in Section 4.3. We evaluated MRNet's performance during tool start-up and while the tool was collecting and processing performance data.

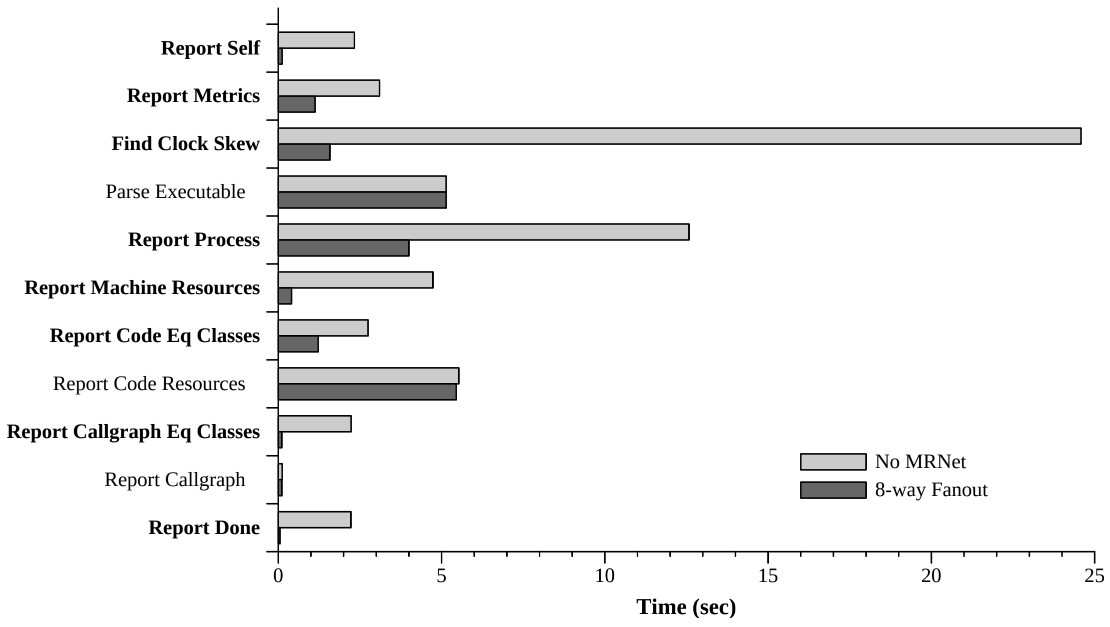
Paradyn's start-up protocol was already highly tuned to reduce redundant data transfer. For several data transfers from tool daemons to the front-end, it used a technique whereby each tool daemon computes a checksum over its own data, the front-end partitions the daemons into equivalence classes based on the checksum values, and then requests the complete data from only a single representative of each equivalence class. We measured the latency of Paradyn's start-up activities when preparing to monitor `smg2000` [14], a parallel linear equation solver included in the benchmark suite for the ASCI Purple system. The `smg2000` executable is relatively small, containing approximately 434 functions in a 290 KB executable. We started the timer when all daemons were known to have been started (but not yet reported themselves to the tool front-end), and stopped the timer after the daemons had reported information about themselves and the application processes they created, and were ready to run the application.

The results of our scalability study with several MRNet topologies are shown in Figure 4.10a. Without MRNet, serialization of the communication between Paradyn's front-end and daemons causes overall start-up latency to rise exponentially as the number of daemons increases. Using MRNet and process topologies with moderate fan-outs, the start-up latency curves are much flatter and growth is nearly linear, indicating a significant improvement in overall tool scalability. To investigate how much of the overall start-up latency that MRNet could affect, we measured the latency of individual start-up activities with and without MRNet for our largest experimental configuration; these results are shown in Figure 4.10b. The individual activities shown in the figure are:

- **ReportSelf:** Using an MRNet concatenation filter, each daemon reports basic characteristics to the front end such as the host on which it is running;
- **ReportMetrics:** The front-end broadcasts Metric Definition Language data to all daemons; the daemons respond using the equivalence class algorithm described above to report all metrics that they support (including internal metrics not specified in the MDL data);
- **Find Clock Skew:** The front-end finds its clock skew with respect to each daemon using the algorithm presented in Section 4.3.1;
- **Parse Executable:** Each daemon examines the application executable and the shared libraries it uses to find names and addresses of all functions, and parses the code to discover the application's static call graph;
- **Report Process:** After creating or attaching to an application process, each daemon reports data about the process to the front end including its process id, its command-line arguments, whether it was created by the daemon or was



(a)



(b)

Figure 4.10 Paradyn start-up latency. For increasing numbers of daemons (a), start-up latency scales much better when using MRNet than without it. When broken down by activity (b), start-up latency for 512 nodes shows some activities benefit from MRNet more than others. Bold activity names indicate use of MRNet for data aggregation or concatenation for at least part of the activity.

already created when the daemon attached to it, and whether the front-end should issue the command to resume the process when all start-up activities are complete;

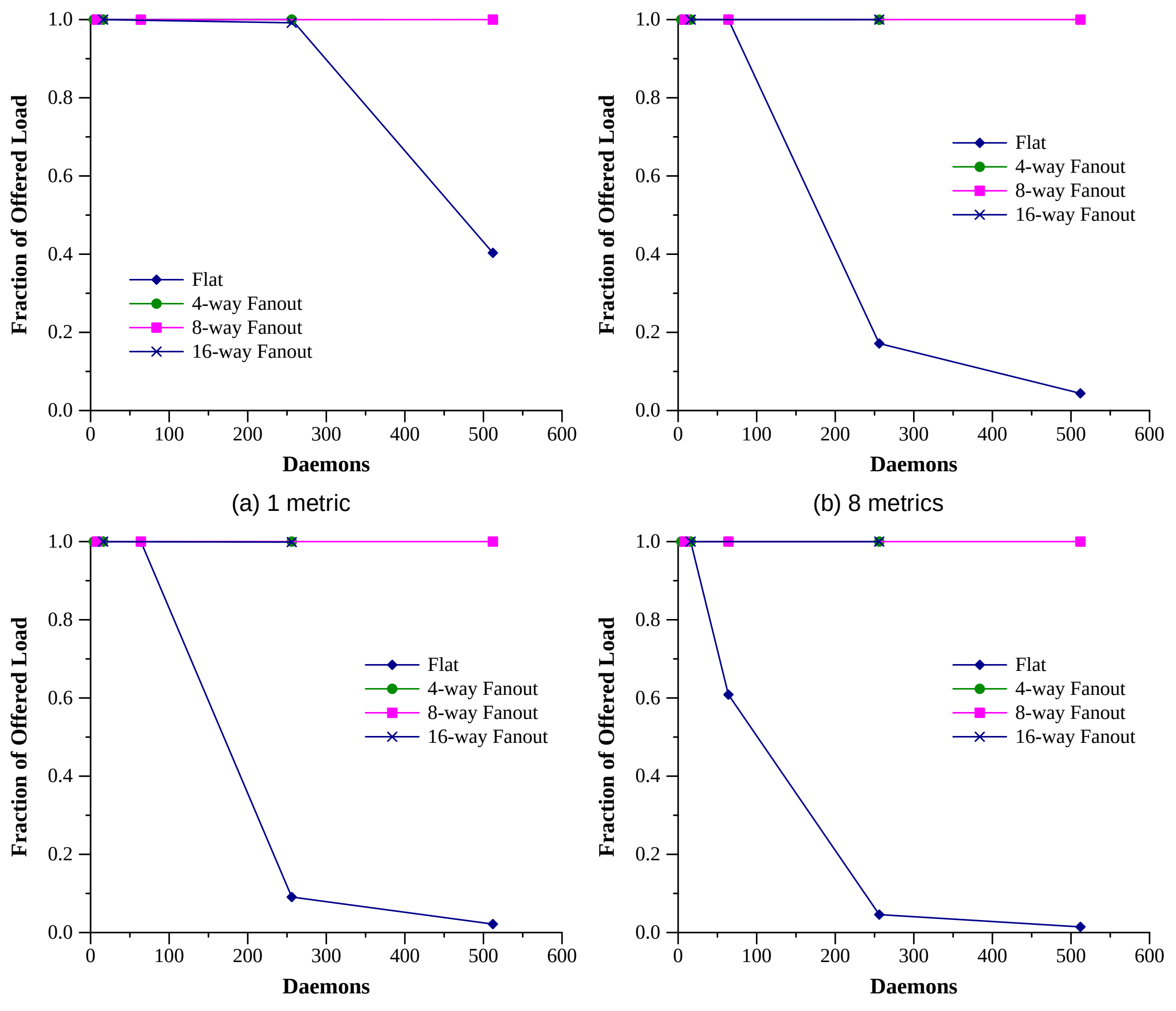
- **Report Machine Resources:** Using a concatenation filter, each daemon defines Paradyn resources for the host, process, and initial thread of its application processes via Paradyn's resource definition protocol;
- **Report Code Eq Classes and Report Code Resources:** Using the equivalence class algorithm, the daemons define resources for all functions and modules in the application executable;
- **Report Callgraph Eq Classes and Report Callgraph:** Using the equivalence class algorithm, the daemons report their static call-graph information (built during the "Parse Executable" activity described above) to the front-end; and
- **Report Done:** The daemons indicate the end of the start-up phase.

Each activity that used MRNet to communicate with all daemons showed a significant latency reduction by using MRNet. The activities that did not show a significant improvement from using MRNet are activities that consist either of work done entirely in parallel by the daemons ("Parse Executable") or point-to-point communication between a small number of daemons and the front-end ("Report Code Resources", "Report Callgraph"). In fact, the point-to-point communication activities transferred data via MRNet; the additional overhead of passing through intermediate MRNet processes was observed to be negligible. Overall, the benefit of using MRNet increased as we increased the number of tool daemons. With our largest configuration of 512 back-ends, the latency for performing all start-up activities was 3.4 times faster with MRNet and a balanced, fully-populated tree configuration with eight-way fan-out than without MRNet. Based on our investigation of MRNet's benefit for each individual activity during Paradyn start-up, we expect this trend to continue with configurations significantly larger than 512 daemons.

Clock skew detection was the Paradyn start-up activity that benefitted most from using MRNet, because it uses repeated broadcast/reduction operations to distribute and collect clock samples and intermediate skew results whereas the other activities perform only one or two collective operations. We evaluated the clock skews computed by the MRNet-based clock skew detection algorithm by comparing them to skews computed using Blue Pacific's SP switch clock (a globally-synchronous clock) and to skew results computed using a commonly-used direct-communication scheme. To compute its clock skew with respect to a given daemon under the direct communication scheme, the front-end sends a small amount of data to the daemon. The daemon samples its clock when it receives the data and sends this sample to the front-end. When the front-end receives the daemon's sample, it samples its own clock and computes the round-trip latency of the sends and receives. The front-end approximates the one-way latency from the round-trip latency, adds the one-way latency to the daemon's clock sample, and uses the difference between this value and the front end's receive timestamp as the clock skew.

In our experiments, the front-end measured the skew using the direct communication scheme 100 times and used the observed skew with the smallest absolute value as the best approximation of the actual clock skew. Using a 64-daemon topology with four-way fan-out (a three-level topology), the MRNet-based clock skew detection algorithm produced skews with an average error of 10.5% as compared to the skews computed using the globally-synchronous switch clock, while the average error in the skews produced by the direct-connection method was 17.5%. However, the standard deviation of the errors produced by the MRNet-based algorithm was 80.4, slightly higher than the standard deviation in the direct connection method's errors at 78.9. In short, MRNet's clock skew detection algorithm produced results sufficient for use even at Paradyn's most demanding sampling rate but is significantly more scalable than the direct-connection method.

To assess the impact of MRNet on Paradyn's performance data processing capabilities, we measured how well Paradyn could consume and process the volume of performance data samples generated by its daemons in a variety of configurations. We varied the load placed on the tool's front-end by varying the number of daemons and the number of performance metrics for which data was collected by each daemon. To simplify the evaluation, we ran Paradyn on a synthetic parallel application with known behavior and easily-controllable run time. To keep the data rate high, we configured the Paradyn daemons to use a fixed sampling rate for the duration of the experiments. We fixed each daemon's sampling rate to Paradyn's default initial rate of five samples per second per metric. Therefore, for a given number of daemons D and metrics M, the overall rate at which samples



(a) 1 metric (b) 8 metrics (c) 16 metrics (d) 32 metrics

Figure 4.11 Fraction of offered load serviced by the Paradyn front-end. When not using MRNet and increasing the number of metrics for which data is being collected (shown by the curves labelled “flat”), Paradyn’s ability to process the offered performance data sample load degrades quickly as the number of daemons increases. However, using MRNet to off-load some of the performance data processing allows Paradyn to scale much better as the number of daemons and metrics increases with four-, eight-, and sixteen-way MRNet fan-outs.

are generated within the tool is *5DM* samples per second.

The results of our integrated performance data processing experiments are shown in Figure 4.11. Each figure shows Paradyn’s performance when collecting data for up to 32 metrics for configurations with between 4 and 256 daemons.

Each data point marks the ratio of the rate at which the Paradyn front-end processed performance data samples to the rate at which the daemons generated the samples. This ratio represents the fraction of offered load processed by the Paradyn front-end. While there were minor start-up transients, the steady-state rate at which the front-end consumed performance data did not fluctuate significantly. Therefore, we report only the steady-state ratio. In these figures, a level curve at value 1.0 indicates the Paradyn front-end was able to keep up with the performance data volume generated by its daemons as the number of daemons was increased.

Our results show that when Paradyn relies on MRNet for some of its performance data processing activity, it scales significantly better with increases in the number of tool daemons and number of metrics for which data is collected. When increasing the number of metrics for which data is being collected, Paradyn's ability to process the offered performance data sample load degraded quickly. For example, when collecting data from only 64 daemons for 32 metrics per daemon without MRNet, the Paradyn front-end processed the data at only about 60% of the rate at which it was generated. With 256 daemons and 32 metrics, the front-end processed data at a rate of less than 5% of the offered load. Note that as the number of metrics per daemon increases, Paradyn increases the size of its messages containing performance data rather than the number of messages.

Using MRNet allowed the Paradyn front-end to scale much better as the number of daemons and metrics were increased. With four-, eight-, and sixteen-way

MRNet fan-outs, the front-end was able to process the entire offered load for all tested configurations.

4.5 Summary

MRONs are scalable overlay networks providing multicast and data reduction services for parallel tools that can be used to address the problem of monitoring and controlling a large number of distributed tool components. MRON-based tools can off-load data processing onto MRON processes to limit the computation and communication load of the tool's front-end.

Our approach for building MRONs is incorporated in MRNet. MRNet uses filters running throughout its overlay network to manipulate data sent between a tool's front-end and its back-ends in parallel. We evaluated the scalability and performance of our MRON approach by measuring the performance of MRNet's low-level collective communication operations. We also integrated MRNet into the Paradyn parallel performance tool to evaluate our MRON approach in the context of parallel tools. Our results showed that our MRON approach can be used to greatly reduce the cost of many important tool activities compared to tools that use a traditional parallel tool organization, resulting in significant improvements in overall scalability.

Using the research described in this chapter as a foundation, there are several directions for future research in the area of scalable parallel tool infrastructure. Our initial work focused on the infrastructure's scalability but reliability and resiliency are also important for large-scale parallel tools. Hence, one direction for future MRON research involves the investigation of techniques for improving an MRON's fault tolerance and fault recovery characteristics without degrading its scalability. In fact, this work has already been started by a member of the Paradyn project.

Another direction for future work is a more complete study of process layout issues for MRON-based tools. In our work with MRNet, we used regular process layouts with moderate fan-outs for their simplicity and regularity, and we placed the internal network processes on computing resources distinct from the tool and application processes. A complete study is needed to determine how best to choose a process layout tailored for a given tool's computation and communication requirements and the capabilities of the underlying system.

A third direction of future research is suggested by the recognition that the process management systems available in some environments may be able to start MRON internal processes more quickly than MRNet's recursive, remote-shell-based approach. Process management systems often use a network of pre-existing, pre-authenticated daemons running on the nodes of a large parallel system to enable low-latency creation of related processes like those that comprise an MPI job. An MRON implementation can use this capability to spawn tool back-end processes, but may also be able to use it for creating its internal network processes. This instantiation approach requires a separate phase for connecting the internal network processes according to the desired process topology. This connection phase is complicated by the need of each process a posteriori to obtain the connection information (host and port number) of the processes to which it connects. Whether any benefit gained from using the process management system is outweighed by the cost of this connection phase is an open issue.

Chapter 5

Scalable Automated Performance Diagnosis for Applications with a Large Numbers of Processes

In this chapter, we address the problem of diagnosing performance problems in applications with thousands or more processes. Existing performance diagnosis tools fail to overcome the two scalability barriers (Chapter 1) encompassed by this problem: the management and processing of large-volume performance data flows generated when monitoring a large number of processes, and communication between a large number of tool components.

To be effective in finding application performance problems, performance diagnosis tools must consider both *local* application behavior, the behavior of individual application processes, and *global* application behavior, the aggregate behavior of all of the application's processes. For example, a tool that monitors only an application's global behavior may be able to determine that it suffers from a global performance problem. However, without examining the behavior of each process individually, the tool will not be able to determine that the global performance problem is caused by an expensive function call sequence being executed by a small number of processes. Conversely, examination of only an application's local behavior may not provide an indication of the application's overall performance.

To address the problem of diagnosing performance problems in applications with a large number of processes, we developed and evaluated an on-line automated performance diagnosis approach that efficiently examines both local and global application behavior. We developed our approach in the context of the Performance Consultant, Paradyn's automated performance diagnosis component (see Chapter 3). We call the enhanced component the Distributed Performance Consultant. Other performance diagnosis tools have adopted the Performance Consultant's search-based approach for finding application performance problems (e.g., [35,36,77,103]), so our approach also is applicable to other search-based other tools.

Our Distributed Performance Consultant research takes two evolutionary steps from the traditional Performance Consultant. The first is a move from the Performance Consultant's centralized approach (hereafter called the *Centralized Approach* or *CA*) to a hybrid strategy that combines a centralized search of global application behavior with distributed searches of local application behavior. We call this approach the *Partially Distributed Approach* or *PDA*. The second step leverages the synergy between the PDA's distributed local bottleneck searches and the Sub-Graph Folding Algorithm (Chapter 7) in a *Truly Distributed Approach* or *TDA*. The TDA retains the functionality of the PDA while avoiding the increase in instrumentation management complexity required to support the PDA's hybrid approach.

The Distributed Performance Consultant's PDA has three parts:

- **A distributed, on-line, automated performance bottleneck search strategy.** Our search strategy has two complementary parts, one to examine the application's local behavior and one to examine its global behavior. For examining local application behavior, our strategy incorporates local decisions regarding the portions of the search that consider the behavior of specific processes. For examining global application behavior, our strategy uses centralized decisions based on performance data aggregated with MRNet (Chapter 4). Our strategy uses MRNet for scalable communication between tool components. Like the traditional Performance Consultant, our strategy uses dynamic instrumentation [49] for collecting performance data.
- **A model for representing the cost of instrumentation in parallel computation.** The traditional Performance Consultant tracks the cost of instrumentation using a single, aggregated value. This model is undesirable for tracking the cost of instrumentation in parallel applications because it does not retain cost information at the granularity of individual application processes. A single, aggregated instrumentation cost value is especially undesirable with the Distributed Performance Consultant's decentralized decision-making approach. To support local decision-making for our bottleneck search strategy, we extend the traditional Performance Consultant's model to retain instrumentation cost information for each application process.

- **A policy for scheduling dynamic instrumentation requests generated during a PDA bottleneck search.** Under the PDA, monitoring and controlling the cost of our search strategy's dynamic instrumentation requires not only a model of the instrumentation's cost, but also a policy for scheduling the search's instrumentation requests. With our search strategy, local and global application behavior may be examined at the same time. Because the examination of global application behavior requires performance data from each application process, instrumentation collecting that data must be present in all application processes to be useful. In contrast, instrumentation that collects performance data about a specific process' behavior does not depend on data generated by instrumentation in other application processes. The Distributed Performance Consultant includes an instrumentation scheduling policy that schedules both local and global instrumentation requests for efficient distributed bottleneck searches.

By leveraging MRNet for scalable aggregation of local application behavior, our approach of distributing the examination of local application behavior while retaining a centralized examination of global application behavior yields the expected scalable search behavior. However, this benefit comes at a cost in terms of increased tool complexity for separate tracking of instrumentation cost and scheduling of local and global instrumentation. Initial success with the PDA and our Sub-Graph Folding Algorithm motivates the TDA that retains the PDA's distributed local bottleneck searches but uses the SGFA to approximate the result of an explicit examination of global application behavior. The TDA retains the PDA's functionality while eliminating the complexity required to support the PDA's hybrid search strategy.

This chapter presents the Distributed Performance Consultant's PDA bottleneck search strategy (Section 5.1), instrumentation cost model, instrumentation scheduling policy (Section 5.2), and TDA bottleneck search strategy (Section 5.3). In our evaluation (Section 5.4), the computation and communication load placed on the tool's front-end was low enough using the PDA and TDA to allow bottleneck searches on up to 1024 application processes (limited by the available system size, not by our approach). In contrast, the CA did not scale beyond 32 application processes. The chapter concludes with a summary of our Distributed Performance Consultant research and a discussion of potential directions for future work (Section 5.5).

5.1 PDA Bottleneck Search Strategy

The local and global behavior of a parallel application must be examined to find application performance problems effectively. Furthermore, because monitoring a large number of application processes can generate a massive volume of performance data, performance diagnosis tools must be able to collect and analyze large-volume data flows efficiently if they are to be useful for tuning parallel applications with a large number of processes. A multicast/reduction overlay network like MRNet enables scalable processing of global performance data (see Section 4.3), but does not address the part of the problem involving data that describes local application behavior.

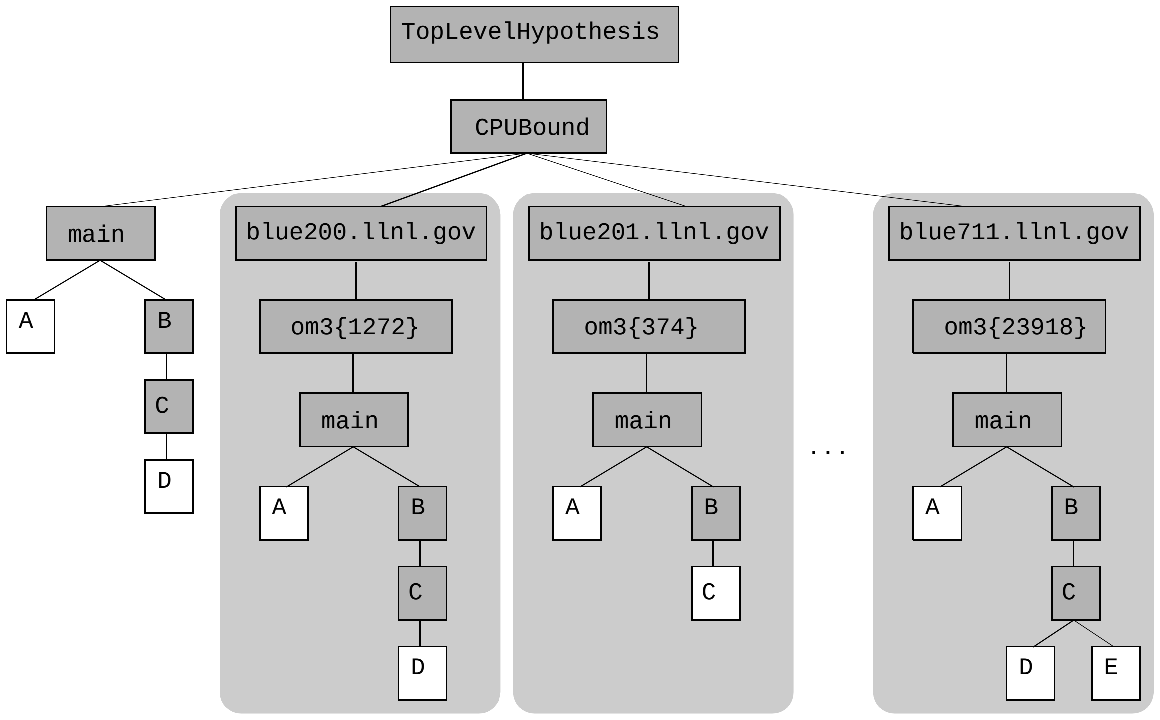


Figure 5.1 Portion of search history graph showing our PDA distributed performance bottleneck search strategy. The figure shows search refinement from an experiment labelled “CPUBound” examining global application behavior. That experiment is refined to an experiment examining the application’s code (left most sub-graph) and a collection of experiments that examine local application behavior. Shaded boxes indicate true experiments. Distributed sub-searches are shown inside lightly-shaded rounded rectangles.

The first part of our performance diagnosis approach is a distributed, on-line automated performance bottleneck search strategy. Our strategy adopts the call-graph-based search refinement rules of Paradyn’s Performance Consultant [15]. However, unlike the Performance Consultant’s centralized design, our strategy distributes the portions of the search that are restricted to specific hosts for efficient examination of local application behavior.

Under the PC’s search refinement rules, the initial experiments of a search investigate a hypothesis about the application’s global behavior, such as whether the application as a whole is CPU bound. When an experiment dealing with global application behavior is refined to examine the application's behavior on specific hosts, our new search strategy delegates control for the host-specific experiments (and all experiments refined from them) to a local search agent running on that host. Figure 5.1 shows an example of such delegation. When an experiment with the hypothesis "CPU bound" and the focus

```
< /Code, /Machine, /SyncObject >
```

is refined to produce a collection of "CPU bound" experiments with foci

```
< /Code, /Machine/blue200.llnl.gov, /SyncObject >
< /Code, /Machine/blue201.llnl.gov, /SyncObject >
< /Code, /Machine/blue202.llnl.gov, /SyncObject >
...
```

our search strategy delegates the sub-search rooted at each host-specific experiment to a search agent running on the host named by the experiment's focus (i.e., `blue200.llnl.gov`, `blue201.llnl.gov`, etc.).

Distributing sub-searches has two benefits for automated performance diagnosis tools. First, distributed sub-searches reduce centralized processing of data and control messages within the tool. Once a sub-search is delegated to a local search agent, the data and control messages for that sub-search are processed by the local search agent instead of the tool's front end. The second benefit is that it can reduce the time needed to complete a search. Each distributed sub-search investigates local application behavior on a single, independent host, so the tool may perform these distributed sub-searches in parallel. Because local sub-searches form a sizeable fraction of the total search for applications with a large number of processes, performing local sub-searches in parallel can reduce the time required to complete a search.

A distributed data analysis like that used by our PDA bottleneck search strategy introduces a data management issue not faced with a centralized data analysis. The data being collected for a local search agent's sub-search may also be used as part of an aggregate value needed for the investigation of the application's global behavior. When local performance data can be used to form a needed global aggregate value, there are a few possible approaches for optimizing the management of performance data. We consider two new scenarios for data collection requests:

- The global search agent issues a request to collect global performance data and some or all of the data needed for the global aggregate is already being collected to support local sub-searches; and
- A local search agent issues a request to collect local performance data and the requested data is already being collected for a global aggregate.

One approach for managing performance data in these two request scenarios is to treat each request independently. That is, even if the requested data is already being collected, the tool does not use the already-collected data to satisfy the later request. Once the later request is issued, subsequent data samples are delivered to both the local and global search agents. This approach has the advantage of being simple to implement, but ignores the opportunity to re-use data that has already been collected to increase the efficiency of the search.

To capitalize on performance data collection that has already occurred, the performance data management approach could cache performance data in each tool daemon. If a global data request is issued for data that is already being collected, tool daemons deliver all of the already-collected data to be aggregated. For the second request scenario, each daemon caches a copy of data collected in response to a global data request. Subsequent local data requests use the already-collected data from the cache if it is present. Depending on how many data samples have been cached, the local search agent may be able to perform its analysis entirely on the cached data without obtaining new data samples. Although this approach could make the overall bottleneck search more efficient, it increases the computation and memory costs incurred by the tool daemons.

To obtain a better idea of the Distributed Performance Consultant's load on Paradyn daemon processes and for ease of implementation, we used the simple, non-caching data management approach in our initial implementation. The caching data management approach is left for future work.

The components that implement the Distributed Performance Consultant's PDA within Paradyn are shown in Figure 5.2. In the figure, the tool's front-end and a tool daemon are shown as shaded ovals. Each process' threads are represented by white rectangles. The tool's front-end contains several threads including a Data Manager thread that issues instrumentation requests to the tool's daemons on behalf of the other front-end threads and holds the performance data produced by those instrumentation requests, and a Performance Consultant thread that provides overall control for the bottleneck search and controls the search's examination of global application behavior. Each tool daemon also con-

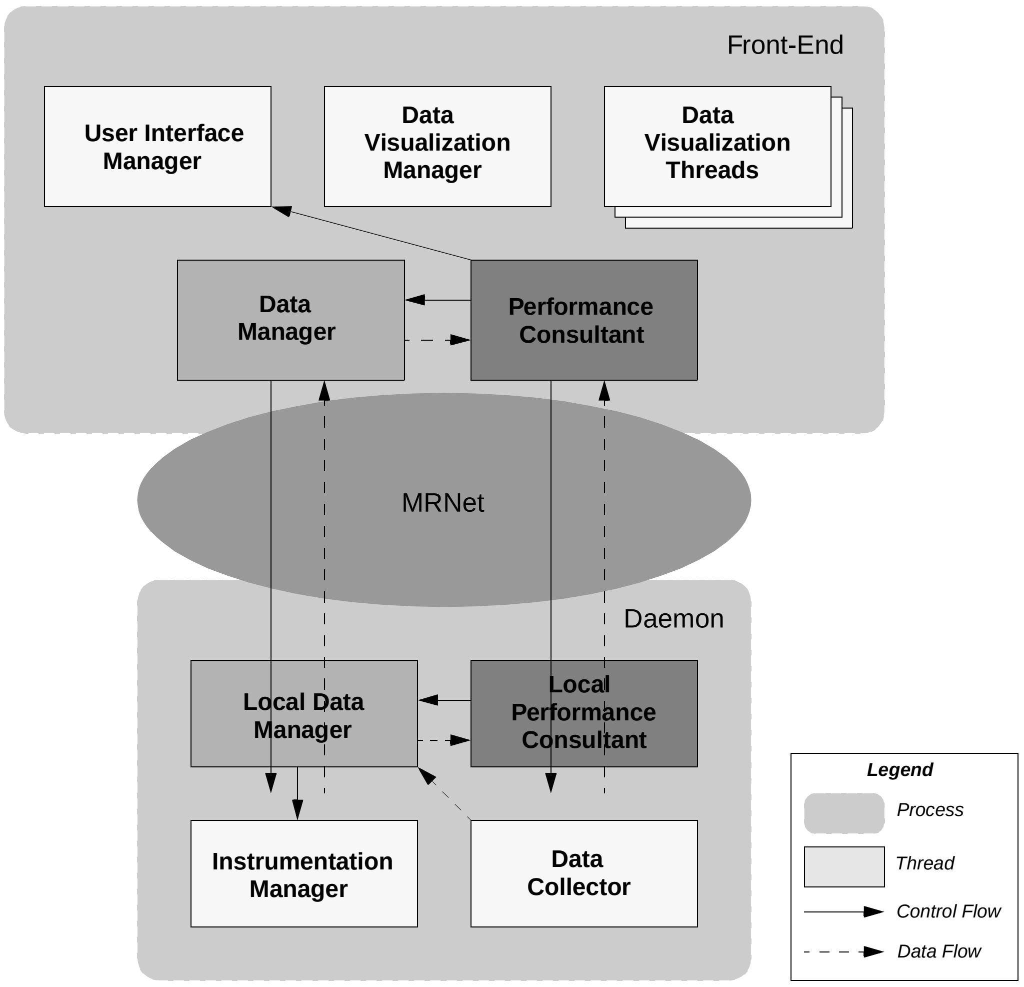


Figure 5.2 The PDA within Paradyn. Only one instance of a daemon process is shown though, in practice, there will be one per application node. MRNet is used for all communication between the front-end and daemon processes. In the front-end, the Performance Consultant obtains global performance data from the Data Manager, delegates local sub-searches to the Local Performance Consultant running in the daemon, and delivers bottleneck search results to the User Interface Manager for visualization. In the daemon, the Local Performance Consultant obtains local performance data from the Local Data Manager, which uses the Instrumentation Manager and Data Collector to obtain the data.

tains several threads. A Local Data Manager thread interfaces with the front-end's Data Manager. The Local Data Manager uses an Instrumentation Manager thread to insert instrumentation into the application processes controlled by the daemon. A Data Collector thread collects performance data generated by that instrumentation and delivers it to the Local Data Manager. The Local Data Manager thread also services requests from the Local Performance Consultant thread. This thread implements the local search agent of our distributed bottleneck search strategy. MRNet is used for all communication between the tool's front-end and daemons.

5.2 Dynamic Instrumentation Management

Our goal in using a distributed performance bottleneck search strategy is to off-load at least some of the performance diagnosis activity from the tool's front-end. However, distributing the bottleneck search places two requirements on the approach used to monitor and control the effects of our search's dynamic instrumentation:

1. Allow local search agents to make independent decisions about inserting and removing the instrumentation that supports their local sub-searches; and
2. Ensure that an instrumentation request for collecting global performance data is satisfied by all tool daemons at approximately the same time so that data from all application processes is available to form the aggregated global data value.

An approach using a single aggregated instrumentation cost value and centralized search control like that used in the traditional Performance Consultant satisfies the second requirement because decisions about when to insert dynamic instrumentation are made centrally with complete instrumentation cost knowledge. However, such a centralized decision-making scheme does not satisfy the first requirement.

5.2.1 Instrumentation Cost Model for Parallel Computation

To allow local search agents under the PDA and TDA to make decisions independently about their own local data instrumentation, the Distributed Performance Consultant represents the cost of instrumentation in parallel computation using a model that maintains the cost of instrumentation in each application process. More specifically, for an application with P processes, the model expresses the instrumentation cost as $C = (c_1, c_2, \ldots, c_P)$ where c_i is the instrumentation cost in application process i, $1 \leq i \leq P$. The benefit of maintaining the instrumentation cost for each application process is that it allows each local search agent to restrict its view of the application's overall instrumentation cost. Each local search agent tracks only the cost of instrumentation in the processes it controls.

Although having each local search agent maintain instrumentation cost information only for local processes allows the PDA to satisfy the first instrumentation management requirement, it complicates our ability to satisfy the second requirement when scheduling a workload with both local and global instrumentation requests. Local search agents do not have complete information about an application's instrumentation cost, so they cannot be guaranteed to make the same scheduling decision based on their own cost information in response to a global instrumentation request from the tool's front-end. A reliable distributed consensus algorithm (e.g., [11,16,58,64]) could be used in the PDA to enable the collection of local search agents to reach the same decision regarding global instrumentation. However, there are several disadvantages to using such an algorithm in the PDA. The algorithm's implementation would place additional computation and communication load on daemon processes. Because distributed consensus algorithms tend to be complex, using such an algorithm would greatly increase the complexity of the Local Performance Consultant's implementation. Also, using such an algorithm would require the addition of communication channels between tool daemons that are not present in our existing tool communication model.

Instead of a distributed consensus algorithm, the PDA centralizes decision making for global instrumentation requests to ensure the same action is taken by all daemons in response to such requests. For efficiency, the PDA uses MRNet to gather instrumentation cost data from the local search agents to the global search agent in the tool's front-end. Because a decision to insert global instrumentation hinges on whether all processes could insert the requested instrumentation without surpassing the PDA's instrumentation cost threshold, and the process with the highest instrumentation cost is the limiting factor in the decision, the PDA uses a "maximum value" MRNet data reduction to collect instrumentation cost information to the global search agent. To limit the overhead of gathering instrumentation cost information, the PDA piggy-backs the cost data with application performance data as it is sent to the tool's front-end.

We extended Paradyn's time-aligned performance data aggregation filter (see Section 4.3.2) to accept an instrumentation cost value and to compute its maximum. Unlike the performance data, the instrumentation cost data is not aggre-

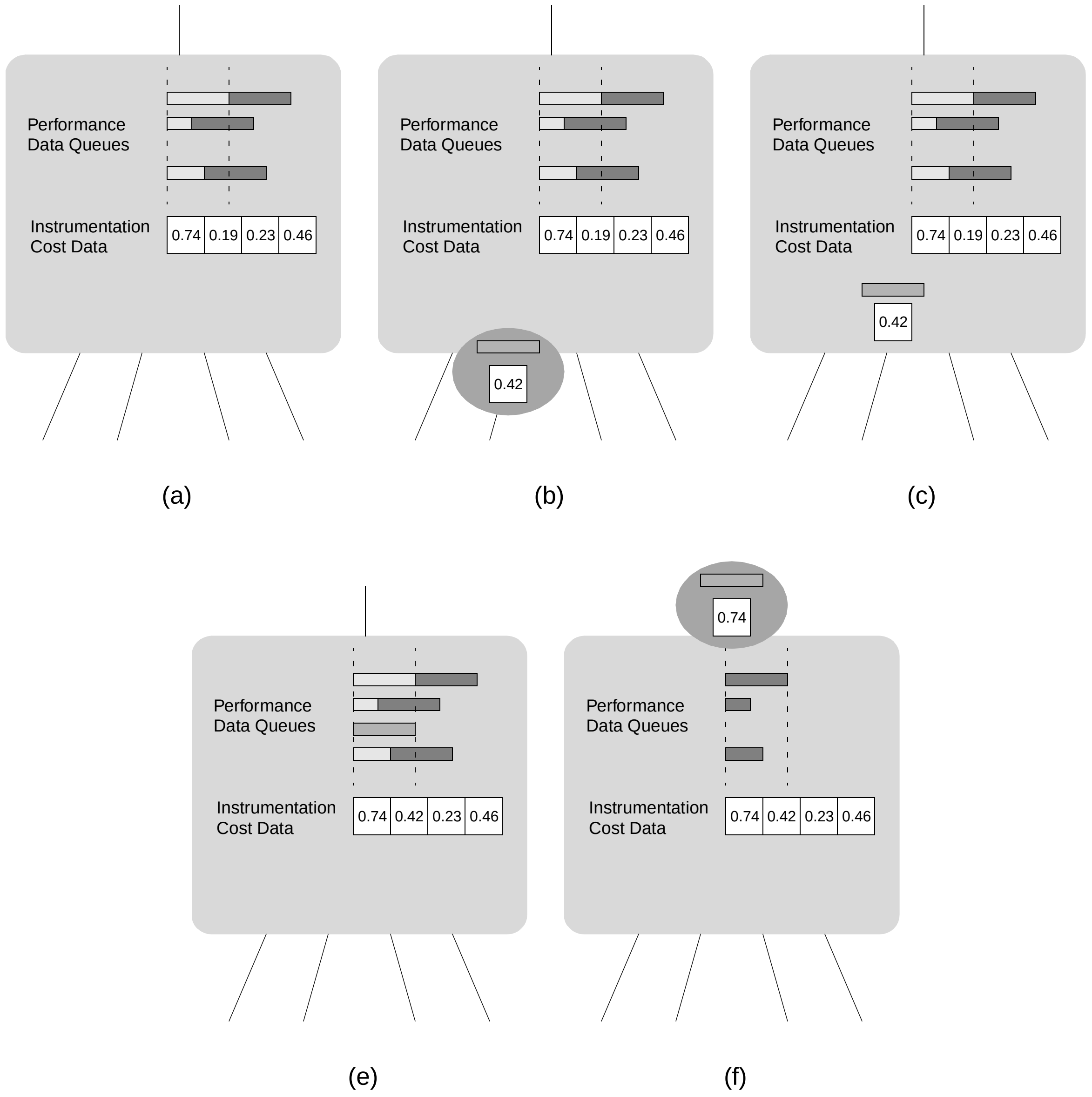


Figure 5.3 Reduction of instrumentation cost data as it is delivered to the Paradyn front-end. In the figure, Paradyn's time-aligned performance data aggregation MRNet filter is shown as a shaded, rounded rectangle. The expanded filter maintains both queues of performance data samples and a vector of instrumentation cost information (a). When a sample arrives from one of the downstream processes (b), the performance data sample and instrumentation cost sample are extracted (c), the instrumentation cost value corresponding to the incoming sample's connection is updated and the performance data is binned (e). If the addition of the newly-arrived performance data sample causes the filter to produce an output packet, the filter's maximum instrumentation cost value is delivered with the outgoing sample.

gated using time-aligned aggregation. Instead, the filter selects the maximum instrumentation cost value using only the most recent cost data available as shown in Figure 5.3. Each filter running throughout MRNet's overlay network maintains a vector containing the most recent instrumentation cost value received on its downstream connections (Figure 5.3a). When a performance data and instrumentation cost sample arrives from a downstream process (Figure 5.3b), the filter extracts the performance data sample and the instrumentation cost value (Figure 5.3c). The filter bins the performance data as described in Section 4.3.2, and updates its instrumentation cost vector so that the element corresponding to the sample's arrival connection contains the new cost value (Figure 5.3d). If the newly-arrived performance data sample causes the filter to produce an output performance data sample, it also determines the maximum value of its instrumentation cost values and attaches it to the outgoing performance data sample (Figure 5.3e). Using this approach, the front-end obtains a timely approximation of the highest instrumentation cost among all application processes.

5.2.2 Dynamic Instrumentation Scheduling Policy

The PDA's problem of scheduling local and global instrumentation requests is similar to (but more restricted than) the problem faced by the scheduler of a parallel system with two job classes. Like a parallel system scheduler, our instrumentation scheduler can use either fixed or dynamic partitioning of the available capacity. Because fixed-partition policies guarantee a portion of the capacity for each class, global instrumentation cannot starve local instrumentation and vice versa. However, such policies can be inefficient; if the instrumentation cost for one class is below its class' capacity, instrumentation of the other class cannot consume the unused capacity. Dynamically-partitioned scheduling policies avoid this disadvantage of fixed-partition policies, but require coordination between local and global search agents when adjusting the partition.

Many job scheduling policies for multiprogrammed parallel systems have been proposed and studied; a recent survey by Feitelson [29] cites over six hundred publications. Typically, only one job scheduler makes scheduling decisions for a computing resource, and information about each requested job is available to the scheduler (though it may not consider every request in each scheduling decision). Distributed queues have been used for load balancing (e.g., [22,93]), but each of the queues is associated with a single resource. In contrast, the Distributed Performance Consultant's global search agent and each local search agent make distributed scheduling decisions about the same resource, the instrumentation capacity of the local application process.

Our prototype PDA instrumentation scheduler uses a dynamically-partitioned policy with per-class soft and hard instrumentation cost limits. The combined instrumentation cost threshold defines the available instrumentation capacity; the soft limits partition this capacity. If a class' instrumentation cost is below its soft limit, our scheduler allows instrumentation of the other class to consume some of the unused capacity. Hard cost limits ensure that instrumentation of one class does not starve instrumentation of the other.

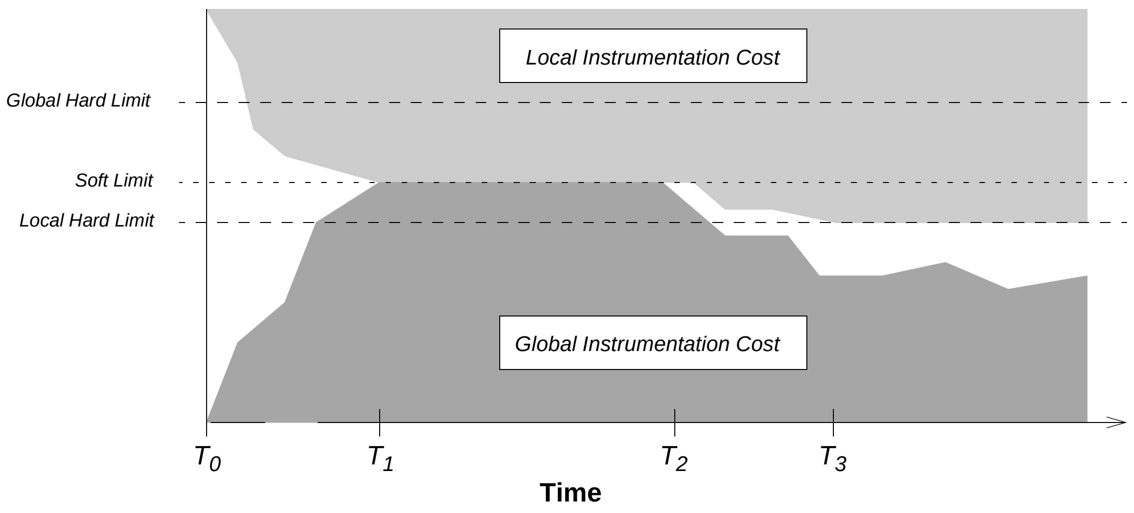


Figure 5.4 Example of instrumentation scheduling by the Distributed Performance Consultant. The figure shows the cost of instrumentation in a single application process over time. The Distributed Performance Consultant begins to insert instrumentation at time T_0. At time T_1, both local and global instrumentation are throttled by the soft instrumentation limit. At time T_2, the removal of global instrumentation causes the global instrumentation cost to fall below its soft limit, and no global instrumentation is available to consume the unused capacity. In response, local instrumentation is inserted to consume the unused capacity until its cost reaches the local instrumentation hard limit at time T_3.

An example of our scheduler's operation is shown in Figure 5.4. In the figure, the local and global search agents each begin inserting instrumentation at time T_0. Both types of instrumentation are throttled at the soft limit at time T_1. At time T_2 in the figure, global is removed that causes the global instrumentation cost to fall below its soft limit. Because there are no pending global instrumentation requests, our scheduler allows local instrumentation to exceed the local instrumentation soft limit. Local instrumentation is inserted until its cost reaches the local instrumentation hard limit at time T_3.

Our instrumentation scheduling policy trades the potential for unused search capacity against the guarantee that a local search will not block the global search and vice versa. Because the local and global search agents use the same search refinement rules, we expect them to produce similar sequences of instrumentation requests. Therefore, once the overall search reaches steady state we expect both local and global instrumentation to be throttled by the soft cost limits under our instrumentation scheduling policy.

5.3 TDA Bottleneck Search Strategy

The PDA bottleneck search strategy (Section 5.1) yields the expected tool scalability benefit, but at a cost in tool complexity due to the need to schedule and track instrumentation cost for both global and local instrumentation (Section 5.2). A truly distributed bottleneck search strategy (i.e., one with no centralized search component for global instrumentation) avoids the increased tool complexity while retaining the scalability advantages of the partially distributed strategy.

Because the TDA performs no explicit examination of the application's global behavior, it must provide insight about global behavior using local behavior information. There are several possible approaches for providing this insight. The first approach is to assume that the behavior across all application processes is so similar that bottleneck search results taken from a limited number of processes is representative of all the others. This approach fails if the chosen processes not truly representative of the other processes. A better approach is to incorporate information from all application processes into the approximation. With this approach, the bottleneck search results can identify not only behavioral variations across all application processes, but also can identify how many and possibly which processes exhibit each type of behavior. Information about each application process must be included to ensure that the Distributed Performance Consultant's global bottleneck results truly represent the application's global behavior.

Our approach for incorporating local application behavior information from each process is to use the Sub-Graph Folding Algorithm to produce a composite search history sub-graph that approximates the application's qualitative global behavior. The SGFA places application processes into qualitative behavioral categories and presents one composite sub-graph of the overall search history graph for each category. The SGFA retains information about the number of processes in each category. By leveraging the SGFA, the Distributed Performance Consultant can use the truly distributed search strategy without sacrificing insight into global application behavior.

5.4 Evaluation

To evaluate the Distributed Performance Consultant, we modified the Paradyn performance tool to use either the CA, PDA, or TDA. We performed a scalability study for the three approaches, comparing the computation and communication load at the tool's front-end, its back-ends, and the MRNet internal processes during a bottleneck search.

5.4.1 Experimental Environment

We implemented the Distributed Performance Consultant within Paradyn version 4.1, modifying it to use either the CA, PDA, or TDA. To support the PDA, we modified Paradyn to track the cost of global and local instrumentation separately, and to implement an instrumentation scheduling policy that uses a simple fixed-partitioning policy. To support the TDA, we implemented the Sub-Graph Folding Algorithm using custom MRNet filters to support scalable presentation of Distributed Performance Consultant results. We modified the Paradyn daemon to be multi-threaded with a Local Data Manager thread, Local Performance Consultant thread, and Local Communications Manager thread that interfaces with MRNet. We also modified the tool to use one daemon process per application node, as opposed to its previous approach of using one daemon process per application process. Finally, to support running our experiments under a batch scheduling system, we modified the tool front-end to present a text-based user interface instead of a graphical user interface and we hard-coded the tool to perform our experiment when our batch jobs were run.

For all experiments, we used `su3_rmd`, a quantum chromodynamics application produced by the MILC collaboration [73] to simulate pure lattice gauge theory. The code is implemented in C and uses MPI for inter-process communication. We used a weak scaling approach in our study.

Our experiments were run on the Multiprogrammatic Capability Cluster (MCR) [61] at Lawrence Livermore National Laboratory. At the time the experiments were performed, MCR contained 1152 nodes (1048 compute nodes) connected with a Quadrics QsNet Elan3 interconnect. Each node had two 2.4 GHz Pentium 4 Xeon processors and 4 GB RAM. Each node ran CHAOS 2.0 [32], a

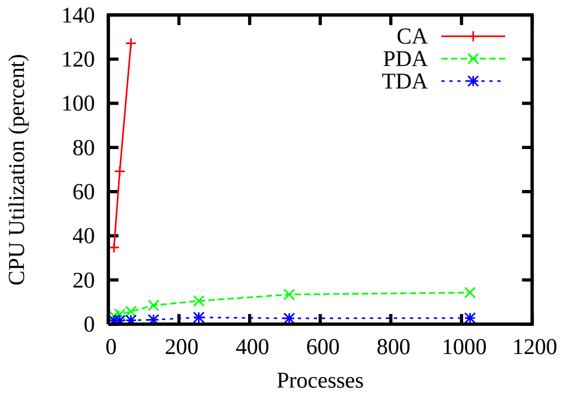
Linux distribution derived from Red Hat Enterprise Linux 3 by LLNL. The MPI implementation on MCR is provided by Quadrics, but is based on the MPICH [41] 1.2 distribution.

5.4.2 Experimental Results

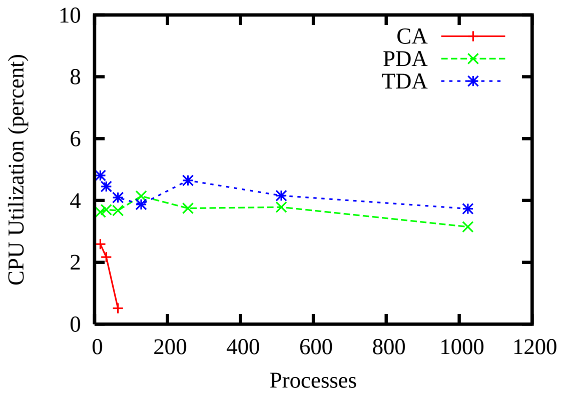
The results of our scalability study are shown in Figures 5.5—5.7. We requested batch job time limits for runs using the CA that were three times longer than for runs using the PDA and TDA. Because CA runs with 64 application processes failed to complete during their batch job's allotted time limit, we did not attempt runs using the CA with more than 64 processes. In contrast, we performed experiments with the PDA and TDA for up to 1024 application processes, limited by the available system size and not by resource saturation. We used the SGFA to verify that the qualitative results produced by each search strategy provided comparable results.

For our experiments we used balanced MRNet topologies with a moderate fan-out of eight. Because the number of data points in our scalability study would be limited if we used only "complete" MRNet topologies (i.e., topologies that used the same fan-out at each level in the process network), we also used topologies with fan-outs of eight at all levels except for the processes directly connected to the daemons. By using a smaller fan-out of two or four at this last level, we were able to run experiments for numbers of application processes at each power of two between 16 and 1024.

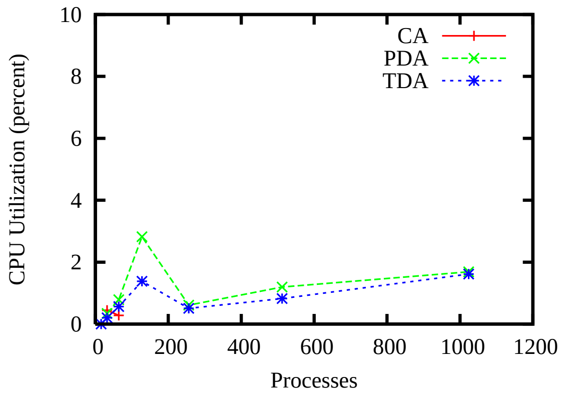
Figure 5.5 compares the computation load at the tool's front-end, daemons,



(a) Front-End CPU Load



(b) Daemon CPU Load



(c) MRNet Internal Process CPU Load

Figure 5.5 Computation load for the CA, PDA, and TDA. The CPU utilization is shown for (a) the tool's front-end, (b) tool daemons, and (c) MRNet internal processes. Runs using the CA with more than 32 application processes did not complete during their batch job's allotted time limit. Note the Y-axis scale differs between plots.

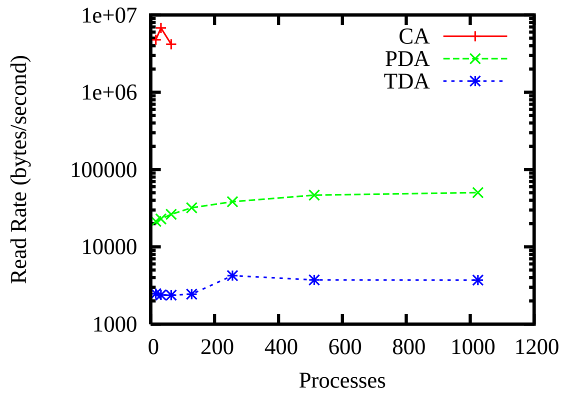
and MRNet internal processes for each bottleneck search strategy under consideration. Each chart in the figure shows the CPU utilization for each search strategy across a range of process counts. We measured CPU utilization by sampling the `getrusage` system call at one-second intervals in the front-end process, all daemon processes, and all MRNet internal processes during the bottleneck search. For each process, we computed the average CPU utilization over the duration of the bottleneck search. The data point for a given search strategy and application process count is the average across all processes of the same type (front-end, daemon, or MRNet internal process) across all runs using the search strategy on that number of application processes.

Our CPU load results show the expected scalability benefit of the TDA search strategy and, to a lesser extent, the PDA search strategy. CA saturates the tool front-end with relatively small numbers of application processes; back-pressure causes the daemon CPU load to decrease. (The CPU load reported by `getrusage` can be larger than 100% on multiprocessor hosts such as the nodes of the MCR cluster.) In contrast, when using TDA the front-end CPU load remained below 5% and relatively constant as we varied the number of application processes.

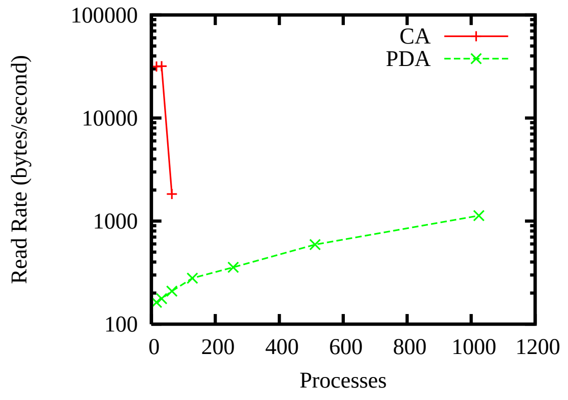
The average CPU load for daemons and MRNet internal processes also remained below 5% in our experiments with the distributed search strategies. As expected, the MRNet internal process CPU load under the PDA tends to be slightly higher than the CPU load under the TDA because the internal processes are aggregating global performance data under the PDA but not under the TDA. However, our CPU load results also revealed unexpected behavior. First, the daemon CPU load tends to be slightly lower under the PDA than the TDA. This behavior may be the result of differences in the way that PDA searches and TDA searches are performed by the daemons. Under the PDA, local sub-searches are started only when the front-end refines a global experiment to a host-specific experiment. If global experiments with hypotheses like ExcessiveSyncWaitingTime and ExcessiveIOBlockingTime are not refined, daemons are not involved in evaluating the performance data for these hypotheses during a search under the PDA. In contrast, under the TDA each daemon begins its search by creating host-specific experiments for all hypotheses, and continues to evaluate performance data for those experiments throughout its bottleneck search.

A second unexpected behavior exposed by our CPU load results is the dip in the TDA daemon CPU load and corresponding spike in MRNet internal process CPU load for 128 application processes. We observed that the daemon TDA load curve follows a sawtooth pattern. The high points in the curve correspond to MRNet topologies where the fan-out at the last level in the process network is two, intermediate points where the last-level fan-out is four, and low points when the last-level fan-out is eight. Whether the variations in CPU load are related to the MRNet topology, and the nature of this relationship, remains an open question.

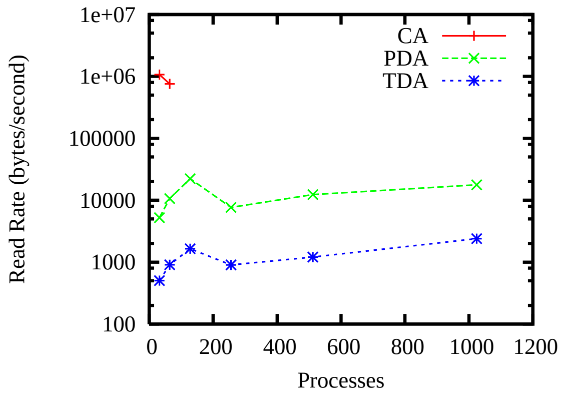
Figures 5.6 and 5.7 compare the network I/O load at the tool's front-end, back-ends, and MRNet internal nodes for each of our three bottleneck search strategies. We instrumented the MRNet library to collect the number of bytes read (Figure 5.6) or written (Figure 5.7) on all MRNet socket connections. We modified each Paradyn process to sample these counts at one-second intervals during a bottleneck search. For each process, we computed the average read or write rate during the bottleneck search. To obtain the chart values for a given search strategy, we averaged the read or write rates for all processes of the same type (front-end, daemon, or MRNet internal process) across all runs that used that search strategy. Variability across runs was low.



(a) Front-End Read Rate



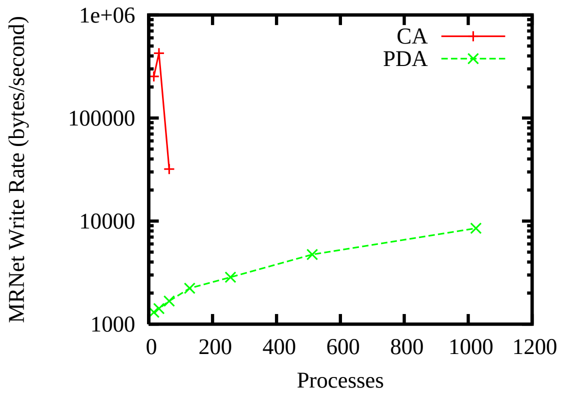
(b) Daemon Read Rate



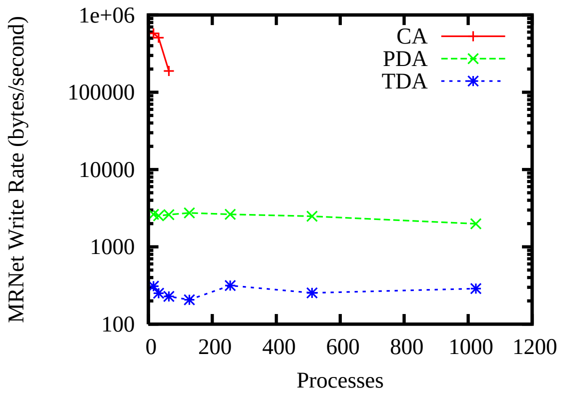
(c) MRNet Internal Process Read Rate

Figure 5.6 MRNet read load for the CA, PDA, and TDA. The number of bytes read by MRNet is shown (a) for the tool's front-end, (b) tool daemons, and (c) MRNet internal processes. Runs using the CA strategy with more than 32 application processes did not complete during their batch job's allotted time limit. The Y-axis is logarithmic in each plot. There is no curve for the TDA search strategy in the daemon read rate plot because daemons do not receive data across MRNet during a TDA search.

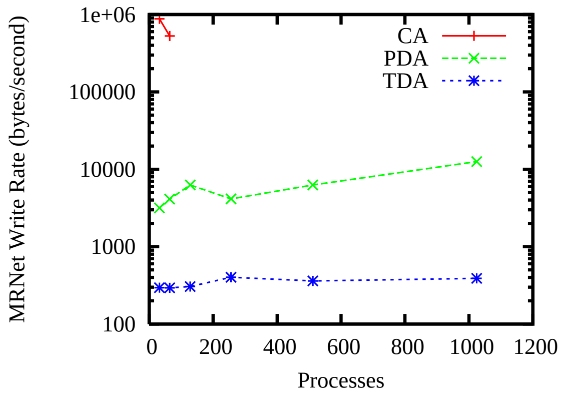
Our MRNet I/O results show the scalability of the TDA. As expected, there were no front-end writes nor daemon reads during the bottleneck search under the TDA strategy. When there were reads and writes under the TDA, the data rate was very low and remained nearly constant as we increased the number of application processes. Our TDA I/O results also exhibit some variability that ech-



(a) Front-End Write Rate



(b) Daemon Write Rate



(c) MRNet Internal Process Write Rate

Figure 5.7 MRNet write load for the CA, PDA, and TDA. The number of bytes written by MRNet is shown for (a) the tool's front-end, (b) tool daemons, and (c) MRNet internal processes. Runs using the CA strategy with more than 32 application processes did not complete during their batch job's allotted time limit. The Y-axis is logarithmic in each plot. There is no curve for the TDA search strategy in the front-end write rate plot because daemons do not send data across MRNet during a TDA search.

oes the variability in the TDA CPU load results. As with the CPU load, it remains an open question whether and how the I/O fluctuation is related to the MRNet topology we used.

5.5 Summary and Future Work

To address the problem of finding and diagnosing performance problems in applications with a large number of processes, we developed, implemented, and evaluated a performance diagnosis approach consisting of three parts. Our approach incorporates an on-line, automated performance bottleneck search strategy that distributes the portions of the sub-search that examine local application behavior. For monitoring and controlling the dynamic instrumentation generated during a bottleneck search, our approach also includes a model of the cost of instrumentation in parallel computation that tracks cost information for each application process and an approach for scheduling dynamic instrumentation requests generated by our bottleneck search. To reduce the complexity introduced by the need to monitor and schedule dynamic instrumentation for explicit examination of global application behavior, our approach includes a truly distributed search strategy that leverages the Sub-Graph Folding Algorithm to approximate the results of a global bottleneck search.

We implemented our performance diagnosis approach in Paradyn, naming the enhanced performance diagnosis component the Distributed Performance Consultant. In experiments with up to 1024 application processes, the Distributed Performance Consultant placed less computation and communication load on the tool's front-end than Paradyn's traditional centralized search strategy while retaining low back-end loads. Even at the highest tested process count, with the truly distributed approach the computation and communication loads on all tool components showed excellent scalability with no sign of resource saturation.

There are several potential directions for future work based on the research described in this chapter. The first is to understand the unexpected variation in

Distributed Performance Consultant computation and communication load as we varied the number of application processes. It is an open question whether the use of MRNet topologies without the same fan-out at each tree level accounts for the load variations we observed.

Another research direction is to explore the sensitivity of SGFA to application characteristics such as its use of adaptive mesh refinement and its sensitivity to the thresholds used in a Distributed Performance Consultant search. Research in these directions would help answer the open question whether information about the qualitative behavioral categories exhibited by application processes (including the number of processes in each category) provides a sufficient approximation to qualitative bottleneck search results based on global performance data for a wide range of parallel applications.

A third direction for potential future work involves the exploration of aspects of instrumentation cost beyond its CPU cost. Our instrumentation cost model tracks only the instrumentation's CPU cost. However, instrumentation exacts a cost in other forms such as memory cost and communication cost. Our instrumentation cost model could be extended to keep a multi-dimensional cost value for each application process. However, introducing multiple cost dimensions complicates the comparison of cost values. For example, if the instrumentation cost for one process has a higher CPU cost component but lower memory cost component than the instrumentation cost for another process, it is not clear which process' instrumentation cost is greater. The use of multi-dimensional instrumentation cost values remains an open issue.

Chapter 6

Scalable Automated Performance Diagnosis for Applications with Large Call Graphs

In this chapter, we address the problem of finding performance problems in applications with a large number of functions and, hence, a large call graph. We and others have shown the benefit of using application structural information such as its call graph to find application bottlenecks [15,36,43,103]. However, a search strategy guided solely by the application's call graph (hereafter called the call graph search strategy) suffers from two problems: it is inefficient for examining the behavior of applications with large or complex call graphs, and it can be ineffective at finding application performance problems that are hidden by the application's call graph structure.

The first problem with the call graph search strategy is that it is inefficient for examining application's with large or complex call graphs. Many factors determine the depth and complexity of an application's call graph, but certain programming styles (e.g., object-oriented styles that favor the use of many simple

$$\begin{tikzcd}
 & & B \arrow[dr] & \\
\cdots \arrow[r] & A \arrow[ur] \arrow[r] \arrow[dr] & C \arrow[r] & E \\
 & & D \arrow[ur] &
\end{tikzcd}$$

Figure 6.1 Part of an application call graph showing a hidden performance bottleneck. In the call graph, function A calls functions B, C, and D. Each of these functions calls a utility function E. If none of the functions B, C, and D are found to be a performance problem, the behavior of E will never be investigated by search strategies that rely strictly on the application's call graph to guide refinement. In contrast, if E is a performance problem, it is likely to be found frequently in Deep Start's stack samples so Deep Start will investigate its behavior even if B, C, and D are not performance problems.

functions over the use of few complex ones) tend to produce applications with such call graphs. High-level libraries that rely on lower-level libraries, such as a radiation transport simulation library that uses a parallel solver library (that itself relies on a parallel communication library and a linear algebra library), may also contribute to increased call graph complexity. The call graph search strategy refines its search step-by-step through the application's call graph, an expensive approach if the call graph is deep or includes many functions to examine at a given level within the call graph.

The second problem with the call graph search strategy is that it may not find performance problems hidden by the application's call graph structure. For example, when searching for performance problems in an application whose call graph is partially shown in Figure 6.1, after finding A to be a bottleneck, if the call graph search strategy determines that none of B, C, and D are bottlenecks, it will not examine E even though E may be a significant performance problem. This scenario can occur when E is a utility function called from many places in an application, but none of the callers individually is a performance problem. Hall presents a similar example to motivate his use of call path profile data in a manual technique for finding application performance problems [43].

To address the two problems affecting the call graph search strategy, we developed Deep Start, a strategy for automatically finding performance problems in applications with large and complex call graphs. Deep Start uses sampling to augment an automated, dynamic-instrumentation-based search for application bottlenecks. Deep Start substantially improves an automated performance diagnosis tool's search effectiveness by locating performance problems more quickly than straightforward search strategies guided only by the application's call graph. Existing automated performance diagnosis tools have used one approach or the other; our work shows the benefit of using a hybrid approach. Furthermore, Deep Start can find problems that are hidden from these more straightforward search strategies. Our Deep Start research is complementary and orthogonal to our other scalable performance diagnosis research.

This chapter presents the design, implementation, and evaluation of the Deep Start search strategy. We begin with by presenting Deep Start's design and implementation within the Paradyn Performance Consultant (Section 6.1) followed by the results of our Deep Start evaluation (Section 6.2). We conclude the chapter with a summary of our Deep Start research and a discussion of potential directions for future work (Section 6.3).

6.1 The Deep Start Search Strategy

The Deep Start bottleneck search strategy augments a call graph search strategy with the examination of *deep starters*, functions likely to be close (in terms of the search strategy) to functions exhibiting actual performance problems. Deep Start uses call stack information, gathered opportunistically from normal bottleneck search activity, to identify deep starters. Once Deep Start has identified deep starters, it extends the traditional call graph search to investigate the behavior of deep starters with higher priority than its normal search.

Although Deep Start benefits greatly from sampling, we feel it is undesirable to replace dynamic instrumentation entirely with sampling for performance data collection. Some types of performance information are more difficult to collect using sampling than instrumentation. For example, collecting inclusive CPU performance data (CPU data for a function and all of its direct and indirect callees) is possible using sampling, but it is complicated and expensive. Also, it is impossible to collect accurate counts of program events using sampling. In contrast, dynamic instrumentation is well suited for collecting inclusive performance data and accurate event counts. Collecting inclusive performance data requires only instrumentation that samples a timer at function entry and exit and the calculation of the difference between these two timer samples. Collecting accurate event counts using dynamic instrumentation is as simple as inserting instrumentation to increment a counter whenever the event occurs. In light of sampling's limitations for collecting some types of performance information, we feel it is not desirable to replace dynamic instrumentation entirely with sampling.

6.1.1 Stack Sampling

Deep Start collects stack samples opportunistically as dynamic instrumentation is inserted and removed to support Deep Start's search activity. In our implementation context, Paradyn daemons insert instrumentation code when requested by some component of the Paradyn front-end process such as the PC or Paradyn's visualization manager. When a daemon inserts instrumentation code into an application process, it must ensure that the process is not executing in code that will be overwritten by the code patch. This safety requirement implies that the daemon must consider not only the location where the process is executing, but also the locations to which control will return when the process completes any function calls that are in progress. That is, the daemon must consider the current program counter and the return addresses of all frames on the call stack. The daemon performs its instrumentation safety check by pausing the process and walking its call stack. To support Deep Start, each time a Paradyn daemon performs a stack walk it saves the stack walk information and the ID of the process whose stack was checked. Each daemon buffers the stack samples it collects, and delivers them in a single batch in response to periodic requests from the PC.

6.1.2 Choosing Deep Starters

Deep Start uses the stack walk samples collected by Paradyn's daemons to choose deep starters. Deep Start selects deep starters whenever a search path is first refined to examine specific application functions such as `main`. Once the deep starter selection algorithm has been triggered for a search path, the selec-

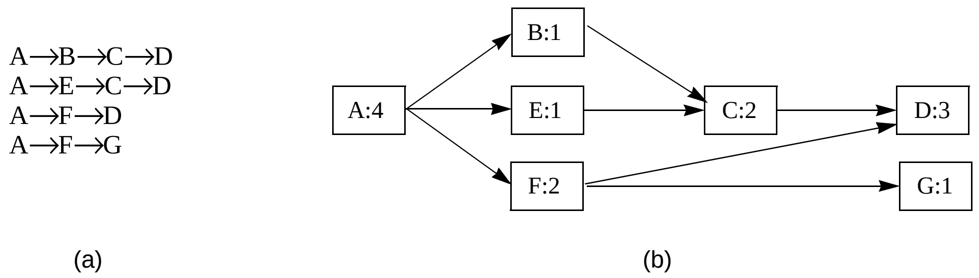


Figure 6.2 Example of a Deep Start function count graph. Given a collection of stack samples (a), Deep Start builds a function count graph (b) with nodes indicating function and a count of times it occurred at that position in the stack samples.

tion algorithm is triggered in response to each subsequent refinment that extends the path.

To select deep starters, Deep Start uses the stack sample information to maintain a data structure we call a *function count graph*. Nodes in this directed graph represent functions of the application, while edges represent a call relationship between two functions as dictated in the stack samples. For each function represented in the graph, the graph keeps a count of the number of times the function was seen in the stack samples. For instance, if Deep Start collects the four stack samples shown in Figure 6.2a (where $x \rightarrow y$ denotes that function x had called function y at the time the stack was sampled), it builds the function count graph shown in Figure 6.2b. In the figure, node labels indicate both the function and its observed count.

One of the strengths of the PC is its ability to examine not only the behavior of an application as a whole but also its behavior at per-host and per-process granularity. This capability is invaluable for finding performance problems caused by workload imbalance or faulty hardware. Deep Start enhances these finer-granu-

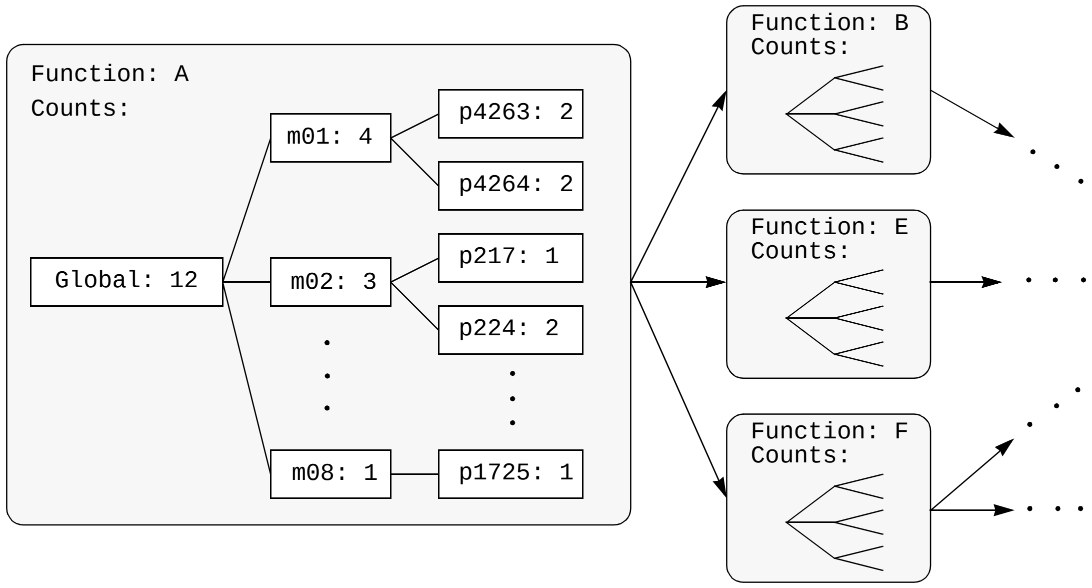


Figure 6.3 Function count graph nodes with count-tree. Unlike the simplified function count graph shown in Figure 6.2, each node of Deep Start's function count graph contains a tree of counts for its function. This count-tree tracks the number of times the function was observed in the stack samples at application, machine, and process granularity.

larity searches with the selection of deep starters using global, per-host, and per-process data. To support choosing deep starters using function count information at varying granularities, Deep Start's function count graph is slightly more complicated than the graph shown in Figure 6.2. In the actual function count graph, each node still represents a single function from the application but instead of a single count per node, Deep Start maintains a tree of counts at each node as shown in Figure 6.3. The root of a count-tree contains an overall count for its associated function, counting the number of times the function was seen in all stack samples. The nodes at the first level of this tree count the number of times that the function was observed in the stack samples from specific hosts. The second level of this tree counts of the number of times that the function was observed in the stack samples of specific application processes. With count-trees, the PC can restrict the stack samples it considers when choosing deep starters to make per-host and per-process deep starter selections. For example, if a search path (a sequence of PC experiments related by refinement; see Chapter 3) has been refined to examine the behavior of a specific process p on a host h, when the PC traverses the function count graph to select deep starters for this search path it uses the count from the count-tree associated with process p on host h when comparing against the deep starter threshold.

Deep Start maintains its function count graph by obtaining call stack samples from the Paradyn daemons and updating the structure and counts in the function count graph based on the information in each stack sample. For each stack sample it receives, the PC processes the stack sample starting at the function on the bottom of the call stack. For each function in a stack sample, the PC walks the function count graph to increment the counts in the count-tree of the node associated with the function. While processing a stack sample, if no function count graph node exists for the function under consideration, the PC creates a new node for the function in the function count graph and initializes the counts in its count-tree to one.

After the PC updates the function count graph, it traverses the graph to find deep starters. Functions that execute frequently or that take a long time to execute are good candidates, so Deep Start looks for functions that occur frequently in the stack samples. We choose those functions whose counts are higher than a user-configurable threshold, where the threshold's value represents a percentage of the total number of stack samples in the function count graph. For most applications, the number of deep starters chosen increases as the threshold is decreased. Setting the threshold too low may have a detrimental effect on the PC search because it tends to increase the number of “false positives” (deep starters that are not application bottlenecks). Based on our experience with the initial implementation, we set the default deep starter threshold to be 60% of the number of stack samples in the function count graph. Furthermore, to focus the PC’s attention on the functions that are likely to be closest to the application’s performance bottlenecks, we choose as deep starters only the deepest (i.e., furthest from the root) of the above-threshold functions in the function count graph.

6.1.3 Adding Deep Starters

Once a deep starter function is selected, Deep Start creates an experiment for the deep starter and adds it to its bottleneck search. The experiment whose refinement triggered the deep starter selection algorithm determines the nature of the deep starter’s experiment. The deep starter experiment uses the same hypothesis and focus as the triggering experiment, except that the portion of the triggering experiment’s focus that specifies Code resources is replaced with the deep starter function. For example, if the triggering experiment is

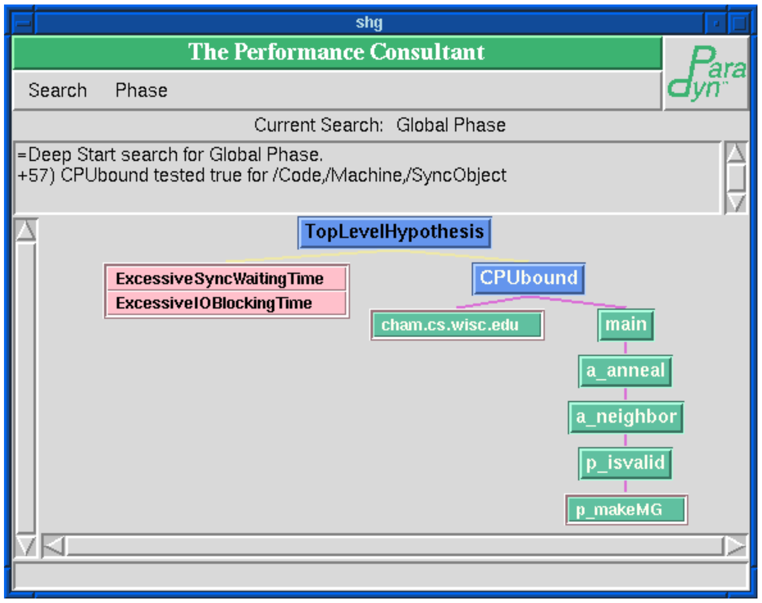
hypothesis:	CPU bound
focus:	`< /code/om3.c/main, /Machine/c2-047/om3{1374} >`

and the function `time_step` is chosen as a deep starter when the selection algorithm is triggered, the deep starter experiment is

hypothesis: CPU bound
focus: `< /code/om3.c/time_step, /Machine/c2-047/om3{1374} >`.

The PC throttles the amount of active instrumentation according to a user-configurable threshold to avoid having an excessive effect on the behavior of an application. To support instrumentation throttling, the PC keeps a priority-ordered queue of experiments that have been defined but are not yet being evaluated. In the Deep Start search strategy, the PC defines deep starter experiments with a higher priority than is used for the experiments generated by normal PC search activity. Since priorities are inherited as the PC refines its search, the sub-search rooted at the deep starter experiment retains precedence over any experiments generated by the normal operation of the PC. Hence, using high priority for deep starter experiments causes the PC to focus its attention in the search space near the deep starters. Since these functions are likely to be near the actual bottlenecks of the application, the PC is likely to find bottlenecks more quickly than with a strict top-down search of its search space.

One of the comforting properties of Paradyn's search history graph (Figure 1.2) is that the search reflects the application's call graph structure when the PC is investigating the application's Code resources. To retain this property with the Deep Start search strategy, when adding a deep starter experiment the PC adds as many experiments as necessary to connect the deep starter experiment to some other experiment that is already present in the search history graph. An example of connecting experiments is shown in the search history graph in Figure 6.4. In the search shown in the figure, the PC added connecting



```
=Deep Start search for Global Phase.
+57) CPUbound tested true for /Code,/Machine,/SyncObject
```

Figure 6.4 Deep starter and connecting experiments in the Search History Graph display. In this example, p_makeMG is the deep starter; a_anneal, a_neighbor, and p_isvalid are experiments that connect the deep starter to the normal PC search activity.

experiments for `a_anneal`, `a_neighbor`, and `p_isvalid` when adding the deep starter experiment for `p_makeMG`.

6.2 Deep Start Evaluation

To evaluate Deep Start, we modified the PC to search using both the Deep Start strategy and its current call graph-based search strategy. We compared the behavior of the Deep Start strategy and the current strategy while searching for performance problems in several scientific applications.

6.2.1 Experimental Environment

We performed our experiments on two sequential and two MPI-based parallel scientific applications (see Table 6.1). The MPI applications were built using version 1.2.2 of the MPICH [41] MPI implementation. We modified the PC within

Name	Version	Type	Language	Domain	Size
DRACO	6.0	Sequential	Fortran 90	Hydrodynamic simulation	68981 lines 18632 KB 398 functions
ALARA	2.4.4	Sequential	C++	Induced radioactivity analysis	19576 lines 2911 KB 720 functions
om3	1.5	Parallel (MPI)	C	Global ocean climate simulation	2674 lines 385 KB 36 functions
su3_rmd	6	Parallel (MPI)	C	Quantum chromodynamics pure gauge lattice theory simulation	35845 lines 511 KB 189 functions

Table 6.1: Characteristics of the applications used to evaluate Deep Start.

the Paradyn version 3.2 software base to search using either the Deep Start search strategy or its current call graph-based search strategy. To support the comparison of searches performed using different search strategies, we also modified Paradyn's search history graph export facility to export complete information about a search's refinement structure.

For all experiments, we ran the Paradyn front-end process on a lightly-loaded Sun Microsystems Ultra 10 system with a 440 MHz UltraSPARC IIi processor and 256 MB RAM. We ran the sequential applications on another Sun Ultra 10 system on the same LAN. We ran the MPI applications on eight nodes of a dedicated Linux cluster. Each node contains a 600 MHz Pentium III processor with 256 KB L2 cache and 128 MB RAM, and runs Linux kernel version 2.2.19. The cluster's nodes are connected by a 100 Mb/s Ethernet switch.

6.2.2 Experimental Methodology

Our experiments consisted of several trials with each application. Each trial consisted of five runs of the application. During each run we used the PC to search for application problems. The same search strategy was used for each run of a trial. During each run, we began the PC search once the application finished its initialization phase so that the PC search investigated the behavior of the application's computation phase. Once the PC search reached a steady state, such that it was not activating any new experiments, we exported the search history graph for post mortem analysis.

6.2.3 Experimental Results

We began by investigating the sensitivity of the Deep Start search strategy to changes in the deep starter threshold (see Section 6.1.2). We performed experiments using a range of deep starter thresholds on one sequential application and one parallel application. Based on the results of these experiments, we selected a single deep starter threshold for use in our remaining experiments. We then compared the performance of the Deep Start and call graph search strategies for each of our test applications.

In our analysis, we often wanted to determine which of two searches showed a better result. To determine whether one run's search is better than another, we borrow the concept of *utility* from microeconomics' consumer choice theory [82]. Utility provides a formal mechanism to compare a person's preferences in various situations; in our case, we wish to reflect a tool user's preferences for obtaining timely results from the tool. To quantify the results of a search, we postulate a utility function that captures a user's preference for obtaining results from the tool, weight the observed search results by the utility function, and sum the weighted results to obtain a single value we call a *utility sum* that describes the observed search results. To capture the idea that users prefer results given earlier in the search, the appropriate utility function is one that is a decreasing function of time. For our analysis, we chose the linear function $U(t) = -t$. This utility function always produces weighted sums with negative values; to make our results easier to understand, we use the absolute value of a search's weighted sum as its utility sum. Hence, the best search result from a collection of search results is the one with the smallest utility sum.

While we evaluated Deep Start, we observed that the PC had difficulty performing a bottleneck search with one of our test applications. The application, su3_rmd, performs frequent gather operations. Instead of using a collective communication operation provided by MPI, su3_rmd implements gather operations using a sequence of point-to-point message transfers. To distinguish between the messages of distinct gather operations, su3_rmd uses a unique message tag for the messages involved with each gather operation. Paradyn dynamically recognizes when new message tags (and MPI communicators) are used, so that they can be considered in a bottleneck search. However, su3_rmd performs gather operations so frequently that Paradyn's ability to recognize and incorporate the new message tags was overwhelmed. Furthermore, it is not profitable for the PC to refine its search for each gather operation's message tag; by the time the PC defines and activates an experiment for a given message tag, the gather operation is usually complete so Paradyn will never see the message tag being used. To deal applications that use resources ephemerally, we implemented a resource discovery control mechanism in Paradyn that ignores the discovery of new resources if their observed discovery rate becomes too high.

6.2.4 Deep Starter Threshold Sensitivity

To investigate Deep Start's sensitivity to changes in the deep starter threshold, we performed trials with the ALARA sequential application and the om3 parallel application using thresholds of 0.2 (i.e., 20% of the collected stack samples), 0.4, 0.6, and 0.8. Table 6.2 summarizes these experiments. In the table, an application's *total known bottleneck* count is the number of distinct bottlenecks found across all runs of the PC on that application, regardless of the search strategy or the deep starter threshold used in the runs. The table shows the number of PC experiments attempted during a Deep Start bottleneck search (averaged across the five trial runs that used the same deep starter threshold), the average number of bottlenecks found as both an absolute count and as a percentage of the total known bottlenecks for the application, and the average utility sum (see Section 6.2.3) for each tested deep starter threshold. Based on the results shown in the table, we report results from experiments using a 0.2 deep starter threshold. For both om3 and ALARA, this threshold yielded the best average utility sum across all runs in a trial and the most bottlenecks found compared to the other thresholds tested.

Application	Total Known Bottlenecks	Deep Starter Threshold	Average Number of Experiments Attempted	Average Number of Bottlenecks Found	Average Utility Sum
ALARA	46	0.8	174.0	39.8 (86.5%)	158.7
		0.6	172.6	40.2 (87.4%)	140.6
		0.4	173.4	39.8 (86.5%)	134.1
		0.2	171.4	39.6 (86.1%)	133.6
om3	145	0.8	260.2	140.2 (96.7%)	154.1
		0.6	260.8	139.6 (96.3%)	132.9
		0.4	265.0	142.0 (97.9%)	131.2
		0.2	269.0	142.2 (98.1%)	124.5

Table 6.2: Summary of deep starter threshold sensitivity experiments.*Total Known Bottlenecks* is the number of unique bottlenecks observed during any search on the application, regardless of search type and deep starter threshold. A *Utility Sum* (see Section 6.2.3) is a measure of the quality of a search; smaller utility sums are better. Results are averaged over five runs with the same deep starter threshold.

Application	Total Known Bottlenecks	Search Type	Average Experiments Attempted	Average Bottlenecks Found	Average Utility Sum
ALARA	46	Call Graph	174.0	39.4 (86%)	191.9
		Deep Start	173.4	89.8 (87%)	134.1
DRACO	18	Call Graph	105.0	18.0 (100%)	152.7
		Deep Start	105.0	18.0 (100%)	75.2
om3	145	Call Graph	261.6	141.8 (98%)	158.1
		Deep Start	269.0	142.2 (98%)	124.5
su3_rmd	85	Call Graph	260.0	75.6 (89%)	141.8
		Deep Start	261.4	82.2 (97%)	114.3

Table 6.3: Summary of Deep Start/Call Graph comparison experiments.*Total Known Bottlenecks* is the number of unique bottlenecks observed during any search on the application, regardless of search type and deep starter threshold. A *Utility Sum* (see Section 6.2.3) is a measure of the quality of a search; smaller utility sums are better. Results are averaged over five runs with the same deep starter threshold.

6.2.5 Comparison of the Deep Start and Call Graph Searches

In general, Deep Start found more bottlenecks than the PC's call graph-based search strategy. Table 6.3 summarizes the Deep Start and the call graph searches for each of our test applications. Although Deep Start tended to perform more experiments than the call graph search strategy, in cases where neither Deep Start nor the call graph strategy found all of an application's known bottlenecks, the Deep Start strategy found more bottlenecks than the call graph strategy.

Deep Start also produced results more quickly than the call graph search strategy. Figure 6.5 shows search profiles comparing the results produced by Deep Start and the call graph strategy for each of our test applications. Each chart in the figure shows the profile of a Deep Start search and a call graph search for one of our test applications. This type of chart relates the bottlenecks found by a search strategy with the time they were found. In this type of chart, a steeper curve is better because it indicates that bottlenecks were found earlier and more rapidly in a search. In the figure, each profile represents the average time across the five runs of a trial to find a specific percentage of the application's total known bottlenecks. Each profile also includes range bars to indicate the best and worst time taken to find each percentage of the total known bottlenecks. For all of our test applications and nearly all percentages considered, Deep Start found the percentage of the total known bottlenecks more quickly than the call graph search strategy, and found all of its search's bottlenecks an average of 10% to 61% faster.

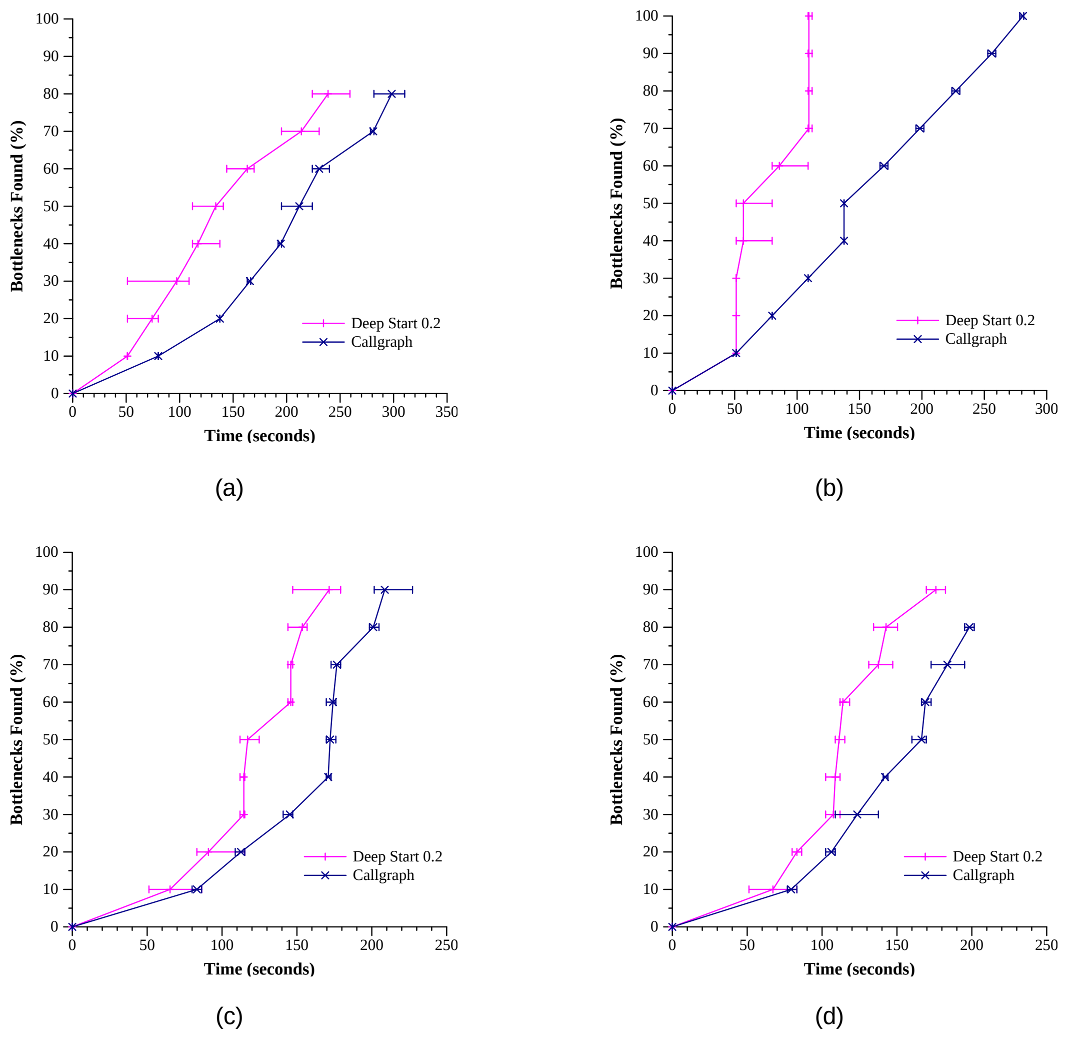


Figure 6.5 Profiles for Deep Start and call graph searches. Search profiles are shown for two sequential applications, ALARA (a) and DRACO (b), and two MPI-based parallel applications, om3 (c) and su3_rmd (d). Each curve represents the average time taken over five runs to find a specific percentage of the application's total known bottlenecks. The range bars indicate the best and worst time taken to find each percentage of the known bottlenecks across the five runs.

6.3 Summary and Future Work

Deep Start is a novel strategy for automatically finding application performance problems that combines call stack sampling with dynamic-instrumentation-based search. During a bottleneck search, Deep Start collects samples from the call stacks of each application process, analyzes the samples to determine which functions are often executing, and then defines high-priority experiments for those functions so that they are examined early in a search. Our evaluation showed that Deep Start can find bottlenecks more quickly than a search strategy guided solely by the structure of an application's call graph, and can find bottlenecks hidden from the more straightforward search strategy.

There are several potential directions for future work based on our Deep Start research. One such direction is to investigate whether the Deep Start approach is beneficial for diagnosing performance problems other than CPU bottlenecks. In its initial implementation, Deep Start looks for deep starters whenever it refines its search through the application's code, regardless of the hypothesis that is being investigated. One open question is whether Deep Start's current approach to collecting and analyzing call stack samples provides any benefit for diagnosing non-CPU-based potential performance problems such as whether an application process is frequently blocked when communicating with other application processes. Another open question is whether the use of semantic information about the functions in the call stack samples can allow Deep Start to make better deep starter selections when investigating non-CPU performance problems. For example, when investigating synchronization bottlenecks, Deep Start may be able to make better deep starter selections if it can recognize when synchronization functions occur in the stack samples.

Another possible direction for research based on our Deep Start work involves selecting deep starters based on sources of performance data other than the application's call stack when making deep starter selections. For example, when investigating synchronization bottlenecks, it may be beneficial for Deep Start to sample the network activity on the application process' hosts; such data may be available via a system function or a file in the hosts' `proc` filesystem. When incorporating new sources of performance data, the cost of sampling the data is a key consideration: one reason Deep Start's call stack sampling is so appealing is because of the incremental cost of obtaining the samples is small over the cost of the PC's normal search activity.

Chapter 7

Scalable Presentation of Performance Bottleneck Search Results

Paradyn's idiom of a hierarchy of application resources is being adopted by other automated performance diagnosis tools, and several tools now provide hierarchical visualizations of search-based performance diagnosis results similar to the Performance Consultant's Search History Graph display [77,96,103]. In the Performance Consultant's display, each node in the graph represents an experiment performed during a bottleneck search and indicates whether the experiment was true (i.e., the observed data for the experiment was above the threshold for its hypothesis), false, or is still unknown. Figure 7.1 shows a portion of a search history graph as it would be shown in the Performance Consultant's Search History Graph display. The figure focuses on the part of the graph that shows the qualitative behavior of four application processes, `myapp{1272}`, `myapp{1273}`, `myapp{7624}`, and `myapp{7625}` running on two hosts, `c33.cs.wisc.edu` and `c34.cs.wisc.edu`. However, because the Search History Graph display shows

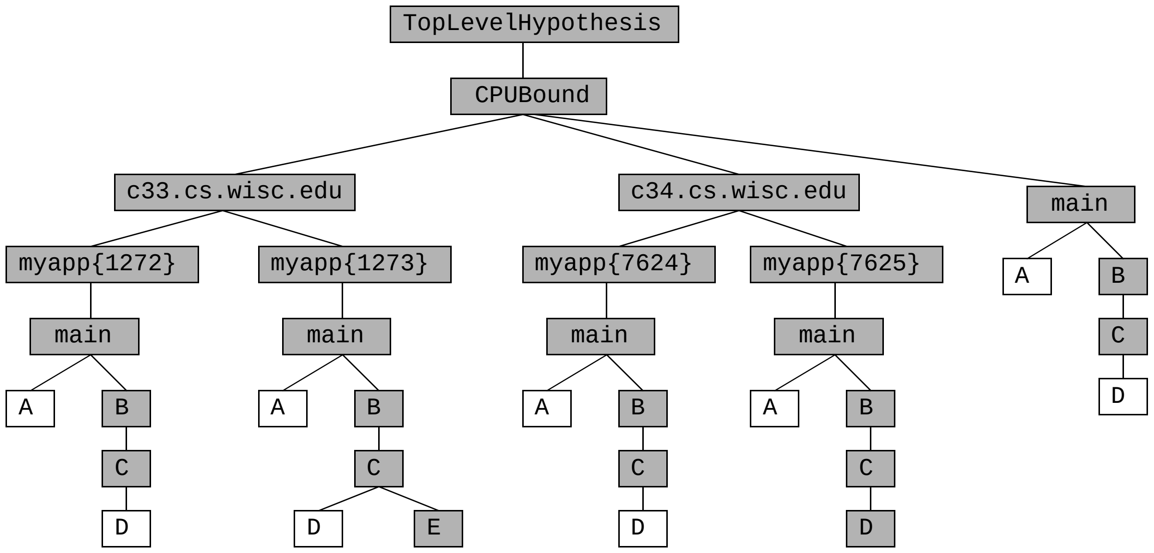


Figure 7.1 Part of a traditional Search History Graph display. The traditional display includes individual results for each local sub-search. True experiments are shown as shaded nodes.

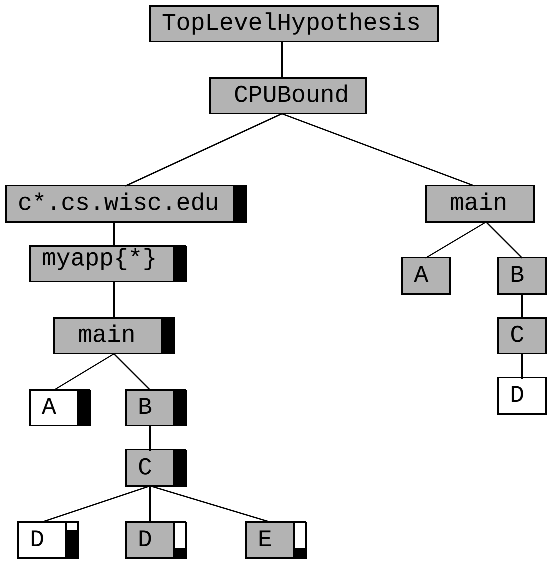


Figure 7.2 Example Search History Graph display after applying the SGFA. Nodes with "thermometer" gauges represent multiple experiments in the un-folded graph (Figure 7.1). Each bar indicates what fraction of the experiments its node represents. The labels "c*.cs.wisc.edu" and "myapp{*}" represent multiple host and process names in the unfolded graph. True experiments are shown as shaded nodes.

search results for individual application processes with one sub-graph per process, the display does not scale.

To address the problem of non-scalable presentation of bottleneck search results, we developed a new technique for producing scalable search result displays called the Sub-Graph Folding Algorithm (SGFA). Using ideas from scalable performance visualization [18,56], experiment management for performance tuning [53], and the PRISM parallel debugger [94], the SGFA combines sub-graphs based on the qualitative behavior of hosts and processes into a composite sub-graph. Sub-graphs indicating similar qualitative behavior are categorized together in the composite sub-graph. We expect the number of such behavioral categories to be small for most applications, allowing SGFA to produce search result displays that are substantially more scalable than the traditional Search History Graph display.

We describe the SGFA in Section 7.1. We present the results of our SGFA evaluation in Section 7.2. In experiments with 1024 application processes, our algorithm converted search history graphs with an average of 30309 nodes into graphs with an average of 44 nodes and a single composite sub-graph. We summarize our SGFA research and discuss future work in Section 7.3.

7.1 The Sub-Graph Folding Algorithm

The SGFA produces scalable displays of bottleneck search results by folding together sub-graphs that represent processes with similar qualitative behavior. The reduction in graph complexity is determined by the number of qualitative behavioral categories in the application (i.e., the number of radically different sub-graphs in the search history graph). Our SGFA approach is based on the hypothesis that the processes of a parallel application exhibit a small number of qualitative behaviors. In earlier work [84,88], we used this hypothesis in an approach for reducing the volume of performance data generated when monitoring a large number of application processes. We do not assume the application uses the same executable file for each of its processes (i.e., that it is an SPMD-style application).

Processes with similar qualitative behavior are represented in the traditional Search History Graph display as sub-graphs with similar shapes and node truth values. The SGFA incrementally produces a composite sub-graph from the similar sub-graphs of the original graph. As each node is added to the original graph, the SGFA traverses the node's sub-graph in the original graph and the composite sub-graph together. If an equivalent node is not already present in the composite sub-graph, the SGFA adds the sub-graph rooted at that node in the original sub-graph to the composite sub-graph. Figure 7.2 shows the result of applying the SGFA to the search history graph from Figure 7.1. In the folded graph, the SGFA has created a single composite sub-graph that represents all of the un-folded sub-graphs. After the sub-graphs for the first three processes (from left to right in Figure 7.1) are folded into the composite sub-graph, the composite sub-graph has a leaf node labelled "D" for a function resource with a truth value of "false." When SGFA folds the final sub-graph into the composite sub-graph, it determines that the node labelled "D" in the final sub-graph is not equivalent to the corresponding node in the composite sub-graph because they have different truth values. It adds another node labelled “D” with a truth value of “true” to the composite sub-graph.

When folding sub-graphs into a composite graph, SGFA must identify node equivalence. SGFA considers several node characteristics when determining whether two nodes are equivalent; the characteristics used depend on the types of the nodes under consideration. For some types of nodes, such as those labelled with host or process names, SGFA does not require node labels to be identical for the nodes to be considered equivalent. For example, when comparing the nodes labelled “`myapp{1272}`” and “`myapp{7624}`” in Figure 7.1, the SGFA uses executable name but disregards process ID values when comparing the nodes labelled “`myapp{1272}`” and “`myapp{7624}`” in Figure 7.1. On the other hand, there are node types whose labels must be identical for the nodes to be considered equivalent. This category of nodes includes nodes whose labels name resource categories (e.g., Message) or specific application functions. The SGFA always considers truth value when determining node equivalence.

Some nodes in a folded graph represent multiple experiments from the original graph. In our presentation approach, each node in a folded sub-graph is shown with a thermometer gauge to indicate the fraction of experiments it represents. For example, there are two nodes labelled “D” in Figure 7.2. The thermometer gauge on the node representing experiments with a “false” truth value is three-quarters shaded, indicating that the experiment on function D was false in three of the four sub-graphs in the original graph. When equivalent nodes from the original search have labels that are similar but not identical, SGFA uses wild card labels in the folded graph. For example, the nodes labelled with host names and process identifiers in Figure 7.1 are labelled “`cs*.wisc.edu`” and “`myapp{*}`”, respectively, in the folded graph. SGFA uses a string generalization algorithm like the longest common subsequence algorithm [46] to construct wild card node labels.

For a visualization technique to be truly scalable, it must have not only a scalable on-screen presentation but also a scalable approach for building the on-screen presentation. A centralized SGFA implementation is a poor match for presenting the results from the Distributed Performance Consultant because the centralized SGFA limits the scalability of the tool as a whole. A distributed SGFA approach is needed to complement the scalability benefit of the Distributed Performance Consultant.

For scalable construction of folded search history graphs, we designed and implemented an MRNet-based SGFA approach that uses custom MRNet data transformation filters (Chapter 4). A stateful SGFA filter running in the MRNet overlay network maintains a folded sub-graph containing results of the local search agents reachable by that MRNet process. When a filter’s folded graph changes, for example to add a node or to change a node’s truth state, the filter delivers a description of the change upstream. By induction, the filter running in the tool’s front-end has the entire folded graph.

7.2 Evaluation

To evaluate the SGFA, we implemented it in a custom MRNet data transformation filter and used it in our Distributed Performance Consultant scalability study (Section 5.4). We compared the complexity of the un-folded search history graph produced by the Distributed Performance Consultant to the graph produced by the SGFA. We used the number of nodes in each search history graph as the measure of its complexity.

The application we used for our study was `su3_rmd`, a quantum chromodynamics code produced by the MILC collaboration [73] for simulating pure lattice gauge theory. The code is written in C and uses MPI for communication. We used a weak scaling approach in our study, and all experiments were performed on the MCR Linux cluster [61] at Lawrence Livermore National Laboratory as part of the Distributed Performance Consultant scalability study (Section 5.4).

The results of our comparison are shown in Figure 7.3. The chart compares the complexity of the un-folded search history graphs produced by the Distributed Performance Consultant's truly distributed bottleneck search strategy with the complexity of the corresponding SGFA-produced graphs.

In our study, the number of nodes in the un-folded graph grew linearly with the number of application processes. This is expected because the un-folded search history graph includes a complete sub-graph for each application process and each sub-graph has approximately the same complexity. In contrast, the complexity of the SGFA-produced graphs remained nearly constant as we varied the number of application processes. Each of the SGFA graphs contained a single composite sub-graph.

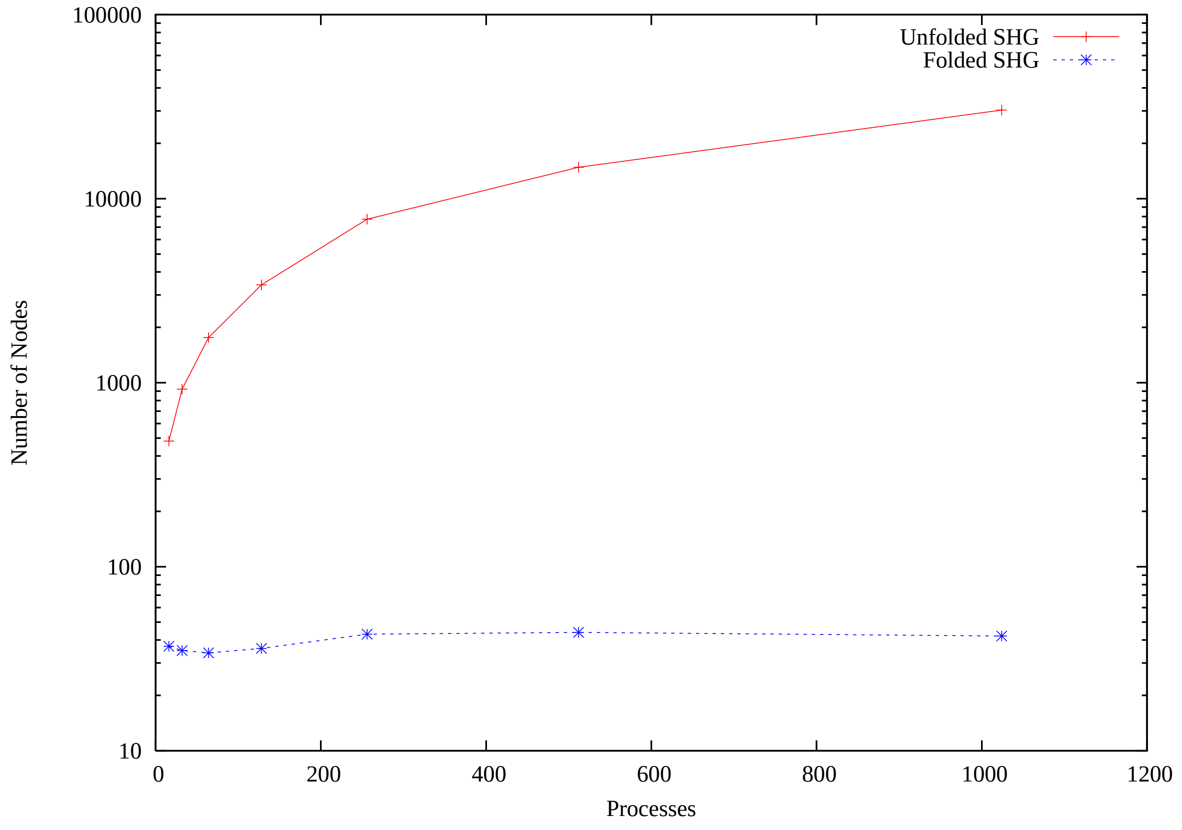


Figure 7.3 The effect of SGFA on search history graph complexity. The chart compares the number of nodes in the un-folded search history graph with the graph produced by the SGFA. The un-folded search history graph was produced using the Distributed Performance Consultant's truly distributed search strategy. Because the un-folded graph contains a results sub-graph for each application process, the number of nodes in the un-folded graph increases by approximately thirty nodes per process as the number of processes is increased. (The chart's y-axis is logarithmic.) The number of nodes in the folded graph remains roughly constant as the number of processes is varied.

7.3 Summary

The Sub-Graph Folding Algorithm produces scalable visualizations of bottleneck search results. The SGFA incrementally folds sub-graphs that show similar qualitative process behavior into a composite sub-graph. Because we expect the number of qualitative behavioral categories to be small, the SGFA approach can produce displays that scale better than the traditional Search History Graph display.

There are a few potential directions for future work based on our SGFA research. The first is to investigate the algorithm's sensitivity to the Performance Consultant's experiment thresholds. These thresholds affect the per-host and per-process sub-graph shape and node truth values, so they may have an effect on the graph complexity reduction possible with SGFA.

Another direction for further SGFA research is to study the computation and communication load in our initial MRNet-based SGFA approach to improve its scalability. For simplicity, our initial implementation transfers complete folded graphs between MRNet SGFA filters. When a filter receives a new folded graph from a downstream connection, the filter constructs its folded graph from scratch using the latest graphs from all downstream connections. The communication load of our approach may be reduced if a filter transfers only information about the graph changes to its upstream parent when the filter's folded graph changes. Also, our approach's computation load may be reduced if a filter, upon receiving graph change information from downstream, applies the changes to its own folded graph without necessarily re-folding its entire graph. Without a better understanding of the computation and communication behavior of our implementation, it is unclear whether these optimizations are necessary or how much scalability benefit they will provide.

Chapter 8

Conclusion

Our goal for this research was to enable performance diagnosis of applications with a large number of processes and a large number of functions. We have identified the barriers that keep existing performance diagnosis techniques from supporting such large-scale applications and devised techniques for overcoming the scalability barriers. In this chapter, we review the contributions of our research and discuss potential directions for future work.

8.1 Contributions

Our research shows that on-line, automated performance diagnosis tools can investigate the behavior of each application process for applications with thousands of processes and a large number of functions.

The challenge for our research was to overcome three barriers to tool scalability: the management of a potentially large volume of performance data, communication between a large number of distributed components, and presentation of performance diagnosis results for a large number of application components. The first two barriers comprise the problem of monitoring and controlling a large number of components.

Our approach for addressing this challenge had four parts:

- **Scalable parallel tool communication and data aggregation.** We presented the concept of Multicast/Reduction Overlay Networks (MRONs). We also introduced MRNet, our MRON implementation that uses a hierarchy of processes to provide scalable communication and data reduction services. We explored a wide variety of traditional and non-traditional data reductions when using MRNet in the Paradyn performance tool.
- **Finding performance problems in applications with a large number of processes.** We presented a new performance diagnosis approach using an on-line automated bottleneck search strategy that distributes decisions regarding local application behavior. To support the distributed bottleneck search that retains a centralized sub-search of global application behavior, the approach includes a model for tracking the cost of the search's instrumentation and a policy for scheduling instrumentation generated by the distributed components that implement a search. We also presented a truly distributed bottleneck search that eliminates the complexity of the augmented instrumentation cost model and instrumentation scheduling policy.
- **Finding performance problems in applications with a large number of functions.** Deep Start augments an instrumentation-based bottleneck search strategy with stack sample information gathered opportunistically during normal search activity. Deep Start finds bottlenecks more quickly than

search strategies guided only by the application's call graph, and can find bottlenecks hidden from more these more straightforward search strategies.

- **Presenting the results of bottleneck searches of applications with a large number of processes.** The Sub-Graph Folding Algorithm dynamically categorizes application processes by their qualitative behavior. For each category, the SGFA produces a composite sub-graph that indicates the behavior of all processes in the category. We expect most applications to exhibit a small number of behavioral categories, allowing the SGFA to produce displays that are more scalable than traditional search history graph displays that show search results for each application process.

Although our research was performed in the context of Paradyn and its Performance Consultant, with varying degrees of adaptation our techniques can be generalized for use in a broader class of parallel tools. MRNet is designed to enable scalable communication and data processing in all types of parallel tools. As automated search is adopted by other tools as a technique for finding application performance problems, the techniques used in the Distributed Performance Consultant, Deep Start, and the Sub-Graph Folding Algorithm can be used in this wider collection of tools.

8.2 Directions for Future Research

We presented several directions for future research in each of this dissertation's main chapters. We highlight two directions involving Multicast/Reduction Overlay Networks and the Distributed Performance Consultant that we feel are especially important.

MRONs and the MRON approach implemented in MRNet provide the basis for a few important lines of inquiry. First, although our MRNet work focused on scalability, reliability and resiliency are also important for tools that involve a large number of back-end processes. Research investigating these MRNet characteristics is currently in progress. Second, tailoring MRNet process layouts to the tool's requirements and the underlying system's capabilities remains largely unexplored. Open questions include how sensitive tools are to small differences in the overlay network's process layout, the use of unbalanced versus balanced tree layouts, and how to automatically tailor the process layout (perhaps by adapting to the tool's observed behavior). These MRNet-related research directions are especially important given MRNet's applicability to a broad class of parallel tools.

The Distributed Performance Consultant and SGFA also provide interesting directions for future research. The feasibility of using the SGFA to approximate an explicit investigation of global application behavior remains largely unexplored. Open questions include how sensitive the SGFA is to application characteristics such as its use of adaptive mesh refinement and how sensitive it is to the thresholds used during a Distributed Performance Consultant bottleneck search. Research along these directions is crucial for evaluating whether the SGFA's behavioral categories provide a sufficient approximation to the global bottleneck search results to support effective parallel application tuning.

References

[1] N.R. Adiga, G. Almasi, G.S. Almasi, Y. Aridor, R. Barik, et al (110 additional authors), "An Overview of the BlueGene/L Supercomputer", *SC 2002*, Baltimore, Maryland, November 2002.

[2] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model", *Journal of Parallel and Distributed Computing* **44**, 1, July 1997, pp. 71–79.

[3] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl, "Continuous Profiling: Where Have All the Cycles Gone?", *ACM Transactions on Computing Systems* **15**, 4, November 1997, pp. 357–390.

[4] T.E. Anderson and E.D. Lazowska, "Quartz: a Tool for Tuning Parallel Program Performance", *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, Colorado, May 1990, pp. 115–125.

[5] APART Working Group on Automatic Performance Analysis: Resources and Tools, http://www.gz-juelich.de/apart/, April 2004.

[6] R.H. Arpaci, A.C. Dusseau, A.M. Vahdat, L.T. Liu, T.E. Anderson, and D.A. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations", *1995 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1995)*, Ottawa, Canada, May 1995, pp. 267–278.

[7] A.C. Arpaci-Dusseau, "Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems", *ACM Transactions on Computer Systems* **19**, 3, August 2001, pp. 283–331.

[8] S.M. Balle, Personal communication, November 2002–November 2003.

[9] S.M. Balle, B.R. Brett, C.-P. Chen, and D. LaFrance-Linden, "A New Approach to Parallel Debugger Architecture", *Sixth International Conference PARA 2002*, Espoo, Finland, June 2002. Published as *Lecture Notes in Computer Science* **2367**, J. Fagerholm, J. Haataja, J. Järvinen, M. Lyly, and P. Raback (Eds), Springer-Verlag, Heidelberg, June 2002, pp. 139–149.

[10] S. Benkner, “VFC: the Vienna Fortran Compiler”, *Scientific Computing* **7**, 1, 1999, pp. 67–81.

[11] P. Berman and J. Garay, “Cloture Voting: (n=4)-Resilient Distributed Consensus in t + 1 Rounds”, *Mathematical Systems Theory 26*, 1, 1993, pp. 3–20.

[12] M. Bernaschi and G. Iannello, “Collective Communication Operations: Experimental Results vs. Theory”, *Concurrency: Practice and Experience* **10**, 5, April 1998, pp. 359–386.

[13] R. Brightwell, L.A. Fisk, D.S. Greenberg, T. Hudson, M. Levenhagen, A.B. Maccabe, and R. Riesen, “Massively Parallel Computing Using Commodity Components”, *Parallel Computing* **26**, 2–3, February 2000, pp. 243–266.

[14] P.N. Brown, R.D. Falgout, and J.E. Jones, “Semicoarsening Multigrid on Distributed Memory Machines”, *SIAM Journal on Scientific Computing* **21**, 5, 2000, pp. 1823–1834.

[15] H.W. Cain, B.P. Miller, and B.J.N. Wylie, “A Callgraph-Based Search Strategy for Automated Performance Diagnosis”, *Sixth International European Conference on Parallel Computing (Euro-Par 2000)*, Münich, Germany, August 2000. Published as *Lecture Notes in Computer Science* **1900**, A. Bode, T. Ludwig, W. Karl, R. Wismüller (Eds), Springer-Verlag, Heidelberg, August/September 2000, pp. 108–122.

[16] B.A. Coan and J.L. Welch, “Modular Construction of an Efficient 1-Bit Byzantine Agreement Protocol”, *Mathematical Systems Theory* **26**, 1, 1993, pp. 131–154.

[17] Compaq Corporation, “21264/EV68A Microprocessor Hardware Reference Manual”, Part Number DS-0038A-TE, 2000.

[18] A.L. Couch, “Categories and Context in Scalable Execution Visualization”, *Journal of Parallel and Distributed Computing* **18**, 2, June 1993, pp. 195–204.

[19] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, E. Santos, K.E. Schauser, R. Subramonian, and T. von Eicken, “LogP: A Practical Model of Parallel Computation”, *Communications of the ACM* **39**, 11, November 1996, pp. 78–85.

[20] J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, G. Chrysos, “ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors”, *30th Annual IEEE/ACM International Symposium on Microarchi-*

tecture (MICRO-30), Research Triangle Park, North Carolina, December 1997, pp. 292–302.

[21] S.E. Deering and D.R. Cheriton, "Multicast Routing in Datagram Internetworks and Extended LANs", *IEEE/ACM Transactions on Networking* **8**, 2, May 1990, pp. 85–110.

[22] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems", *IEEE Transactions on Software Engineering* **12**, 5, May 1986, pp. 662–675.

[23] G. Eisenhauer and L.K. Daley, "Fast Heterogeneous Binary Data Interchange", *Ninth Heterogeneous Computing Workshop (HCW 2000)*, Cancun, Mexico, May 2000, pp. 90–101.

[24] A. Espinosa, T. Margalef, and E. Luque, "Automatic Performance Evaluation of Parallel Programs", *Sixth Euromicro Workshop on Parallel and Distributed Processing (PDP '98)*, Madrid, Spain, January 1998, pp. 43–49.

[25] Etnus, Inc., "TotalView", http://www.etnus.com/Products/TotalView/, 2004.

[26] D.A. Evensky, Personal communication, November 2001.

[27] D.A. Evensky, A.C. Gentile, L.J. Camp, and R.C. Armstrong, "Lilith: Scalable Execution of User Code for Distributed Computing", *Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC '97)*, Portland, Oregon, August 1997, pp. 306–314.

[28] T. Fahringer, M. Gerndt, G. Riley, and J.L. Träff, "Specification of Performance Problems in MPI Programs with ASL", *2000 International Conference on Parallel Processing (ICPP'00)*, Toronto, Canada, August 2000, pp.51–58.

[29] D.G. Feitelson, "Job Scheduling in Multiprogrammed Parallel Systems", IBM Research Technical Report RC 19790 (87657), second revision, August 1997.

[30] I. Foster, C. Kesselman (Eds.), "The Grid: Blueprint for a New Computing Infrastructure", (2nd Ed.) Morgan Kaufmann, San Francisco, November 2003.

[31] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke, "The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration", Draft, Global Grid Forum Open Grid Services Architecture Working

Group, May 2003. Available from https://forge.gridforum.org/projects/ogsa-wg/document/The_Physiology_of_the_Grid/en/1/.

[32] J.E. Garlick and C.M. Dunlap, "Building CHAOS: an Operating Environment for Livermore Linux Clusters", Lawrence Livermore National Laboratory Technical Report UCRL-ID-151968, February 2002.

[33] A.S. Grimshaw, A.J. Ferrari, G. Lindahl, and K. Holcomb, "Wide-Area Computing: Resource Sharing on a Large Scale", *IEEE Computer* **32**, 5, May 1999, pp. 29–37.

[34] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam, "PVM: Parallel Virtual Machine—A User's Guide and Tutorial for Network Parallel Computing", MIT Press, 1994.

[35] H.M. Gerndt and A. Krumme, "A Rule-Based Approach for Automatic Bottleneck Detection in Programs with Shared Virtual Memory Systems", *Second International Workshop on High-Level Programming Models and Supportive Environments*, Geneva, Switzerland, April 1997.

[36] M .Gerndt, A. Schmidt, M. Schulz, and R. Wismüller, "Performance Analysis for Teraflop Computers: a Distributed Automatic Approach", *Tenth Euromicro Workshop on Parallel, Distributed, and Network-based Processing (PDP 2002)*, Canary Islands, Spain, January 2002, pp. 23–30.

[37] Global Grid Forum Open Grid Services Architecture Working Group, "Open Grid Services Architecture", https://forge.gridforum.org/-projects/ogsa-wg/, February 2004.

[38] S.L. Graham, P.B. Kessler, and M.K. McKusick, "Gprof: a Call Graph Execution Profiler", *SIGPLAN 1982 Symposium on Compiler Construction*, Boston, Massachusetts, June 1982. Published as *SIGPLAN Notices* **17**, 6, ACM Press, June 1982, pp. 120–126.

[39] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys* **25**, 2, June 1993, pp. 73–170.

[40] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data Cube: a Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals", *Data Mining and Knowledge Discovery* **1**, 1, April 1997, pp. 29–53.

[41] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", Argonne National Laboratory Report MCS-P567-0296, February 1996.

[42] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter, “Falcon: On-line Monitoring for Steering Parallel Programs”, *Concurrency: Practice and Experience* **10**, 9, August 1998, pp. 699–736.

[43] R.J. Hall, “Call Path Refinement Profiles”, *IEEE Transactions on Software Engineering* **21**, 6, June 1995, pp. 481–496.

[44] M.T. Heath and J.A. Etheridge, “Visualizing the Performance of Parallel Programs”, *IEEE Software* **8**, 5, September 1991, pp. 29–39.

[45] B.R. Helm, A.D. Malony, and S.F. Fickas, “Capturing and Automating Performance Diagnosis: the Poirot Approach”, *1995 International Parallel Processing Symposium (IPPS ’95)*, Santa Barbara, California, April 1995, pp. 606–613.

[46] D.S. Hirschberg, “Algorithms for the Longest Common Subsequence Problem”, *Journal of the ACM* **24**, 4, October 1977, pp. 664–675.

[47] J.K. Hollingsworth and B.P. Miller, “An Adaptive Cost Model for Parallel Program Instrumentation”, *Second International European Conference on Parallel Computing (Euro-Par ’96)*, Lyon, France, August 1996. Published as *Lecture Notes in Computer Science* **1123**, L Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Springer, pp. 88–97.

[48] J.K. Hollingsworth, R.B. Irvin, and B.P. Miller, “The Integration of Application and System Based Metrics in a Parallel Program Performance Tool”, *Third ACM SIGPLAN Symposium on Principals & Practice of Parallel Programming (PPoPP 1991)*, Williamsburg, Virginia, April 1991, pp. 189–200.

[49] J.K. Hollingsworth, B.P. Miller, and J. Cargille, “Dynamic Program Instrumentation for Scalable Parallel Tools”, *1994 Scalable High Performance Computing Conference (SHPCC ’94)*, Knoxville, Tennessee, pp. 841–850, May 1994.

[50] J.K. Hollingsworth, B.P. Miller, M.J.R. Goncalves, O. Naim, Z. Xu, and L. Zheng, “MDL: A Language and Compiler for Dynamic Program Instrumentation”, *International Conference on Parallel Architectures and Compilation Techniques (PACT’97)*, San Francisco, California, November 1997, pp. 201–213.

[51] R. Hood, “The p2d2 Project: Building a Portable Distributed Debugger”, *1996 SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT ’96)*, Philadelphia, Pennsylvania, May 1996, pp. 127–136.

[52] International Business Machines Corp., “Parallel Environment for AIX: Operations and Use, Volume 1”, Document number SA22-7425-01, December 2001.

[53] K.L. Karavanic and B.P. Miller, “Experiment Management Support for Performance Tuning”*, SC99*, Portland, Oregon, November 1999.

[54] R.E. Kessler, M.D. Hill, and D.A. Wood, “A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches”, *IEEE Transactions on Computers* **43**, 6, June 1994, pp. 664–675.

[55] T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, and R.A.F. Bhoedjang, “MagPIe: MPI’s Collective Communication Operations for Clustered Wide Area Systems”*, Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, Georgia, May 1999, pp. 131–140.

[56] D. Kimelman, B. Leban, T. Roth, and D. Zernik, “Reduction of Visual Complexity in Dynamic Graphs”, *Graph Drawing ’94: DIMACS International Workshop (GD ’94)*, Princeton, New Jersey, October 1994. Published as *Lecture Notes in Computer Science* **894**, R. Tamassia and I.G. Tollis (Eds.), Springer-Verlag, Heidelberg, 1994, pp. 218–225.

[57] J. Kohn and W. Williams, “ATExpert”, *Journal of Parallel and Distributed Computing* **18**, 2, June 1993, pp. 205–222.

[58] L. Lamport, R. Shostak, and M. Pease, “The Byzantine Generals Problem”, *ACM Transactions on Programming Languages and Systems* **4**, 3, July 1982, pp. 382–401.

[59] Lawrence Livermore National Laboratory, “ASCI Blue Pacific”, http://www.llnl.gov/asci/platforms/bluepac/, February 2003.

[60] Lawrence Livermore National Laboratory, “ASCI Purple”, http://www.llnl.gov/asci/purple/, May 2004.

[61] Lawrence Livermore National Laboratory, “M&IC Capability Cluster”, http://www.llnl.gov/linux/mcr/, April 2005.

[62] S.T. Leutenegger and M.K. Vernon, “The Performance of Multiprogrammed Multiprocessor Scheduling Policies”, *1990 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1990)*, Boulder, Colorado, May 1990, pp. 226–236.

[63] Los Alamos National Laboratory, "The Pink Page", http://www.lanl.gov/projects/pink/, 2004.

[64] N.A. Lynch, M. Fischer, and R.J. Fowler, "A Simple and Efficient Byzantine Generals Algorithm", *Second Symposium on Reliability in Distributed Software and Database Systems*, July 1982, pp.46–52.

[65] B. Lyon, "Sun External Data Representation Specification", Sun Microsystems, Inc. Technical Report, 1985.

[66] S. Madden, M.J. Franklin, J.M Hellerstein, and W. Hong, "TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks. *Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, Massachusetts, December, 2002.

[67] A.D. Malony, D.H. Hammerslag, and D.J. Jablonowski, "Traceview: a Trace Visualization Tool", *IEEE Software* **8**, 5, September 1991, pp. 19–28.

[68] A.D. Malony and D.A. Reed, "Models for Performance Perturbation Analysis", *ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, May 1991, pp. 15–25.

[69] M.L. Massie, B.N. Chun, and D.E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience", University of California, Berkeley Technical Report, http://ganglia.sourceforge.net/talks/parallel_computing/ganglia-twocol.pdf, February 2003.

[70] W. Meira Jr., T.J. LeBlanc, and A. Poulos, "Waiting Time Analysis and Performance Visualization in Carnival", *SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '96)*, Philadelphia, Pennsylvania, May 1996, pp. 1–10.

[71] Message Passing Interface Forum, "MPI: a Message Passing Interface Standard", *International Journal of Supercomputing Applications* **8**, 3/4, Fall/Winter 1994.

[72] Message Passing Interface Forum, "MPI-2: A Message-Passing Interface Standard", *International Journal of Supercomputer Applications and High Performance Computing* **12**, 1/2, 1998.

[73] The MIMD Lattice Computation (MILC) Collaboration, http://physics.indiana.edu/~sg/milc.html.

[74] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Paral-

lel Performance Measurement Tool", *IEEE Computer*, **28**, 11, November 1995, pp. 37–46.

[75] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski, "IPS-2: the Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Computing* **1**, 2, April 1990, pp. 206–217.

[76] B. Mohr, A.D. Malony, and J.E. Cuny, "TAU: Tuning and Analysis Utilities for Portable Parallel Programming", In *Parallel Programming using C++*, G. Wilson, Ed., MIT Press, Cambridge, Massachusetts, 1996.

[77] B. Mohr and F. Wolf, "KOJAK: A Tool Set for Automatic Performance Analysis of Parallel Applications", *Ninth International Euro-Par Conference (Euro-Par 2003)*, Klagenfurt, Austria, August 2003, published as *Lecture Notes in Computer Science* **2790**, H. Kosch, L. Böszörményi, and H. Hellwagner (Eds.), Springer-Verlag, Heidelberg, pp. 1301–1304.

[78] A. Morajko, "Dynamic Tuning of Parallel/Distributed Applications", Doctoral dissertation, Universitat Autonoma de Barcelona, Barcelona, Spain, December 2003.

[79] N. Mukherjee, G.D. Riley, and J.R. Gurd, "FINESSE: a Prototype Feedback-Guided Performance Enhancement System", *Eighth Euromicro Workshop on Parallel and Distributed Processing (PDP 2000)*, Rhodes, Greece, January 2000.

[80] W.E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and Analysis of MPI Resources", *Supercomputer 63* **12**, 1, 1996, pp. 69–80.

[81] F. Petrini, D.J. Kerbyson, and S. Pakin, "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q", *SC2003*, Phoenix, Arizona, November 2003.

[82] R.S. Pindyck and D.L. Rubinfeld, "Microeconomics", Prentice Hall, Upper Saddle River, New Jersey, 2000.

[83] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B.W. Schwartz, and L.F. Tavera, "Scalable Performance Analysis: the Pablo Performance Analysis Environment", *Scalable Parallel Libraries Conference (SPLC 1993)*, Mississippi State, Mississippi, October 1993, pp. 104–113.

[84] D.A. Reed, O.Y. Nickolayev, and P.C. Roth, "Real-Time Statistical Clustering for Event Trace Reduction", *International Journal of Supercomputing*

Applications and High-Performance Computing **11**, 2, Summer 1997, pp. 144–159.

[85] R.L. Ribler, H. Simitci, and D.A. Reed, "The Autopilot Performance-Directed Adaptive Control System", *Future Generation Computer Systems* **18**, 1, September 2001, pp. 175–187.

[86] E. Rich and K. Knight, "Artificial intelligence", McGraw-Hill, New York, 1991.

[87] R. Riesen, R. Brightwell, L.A. Fisk, T. Hudson, J. Otto, and A.B. Maccabe, "Cplant", *Second Extreme Linux Workshop* at the *1999 USENIX Annual Technical Conference*, Monterrey, California, June 1999.

[88] P.C. Roth, "ETRUSCA: Event Trace Reduction Using Statistical Data Clustering Analysis", Master's Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1996.

[89] P.C. Roth, D.C. Arnold, and B.P. Miller, "MRNet: a Software-Based Multicast/Reduction Network for Scalable Tools", *SC 2003*, Phoenix, Arizona, November 2003.

[90] P.C. Roth and B.P. Miller, "Deep Start: a Hybrid Strategy for Automated Performance Problem Searches", *Concurrency and Computation: Practice and Experience* **15**, 11–12, September 2003, pp. 1027–1046. Also appeared in shorter form in *Eighth International Euro-Par Conference (Euro-Par 2002)*, Paderborn, Germany, August 2002, published as *Lecture Notes in Computer Science* **2400**, B. Monien and R. Feldmann (Eds.), Springer-Verlag, Heidelberg, pp. 86–96.

[91] A. Shatdal and J.F. Naughton, "Adaptive Parallel Aggregation Algorithms", *ACM SIGMOD Record* **24**, 2, May 1995, pp. 104–114.

[92] T. Sheehan, A. Malony, and S. Shende, "A Runtime Monitoring Framework for the TAU Profiling System", *Third International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'99)*. Published as *Lecture Notes in Computer Science* **1732**, Springer-Verlag, Heidelberg, December 1999, pp. 170–181.

[93] N.G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems", *IEEE Computer* **25**, 12, December 1992, pp. 33–44.

[94] S. Sistare, D. Allen, R. Bowker, K. Jourenais, J. Simons, and R. Title, "A Scalable Debugger for Massively Parallel Message-Passing Programs", *IEEE Parallel & Distributed Technology* **1**, 2, Summer 1994, pp. 50–56.

[95] S. Sistare, E. Dorenkamp, N. Nevin, and E. Loh, “MPI Support in the Prism Programming Environment”, *1999 ACM/IEEE Conference on Supercomputing (SC1999)*, Portland, Oregon, November 1999.

[96] F. Song and F. Wolf, “CUBE User Manual”, ICL Technical Report ICL-UT-04-01, University of Tennessee, February 2004.

[97] M.J. Sottile and R.G. Minnich, “Supermon: a High-Speed Cluster Monitoring System”, *Cluster 2002*, Chicago, Illinois, September 2002.

[98] J.T. Stasko and J. Muthukumarasamy, “Visualizing Program Executions on Large Data Sets”, *1996 IEEE Symposium on Visual Languages*, Boulder, Colorado, September 1996, pp. 166–173.

[99] T.L. Sterling, J. Salmon, D.J. Becker, and D.F. Savarese, “How to Build a Beowulf: a Guide to the Implementation and Application of PC Clusters”, MIT Press, Cambridge, Massachusetts, 1999.

[100] C. Tapus, I-H. Chung, and J.K. Hollingsworth, “Active Harmony: Towards Automated Performance Tuning”, *SC 2002*, Baltimore, Maryland, November 2002.

[101] K. Tani, “Status of the Earth Simulator System”, *16th International Supercomputer Conference*, Heidelberg, Germany, June 2001.

[102] B. Tierney, R. Aydt, D. Gunter, W. Smith, V. Taylor, R. Wolski, and M. Swany, “A Grid Monitoring Architecture”, Technical report GWD-Perf-16-3, Global Grid Forum, August 2002. Available from http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-Perf-16-3.pdf.

[103] H.-L. Truong and T. Fahringer, “SCALEA: a Performance Analysis Tool for Parallel Programs”, *Concurrency and Computation: Practice and Experience*, **15**, 11–12, September 2003, pp. 1001–1025. Also appeared in shorter form in *Eighth International Euro-Par Conference (Euro-Par 2002)*, Paderborn, Germany, August 2002, published as *Lecture Notes in Computer Science* **2400**, B. Monien and R. Feldmann (Eds.), Springer-Verlag, Heidelberg, pp. 75–85.

[104] H.-L. Truong and T. Fahringer, “SCALEA-G: a Unified Monitoring and Performance Analysis System for the Grid”, *Second European Across Grids Conference*, Nicosia, Cyprus, January 2004.

[105] S.S. Vadhiyar, G.E. Fagg, and J.J. Dongarra, “Automatically Tuned Collective Communications”, *SC 2000*, Dallas, Texas, November 2000.

[106] A. Waheed, D.T. Rover, and J.K. Hollingsworth, "Modeling and Evaluating Design Alternatives for an On-Line Instrumentation System: A Case Study", *IEEE Transactions on Software Engineering* **24**, 6, June 1998, pp. 451–470.

[107] W.-H. Wang and J.-L. Baer, "Efficient Trace-Driven Simulation Methods for Cache Performance Analysis", *ACM Transactions on Computer Systems* **9**, 3, August 1991, pp.222–241, August 1991.

[108] W. Williams, T. Hoel, and D. Pase, "The MPP Apprentice™ Performance Tool: Delivering the Performance of the Cray T3D®", *IFIP WG10.3 Working Conference on Programming Environments for Massively Parallel Distributed Systems,* Monte Verità, Switzerland, April 1994, pp. 333–345. Published as *Programming Environments for Massively Parallel Distributed Systems*, K.M. Decker and R.M. Rehmann (Eds.), Birkhäuser Verlag, 1994.

[109] J.C. Yan and S. Listgarten, "Intrusion Compensation for Performance Evaluation of Parallel Programs on a Multicomputer", *Sixth International Conference on Parallel and Distributed Computing Systems (ISCA 1993)*, Louisville, Kentucky, October 1993, pp. 427–431.

[110] J.C. Yan and S.R. Sarukkai, "Analyzing Parallel Performance Using Normalized Performance Indices and Trace Transformation Techniques", *Parallel Computing* **22**, 9, November 1996, pp. 1215–1237.

[111] J. Yan, S. Sarukkai, and P. Mehra, "Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs Using the AIMS Toolkit. *Software: Practice and Experience* **25**, 4, April 1995, pp. 429–461.

[112] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward Scalable Performance Visualization with Jumpshot", *High Performance Computing Applications* **13**, 2, Fall 1999, pp. 277–288.