On-line Automated Performance Diagnosis on Thousands of Processes

Philip C. Roth[†]
Computer Science and Mathematics Division
Oak Ridge National Laboratory
One Bethel Valley Rd.
Oak Ridge, TN 37831-6173 USA
rothpc@ornl.gov

Barton P. Miller
Computer Sciences Department
University of Wisconsin, Madison
1210 W. Dayton St.
Madison, WI 53706-1685 USA
bart@cs.wisc.edu

Abstract

Performance analysis tools are critical for the effective use of large parallel computing resources, but existing tools have failed to address three problems that limit their scalability: (1) management and processing of the volume of performance data generated when monitoring a large number of application processes, (2) communication between a large number of tool components, and (3) presentation of performance data and analysis results for applications with a large number of processes. In this paper, we present a novel approach for finding performance problems in applications with a large number of processes that leverages our multicast and data aggregation infrastructure to address these three performance tool scalability barriers.

First, we show how to design a scalable, distributed performance diagnosis facility. We demonstrate this design with an on-line, automated strategy for finding performance bottlenecks. Our strategy uses distributed, independent bottleneck search agents located in the tool agent processes that monitor running application processes. Second, we present a technique for constructing compact displays of the results of our bottleneck detection strategy. This technique, called the Sub-Graph Folding Algorithm, presents bottleneck search results using dynamic graphs that record the refinement of a bottleneck search. The complexity of the results graph is controlled by combining sub-graphs showing similar local application behavior into a composite sub-graph.

Using an approach that combines these two synergistic parts, we performed bottleneck searches on programs with up to 1024 processes with no sign of tool resource saturation. With 1024 application processes, our visualization technique reduced a search results graph containing over 30,000 nodes to a single composite 44-node graph sub-graph showing the same qualitative performance information as the original graph.

Categories and Subject Descriptors C.4 [Performance of Systems]: Measurement techniques, Performance attributes.

General Terms Performance, Measurement.

Keywords Scalability, performance diagnosis, tools, automation, Paradyn.

1 Introduction

Computational science has become a primary approach for addressing open questions in areas like biology, nuclear physics, and climate studies. The desire to address such problems at high fidelity has driven the increasing deployment of high-end parallel systems and clusters with thousands or even hundreds of thousands of processors. However, achieving the computational potential of these large-scale systems has proven difficult. Applications that achieve more than a small percentage of a system's peak performance are the exception, not the rule.

Tuning the performance of a parallel program to make better use of a large-scale system requires a thorough understanding of the program's behavior. Tools for identifying and diagnosing performance problems are critical for understanding program behavior on large-scale systems, but such tools must be scalable to be effective. We have identified three barriers that keep performance tools from scaling:

- management of large-volume performance data flows generated when monitoring a large number of processes;
- efficient control of a large number of tool agent processes; and
- presentation of performance diagnosis results for a large number of processes.

In previous work [27], we partially addressed these scalability barriers with infrastructure called MRNet that uses an overlay network of processes to provide multicast and data aggregation services for tools. Because it provides a scalable mechanism for obtaining *global* application performance data (data describing the behavior of all application processes) such as average CPU utilization across all processes, this infrastructure allowed us to partially

[†]Significant parts of this work were undertaken while the author was a student in the Computer Sciences Department at the University of Wisconsin, Madison.

This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, Lawrence Livermore National Lab grant B504964, and NSF grants CDA-9623632 and EIA-9870684. This research was also sponsored in part by the Office of Mathematical, information, and Computational Sciences, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

address the first scalability barrier. The infrastructure's multicast functionality enables us to address the second scalability barrier by providing a scalable mechanism for communication between a large number of tool components. Nevertheless, because a tool must consider *local* application behavior (the behavior of individual application processes) in addition to global behavior, and because the data describing local behavior is not aggregated before it is analyzed, our MRNet approach alone is insufficient to address the three scalability barriers.

To address all three scalability barriers, we propose a novel performance diagnosis approach for finding performance problems in applications with a large number of processes. Our approach has two synergistic parts. The first part is a scalable and distributed performance diagnosis facility. We demonstrate this design in the form of an on-line, automated strategy for finding performance bottlenecks. The strategy uses *search*, a well-established technique for methodically examining a solution space. The second part is a technique for constructing compact displays of the results of our search-based bottleneck detection strategy. In this technique, called the Sub-Graph Folding Algorithm (SGFA), bottleneck search results are presented using dynamic graphs that record the refinement of a bottleneck search. To control the complexity of the results graph, sub-graphs showing similar local application behavior are combined into a composite sub-graph. In effect, the SGFA dynamically clusters application processes based on their qualitative behavior. Both parts of our performance diagnosis approach rely on a multicast and data aggregation infrastructure like MRNet for scalable communication and data processing.

We focus on *on-line automated* performance bottleneck detection techniques (those that perform the bulk of their analysis while the application runs, with minimal user involvement) because we believe they are highly effective for large-scale performance tuning. First, they relieve the user from the difficult task of identifying which performance data is important. Because the data volume generated when monitoring large-scale applications can be massive, the 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. The user need not be a performance tuning expert to use an on-line automated tool effectively because the expertise is built into the tool. Finally, they can adapt their activity in response to the application's behavior as it runs. This ability allows on-line automated techniques to adapt the data being collected to obtain useful results from only a single application run [22] and to collect and process a smaller volume of performance data than tools that cannot adapt the data they collect during a run.

The overall contribution of this research is in the area of performance tool scalability. More specifically, the contributions of this work are:

- the design and evaluation of a new, distributed search strategy for finding performance bottlenecks in applications with a large number of processes;
- a new model for expressing the cost of instrumentation in parallel computation, designed for use with our distributed bottleneck search strategy;
- a new approach for making concise graphical presentations of the results of a bottleneck search (i.e., the SGFA); and
- the recognition that the synergy between a distributed bottleneck search strategy and the SGFA allows us to avoid explicit examination of the application's global behavior.

To evaluate our new approach to automated performance diagnosis, we performed a scalability study using a prototype implementation of our performance diagnosis approach to find bottlenecks in a quantum chromodynamics (QCD) application running on a Linux cluster. Because the goal of our work is to improve the scalability of performance tools as opposed to parallel applications, we measured characteristics of the performance tool that indicate a scalability bottleneck within the tool system itself. In particular, for each process in the prototype tool implementation, we measured the tool process' CPU utilization and I/O behavior, looking for signs of resource saturation caused by the tool processes. In experiments with up to 1024 application processes, we performed bottleneck searches of the QCD application with no sign of tool resource saturation. We describe these results in more detail in Section 4.1.

During our study, we also considered the scalability of our new performance bottleneck search results visualization technique, the SGFA. We quantified the scalability of the SGFA by considering the number of nodes in a bottleneck search results graph as a measure of its complexity, and by assuming that it is preferable to minimize the number of graph nodes. In the most demanding scenario with 1024 application processes, the SGFA reduced a bottleneck search results graph containing over 30,000 nodes to a 44-node graph containing a single composite sub-graph showing the same qualitative performance diagnosis information as the original graph. The complete results of this part of our evaluation are provided in Section 4.2.

In the next two sections, we detail our performance diagnosis approach for programs with a large number of processes. In Section 2 we present our design for a scalable data analysis and performance diagnosis facility. In Section 3 we describe the Sub-Graph Folding Algorithm. We present the results of the evaluation of our approach in Section 4, and conclude in Section 5 with a discussion of previous work related to our research.

2 Scalable Bottleneck Detection

The first part of our scalable performance diagnosis approach is a distributed data analysis and performance diagnosis facility, embodied in the design of an on-line, automated strategy for finding performance bottlenecks. The strategy addresses the problem of efficiently examining both the local and global behavior of applications with a large number of processes.

Our bottleneck search strategy adopts the search refinement rules of the Performance Consultant [5,16,22], Paradyn's automated performance diagnosis component. Like the Performance Consultant, our bottleneck search strategy performs *experiments* that determine whether an application is exhibiting performance problems and, if so, where the problem is occurring. Each experiment consists of a *hypothesis*, a reason why the application may be exhibiting a performance problem, and a *focus* that indicates where in the application the hypothesis will be tested. The Performance Consultant uses a small number of built-in hypotheses such as "too much time spent blocked for I/O" and "CPU bound." Using a hardware counter access facility like the Performance Application Programming Interface (PAPI) [4], the Performance Consultant can also create hypotheses based around hardware counters such as L2 cache misses. An experiment's focus is constructed using program

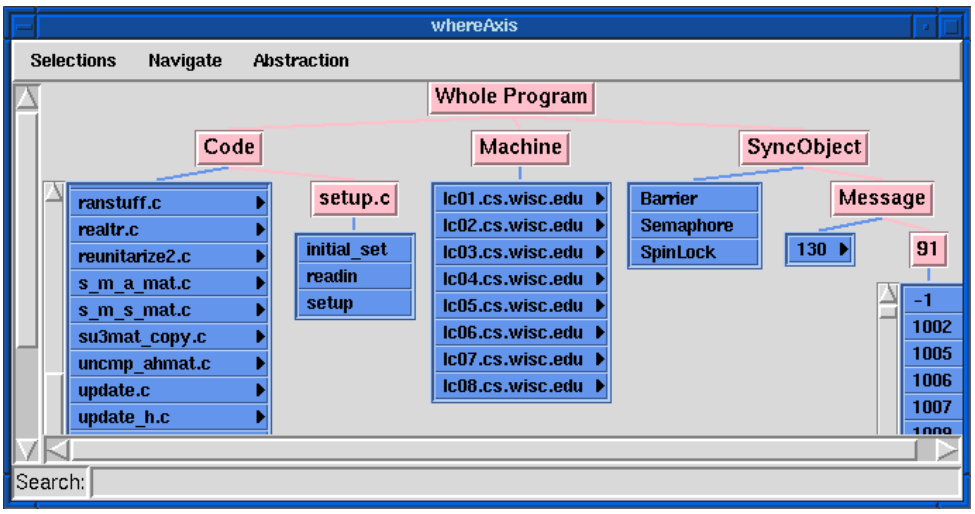


Figure 1: Program resources arranged into hierarchies.

resources that name the static and dynamic entities that comprise the program. The functions that constitute the program'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. Program resources are arranged in hierarchies as shown in Figure 1. In a resource hierarchy, each resource represents all resources organized beneath it within its hierarchy. For example, the `Code` resource represents all functions in the application's executable. An experiment's focus is a tuple is a tuple containing one resource from each resource hierarchy, providing a concise name for a potentially large set of resources. For example, one experiment performed during our scalability study with the QCD application had the focus `</Code/control.c/main,/Machine, /SyncObject>`. Because this focus specifies the top level `Machine` resource, the focus does not constrain the set of resources it names to any particular host—it names the collection of main functions from all application processes. Another experiment used the focus `</Code/update_h.c/update_h,/Machine/mcr498/-su3_rmd{2473},/SyncObject>`, which denotes only the `update_h` function from process 2473 on host `mcr498` because it names specific resources from both the `Code` and `Machine` resource hierarchies.

Dynamic instrumentation [15] is used to collect performance data to test an experiment's hypothesis at the program locations named by the experiment's focus. If the performance data collected for an experiment is above the user-configurable threshold associated with the experiment's hypothesis, the search is *refined* to create more specific experiments. A bottleneck search begins with experiments that test general performance problems, such as whether the entire application is spending too much time waiting for communication operations. With successive search refinements, the search pinpoints the nature and location of application bottlenecks. The Performance Consultant can refine its search to the level of individual functions and loops.

Because the Performance Consultant inserts instrumentation into application processes as they run, each process' behavior is altered slightly from what it would have been if it were uninstrumented. With a user-configurable threshold for the cost of instrumentation, the Performance Consultant can limit the effect of its instrumentation on the application's behavior. For sensitive applications, the search can be made less aggressive to reduce the tool-induced load and perturbation on system resources. Our new performance diagnosis approach improves on our ability to control instrumentation overhead with our new instrumentation cost model (described in Section 2.1). Furthermore, our distributed bottleneck search strategies add little new perturbation on the system nodes, as demonstrated by our evaluation results (Section 4.1).

The results of a performance bottleneck search are shown effectively in a search history graph (Figure 2) that records the cumulative refinements of a search. Figure 2 focuses on the part of the graph that shows the qualitative CPU utilization behavior of several application processes. Each node in the graph represents an experiment. The node's background color indicates whether the experiment is true (blue, or dark gray in black-and-white), false (pink, or light gray), or not yet known (none shown in Figure 2). The color of a node's label indicates whether the experiment's dynamic instrumentation has been inserted (white) or is not currently active (black). At the root of the graph is a node labeled "`TopLevelHypothesis`" representing a virtual experiment that tests whether there are any performance problems in the application as a whole. This virtual experiment involves no instrumentation. Rather, at the start of a bottleneck search this virtual experiment is immediately refined to create a small number of experiments for the entire application with specific hypotheses like "the application is CPU bound" and "the application is spending too much time waiting for I/O operations." Figure 2 focuses on experiments testing the CPUBound hypothesis. In a search history graph, two nodes are connected by an edge if their associated experiments are related by refinement (i.e., one experiment was refined to create the other, more specific experiment). A node's label and its parent node's label indicate the nature of the refinement. For instance, the node labeled `CPUbound:mcr128.llnl.gov` in the figure represents an experiment testing whether all application processes running on

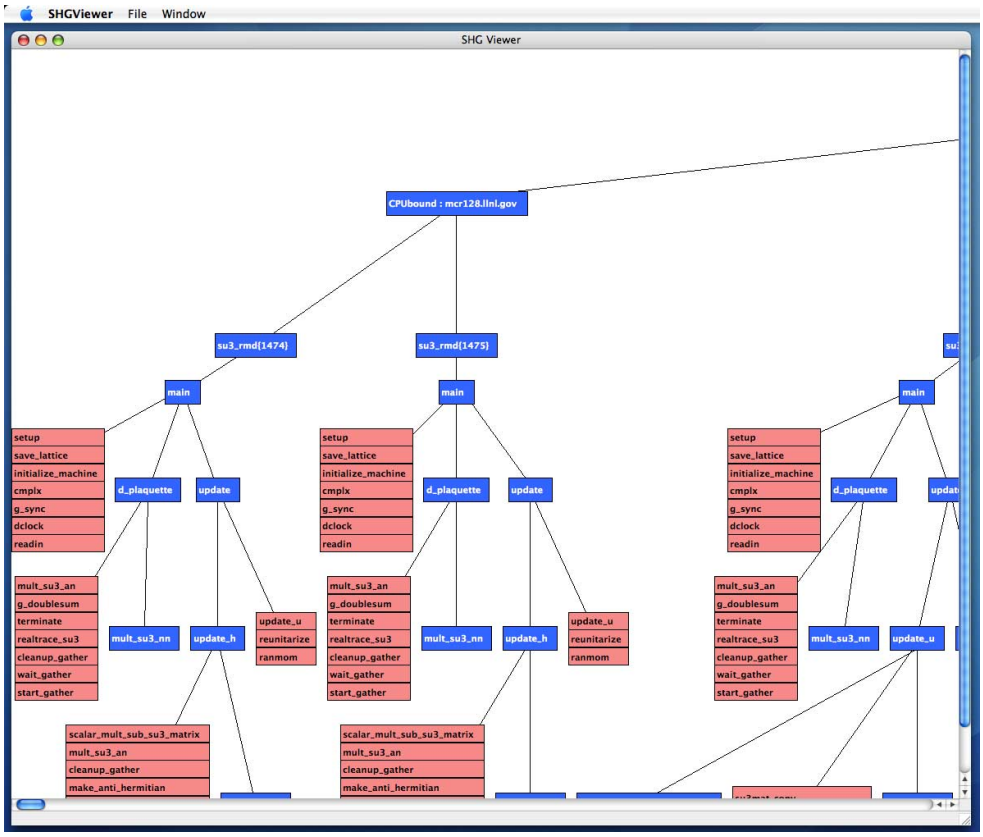


Figure 2: Part of a traditional Search History Graph display

True experiments are shown as blue nodes (dark gray in black-and-white), false experiments are shown as pink nodes (light gray).Only a portion of the full search history graph can be shown—the full graph has over 30,000 nodes.

the host `mcr128.llnl.gov` are CPU bound. This experiment was found to be true, and the experiment was refined to create two new experiments, each of which test whether a specific application process (with process ID 1474 and 1475, respectively) are CPU bound. Although we adopt Paradyn's Search History Graph display, other tools have adopted hierarchical visualizations of search-based performance diagnosis results similar to the approach described here [24,31].

In traditional on-line parallel performance tools, the tool *front-end* process identifies application performance problems by analyzing data generated by tool *daemons* that monitor the application's processes. For example, in the traditional Performance Consultant, the bottleneck search strategy described above is performed entirely within the tool's front-end process. (Tool daemons generate the raw performance data, but the tool front-end is responsible for all analysis and search control.) Because centralized performance data and tool control processing limit tool scalability as the number of application processes grows, our new search strategy distributes portions of the search that examine local application behavior (hereafter called "sub-searches") to reduce centralization of tool activities. In particular, our search strategy distributes sub-searches that examine the behavior of each application process to a search agent within the daemon that monitors the process. It is important to note that the new bottleneck search strategy uses the same number and placement of tool daemons as the traditional centralized strategy.

Distributing local sub-searches has two benefits for automated performance diagnosis tools over a centralized approach. First, distributed sub-searches reduce centralized processing of data and control messages within the tool. Under our search strategy, the parts of a search that examine the behavior of a particular application process is delegated to a local search agent within the tool daemon that is monitoring the process. Data and control messages for that sub-search are processed entirely within the daemon instead of being communicated to the tool's front-end process. The second benefit is that distributed sub-searches 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.

Our bottleneck search strategy adopts the call-graph-based search refinement of Paradyn's Performance Consultant (the "centralized approach" or CA). We investigated two alternatives for distributing local bottleneck searches: a partially distributed approach (PDA) that requires complex management of the search's dynamic instrumentation, and a truly distributed approach (TDA) that retains the PDA's functionality while eliminating instrumentation management complexity.

2.1 Partially Distributed Approach

The PDA consists of three parts: a partially distributed performance bottleneck search strategy, a model for expressing the cost of dynamic instrumentation in parallel computation, and a policy for scheduling instrumentation generated by both the centralized and distributed parts of the search.

Under the traditional Performance Consultant'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. For example, when an experiment labelled "`CPUBound`" is refined to an experiment labelled "`mcr128.llnl.gov`", that experiment will be performed by the local search agent running within the tool daemon on the host `mcr128.llnl.gov`. Furthermore, that experiment becomes the root of a sub-search performed by the local search agent that examines local behavior of the application processes on that host.

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 introduces two requirements on the approach used to monitor and control our search's dynamic instrumentation:

- Local search agents must be able to make independent decisions about inserting and removing the instrumentation that supports their local sub-searches; and
- Instrumentation requests for collecting global performance data must be 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.

The traditional Performance Consultant's centralized search control and single instrumentation cost value satisfy 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.

To allow local search agents to make decisions independently about their own local data instrumentation, our distributed performance diagnosis strategy 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, $1 \le i \le P$. 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., [6,18]) could be used in the PDA to enable the collection of local search agents to reach the same decision regarding global instrumentation, however such an algorithm implementation would place unacceptable computation and communication load on daemon processes. It also would require communication channels between daemon processes that do not exist in our existing tool communication model. The PDA centralizes decision making for global instrumentation requests, using MRNet to gather and aggregate instrumentation cost information.

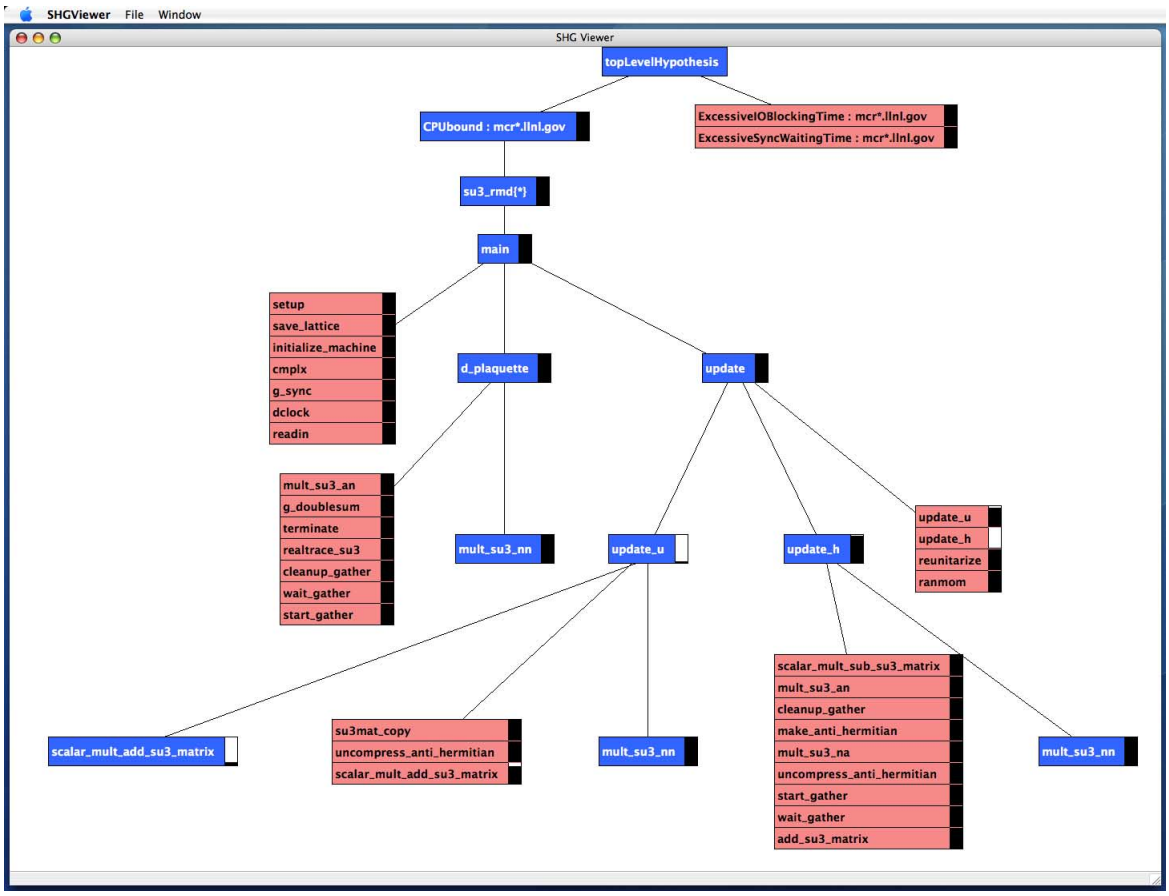
For simplicity, the PDA uses a fixed-partition policy for scheduling both local and global instrumentation. The combined instrumentation cost threshold defines the available instrumentation capacity. The fixed partition ensures that instrumentation of one class does not starve instrumentation of the other. This 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 cost limits under instrumentation scheduling policy.

2.2 Totally Distributed Approach

Initial experience with the PDA showed that it 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. A truly distributed bottleneck search strategy (i.e., one with no centralized search component for global instrumentation) would avoid 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 infer global behavior using local behavior information. There are several possible approaches for making this inference. 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 are 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 our search strategy's global bottleneck results truly represent the application's global behavior.

In our TDA design, we use the Sub-Graph Folding Algorithm (Section 3) to approximate the application's qualitative global behavior. Because the SGFA dynamically clusters application processes based on their qualitative local behavior, and retains information about the number of processes in each category, our TDA technique does not sacrifice insight into an application's global behavior.



Example of a Search History Graph display after applying the SGFA

This is the complete folded form of the Search History Graph partially shown in Figure 2; whereas the un-folded graph has over 30,000 nodes, the folded graph contains 44 nodes. Nodes with "thermometer" gauges represent multiple experiments in the un-folded graph. Each bar indicates what fraction of the experiments its node represents. The label "su3_rmd{}" represents multiple process names in the un-folded graph. True experiments are shown as blue nodes (dark gray in black-and-white), false experiments are shown as pink nodes (light gray in black-and-white). Like Paradyn's traditional Search History Graph display, nodes with the same parent but no children of their own are collapsed into a meta-node to reduce the screen area required for rendering the graph. The thermometer gauge for each sub-node in a meta-node are still drawn immediately to the right of the sub-node within the collapsed node, separated by a thin line.*

3 Sub-Graph Folding Algorithm

With the Sub-Graph Folding Algorithm, our performance diagnosis approach also addresses the scalability barrier of qualitatively aggregating performance diagnosis results for a large number of processes. For a handful of application processes, the search history graph (Figure 2) is effective for showing the results of a bottleneck search. However, because the Search History Graph display shows search results for individual application processes with one sub-graph per process, the display does not scale.

Using ideas from scalable performance visualization [7,17], experiment management for performance tuning [16], and the PRISM parallel debugger [28], we developed the Sub-Graph Folding Algorithm (SGFA) for visualizing the results of a performance bottleneck search. 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 compact than the traditional Search History Graph display. Figure shows the result of applying the SGFA to the result of a TDA bottleneck search on 1024 application processes. There are 44 nodes in the folded graph; the original graph had over 30,000. Like Paradyn's traditional Search History Graph display, nodes with the same parent and no children of their own are collapsed into a meta-node to reduce screen area demands.

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 from each host. 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. In the example folded graph shown in Figure , the SGFA has created a single composite sub-graph that represents all of the un-folded sub-graphs.

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 "`su3_rmd{3751}`" and "`su3_rmd{1634}`" in Figure 2, the SGFA uses executable name but disregards process ID values. 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 "`update_h`" in Figure . The thermometer gauge on the node representing experiments with a "true" truth value is mostly shaded, indicating that the experiment on function `update_h` was true in most of the sub-graphs in the original graph, but not all. Such a situation can arise, for example, if there are small variations in a program's behavior across processes and the observed performance data for an experiment is close to the experiment's user-configurable threshold. 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 process identifiers in Figure 2 are labelled "`su3_rmd{*}`" in the folded graph. Because a TDA bottleneck search used to generate the Search History Graph display shown in Figure , the experiments labeled "CPUBound," "ExcessiveSyncWaitingTime," and "ExcessiveIOBlockingTime" reflect refinement of both hypothesis and focus (to each individual host); hence, there is a node with a composite generalized label containing both a hypothesis and part of a focus. SGFA uses a string generalization algorithm like the longest common subsequence algorithm [13] 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 our distributed bottleneck search strategy 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 bottleneck search strategy.

We designed and implemented an MRNet-based SGFA approach that uses custom MRNet data transformation filters. A stateful SGFA filter running in each process of the MRNet overlay network and the tool front-end maintains a folded sub-graph containing results of the local search agents reachable by that 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.

Because we performed our experimentation in a batch environment, we modified Paradyn to operate without its traditional interactive graphical user interface. To visualize the result of bottleneck searches, we developed a minimal post-mortem Search History Graph visualization tool (shown in Figure). Positive experience with this approach has led us to consider future work investigating another performance diagnosis strategy for large-scale programs in batch environments. The proposed approach uses completely independent bottleneck search agents like the CDA, but instead of using an on-line implementation of the SGFA to combine results, each agent writes its own search results to local storage (or a parallel file system) and then uses a post-mortem tool to apply the SGFA and to visualize the results.

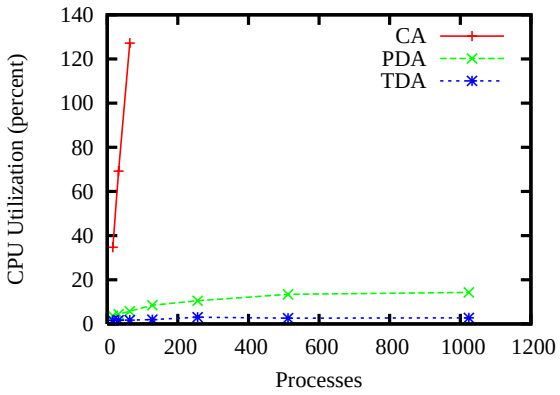
4 Evaluation

To evaluate our distributed bottleneck search strategy and the SGFA, we modified Paradyn version 4.1 to search for performance bottlenecks using one of the CA, PDA, or TDA approaches, and to apply the SGFA to search results. Because we implemented our bottleneck search strategy in the context of Paradyn's Performance Consultant, we call the augmented bottleneck detection component the *Distributed Performance Consultant*. We used the modified version of Paradyn to search for bottlenecks in a parallel application with 16 to 1024 processes. For all experiments we used `su3_rmd`, a simulation of the pure lattice gauge theory of quantum chromodynamics produced by the MILC collaboration [23] for simulating the Standard Model of nuclear physics. The code is implemented in C and we configured it to use MPI for inter-process communication. We used a weak scaling approach in our study.

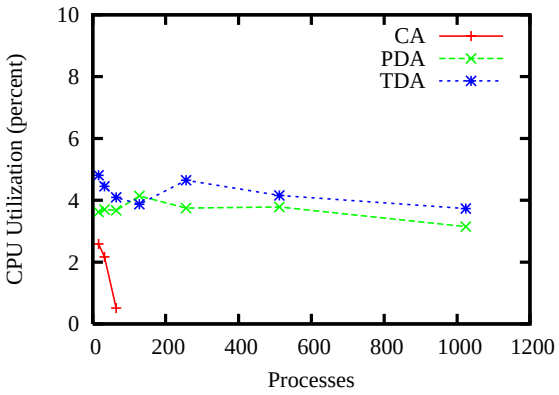
In our experiments we used balanced MRNet topologies with moderate fan-out at each process. Because the number of data points in our scalability study would be limited if we used the same fan-out at each level in the process network, we used topologies with fan-outs of eight at all levels except for those processes connected directly 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.

Our experiments were run on the Multiprogrammatic Capability Cluster (MCR) [19] 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 [9], 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 1.2 distribution [11].

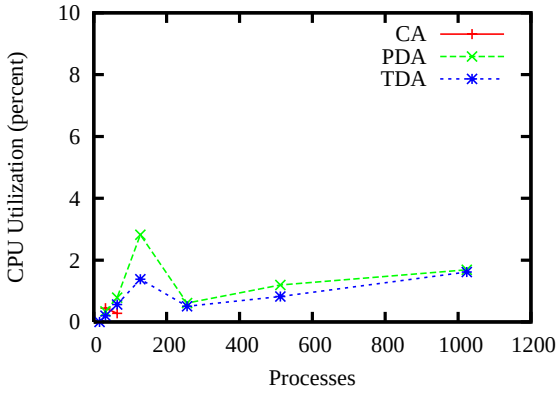
The goal of this work is to evaluate the scalability of our new approach for finding performance bottlenecks in programs with a large number of processes. Consequently, we focused on the performance characteristics of the tool itself, rather than the application that provided a test workload for our tool. Nevertheless, to ensure the performance diagnosis results provided by our new approach were qualitatively similar to those provided using the Performance Consultant's traditional approach, we examined the results produced by our prototype implementation. For example, with 1024 application processes, our performance bottleneck searches indicated CPU time bottlenecks (i.e., the observed CPU utilization was above the CPU utilization hypothesis threshold) in the `mult_su3_nn` and `scalar_mult_add_su3_matrix` functions of the MILC `su3_rmd` code. Because the Performance



(a) Front-End CPU Load



(b) Daemon CPU Load



(c) MRNet Internal Process CPU Load

Figure 3: Computation load for the CA, PDA, and TDA bottleneck search strategies

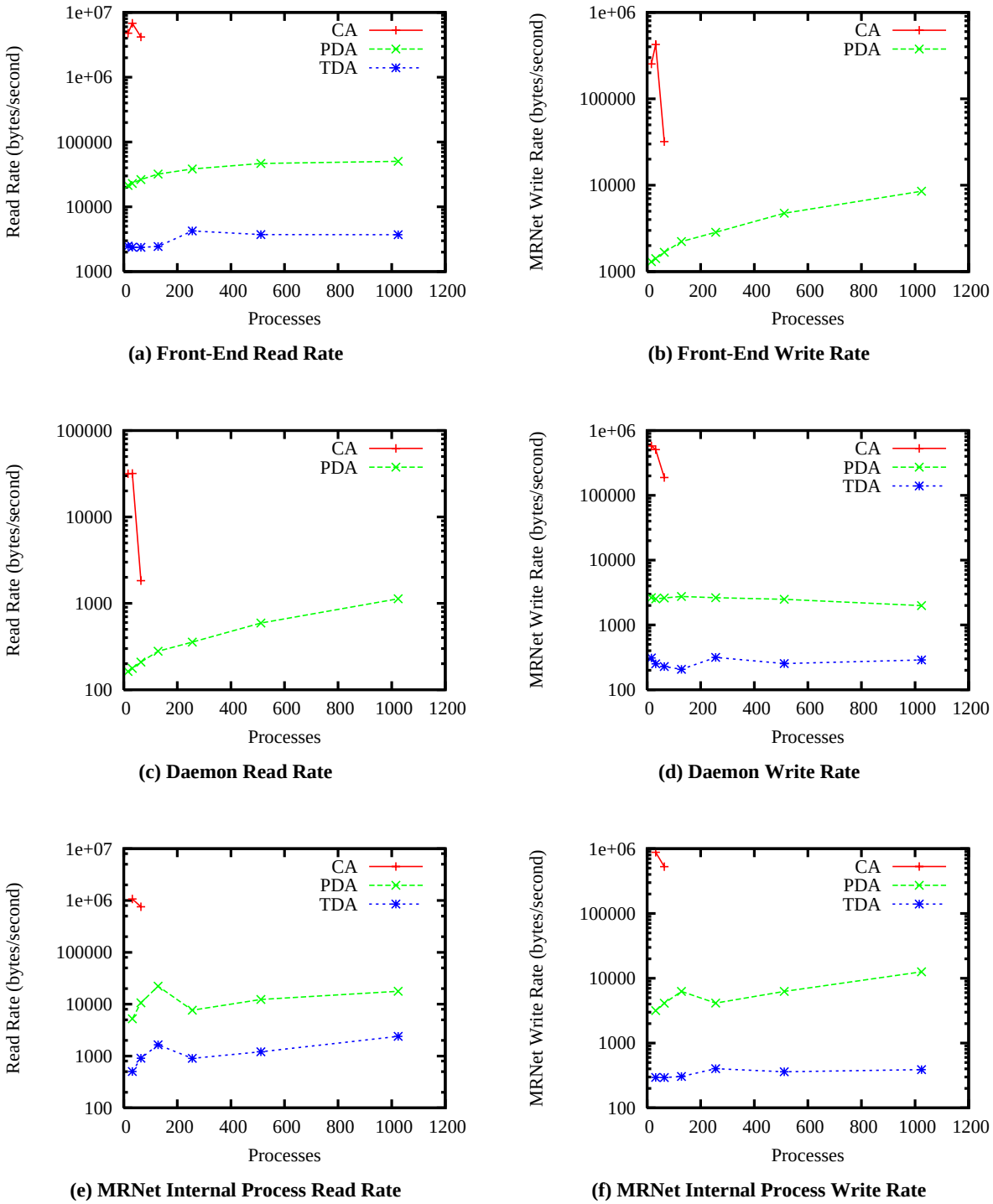
The CPU utilization is shown for (a) the tool's front-end, (b) tool daemons, and (c) MRNet internal processes. Note the Y-axis scale differs significantly between plots. Because CA runs with 64 application processes failed to complete during their batch job's allotted time limit, we did not attempt CA runs with more than 64 processes.

Consultant uses inclusive CPU time metrics that measure the time required by a function and all of its callees, the Performance Consultant also reported CPU time bottlenecks in the `main`, `update`, `update_u`, and `update_h` functions. Except for these experiments, all other experiments performed by the Performance Consultant were false. It was unexpected that no communication-related performance bottlenecks were identified, but further inspection into the su3_rmd code shows that although the program uses a gather operation, it implements the gather using point-to-point operations instead of the MPI_Gather collective communication operation. Individual gather operations are distinguished using a unique message tag for each gather operation. The use of point-to-point operations appears to allow the program to avoid significant blocking for each gather.

4.1 Distributed Performance Consultant Results

To evaluate the Distributed Performance Consultant, we modified Paradyn to use either the CA, PDA, or TDA approach when performing a bottleneck search. We performed a scalability study for the three approaches, watching the computation and communication load at all tool processes (including MRNet internal processes) for signs of resource saturation that would indicate a scalability bottleneck within the tool system itself. We used a tool process' CPU utilization as a measure of its computation load and the rate at which it reads and writes data to the MRNet infrastructure to quantify its communication load. 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. To support running our experiments under MCR's batch scheduling system, we modified the tool front-end to present a text-based user interface instead of a graphical user interface and we automatically perform our experiment when our batch jobs were run.

The results of our scalability study are shown in Figures 3 and 4. 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 CA runs 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.



(a) Front-End Read Rate

(b) Front-End Write Rate

(c) Daemon Read Rate

(d) Daemon Write Rate

(e) MRNet Internal Process Read Rate

(f) MRNet Internal Process Write Rate

Figure 4: MRNet I/O load for the CA, PDA, and TDA bottleneck search strategies.

The number of bytes read from MRNet is shown (a) for the tool's front-end, (c) tool daemons, and (e) MRNet internal processes. The number of bytes written onto MRNet is shown (b) for the tool's front-end, (d) tool daemons, and (f) MRNet internal processes. The Y-axis is logarithmic in each plot. Plots (b) and (c) do not contain a TDA curve because the front-end does not send data and the daemons do not receive data across MRNet during a TDA search. Because CA runs with more than 64 application processes failed to complete during their batch job's allotted time limit, we did not attempt CA runs with more than 64 processes.

Figure 3 compares the computation load at the tool's front-end, daemons, 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 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 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, still needs to be investigated.

Figure 4 compares 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 and written on all MRNet socket connections. We modified each Paradyn process to sample these counts at one-second intervals during a 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.

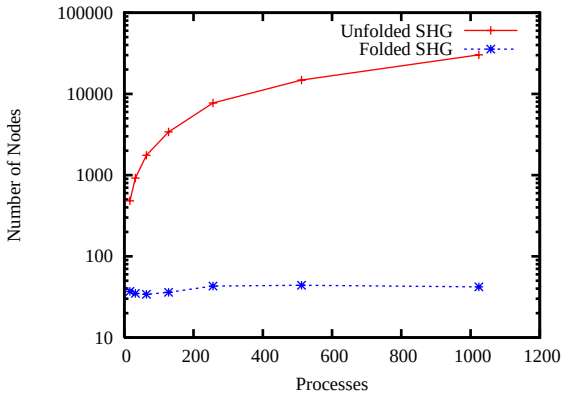


Figure 5: Effect of SGFA on search history graph complexity. *The chart compares the number of nodes in the un-folded search history graph resulting from a TDA bottleneck search with the graph produced by the SGFA. The Y-axis is logarithmic.*

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 variability that echoes the variability in the TDA CPU load results. As with the CPU load, this remains to be investigated as to whether and how the I/O fluctuation is related to the MRNet topology we used.

4.2 SGFA Results

We evaluated the SGFA by implementing it in a custom MRNet data transformation filter as described in Section 3 and measuring its ability to reduce the complexity of Search History Graph results during our Distributed Performance Consultant scalability study. We used the number of nodes in a search history graph as a measure of its complexity. We instrumented the Paradyn front-end to report the number of nodes in the un-folded and folded search history graphs at the completion of a bottleneck search.

The results of our comparison are shown in Figure 5. The chart compares the complexity of the un-folded search history graphs produced by the TDA Distributed Performance Consultant approach with the complexity of the corresponding SGFA-produced graphs. 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 (around thirty nodes). 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.

5 Related Work

The Distributed Performance Consultant and the Sub-Graph Folding Algorithm are two synergistic parts of our approach for

scalable on-line automated performance diagnosis. The Distributed Performance Consultant is related to previous work in on-line automated performance tools, instrumentation cost models, and workload scheduling. The Sub-Graph Folding Algorithm builds on previous work in scalable visualization techniques, especially those for building scalable graph-based displays.

Despite their desirability for large-scale performance tuning, few automated performance diagnosis tools exist. Paradyn [5,22] is the seminal instance of this class of tools, but recent projects KOJAK [24], Peridot [10], and SCALEA [31] also provide automated performance diagnosis functionality. KOJAK is a trace-based off-line automated tool, whereas Paradyn, Peridot, and SCALEA each use on-line analysis. On-line automated computational steering tool kits and tools like Falcon [12], Autopilot [26], and Active Harmony [30] share several characteristics with on-line automated performance diagnosis tools. Falcon and Autopilot use a distributed system of sensors to collect data about an application's behavior and actuators to make modifications to application variables to change its behavior. Active Harmony extends the computational steering toolkit idea to include a component that automatically tunes an application's performance by adjusting application parameters. Each of these systems requires the application and the libraries it uses to be modified to expose steerable parameters. MATE [25] augments on-line automated performance diagnosis with dynamic code optimization to combine the advantages of both automated performance diagnosis and computational steering. MATE does not require program modifications to expose steerable parameters. Instead, it uses dynamic instrumentation to adjust program parameters.

The projects described here have placed varying degrees of emphasis on the issue of scalability. Many 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. Of the projects discussed here, Peridot's approach is the most similar to our Distributed Performance Consultant for scalability as the number of application processes increases. Although its design has not yet been tested at scale, Peridot proposed to use a hierarchy of agents that monitor and evaluate hierarchically-specified "performance properties" as defined by the APART Specification Language [8] or ASL. (The ASL is a product of the APART Working Group [1], a working group funded by the European Union for studying automated performance analysis techniques and tools.) We also use a hierarchy of processes for scalability (i.e., MRNet), but our preferred approach (TDA) uses a single level of distributed performance diagnosis agents and has been evaluated at scale. Falcon and Autopilot also use a distributed analysis approach like ours, but these computational steering tool kits provide only distributed decision-making mechanism by design. Finally, none of these projects addresses the issue of scalable visualization of performance diagnosis results that we address with the SGFA.

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 [21] and Yan and Listgarten [32] proposed cost models for trace-based instrumentation systems that try to compensate for the effects of instrumentation in event traces. Their models assume simple instrumentation such as the writing of fixed-length record to a trace file. Hollingsworth and Miller [14] presented a simple model for predicting and measuring the cost of general instrumentation in an on-line performance tool. Paradyn's Performance Consultant [22] uses their model in a feedback system that controls instrumentation cost during automated bottleneck searches.

None of these instrumentation cost models is well-suited for our scalable automated performance diagnosis approach. Our instrumentation is more complex than the instrumentation assumed by Reed and Malony and Yan and Listgarten. The Hollingsworth and Miller model is not well suited because it aggregates the instrumentation cost for parallel computation into a single value. Our PDA and especially TDA approaches require cost information at the granularity of individual nodes, but that information is lost in the Hollingsworth and Miller model.

The PDA's local/global instrumentation request scheduling policy draws upon previous research in scheduling mixed sequential/parallel workloads, including studies by Leutenegger and Vernon [20] for shared-memory multiprocessors, Arpaci et al [2] for a network of workstations, and Arpaci-Dusseau [3] for avoiding the scalability problems of traditional gang scheduling systems with implicit co-scheduling. The PDA's scheduling problem is slightly different and more restricted than the problems addressed in this previous work. In the PDA, gang scheduling of global instrumentation is a requirement, not just a performance benefit.

The Sub-Graph Folding Algorithm is related to previous work in scalable visualization techniques, especially those for building scalable graph-based displays. Couch [7] proposed an abstract technique for constructing scalable performance visualizations that groups processes into categories and displays only per-category statistics. Kimelman et al [17] described a technique for reducing the complexity of dynamic graph-based displays like search history graphs by manually combining nodes of the graph. Stasko and Muthukumarasamy [29] proposed the semantic zooming technique for constructing scalable graph-based displays by abstracting the graph into a matrix of blocks. Each block's characteristics (e.g., color and texture) indicate some characteristics of the sub-graph represented by that block. Unlike Couch and Kimelman's approaches, the SGFA requires no user intervention to construct its scalable display. 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 familiar graph-like characteristics (i.e., nodes and edges) whereas semantic zooming does not.

SGFA's folding operation is similar to that used by the PRISM parallel debugger [28] and Karavanic's structural merge operator [16]. PRISM's Where Tree provides a concise, tree-based representation of the call stacks of all application processes by folding together equivalent call sequences to form a tree. Karavanic's structural merge operator combines two trees of program resources called EventMaps. The operator combines equivalent nodes from the two EventMaps to form a composite tree containing the resources that are present in both EventMaps. In contrast to the PRISM folding operation, SGFA folds trees rather than linear sequences of nodes, and considers node types and truth values in addition to node name when folding. SGFA differs from Karavanic's structural merge operator in that the 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.

Acknowledgments

This paper benefited from the hard work of many past and present members of the Paradyn research group, especially Dorian Arnold with his work on MRNet. We also thank Scott Futral, Chris Chambreau, Bronis de Supinski, John Gyllenhaal, and Barbara Herron for their help with the computing environment at Lawrence Livermore National Laboratory, and Jeffrey Vetter of Oak Ridge National Laboratory for his support and guidance.

References

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

[2] 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 Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS 1995)*, Ottawa, Canada, May 1995, pp. 267–278.

[3] A.C. Arpaci-Dusseau. Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems. *ACM Trans. on Computer Systems* **19**, 3, August 2001.

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Intl. Journal of High Performance Computing Applications* **14**, 3, Fall 2000, pp. 189–204.

[5] H.W. Cain, B.P. Miller, and B.J.N. Wylie. A Callgraph-Based Search Strategy for Automated Performance Diagnosis. *Sixth Intl. European Conf. 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.

[6] B.A. Coan and J.L. Welch. Modular Construction of an Efficient 1-Bit Byzantine Agreement Protocol. *Mathematical Sys. Theory* **26**, 1, 1993.

[7] A.L. Couch. Categories and Context in Scalable Execution Visualization. *Journal of Parallel and Distributed Computing* **18**, 2, June 1993.

[8] T. Fahringer, M. Gerndt, G. Riley, and J.L. Träff. Specification of Performance Problems in MPI Programs with ASL. *2000 Intl. Conf. on Parallel Processing (ICPP'00)*, Toronto, Canada, Aug. 2000.

[9] 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.

[10] M. Gerndt. Automatic Performance Analysis Tools for the Grid. *Concurrency and Computation: Practice and Experience* **17**, 2–4, February 2005.

[11] 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.

[12] 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.

[13] D.S. Hirschberg. Algorithms for the Longest Common Subsequence Problem. *Journal of the ACM* **24**, 4, October 1977.

[14] J.K. Hollingsworth and B.P. Miller. An Adaptive Cost Model for Parallel Program Instrumentation. *Second Intl. European Conf. 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.

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

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

[17] D. Kimelman, B. Leban, T. Roth, and D. Zernik. Reduction of Visual Complexity in Dynamic Graphs. *Graph Drawing '94: DIMACS Intl. Workshop (GC '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.

[18] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. on Programming Languages and Systems* **4**, 3, July 1982.

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

[20] S.T. Leutenegger and M.K. Vernon. The Performance of Multiprocessor Scheduling Policies. *1990 ACM Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS 1990)*, Boulder, Colorado, May 1990.

[21] 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.

[22] 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 Parallel Performance Measurement Tool. *IEEE Computer*, **28**, 11, November 1995.

[23] The MIMD Lattice Computation (MILC) Collaboration. http://physics.indiana.edu/~sg/milc.html, April 2005.

[24] B. Mohr and F. Wolf. KOJAK: A Tool Set for Automatic Performance Analysis of Parallel Applications. *Ninth Intl. Euro-Par Conf. (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.

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

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

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

[28] 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.

[29] 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.

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

[31] H.-L. Truong and T. Fahringer. SCALEA: a Performance Analysis Tool for Parallel Programs. *Concurrency and Computation: Practice and Experience* **15**, 11–12, Sept. 2003.

[32] J.C. Yan and S. Listgarten. Intrusion Compensation for Performance Evaluation of Parallel Programs on a Multicomputer. *Sixth Intl. Conf. on Parallel and Distributed Computing Systems (ISCA 1993)*, Louisville, Kentucky, October 1993.