The Paradyn Parallel Performance Measurement Tools

Barton P. Miller, Mark D. Callaghan
Jonathan M. Cargille, Jeffrey K. Hollingsworth
R. Bruce Irvin, Karen L. Karavanic
Krishna Kunchithapadam, Tia Newhall

{bart,markc,jon,hollings,rbi,karavan,krishna,newhall}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin–Madison
1210 W. Dayton Street
Madison, WI 53706

Abstract

Paradyn is a performance measurement tool for parallel and distributed programs. Paradyn uses several novel technologies so that it scales to long running programs (hours or days) and large (thousand node) systems, and automates much of the search for performance bottlenecks. It can provide precise performance data down to the procedure and statement level.

Paradyn is based on a dynamic notion of performance instrumentation and measurement. Unmodified executable files are placed into execution and then performance instrumentation is inserted into the application program and modified during execution. The instrumentation is controlled by the Performance Consultant module, that automatically directs the placement of instrumentation. The Performance Consultant has a well-defined notion of performance bottlenecks and program structure, so that it can associate bottlenecks with specific causes and specific parts of a program. Paradyn controls its instrumentation overhead by monitoring the cost of its data collection, limiting its instrumentation to a (user controllable) threshold.

The instrumentation in Paradyn can easily be configured to accept new operating system, hardware, and application specific performance data. It also provides an open interface for performance visualization, and a simple programming library to allow these visualizations to interface to Paradyn.

Paradyn can gather and present performance data in terms of high-level parallel languages (such as data parallel Fortran) and can measure programs on massively parallel computers, workstation clusters, and heterogeneous combinations of these systems.

1. INTRODUCTION

Paradyn is a tool for measuring the performance of large-scale parallel programs. Our goal in designing a new performance tool was to provide detailed, flexible performance information without incurring the space (and time) overhead typically associated with trace-based tools. Paradyn achieves this goal by dynamically instrumenting the application and automatically controlling this instrumentation in search of performance problems. Dynamic instrumentation allows us to defer inserting instrumentation into an executing program until the moment it is needed (and removing it when it is no longer needed);

This work is supported in part by Department of Energy Grant DE-FG02-93ER25176. Office of Naval Research Grant N00014-89-J-1222, NSF Grant CCR-9100968, and an NSF Infrastructure Grant CDA-9024618. Hollingsworth was supported in part by an ARPA Graduate Fellowship in High Performance Computing.

Paradyn's Performance Consultant decides when and where to insert instrumentation.

1.1. Guiding Principles and Characteristics

Below we describe several principles that guided the design of Paradyn. With each description, we also summarize the features of Paradyn that incorporate these principles.

Scalability:

We must be able to measure long running programs (hours or days) on large (about 1000 node) parallel machines using large data sets. For correctness debugging, you can often test your program on small data sets and be confident that the program will work correctly on larger data sets. For performance tuning, this is often not the case. We have seen, in many real application programs, that as program and data set size increase, different resources saturate and become bottlenecks.

We must also be able to measure large programs that have hundreds of modules and thousands of procedures. The mechanisms for instrumentation, program control, and data display must gracefully handle this large number of program components.

Paradyn uses dynamic instrumentation to instrument only those parts of the program relevant to finding the current performance problem. It starts looking for high-level problems (such as too much total synchronization blocking, I/O blocking, or memory delays) for the whole program. Only a small amount of instrumentation is inserted to find these problems. Once one of these general problems is found, instrumentation is inserted to find more specific causes of the problem. No detailed instrumentation is inserted for classes of problems that do not exist.

Automate the search for performance problems:

Our approach to performance measurement is to try to identify the parts of the program that are consuming the most resources and direct the programmer to these parts. Automating the search for performance information enables Paradyn to dynamically select which performance data to collect (and when to collect it). The goal is for the tool to identify the parts of the program that are limiting performance, rather than require the programmer to do this.

The Performance Consultant module of Paradyn has a well-defined notion, called the W^3 Search Model, of the types of performance problems found in programs and of the various components contained in the current program. The Performance Consultant uses the information in the W^3 Search Model to guide placement and modification of the dynamic instrumentation.

Provide well-defined data abstractions:

Simple data abstractions can unify the design of a performance tool and simplify its organization. Paradyn uses two important abstractions in collecting, communicating, analyzing, and presenting performance data: *metric-focus grids* and *time-histograms*.

A metric-focus grid is based on two lists (vectors) of information. The first vector is a list of performance metrics, such as CPU time, blocking time, message rates, I/O rates, or number of active processors. The second vector is a list of individual program components, such as a selection of procedures, processor nodes, disks, message channels, or barrier instances. The combination of these two vectors produces a matrix with each metric listed for each program component.

The elements of the matrix can be single values (such as current value, average, min, or max) or time-histograms. Time-histograms are fixed-size data structures that record the behavior of a metric as it varies over time. The time-histogram is an important tool in recording time-varying data for long running programs.

Support heterogeneous environments:

Parallel computing environments range from clusters of workstations to massively parallel computers. Heterogeneity arises in processor architectures (e.g., SPARC vs. PA-RISC), operating systems (e.g., OSF/1 vs. Solaris 2 vs. SunOS 4.1), programming models (native operating system functions vs. PVM vs. P4), and programming languages (e.g., C vs. HPF vs pC++). Isolating each of these different dimensions into abstractions within the performance tool can simplify porting. For example, adding support for PVM requires only that you know the name of the new communication and process creation operations; the underlying support for the UNIX operating system, chip architecture, and programming language support stays the same.

Paradyn already works well in several of these domains and measures programs running on heterogeneous combinations of these domains. Current hardware platforms include the TMC CM-5, SPARCstation (including multiprocessors), and HP PA-RISC; operating systems include TMC CMOST, SunOS 4.1, Solaris 2, and HP/UX; programming models include UNIX IPC, Solaris 2 thread and synchronization primitives, CM-5 CMMD, CM Fortran CM-RTS, and PVM.

Support high-level parallel languages:

Users of high-level parallel programming languages need accurate performance information that is relevant to their source code. When their programs experience performance problems at the lowest levels of their hardware and software systems, programmers need the ability to peel back layers of abstraction to examine low-level problems while maintaining references to the high-level source code.

Paradyn supports a facility for allowing high-level language programmers to view the performance of their program in terms of the high-level objects (such as arrays and loops for data parallel Fortran) or in terms of the primitive objects (such as nodes and messages).

Open interfaces for visualization and new data sources:

Graphical and tabular displays are important mechanisms for understanding performance data. There are several projects, such as Paragraph [2] and Pablo [8], that have developed a large collection of visualization routines. We want our tool to leverage off these existing visualizations. Paradyn has a set of standard visualizations (time-histograms, bar graphs, tables, and profiles) and provides a simple interface to incorporate displays from other sources.

Equally important is the ability to incorporate new sources of performance data, such as cache miss data from the processor, network traffic from the network interfaces, or paging activity from the operating system. Paradyn's instrumentation is configurable to use any performance quantity that can be mapped into a program's address space. Including these new sources of data in Paradyn requires only a change to a configuration file.

Streamlined use:

There should be few impediments to the use of a new tool. Installation should require only fetching one or more files via ftp and running the tool – there should be no need to have special system privileges or to modify system directories. In the best case, users should not have to modify their source program or use special compiling techniques. Dynamic instrumentation in Paradyn avoids the need to modify, recompile, or re-link an application. This same characteristic is shared by some implementations of binary rewriting, such as qpt [1] and Pixie [6]. Dynamic instrumentation also allows us to attach to an already-running program (such as a parallel database server), monitor its performance for some interval, and then detach.

1.2. Dynamic Performance Measurement

Paradyn differs from previous performance measurement tools in that program instrumentation and performance evaluation are done *during* execution of the application program. Since all the work is done during execution, comparing Paradyn to previous tools raises several questions. These questions include: What about transient effects? What about brief periodic effects? What if the program does not run for a long enough time?

The answers to these questions are based on the observation that we are measuring long-running programs, with execution times of hours (or even days). Although it might take seconds to insert new instrumentation and start evaluating the data, there is little chance that interesting behaviors will be missed. If a program runs for 10 hours, then even a 5 minute ‘‘transient’’ operation is less than 1% of the total execution time (and therefore not interesting for performance tuning). If a program repeatedly performs a brief (few second) operation, we will detect this behavior if the *cumulative* effect of these brief operations is large enough. Short running programs might finish before Paradyn has had a chance to isolate the performance problem(s). In this case, Paradyn can save the state of its search for performance problems and re-run the program to complete the search.

In the next section, we describe our measurement methodology and present an overview of the structure of the Paradyn tools. Section 3 presents our mechanism for dynamic instrumentation, along with some of the implementation details. Section 4 describes Paradyn’s Performance Consultant, an automated module that controls dynamic instrumentation and searches for performance problems. In Section 5, we describe Paradyn’s open interface for visualizations. To provide a better feel for how Paradyn is used, Section 6 includes examples of measurement sessions on real application programs. We conclude in Section 7.

2. SYSTEM OVERVIEW

This section provides an overview of the Paradyn system. We first present the basic abstractions used in Paradyn and then describe its basic components.

2.1. Basic Abstractions

Paradyn is built around two simple but powerful data abstractions. These abstractions unify the internal structure of the system, giving users a consistent view of the system and of the data that it presents.

The first abstraction is the metric-focus grid. Metrics are time-varying functions that characterize some aspect of a parallel program’s performance; examples include CPU utilization, memory usage, and counts of floating point operations. A focus is a specification of a part of a program expressed in terms of program resources. Typical resource types include synchronization objects, source code objects (procedures, basic blocks), threads and processes, processors, and disks. Resources are separated into several different hierarchies, each representing a class of objects in a parallel application. For example, there is a resource hierarchy for CPUs, containing each processor. A focus contains one or more components from each resource hierarchy. For example, one focus might be all synchronization objects accessed by a single procedure on one processor. The combination of a list of metrics with a list of foci forms a matrix (called a *grid* in Paradyn) containing the value of each metric for each focus. The Performance Consultant and visualizations receive performance data by specifying one or more metric-focus grids.

Paradyn stores performance data internally in a data structure called a time histogram [3]. A time histogram is a fixed-size array whose elements (buckets) store values of a metric for successive time intervals. Two parameters determine the granularity of the data stored in time histograms: initial bucket width (time interval) and number of buckets. Both parameters are supplied by higher level consumers of the performance data. If a program runs longer than the initial bucket width times the number of buckets, we double the bucket width and re-bucket the previous values. The change in bucket width (time interval) can cause a corresponding change in the sampling rate for performance data, reducing instrumentation overhead. This process repeats each time we fill all the buckets. As a result, the rate of data collection decreases logarithmically, while maintaining a reasonable representation of the metric's time-varying behavior.

2.2. Components of the System

Paradyn consists of the main Paradyn process, one or more Paradyn daemons, and zero or more external visualization processes. The central part of the tool is a multi-threaded process that includes the Performance Consultant, Visualization Manager, Data Manager, and User Interface Manager. Communication between threads is defined by a set of interfaces constructed to allow any module to request a service of any other module. Figure 1 shows the Paradyn architecture.

The Data Manager handles requests from other threads for data collection, delivers performance data from the Paradyn daemon(s) to the requesting thread(s), and maintains and distributes information about the metrics and resource hierarchies for the currently defined application. The User Interface Manager thread was developed using Tcl/Tk [7], and provides the user with visual access to the system's main controls and performance data. The Performance Consultant controls the automated search for performance problems, requesting and receiving performance data from the Data Manager. The Visualization Manager starts visualization processes; one visualization thread is created in Paradyn for every external visualization process that is started. The job of a visualization thread is to handle communication between the external visualization process and the other Paradyn modules.

The Paradyn daemon contains the platform-dependent parts of Paradyn. The Instrumentation Manager implements the dynamic instrumentation; i.e., it is responsible for inserting the requested

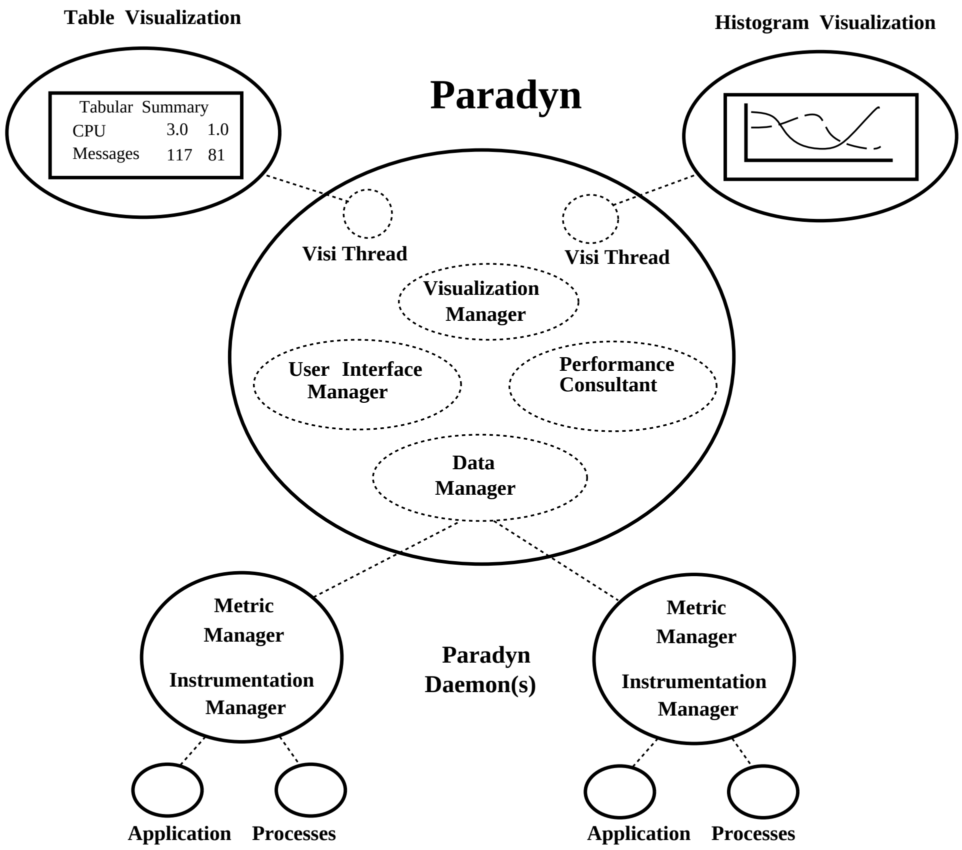

Figure 1. Overview of Paradyn Structure.

Dotted ovals are threads, and solid ovals are processes.

instrumentation into the executing processes that it monitors. The interface between the Paradyn process and the Paradyn daemon supports four main functions: process control, performance data delivery, performance data requests, and the delivery of high-level language mapping data. The daemon services requests from Paradyn for process control and performance data, and delivers performance data from the application(s) to Paradyn. By encapsulating the platform dependencies in the daemons and using a remote procedure call interface, we can easily handle heterogeneous applications. We currently support daemons for various versions of UNIX, the TMC CM-5, and networks of workstations running PVM.

2.3. Configuration Files

Paradyn uses a configuration language (Paradyn Configuration Language, PCL) to describe all architecture, operating system, environment, and language dependent characteristics of applications and platforms. PCL allows users to create new metrics and instrumentation, incorporate new visualizations, specify alternate Paradyn demons, set various display and analysis options, and specify command lines for starting applications. More details on the metric definition part of PCL are presented in Section 3.

The default PCL file describes Paradyn's basic metrics, instrumentation, visualizations, and daemons. Each user can provide an additional PCL file with personalized settings and options. Users can also create an application-specific PCL file that describes details of the application and how it is run.

3. DYNAMIC INSTRUMENTATION

Paradyn uses dynamic instrumentation to instrument only those parts of the program relevant to finding the current performance problem [4]. Dynamic instrumentation defers instrumenting the program until it is in execution and dynamically inserts, alters, and deletes instrumentation during program execution. This section describes the dynamic instrumentation interface and implementation, how dynamic instrumentation collects mapping information for high-level language views, and how users may describe their own metrics. In Section 4, we discuss how Paradyn's Performance Consultant controls dynamic instrumentation to find performance problems.

3.1. Dynamic Instrumentation Interface

Requests for dynamic instrumentation are made in terms of a metric-focus grid (described in Section 2.1). The Paradyn daemon translates instrumentation requests into instructions to be inserted into the application. Translation is a two-step process. First, metric-focus requests are translated into machine independent abstractions by the Metric Manager. Second, the machine independent representation is converted into machine instructions by the Instrumentation Manager.

Counters and timers are the two types of instrumentation that can be inserted into an application. Counters are integer counts of the frequency of some event, and timers measure the amount of time spent performing various tasks (in either wall-time or process-time units).

3.2. Points, Primitives, and Predicates

Points, primitives, and predicates provide a simple, machine independent set of operations that are used as building blocks for dynamic instrumentation. *Points* are locations in the application's code where instrumentation can be inserted. *Primitives* are simple operations that change the value of a counter or a timer. *Predicates* are boolean expressions that guard the execution of primitives (essentially **if** statements). By inserting predicates and primitives at the correct points in a program, a wide variety of metrics can be computed.

The points currently available in our system are procedure entry, procedure exit, and individual call statements. In the future, points will be extended to include basic blocks and individual statements. We provide six primitives: set counter, add to counter, subtract from counter, set timer, start timer, and stop timer. Predicates are simple conditional statements that consist of an expression and an action. The expression can be computed using counters, constants, parameters to a procedure, or a procedure return value, as well as numeric or relational operators.

An example of how primitives and predicates can be combined to create metrics is shown in Figure 2. It computes the amount of time spent sending messages on behalf of the procedure `foo` and its descendents. The `fooFlg` counter keeps track of whether `foo` is on the stack; it is incremented on entry and decremented on exit from `foo`. The value of `fooFlg` is then used as a predicate to control whether the `msgTime` timer primitives will be called upon entry and exit from `SendMsg`. When `foo` is not active, the primitives will not be executed.

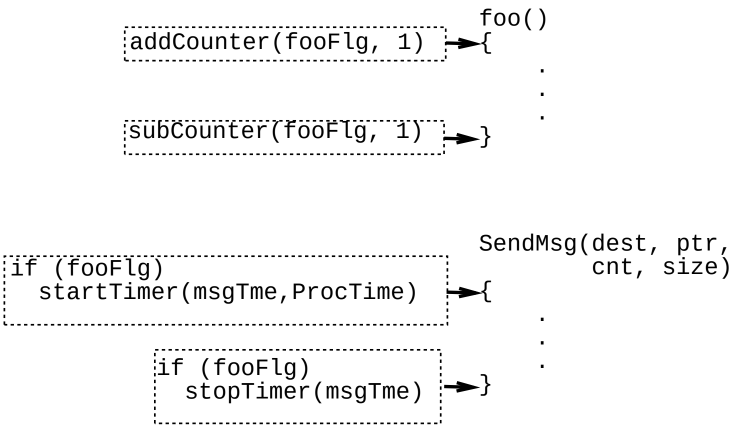

Figure 2. Example Metric Computation.

The translation from metric-focus specifications to points, primitives, and predicates is described by metric definitions contained in the PCL configuration files. These definitions simplify the addition of new metrics, porting of Paradyn to new systems, and user customization. We provide a standard library of metric descriptions, and users (and other tools) can add to this library. The metric descriptions in PCL consist of definitions and constraints. Some metric definitions and resource constraints are generic and apply to all platforms; others are specific to a platform or programming model.

A metric definition is a template that describes how to compute a metric for different resource combinations. It consists of a series of code fragments that create the primitives and predicates to compute the desired metric. We need to be able to compute each metric for any combination of resource constraints. To make these metric definitions compact and modular, we divide metric definition into two parts: a base metric, and a series of resource constraints. The base metric defines how a metric is computed for an entire application (e.g., all procedures, processes, or processors). A resource constraint defines how to restrict the base metric to an instance of a resource in one of the resource hierarchies. Resource constraints usually translate to an instrumentation predicate.

3.3. Instrumentation Generation

The Instrumentation Manager encapsulates the architecture-specific knowledge; it locates the allowable instrumentation points and performs the final translation of points, primitives, and predicates into machine-level instrumentation. When Paradyn is initially connected to an application process, the Instrumentation Manager identifies all potential instrumentation points by scanning the application (or applications) binary image(s). Procedure entry and exit, as well as procedure call sites are detected and noted as points.

After Paradyn is connected to the application, the Instrumentation Manager waits for requests from the daemon's Metric Manager, translates them into small code fragments, called *trampolines*, and inserts them into the program. Two types of trampolines, base trampolines and mini-trampolines, are used. There is one base trampoline per point with active instrumentation. A base trampoline is inserted into the program by replacing the machine instruction at the point with a branch to the trampoline, and relocating the replaced instruction to the base trampoline. A base trampoline has slots for calling mini-trampolines both before and after the relocated instruction.

Mini-trampolines contain the code to evaluate a specific predicate, or invoke a single primitive. There is one mini-trampoline for each primitive or predicate at each point. Creating a mini-trampoline requires generating appropriate machine instructions for the primitives and predicates requested by the Metric Manager. The necessary instructions are assembled by the Instrumentation Manager, and then transferred to the application process using a variation of the UNIX ptrace interface. The generated code also includes appropriate register save and restore operations.

3.4. Data Collection

Once instrumentation has been inserted into the application, data begins flowing back to the higher-level clients. The current value of each active timer and counter is periodically sampled, and transported by the Paradyn daemon to the Data Manager. Note that the instrumentation keeps track of the precise value of each performance metric, and the sampling rate determines only how often Paradyn sees the new value.

Since samples from counters and timers are Paradyn's basic data type, we are able to easily integrate performance data from external sources. For example, most operating systems keep a variety of performance data that can be read by user processes; examples include statistics about I/O, virtual memory, and CPU use. Several machines also provide hardware-based counters that are a source of useful performance information. For example, the IBM Power2, Cray Y-MP, and Sequent Symmetry systems provide detailed counters of processor events. Data from external sources is treated identically to Paradyn's own instrumentation. External data can be constrained in the same way as other performance metrics, to relate it back to specific parts of a program. For example, if we have a way to read the cumulative number of page faults taken by a process, we can read this counter before and after a procedure call to approximate the number of page faults taken by that procedure.

3.5. Internal Uses of Dynamic Instrumentation

Resource discovery is an important use of dynamic instrumentation. Resource discovery is the process of determining which resources are used by an application, and using this information to build the resource hierarchies. Much of the resource information for an application can be determined statically when Paradyn is first connected to the application; for example, at this point we know all of the procedures that might be called, and what types of synchronization libraries are linked into the application. However, some aspects of resource discovery must be deferred until the program is executing. For example, information about which files are read or written during execution can only be determined when the files are first accessed. To collect this runtime resource information, instrumentation is inserted into the application program. We insert instrumentation (using the same technique as normal instrumentation) to record the file names on open requests.

Another important use of dynamic instrumentation is the collection of dynamic mapping information for high-level languages. Paradyn daemons collect most mapping information statically from such sources as symbol tables, but many parallel languages defer mapping data structures to processor nodes until runtime, and some languages change data mappings during execution. In these cases, we dynamically instrument runtime mapping routines, and the Paradyn daemons send the information to the Data Manager. The Data Manager uses the information to support language specific views of performance data as described in Section 4.2.

4. THE W^3 SEARCH MODEL AND THE PERFORMANCE CONSULTANT

The goal of Paradyn is to assist the user in locating performance problems in a program; a performance problem is a part of the program that contributes a significant amount of time to its execution. A single execution of a program may contain several problems. To assist in finding performance problems, Paradyn has a well-defined model, called the W^3 Search Model [5], that organizes information about a program's performance. Performance problems are found by *searching* through the space defined by W^3. Paradyn's Performance Consultant module uses the W^3 Search Model to automate the searching for performance problems. To conduct this search, the Performance Consultant uses data gathered by dynamic instrumentation.

We first describe the W^3 Search Model in more detail, and then describe how the Performance Consultant automates searching in Paradyn.

4.1. THE W^3 SEARCH MODEL

The W^3 Search Model abstracts those aspects of a parallel program that can affect performance, and is based on answering three questions: *why* is the application performing poorly, *where* is the performance problem, and *when* does the problem occur? To answer the ''why'' question, our system includes hypotheses about potential performance problems in parallel programs. We collect performance data to test whether these problems exist in the program. In answering the ''where'' question, we isolate a performance problem to a specific program resource (e.g., a disk system, a synchronization variable, or a procedure). To identify when a problem occurs, we try to isolate a problem to a specific phase of the program's execution. Finding a performance problem is an iterative process of refining our answers to these three questions.

4.1.1. The ''Why'' Axis

The first performance question often asked by programmers is ''why is my application running so slowly?'' The ''why'' axis represents the broad types of problems that can cause a parallel program to run slowly. Potential performance problems are represented by *hypotheses* and *tests*. Hypotheses represent the fundamental types of performance problems that occur in parallel programs, independent of the program being studied and the algorithms it uses. For example, a hypothesis might be that a program is synchronization bound. Hypotheses represent activities universal to all parallel computation, so a small set of them (a couple of dozen), provided by the tool builder, can cover most performance problems.

Hypotheses can be refined into more precise hypotheses. The dependence relationships between hypotheses define the search hierarchy for the ‘‘why’’ axis. These dependencies form a directed acyclic graph, and searching the ‘‘why’’ axis involves traversing this graph. Figure 3 shows a partial ‘‘why’’ axis hierarchy; the current hypothesis is HighSyncBlockingTime. This hypothesis was reached after first concluding that a SyncBottleneck exists in the program.

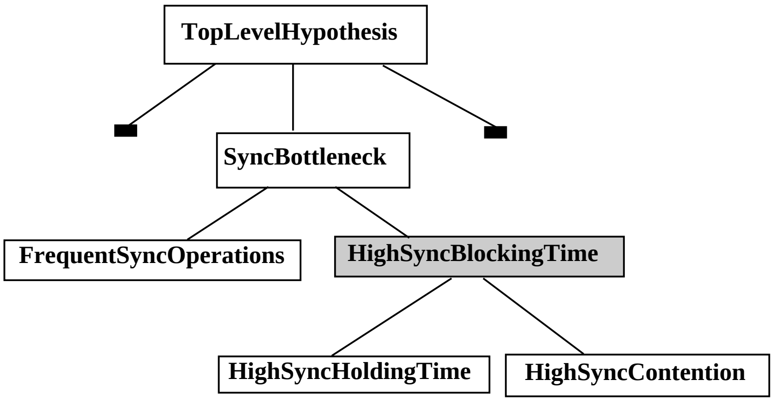

Figure 3. A sample ‘‘why’’ axis with several hypotheses.

This figure shows a portion of the ‘‘why’’ axis, representing several types of synchronization bottlenecks. The shaded node shows the hypothesis currently being considered.

Tests are boolean functions that evaluate the validity of a hypothesis. Tests are expressed in terms of a threshold and (one or more) metrics calculated by the Instrumentation Manager (e.g., synchronization blocking time is greater than 20% of the execution time).

4.1.2. The ‘‘Where’’ Axis

The second performance question most programmers ask is ‘‘what part of my application is running slowly?’’ The ‘‘where’’ axis represents the classes of resources (parts of a program) in which a performance problem lies. Searching along the ‘‘why’’ axis classifies the type of a performance problem, while searching along the ‘‘where’’ axis pinpoints the problem to specific program components. For example, a ‘‘why’’ search may show that a program is synchronization bound, and a subsequent ‘‘where’’ search may isolate one hot synchronization object among many thousands in the program.

The ‘‘where’’ axis represents the different foci that can be measured. Each hierarchy in the ‘‘where’’ axis has multiple levels, with the leaf nodes being the instances of the resources used by the application. Each resource hierarchy can be refined independently.

The trees in Figure 4 represent sample resource hierarchies. A ‘‘where’’ axis display from a real application is shown in Figure 5. The root of the leftmost hierarchy in Figure 4 is SyncObject. The next level contains four types of synchronization (Semaphore, Message, SpinLock, and Barrier). Below the SpinLock and Barrier nodes are the individual locks and barriers used by the application. The children of the Message node are the types of messages used. The children of the Semaphore node are the semaphore groups used in the application. Below each semaphore group are the individual semaphores.

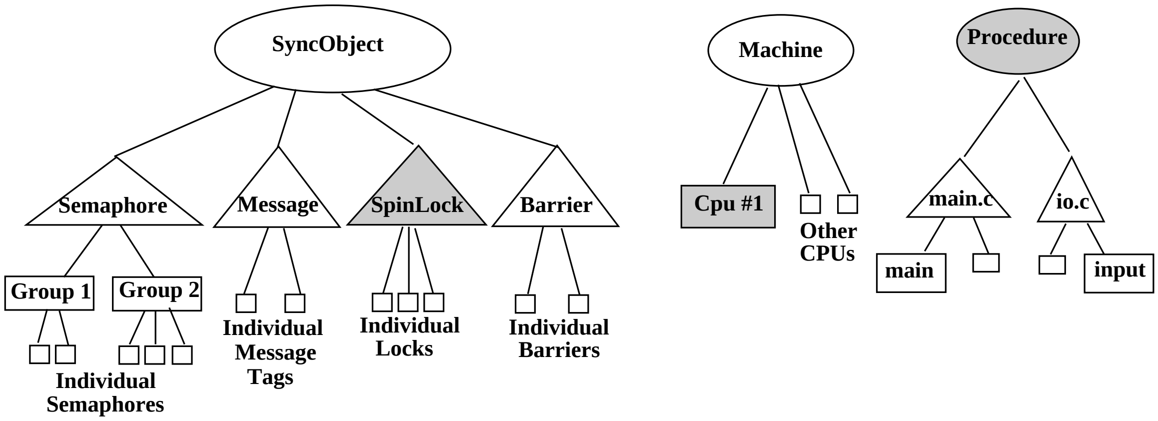

Figure 4. A sample ‘‘where’’ axis with three class hierarchies.

The shaded nodes show the current focus. The oval objects are defined in the W^3 Search Model. The triangles are static based on the application, and the rectangles are dynamically (runtime) identified. The Paradyn resource hierarchies include several other classes, such as I/O, Memory *and* Process, *which are not shown.*

Different components of the ‘‘where’’ axis may be created at different times. Some nodes are defined statically, some when the application starts, and others during the application’s execution. The root of each resource hierarchy is statically defined.

The W^3 Search Model can also represent resources specific to high-level parallel programming languages, by representing each high-level language abstraction with its own ‘‘where’’ axis. A language-specific axis contains only those resource hierarchies that correspond to resources found in that language. For example, a data-parallel Fortran ‘‘where’’ axis would include a data-parallel array hierarchy.

A high-level language resource may map to collections of resources at the base level. Therefore, W^3 maintains mappings between resources in different ‘‘where’’ axes. For example, W^3 builds mappings between data-parallel arrays (which are understood by programmers) and the details of related message communication (which most programmers would like to ignore). The mappings allow us to translate between a focus at a higher-level to the corresponding focus at a lower-level.

Separating programming abstractions into different ‘‘where’’ axes allows searches to concentrate on one abstraction at a time. This allows tools that use W^3 to take advantage of language abstractions to explore the search space. Furthermore, if a performance problem cannot be refined within a high-level abstraction, the programmer may peel back layers of abstraction and continue the search at a lower level. The low-level search starts with the focus generated by applying the mappings to the current higher-level focus.

4.1.3. The ‘‘When’’ Axis

The third performance question programmers may ask is ‘‘at what time did my application run slowly?’’ Programs have distinct phases of execution and the ‘‘when’’ axis represents periods of time during which different types of performance problems can occur. For example, a simple program might have three phases of execution: initialization, computation, and output. Within a single phase of a program, its performance tends to be uniform. However, when a program enters a new phase, its behavior, and therefore its performance problems, can change radically. As a result, decomposing a program’s execution into phases provides a convenient way for programmers to understand the performance of their program.

Searching along the ‘‘when’’ axis involves testing the hypotheses for a focus during different intervals of time during the application’s execution. A full description of searching the ‘‘when’’ axis is beyond the scope of this paper.

4.2. The Performance Consultant

The Performance Consultant module of Paradyn discovers performance problems by searching through the space defined by the W^3 Search Model. The ability to automatically search for performance problems is a key feature of the Performance Consultant. Refinements are made across the ‘‘where’’, ‘‘when’’, and ‘‘why’’ axes without requiring the user to be involved. We determine a list of possible refinements by considering the children of the current nodes along each axis, then order this list using internally-defined hints. Finally, we select one or more refinements to try from the ordered list. If a selected refinement is not true, we consider the next item from the ordered refinement list. Paradyn will conduct a fully automatic search, allow the user to make individual manual refinements to direct the search, or combine these two methods.

Feedback about the search process currently underway is provided by the *Search History Graph* (SHG). The Search History Graph records refinements considered along the ‘‘why’’, ‘‘where’’, and ‘‘when’’ axes, and the result of testing the refinements. Figure 6 shows an actual SHG display from one of our sample application programs. Each node in the graph represents a single step in the overall search process, which is a refinement along one of the three axes. The nodes are colored according to the current state of the particular hypothesis it represents: currently being tested (pink), tested true (blue), tested false (green), or never tested (orange). The arcs indicate refinements, and are color-coded according to the particular axis along which the refinement was made: refinements along the ‘‘why’’ axis in blue and refinements along the ‘‘where’’ axis in purple. The node label indicates a particular node of an axis being explored; for example, refinements considered from the root node include a node from the ‘‘why’’ axis for cpuBottleneck. Because each step of the search is limited to a single refinement, the complete focus in the search space represented by any node can be determined by reading along a path from the root node to the node under consideration. The SHG is useful because it represents refinements that were made, those that were tried and rejected, and those that were possible but not tried. The current path of exploration can easily be determined by following the blue (‘‘true’’) nodes from the root to a leaf. The display options in Figure 6 de-emphasize other node types by using a smaller font size. If you follow the blue nodes in Figure 6, you see the search discovered that of the program was CPU bound in procedure PostCallBookwork (in module X_noncom_new.C) for machine partition mendota.

5. OPEN VISUALIZATION INTERFACE

Paradyn provides a simple library and remote procedure call interface to access performance data in real-time. Visualization modules (*visi’s*) in Paradyn are external processes that use this library and interface. All performance visualizations are implemented as visi’s. Paradyn currently provides visi’s for time-histograms (‘‘strip plots’’), bar charts, and tables; examples of these displays are given in the next section. It is not difficult to build a visi to provide data to commercial data visualization packages such as AVS [9], or incorporate the visualization displays of systems such as Paragraph [2] or Pablo [8]. The visi interface and library also can provide performance data for other uses, such as evaluating performance predicates for application steering, or logging performance data for experiment management.

After the user selects a visi from the menu, Paradyn provides the menus to select the foci (program components) and metrics to display. The visi is then started and sent the initial list of foci and metrics to display. When the visi needs to add or delete foci or metrics, it calls a procedure in the visi library, which then handles the menus and selection. The result is that the visi is isolated from the details of Paradyn’s internal structure and user interface.

The selection of a list of performance metrics for a list of foci can most easily be pictured as a two dimensional array – basically, a table. The visi library provides a C++ class, called the ‘‘DataGrid’’, that is the visi programmer’s interface to performance data. The DataGrid appears to the visi programmer as an array; the array is indexed using a metric and a focus ID. Each element of the DataGrid can be either a single value, representing the current, maximum, or average value, or can be a time-histogram, representing the time-varying behavior of the metric.

When a visi requests performance data from Paradyn, that request is sent to the Data Manager. If the requested data is already being collected, the Data Manager will send the current values to the visi, and provide continuous updates as additional data is collected. If the requested data is not being collected, the Data Manager will ask the Instrumentation Manager to start collecting it. When the visi no longer needs the data (and if no other part of the system is also using it), then instrumentation for that data will be removed.

A visi accesses performance data in the DataGrid by using standard C++ (overloaded) array operations. Optionally, the visi library can notify the visi when potentially interesting events occur. These events include the arrival of new data samples, the disabling of some currently selected focus-metric pair, a ‘‘fold’’ event associated with the time histograms, or a new phase being created on the ‘‘when’’ axis. The visi programmer can register a callback procedure for any of these events. For example, in a time histogram display, a fold event means that the curves being displayed should be redrawn, doubling the time interval on the x-axis; a tabular display (displaying single values for each focus-metric) can ignore these events.

6. EXAMPLES OF USE

We have used Paradyn to study several parallel, distributed, and sequential applications. In this section we demonstrate Paradyn's basic features and show how programmers use Paradyn to find performance problems, and illustrate those problems with visualization displays. We draw our examples from measurements of two real applications, a graph coloring program based on a branch-and-bound search, and a linear programming optimization code that uses a domain decomposition method. Both application programs were written by people outside of the Paradyn project and were intended to solve real application problems; both ran on a TMC CM-5 in a 32 node partition.

When we ran these applications with Paradyn, the Performance Consultant was able to discover and isolate a CPU bottleneck in the coloring application, and to discover multiple problems (two synchronization problems and a CPU bottleneck) in the linear programming code. For each application, we show Performance Consultant displays that illustrate each performance problem.

Figure 5 shows Paradyn's main window. At this stage, the programmer has started their application and Paradyn is displaying a ‘‘where’’ axis (only a few of the resource trees and a few nodes from each of

-- --

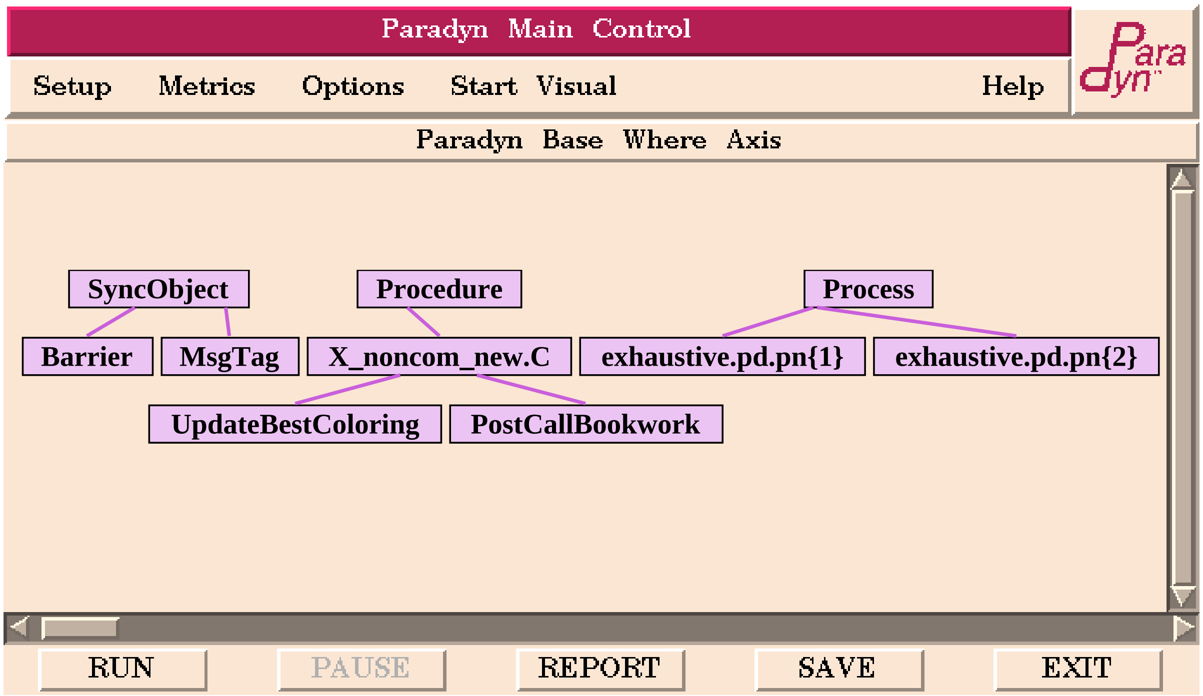

Figure 5. The Main Paradyn Control Window

these trees have been selected for display). Paradyn is ready to accept user commands to control the application, display visualizations of performance data, or invoke the Performance Consultant to find bottlenecks. The user may start or stop the application as many times as desired during execution. Stopping the application stops the flow of data to visualizations and also stops the Performance Consultant.

The ‘‘where’’ axis display shows each resource hierarchy. To select a particular focus, a user selects up to one node from each resource hierarchy. For example, if the user selects the root of the Procedure hierarchy, and a leaf node in the Machine hierarchy, then they have requested all procedures on a particular machine. To display a visualization of a metric for a focus, a user simply selects a focus in the ‘‘where’’ axis display and selects a visualization from the **Start Visual** menu. Paradyn will then prompt the user for a list of metrics and will start the visualization.

Alternate high-level language ‘‘where’’ axis views are displayed in separate windows (not shown). Paradyn uses static and dynamic mapping information to map each abstract focus to the base view. When the user selects a focus in an abstract view, Paradyn automatically highlights the corresponding resources in the base view.

Typically, users start the Performance Consultant on an automated search, and wait for the Paradyn to find a performance bottleneck. When the Performance Consultant is running it displays a window similar to the one shown in Figure 6. The top row of the Performance Consultant window contains pull down menus for display configuration. The middle area reports the status of the search (such as a description of the current bottleneck, an indication that a previously true bottleneck is no longer present, or notice that a new set of refinements is being considered). The largest area is a display of the Search History Graph (SHG). Nodes in the graph are colored to indicate the state of their corresponding hypotheses, as discussed in section 4.2. Nodes are added to the SHG as new refinements are made, and change color to reflect the current state of the search for bottlenecks. The bottom of the window contains buttons for controlling the search process.

6.1. Graph Coloring Application

Our first example demonstrates Paradyn's analysis of a graph coloring program called ''match-maker''. Match-maker is a branch-and-bound search program with a central manager that brokers work to idle processors. It uses CMMD, the CM-5 explicit message passing library. The program is written in C++ and contains 4,600 lines of code in 37 files.

The Performance Consultant discovered an initial CPU bottleneck in match-maker after 10 seconds of execution. Figure 6 shows some of the hypotheses and foci considered by the Performance Consultant. Starting from the root node, the Performance Consultant considered several types of bottlenecks (synchronization, I/O, CPU, virtual memory, and instrumentation). At this point, it identified a CPU bottleneck. At the next step, it considered refinements to the CPU bottleneck and confirmed that the program was CPU bound. Next, the Performance Consultant refined the CPU bottleneck to a specific module in the program, X_noncom_new.C. The Performance Consultant then isolated the bottleneck to the procedure PostCallBookwork in that module. Since the problem was diffused across all the processes, the Performance Consultant could not further refine the bottleneck.

We then displayed a visualization of the bottleneck in the graph coloring application with a time histogram display of CPU time for procedure PostCallBookwork and for the whole program. The time histogram display in Figure 7 verifies that PostCallBookwork was responsible for a large percentage of the application's CPU time.

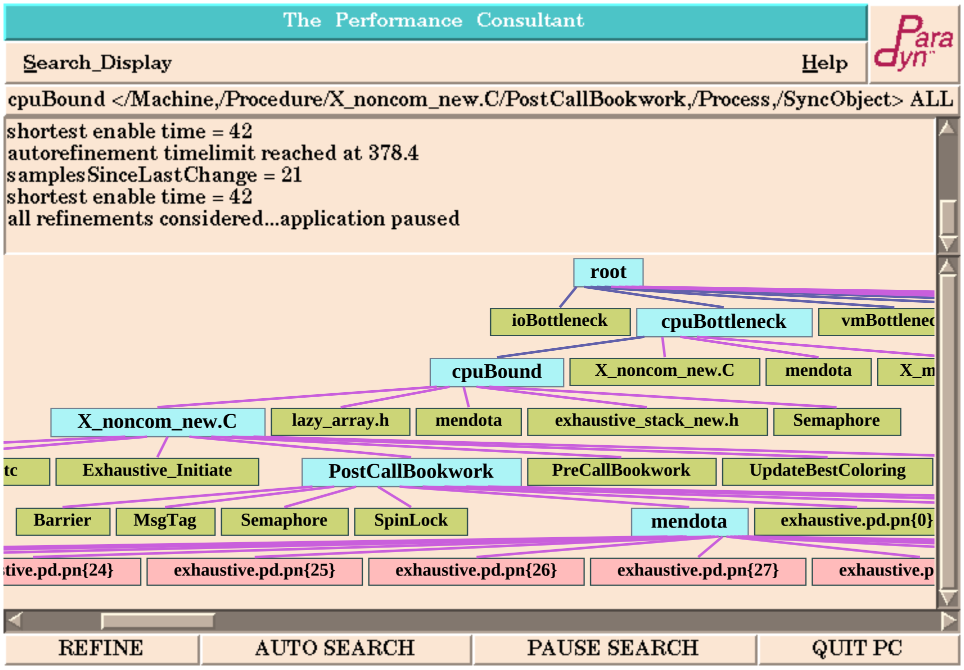

Figure 6. Performance Consultant Search for the Graph Coloring Application

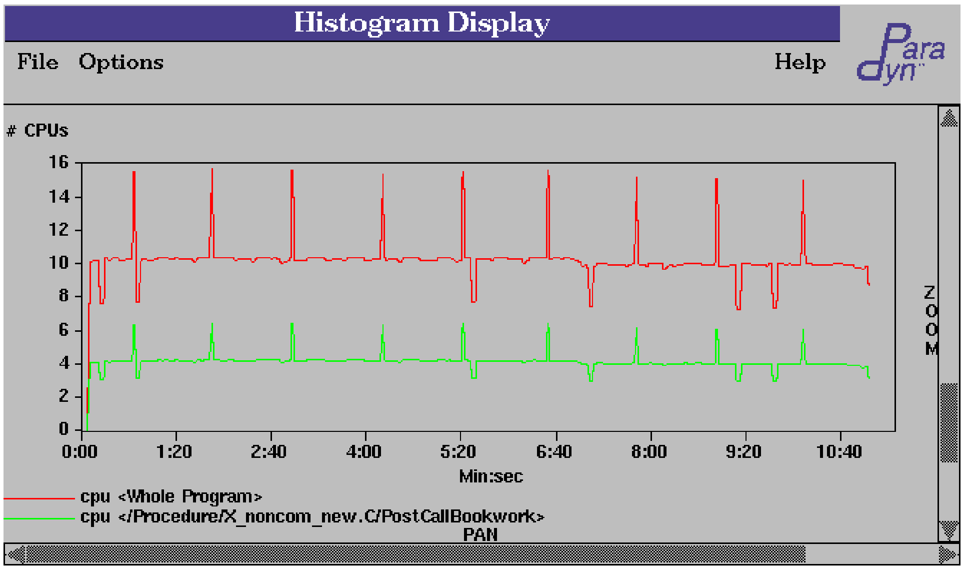

Figure 7. Visualization of Bottleneck in Graph Coloring Application

6.2. Message Passing Optimization Application

Our second application, called ‘‘msolv’’, uses a domain decomposition method for optimizing large-scale linear models. The application consists of 1,793 lines of code in the C programming language, and makes use of a sequential constrained optimization library package called Minos. Uninstrumented, msolv runs for 1 hour 48 seconds on a 32-node CM-5 partition.

Paradyn found three bottlenecks in this program. First, during an initialization phase, it found a synchronization bottleneck as the nodes initialized. This bottleneck lasted less than one minute, and was not further refined by the Performance Consultant. Second, it found a CPU bottleneck in the module minos_part.c during an initial computation phase. Third, the Performance Consultant located a key synchronization bottleneck that persisted for the rest of the program's execution. The Search History Graph of the isolation of this third bottleneck appears in Figure 8.

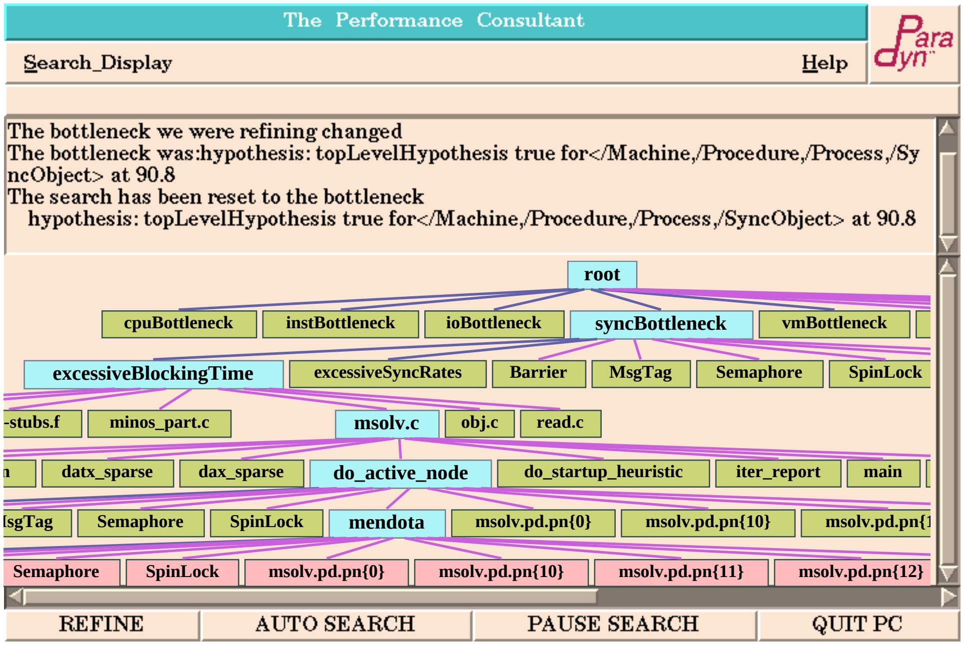

Figure 8. Performance Consultant Search for the Msolv Application

The Performance Consultant made five refinements while locating the synchronization bottleneck. First, it discovered a synchronization bottleneck in the program. Second, it identified that the bottleneck was due to excessiveBlockingTime (as opposed to many short synchronization operations performed too frequently). Third, it isolated the synchronization bottleneck to the file msolv.c. Fourth, it refined the bottleneck to the procedure do_active_node. Finally, it (trivially) isolated the problem to the single partition used. The partition refinement is labeled mendota, the name of the partition's manager. The Performance Consultant attempted to isolate the bottleneck to a particular processor node, but the refinement failed because the bottleneck was diffused across all the processors.

To gain a better understanding of the synchronization problem, we displayed a bar chart showing the amount of time spent synchronizing by the application. The display in Figure 9 shows the synchronization time of the whole program, and the synchronization time spent in each node. The display shows

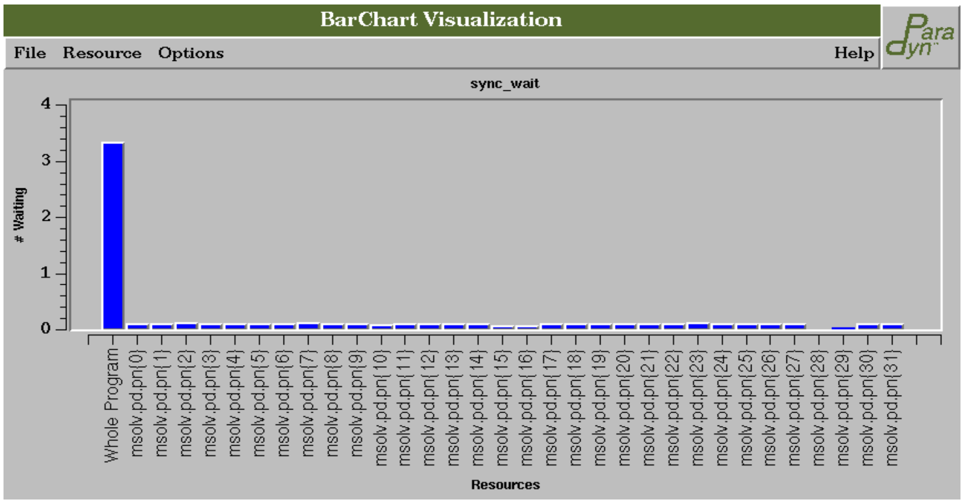

Figure 9. Bar Chart showing Msolv synchronization time

that the synchronization bottleneck was diffused across all the nodes and could not be refined further.

7. CONCLUSIONS

The Paradyn parallel performance measurement tools incorporate several novel technologies. Dynamic instrumentation offers the chance to significantly reduce measurement overhead, and the W^3 Search Model, as embodied in the Performance Consultant, provides the means to control the instrumentation. The synergy between these two technologies results in a performance tool that can automatically search for performance problems in large-scale parallel programs. Paradyn's support for high-level parallel languages lets programmers study the performance of their programs using the native abstractions of the language. In addition, we provide detailed, time-varying data about a program's performance. As a result, programmers with large applications can use Paradyn as easily as someone with a small prototype application. Uniform data abstractions, such as the metric-focus grid and time histogram, allow simple interfaces within Paradyn and provide easy-to-understand interfaces to the program.

While Paradyn is a working system, there remain many directions for growth. Over the next few years, we will be expanding to new machine environments (such as the Cray T3D), new high-level languages (such as HPF and sparse matrix languages), and new problem domains (such as application steering).

8. ACKNOWLEDGMENTS

The authors thank Sherry Frizell who initially implemented the bar chart visualization; we also thank the authors of the applications used in our study, Gary Lewandowski (Graph Coloring), and Spyros Kontogiorgis (Msolv).

9. REFERENCES

1. T. Ball and J. R. Larus, "Optimally Profiling and Tracing Programs", *19th ACM Symposium on Principles of Programming Languages*, Albuquerque, NM, January 19-22, 1992, pp. 59-70.
2. M. T. Heath and J. A. Etheridge, "Visualizing Performance of Parallel Programs", *IEEE Software 8*, 5 (Sept 1991), .
3. J. K. Hollingsworth, R. B. Irvin and B. P. Miller, "The Integration of Application and System Based Metrics in A Parallel Program Performance Tool", *1991 ACM SIGPLAN Notices Symposium on Principles and Practice of Parallel Programming*, April 1991, pp. 189-200.
4. J. K. Hollingsworth, B. P. Miller and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools", *1994 Scalable High-Performance Computing Conf.*, Knoxville, Tenn., 1994.
5. J. K. Hollingsworth and B. P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems", *7th ACM International Conf. on Supercomputing*, Tokyo, July 1993, pp. 185-194.
6. *RISCompiler Languages Programmer's Guide*, MIPS Computer Systems, Inc., December, 1988.
7. J. K. Ousterhout, "An X11 Toolkit Based on the Tcl Language", *Proc. USENIX Winter Conference*, January 1991.
8. 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, in *Scalable Parallel Libraries Conference*, IEEE Computer Society, 1993.
9. C. Upson, T. F. Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz and A. Dam, The Application Visualization System: A Computational Environment for Scientific Visualization, Vol. 9, July 1989.