Chapter 1
The Paradyn Parallel Performance Tools and PVM*

Barton P. Miller[†] Jeffrey K. Hollingsworth[‡] Mark D. Callaghan[§]

Abstract

Paradyn is a performance tool for large-scale parallel applications. By using dynamic instrumentation and automating the search for bottlenecks, it can measure long running applications on production-sized data sets. Paradyn has recently been ported to measure native PVM applications.

Programmers run their unmodified PVM application programs with Paradyn. Paradyn automatically inserts and modifies instrumentation during the execution of the application, systematically searching for the causes of performance problems. In most cases, Paradyn can isolate major causes of performance problems, and the part of the program that is responsible the problem.

Paradyn currently runs on the Thinking Machine CM-5, Sun workstations, and PVM (currently only on Suns). It can measure heterogeneous programs across any of these platforms.

This paper presents an overview of Paradyn, describes the new facility in PVM that supports Paradyn, and reports experience with PVM applications.

1 Introduction

Performance monitoring creates a dilemma: identifying a bottleneck necessitates collecting detailed information, yet collecting all this data can introduce serious data collection bottlenecks. At the same time, users are being inundated with volumes of complex graphs and tables that require a performance expert to interpret. Paradyn takes a new approach that addresses both these problems by combining dynamic on-the-fly selection of what performance data to collect with decision support to assist users with the selection and presentation of performance data. The approach is called the W^3 *Search Model.* We have also developed a new monitoring technique for parallel programs called *Dynamic Instrumentation.*

This paper presents an overview of Paradyn, describes the new facility in PVM that supports Paradyn, and reports experience with PVM applications. The initial implementation of the Paradyn Performance Tools was for a Thinking Machines CM-5. We have ported the Paradyn tools to a network of workstations running PVM[2]. To do this we took advantage of the new features of PVM 3.3 that permit external tools to easily integrate with PVM. This effort was the first port of the Paradyn tools, and one of the first users of this new interface to PVM.

[*] This research supported in part by Department of Energy grant DE-FG02-93-ER25176, Office of Naval Research grant N00014-89-J-1222, and National Science Foundation grants CCR-9100968 and CDA-9024618. Hollingsworth is supported in part by an ARPA Graduate Fellowship in High Performance Computing.

[†] University of Wisconsin, Madison

[‡] University of Maryland, College Park

[§] University of Wisconsin, Madison

2 Overview of the Paradyn System

The Paradyn Performance Tools Suite is a modular collection of tools to measure and understand the performance of large-scale parallel programs. It is designed for the runtime monitoring and analysis of programs. Paradyn consists of a flexible data collection facility, a tool for the automatic isolation of performance bottlenecks, and an open visualization interface. In addition, several performance visualizations are provided. Before describing how the Paradyn tools work together with PVM, we briefly review each of the components of the Paradyn system.

2.1 The W^3 Search Model

For a given parallel program, the amount of performance data that might be useful to understand its performance can be huge. However, in practice a small amount of information is often sufficient to reveal the key bottlenecks. Performance debuggers exist to help programmers find the gems of understanding among the large space of available performance data. In this section we review the W^3 Search Model[4], a system that provides a structured way for programmers to quickly and precisely isolate a performance problem without having to examine a large amount of extraneous information. It is based on answering three separate questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. By iteratively refining the answer to these three questions, we can give programmers a precise description of why their program is not performing as expected. To deliver answers rather than just posing the questions, we automate this search process.

The first performance question most programmers ask is “why is my application running so slowly?” To answer this question we need to consider what types of problems can cause a bottleneck in a parallel program. We represent these potential bottlenecks with *hypotheses* and *tests*. Hypotheses represent the fundamental types of bottlenecks that occur in parallel programs independent of the program being studied. For example, a hypothesis might be that a program is synchronization bound. Tests are boolean functions that indicate if a program exhibits a specific performance behavior related to the hypothesis. They are expressed in terms of thresholds that indicate when a test should evaluate to true (e.g., more than 20% of the time is spent waiting for synchronization).

Hypotheses can have other hypotheses as pre-conditions. The dependence relationships between hypotheses define the search hierarchy for the “why” axis. Figure 1 shows a partial “why” axis hierarchy with the current hypothesis being that the application has a **HighSyncBlockingTime** bottleneck. This hypothesis was reached by first concluding that a **SyncBottleneck** exists in the program.

By searching along the “why”, axis we classify the type of problem in a parallel application; to fix the problem, more specific information is required. For example, knowing that a program is synchronization bound suggests we look at the synchronization operations, but a large application might contain hundreds or thousands of these operations. We must also find which synchronization operation is causing the problem. To isolate a bottleneck to a specific resource, we search along the “where” axis.

The “where” axis is formed by a collection of logically independent resource hierarchies. Resource hierarchies include: synchronization objects, source code, threads, processes, processors, and disks. There are multiple levels in each hierarchy, and the leaf nodes are the instances of the resources used by the application. Searching the “where” axis is iterative and consists of traveling down each of the individual resource hierarchies.

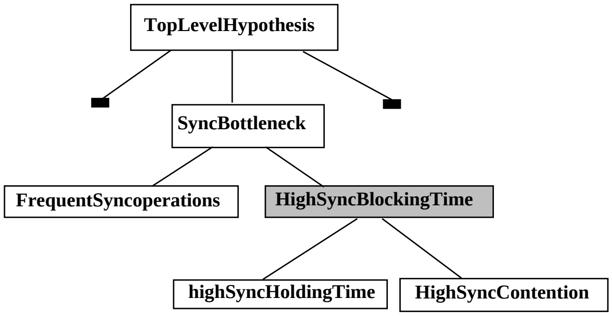

Fig. 1. *A Partial "Why" Axis.*

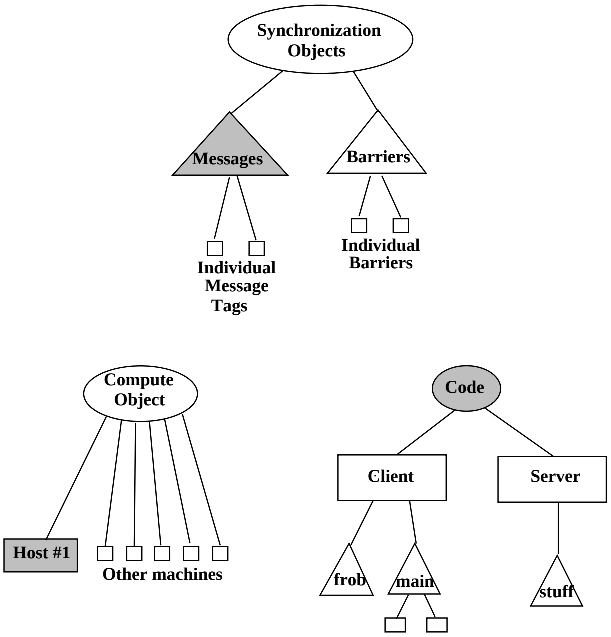

Fig. 2. *A Partial "Where" Axis.*

The top tree in Figure 2 shows a sample resource hierarchy. The root of the hierarchy is **Synchronization Objects**. The next level contains two types of synchronization (**Messages** and **Barriers**). Below the **Barriers** node are the individual barriers used by the application. The children of the **Messages** node are the types of messages (tags) used.

The current status of the search along the "where" axis is called the *focus*, and consists of the current state of each resource class hierarchy. Figure 2 shows a sample "where" axis containing three resource hierarchies. The highlighted nodes show the current focus component of each hierarchy. The focus shown in the figure is all **Messages** on **Host #1** used in any procedure.

Programs in general, and especially parallel programs, have distinct phases of execution. 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 similar. However, when it enters a new phase, behavior of the program can change radically. When a program enters a new phase of execution, its performance bottlenecks also change. As a result, decomposing a program's execution into phases provides a convenient way for programmers to understand the performance of their program.

The third component of the W^3 Search Model is the "when" axis. The "when" axis is a way for programmers to exploit the phase behavior of their programs to find performance bottlenecks. Searching along the "when" axis involves testing the current hypotheses for the current focus during different intervals of time during the application's execution. For example, we might choose to consider the interval of time when the program is doing initialization.

A key component of the W^3 Search Model is its ability to automatically search for performance bottlenecks. This automation is accomplished by making refinements across the "where", "when", and "why" axes without requiring the user to be involved. Automated refinement is exactly like manual (user directed) searching, and hybrid combinations of manual and automated searching are possible.

2.2 Dynamic Instrumentation

Data collection is a critical problem for any parallel program performance measurement system. To understand the performance of parallel programs, it is necessary to collect data for full-sized data sets running on large numbers of processors. However, collecting large amounts of data can excessively slow down a program's execution, and distort the collected data. A variety of different approaches have been tried to efficiently collect performance data. Two common approaches are event tracing and statistical sampling. Both of these techniques have limitations in either the volume of data they gather or granularity of data collected. Paradyn takes a new approach to data collection, called *dynamic instrumentation*[3] that defers instrumenting the program until it is in execution. This approach permits dynamic insertion and alteration of the instrumentation during program execution. We also describe a new data collection model that permits efficient, yet detailed measurements of a program's performance.

To meet the challenges of providing an efficient, yet detailed instrumentation we needed to make some radical changes in the way traditional performance data collection has been done. However, since we wanted our instrumentation approach to be usable by a variety of high level tools, we needed a simple interface. The interface we developed is based on two abstractions: *resources* and *metrics*. Resources are similar to nodes in the "where" axis of the W^3 Search Model. Metrics are time varying functions that characterize some aspect of a parallel program's performance. Metrics can be computed for any subset of the resources in the system. For example, CPU utilization can be computed for a single procedure executing on one processor or for the entire application.

A key question about any performance data gathering system is what data is collected and how is it collected. We have developed a new data collection model that combines the advantages of tracing and sampling. Tracing inserts instrumentation to generate a log of all of the interesting events during a program's execution; logged events generally include inter-processor communication and procedure invocations. Trace-based systems can collect detailed information about specific events during a program's execution. However, they can generate vast amounts of data that are difficult to manage. A second approach to collecting performance data is to summarize interesting information as counts and times that are reported at the end of program execution. Using summary counters and timers greatly reduces the volume of performance data collected; however summary data loses important temporal information about usage patterns and relationships between different components. For example, it is impractical to collect performance data about how each procedure uses different synchronization variables on each processor for a large parallel machine. Instead, Paradyn dynamically modifies the program to record precise information about relevant state transitions in counter and timer data structures. These structures are then periodically sampled to report performance information to the higher layers of our system. Periodic sampling of these structures provides accurate information about the time varying performance of an application without requiring the large amount of data needed by full tracing.

By using simple counters and timers, we are also able to easily integrate performance data from external sources. For example, most operating systems keep a variety of performance data internally: I/O counts, virtual memory statistics, and CPU time. Usually, this information can be made available to user processes by reading kernel data structures. Also, several machines provide hardware based counters that are a source of useful performance information. For example, the Power2[6], Cray Y-MP[1], and Sequent Symmetry[5] systems provide detailed counters of cache utilization, floating point operations, and bus contention. We can combine external information with direct instrumentation to get precise information to relate the external events back to specific parts of the program. For example, if we have a counter of page faults by a single process, we can read this counter before and after a procedure call to compute the number of page faults taken by that procedure.

Dynamic Instrumentation is a new approach to data collection that permits customized instrumentation to be inserted at runtime. We also use a hybrid of sampling and tracing that permits efficient collection of intermediate (temporal) values of counters and timers without having to do full event tracing.

2.3 Visualization Interface

Performance Visualizations are useful to explain the performance of a parallel program; but developing visualizations is not a primary goal of the Paradyn project. However, a number of visualization tools have been built, and we would like to be able to use these tools as part of a Paradyn session. To do this, we have created an Application Programmer Interface (API) to permit the performance data gathered by Paradyn to be exported to visualization tools.

The interface is a simple grid (array) that permits visualization tools to request data for any combination of resources and metrics in the system. In addition, data can be requested in either summary (single value) form or as a time series (time histogram). Since Paradyn is designed to work during program execution, the Visualization Interface also needs to be able to provide timely delivery of performance data to visualizations. This is accomplished by delivering data to all components of the data grid for a single time step at once.

3 Porting Paradyn to Work with PVM

The Paradyn Performance Tools are divided into three parts: the Paradyn controller, the Paradyn daemon, the application processes. Figure 3 displays their relationship. The Paradyn controller performs automated searches for performance bottlenecks and supports the user interface to the rest of the Paradyn system. The Paradyn daemon acts as an intermediary between the Paradyn controller and the application processes, and each application process is controlled by one Paradyn daemon. The operating system and machine specific dependencies have been isolated to the Paradyn daemon. Only the Paradyn daemon had to be changed to enable the Paradyn system to work in the PVM environment. The application processes contain the dynamically inserted instrumentation

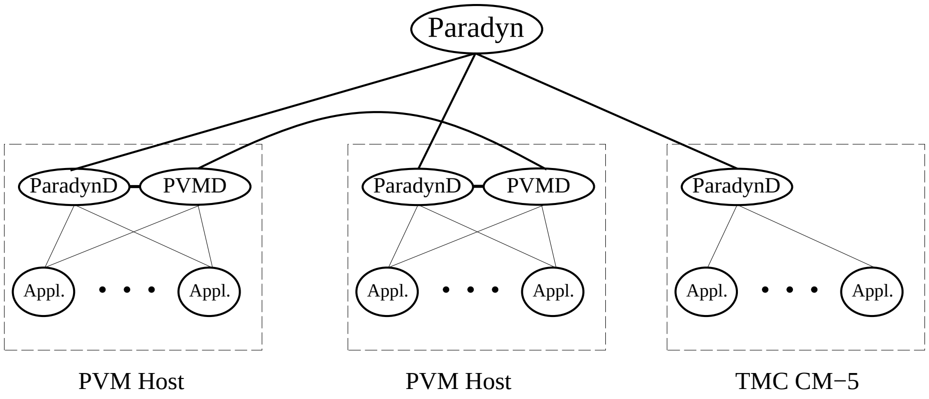

Fig. 3. *Structure of Paradyn and PVM.*

and periodically report the values of metrics.

Restricting the machine and programming environment specific code in Paradyn to the daemon enables the controller to work with any daemon that supports the Paradyn controller interface. As an intermediary, the Paradyn daemon instruments the application processes when requests from the controller process are received, and forwards performance data from the application process to the controller process.

3.1 Interface

The interface between the Paradyn controller and daemon supports three main functions: process control, performance data delivery, and performance data requests. The daemon services the process control requests and performance data requests from the controller. The daemon also delivers performance data to the controller in response to prior performance data requests.

Process control functions include the ability to start, stop, pause, continue, and write to the address space of application processes. Not only is the ability to start application processes required, but the Paradyn daemon must also intercept all process start requests (e.g., *pvm_spawn* calls) made by the application. There are two reasons for this requirement. First, the Paradyn controller must be aware of all processes that make up a parallel application. Second, UNIX ptrace system calls are used to insert instrumentation into the application; therefore, the Paradyn daemon must be the parent of the application process, unless an attachable version of ptrace is provided by the machine vendor.

To deliver performance data, the Paradyn daemon forwards data it receives from the application processes to the Paradyn controller. Performance data for a particular resource and metric combination might require data to be gathered from several application processes. In this case, the data from each application process must be combined to form the desired aggregate data. Data aggregation could be done either by the daemon or the controller. We choose to aggregate data in the daemon when possible. When more than one daemon forwards data to the controller for the same resource and metric combination, the data is aggregated by the controller. Data aggregation by the daemon reduces message traffic to the controller.

Requests for performance data may cause the Paradyn daemon to instrument the processes that it manages. The request for performance data uses a metric and a resource as parameters. The daemon translates this request to machine language instructions that are written into the address space of the application process. Using the Paradyn daemon as an intermediary allows the controller to maintain a high level, machine independent view of performance data.

3.2 Implementation

The implementation of the interface between the Paradyn daemon and the application process is machine and parallel programming model specific. It takes advantage of new features of PVM version 3.3. PVM allows processes to register with PVM to provide services that are by default provided by PVM. These services include starting new application processes and starting new PVM daemons. A process that registers with its local PVM daemon to start processes is called the *tasker*. A process that registers with its local PVM daemon to start new PVM daemons is called a *hoster*. There is only one hoster process at any given time within a given Parallel Virtual Machine. Providing these open interfaces in PVM has increased its utility by enabling it to work in close coordination with Paradyn (and other similar tools).

Supporting process control functions in the PVM-Paradyn daemon required significant changes whereas supporting data delivery and instrumentation did not. The hoster and tasker features allow the PVM-Paradyn daemon to start all processes for PVM. We assume that PVM daemons are running when a PVM-Paradyn daemon is started. The first PVM-Paradyn daemon queries PVM to determine all hosts that have a PVM daemon, and starts a PVM-Paradyn daemon on each of those hosts. Each PVM-Paradyn daemon registers with the local PVM daemon as the tasker for that PVM daemon. The first PVM-Paradyn daemon also registers as the hoster. The PVM-Paradyn daemon now performs all process starts for the PVM environment. In addition, the hoster starts a new PVM-Paradyn daemon on any machine that is added to the Parallel Virtual Machine.

The following sequence of steps occurs when a process is started by a PVM-Paradyn daemon.

1. The PVM-Paradyn daemon receives a start task message from PVM;

2. The PVM-Paradyn daemon starts the application process after creating a pipe for communication;

3. The PVM-Paradyn daemon notifies the controller that a new application process has been started;

4. The application process connects to its local PVM daemon; and

5. The PVM-Paradyn daemon informs the PVM daemon that the start was successful.

The code that instruments application processes required some PVM specific changes. It must know how to translate instrumentation requests from the Paradyn controller to machine instructions. An example is the case in which the Paradyn controller wants to determine how many messages have been sent in an application process. The Paradyn daemon must determine the points in an application process at which messages are sent, and instrument these locations to increment a counter. The PVM-Paradyn daemon is required to know which PVM functions send messages. The instrumentation is inserted into the application process using ptrace system calls.

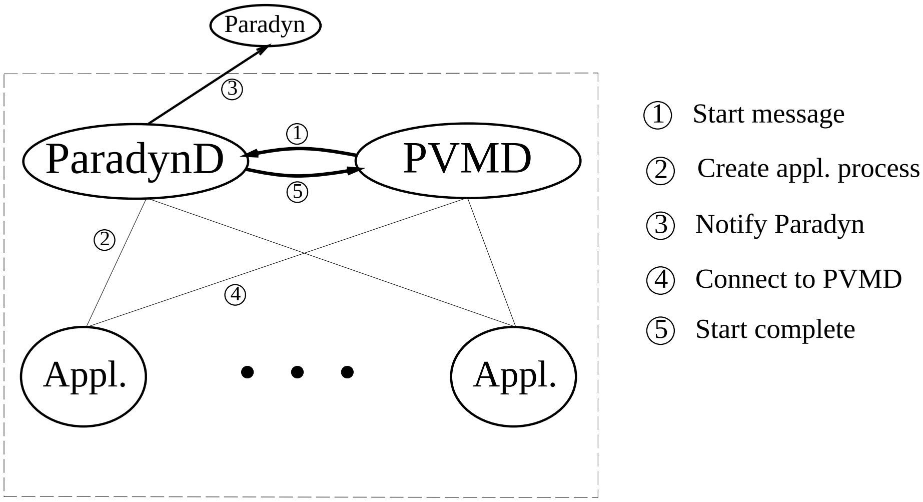

Fig. 4. *Starting a new Process with Paradyn and PVM.*

3.3 Summary

The design of the Paradyn system was driven by the principle of the separation of the interface from the implementation. A well-defined interface for communication between the Paradyn controller and the Paradyn daemon allowed the development of a Paradyn controller that contains no machine or parallel programming environment specific code. All parallel programming specific code was contained in the Paradyn daemon. The new features of PVM 3.3, the hoster and tasker functionality, enabled the Paradyn system to be ported to the PVM environment.

4 An Example of using Paradyn with PVM

The PVM-Paradyn daemon has been used to find performance bottlenecks in PVM applications. In this section, we describe using Paradyn on a simple PVM application. The processes in the application were organized as a ring, and a token message was passed around the ring. There were three SPARC workstations in the Parallel Virtual Machine, and four processes in the application. All messages sent after initialization used the same message tag. Paradyn dynamically determines the message tags that have been used and is capable of making the instrumentation in a message send function dependent on the message tag used in the message send.

Paradyn used this ability to refine the performance bottleneck. The results of the search are displayed in Figure 5. The white nodes in the graph represent the current bottleneck. Paradyn initially refined along the "why" axis and found that **syncBottleneck** evaluated to true. A **syncBottleneck** means that the Paradyn test criteria for a synchronization bottleneck has been satisfied. The initial bottleneck was further refined to **excessiveBlockingTime**. This means that the bottleneck with the synchronization operations is due to too much time spent waiting for the operations to execute (as opposed to too fine grained of synchronization). Paradyn refined along the "where" axis when no further "why" axis refinements could be made. It determined that excessive blocking time was being spent on message receives (**MsgTag**), and it refined this bottleneck to a specific message tag (**4**). Paradyn next determined that the bottleneck was in the process **spmd**{**1**}. Finally, Paradyn determined that the bottleneck was in a specific procedure, **dowork** on the machine named **poona**. Since

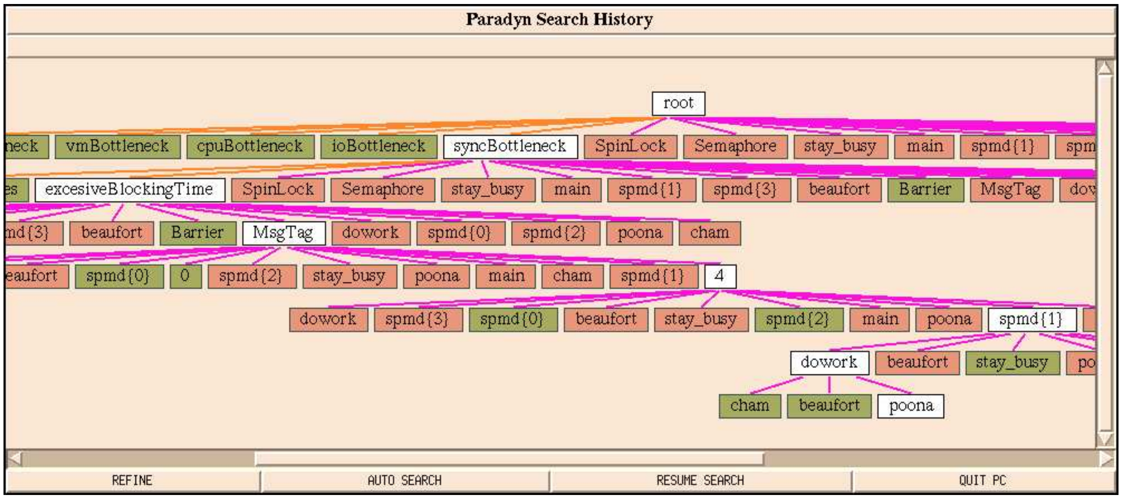

Fig. 5. *A Sample Session of the Performance Consultant.*

all of the processes used the same executable file, Paradyn could have chosen any of the processes and any of the machines. At this point no further refinements can be made since Paradyn has refined along the “why” axis and each component of the “where” axis.

The bottleneck that Paradyn had found is further illustrated by a visualization that is supported by the Paradyn controller. This is shown in Figure 6. In this figure, the amount of time spent in synchronization operations for the **process spmd{1}** on machine **poona** is displayed along with the CPU utilization for **process spmd{1}** on machine **poona**. The visualization shows that the majority of time is spent waiting for synchronization operations to complete. For the selected process and machine, the vast majority (greater than 90%) of the time is spent waiting for synchronization operations to complete.

This particular visualization is a time histogram. Each line on the visualization represents a resource and metric combination. The visualization can display multiple metric and resource combinations and is updated in real time by data that is forwarded to it from the Paradyn controller. The x-axis represents time and the y-axis represents the metric amount. The visualization can plot any combination of metrics and resources along the y-axis.

5 Conclusions

In this paper, we described the Paradyn Parallel Performance tools. We also reported on the techniques used to port Paradyn to work with PVM applications. We illustrated how Paradyn was used to automatically isolate a performance bottleneck in a simple PVM application. Finally, we presented an example of a performance visualization that can be used in conjunction with the Paradyn system.

6 Acknowledgments

We wish to thank Jack Dongarra and Bob Manchek of the University of Tennessee for their invaluable help in defining and implementing the tasker and hoster interfaces to PVM.

References

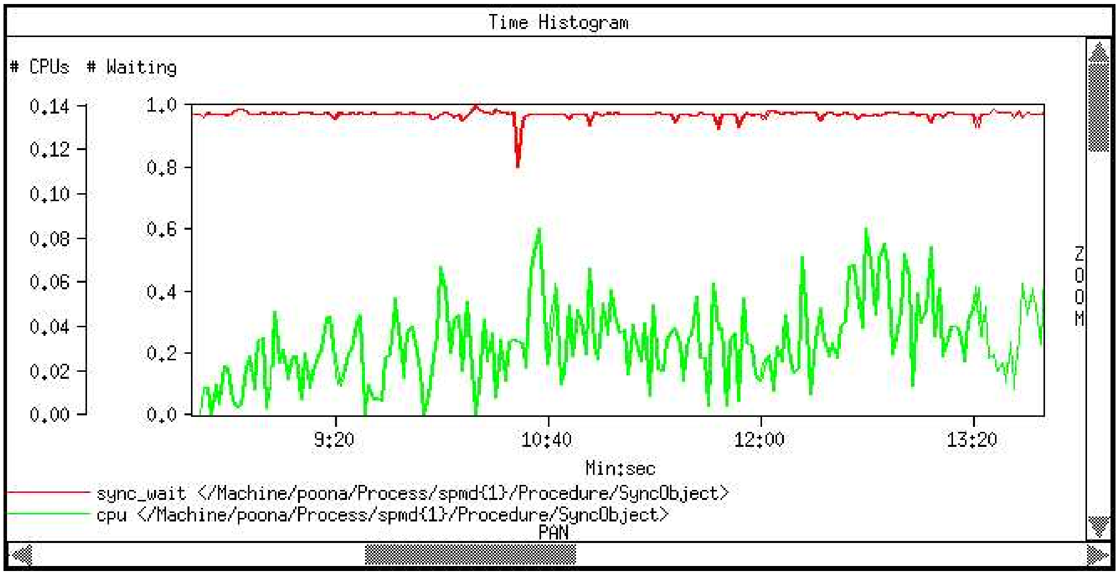

Fig. 6. *A Sample Performance Visualization.*

The graph displays synchronization waiting time and CPU utilization over time. Both metrics are shown for the **process spmd{1}** on the host **poona**.

[1] Cray Research Inc, *UNICOS File Formats and Special Files Reference Manual*, SR-2014 5.0 ed.

[2] J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam, *Integrated PVM framework supports heterogeneous network computing*, Computers in Physics, 7 (1993), pp. 166–74.

[3] J. K. Hollingsworth, J. Cargille, and B. P. Miller, *Dynamic Program Instrumentation for Scalable Performance Tools*, in Proceedings of the 1994 Scaleable Highe Performance Computing Conference, Knoxville, TN, May 1994, pp. 841–850.

[4] J. K. Hollingsworth and B. P. Miller, *Dynamic Control of Performance Monitoring on Large Scale Parallel Systems*, in Proceedings of the 7th ACM International Conference on Supercomputing, Tokyo, July 1993, pp. 185–194.

[5] S. S. Thakkar, *Performance of Parallel Applications on a Shared-Memory Multiprocessor System*, in Performance Instrumentation and Visualization, M. Simmons and R. Koskela, eds., Addison-Wesley, 1990, pp. 233–256.

[6] E. H. Welbon, C. C. Chen-Nui, D. J. Shippy, and D. A. Hicks, *The POWER2 Performance Monitor*, IBM Journal of Research and Development (submitted for publication), (1994).