IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs

Barton P. Miller
Cui-Qing, Yang

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706

ABSTRACT

We have designed an interactive tool, called IPS, for performance measurement and analysis of parallel and distributed programs. IPS is based on two main principles. First, programmers should be supplied with the maximum information about the execution of their program. This information should be available from all levels of abstraction – from the statement level up to the process interaction level. To prevent the programmer from being inundated with irrelevant details, there must be a logical and intuitive organization to this data. Second, programmers should be supplied with answers, not numbers. The performance tool should be able to guide the programmer to the location of the performance problem, and describe the problem in terms of the source program.

IPS uses a hierarchical model as the framework for performance measurement. The hierarchical model maps program's behavior to different levels of abstraction, and unifies performance data from the whole program level down to procedure and statement level. IPS allows the programmer to interactively evaluate the performance history of a distributed program. Users are able to maneuver through the hierarchy and analyze the program measurement results at various levels of detail. The regular organization of the hierarchy allows us to easily reason about the program's execution and provides information to automatically guide the programmer to the cause of performance bottlenecks. Critical path analysis, in conjunction with hierarchically organized performance metrics, is one method used to direct the programmer in identifying bottlenecks.

An initial implementation of IPS has been made on the Charlotte distributed operating system and a new implementation is currently being built on 4.3BSD UNIX.

1. INTRODUCTION

An important motivation for writing parallel and distributed programs is to achieve performance speed-up. There is a steadily increasing number of systems that support loosely-coupled or tightly-coupled parallel programming, and there is an increasing number of parallel applications. The techniques and tools needed to aid in program performance evaluation and debugging are lagging behind the development of parallel programs. In this paper, we present a performance measurement system for distributed programs that provides a broad range of performance information for evaluating the behavior of program's execution.

Our research in the area of performance measurement emphasizes two points. First, we must supply the programmer with a complete picture of a program's execution. Information must be available about all aspects of the program's behavior and the programmer should be able to view this information from different levels of abstraction. We must supply this large body of performance information in such a way that the programmer is not overwhelmed and can easily and intuitively access the information. Second, programmers need more than a tool that provides extensive lists of performance metrics; they need tools that will direct them to the location of performance problems. Performance tools must be able to summarize a program's behavior and automatically guide the programmer to the cause of performance bottlenecks. These results should be presented to the programmer in terms of the source program and able to be viewed from different levels of abstraction. A system that incorporates these ideas will provide an integrated performance system.

The basis of our performance system is to provide a regular structure to the performance data. We define a hierarchical model of the execution of distributed programs as a framework for the performance measurement. We use the hierarchy to organize performance information, provide views of performance data at different levels of abstraction, give the programmer an intuitive way to view and manipulate the data, and simplify the construction of tools that automatically reason about a program's behavior.

A hierarchical model naturally fits the way in which we construct and define distributed and parallel programs. Our hierarchy is a regular structure that reflects the semantics and organization of the program. A complete picture of the program's execution can be presented at different levels of detail in the hierarchy. An interactive interface allows users to easily traverse through the hierarchy and zoom-in/zoom-out at different levels of abstraction. Users can always concentrate on the spots in the picture where the most interesting activities have occurred and interactively shift that focus. Our efforts are aimed at integrating the performance tool with automatic guiding techniques for locating performance problems.

Much of the research on performance measurement of distributed systems and programs[1-6] shows that we can describe a program's behavior at many levels of detail and abstraction. These levels include the hardware architecture, operating system, single process, and entire program. Often, we need information from several of these levels and some way to coordinate the information received from these various levels of abstraction. Most existing performance tools work at one of two levels: those that monitor the internal activities of the processes in a program (*intraprocess level*) and those that monitor the interactions between processes (*interprocess level*). Each level provides a part of the picture of the program's performance, but neither level offers a way to combine these results for a complete picture of the program's behavior.

A more recent and promising approach is that of the PIE system[7]. PIE includes performance data from both the intra- and interprocess levels. This data is stored using a relational data model. Queries on the relational data provide an integrated view of the program's behavior. The integrated view is important to the programmer's ease of using the performance tool.

Our approach to the organization of performance data is to integrate it into a single structure as suggested in PIE, but into a regular structure that reflects the semantics and organization of the program. This regular structure should provide easy and intuitive

7th Int'l Conf on Distr Computing Systems Sept 1987

CH2439-8/87/0000/0482$01.00 © 1987 IEEE

access for performance information, straight-forward mapping onto user interfaces, and the ability to easily automate reasoning about the program's behavior. The ability to reason about a program's behavior allows us to build tools to guide the programmer to performance problems. The development of techniques for automatically guiding the programmer to locate performance problems will complement designs such as PIE. We expect the results of our research to be applied to a variety of systems.

In Section 2, we describe a sample hierarchy for distributed programs and a corresponding hierarchy for program measurement. These models unify many levels of performance data and provide the basis of our research. Section 3 briefly presents details of the pilot implementation of data collection and analysis facilities that support the hierarchical structure. Section 4 focuses on techniques for automatically guiding the programmer to locate performance problems and improve program efficiency. A summary and report of current status of IPS is given in Section 5.

2. THE PROGRAM AND MEASUREMENT HIERARCHIES

Our performance system of distributed programs, called IPS, is based on a hierarchical model of parallel and distributed computation. A hierarchical model presents multiple levels of abstraction, provides multiple views of the data, and demonstrates a regular structure. The objects in a hierarchical model are organized in well-defined layers separated by interfaces that insulate them from the internal details of other layers. Therefore, we can view a complex problem at various levels of abstraction. We can move vertically in the hierarchy, increasing or decreasing the amount of detail that we see. We can also move horizontally, viewing different components at the same level of abstraction.

In this section we present the sample hierarchy of IPS that is based on our initial target systems — the Charlotte Distributed Operating System[8, 9] and 4.3BSD Berkeley UNIX[10]. Both systems consist of processes communicating via messages. These processes execute on machines connected via high-speed local networks. The hierarchy presented here serves as a test example of our hierarchy model and reflects our current implementation. It is easy to extend these ideas to incorporate new features and other programming abstractions. For example, we can add the light-weight processes (processes in the same address space) from the LYNX parallel programming language[11] to our hierarchy with little effort. Our hierarchical structure could be also applied to systems such as HPC[12], which has a different notion of program structuring, or MIDAS[13], which has a 3-level programming hierarchy.

2.1. The Program Hierarchy

In our sample hierarchy, a program consists of parallel activities on several machines. Machines are each running several processes. A process itself consists of the sequential execution of procedures. An overview of our computation hierarchy is illustrated in Figure 1. This hierarchy can be considered a subset of a larger hierarchy, extending upwards to include local and remote networks and downward to include machine instructions, microcode, and gates.

(A) *Program Level*

This level is the top level of the hierarchy, and is the level in which the distributed system accounts for all the activities of the program on behalf of the user. At this level, we can view a distributed program as a black box running on certain system to which a user feeds inputs and gets back outputs. The general behavior of the whole program, such as the total execution time is visible at this level; the underlying details of the program are hidden.

(B) *Machine Level*

At the machine level, the program consists of multiple threads that run simultaneously on the individual machines of the system.

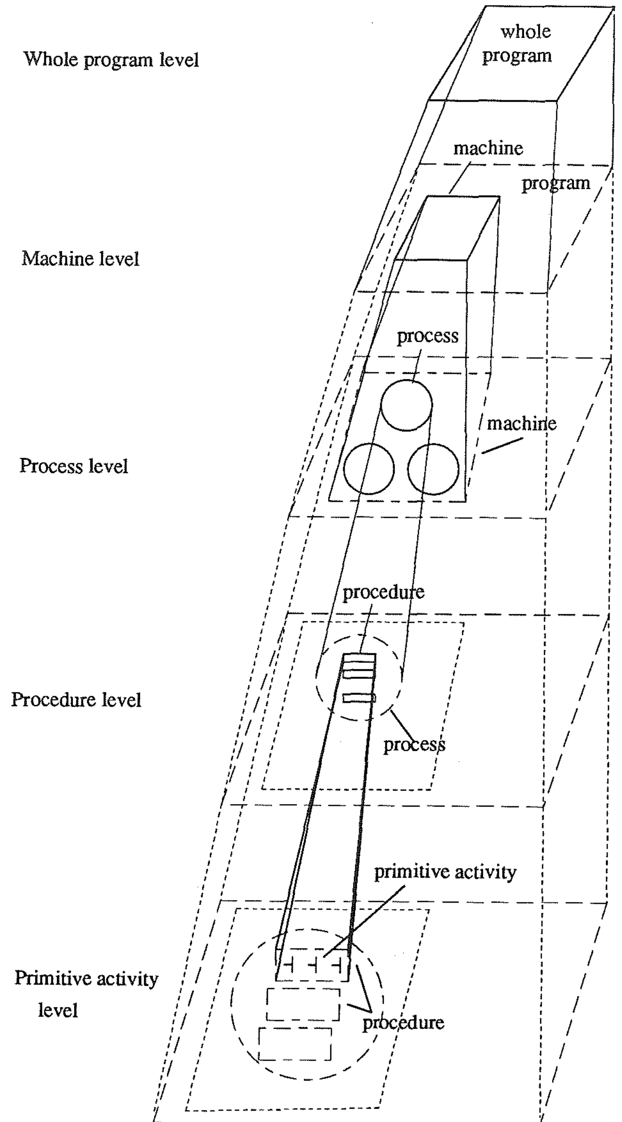

Figure 1: Overview of Computation Hierarchy

We can record summary information for each machine, and the interactions (communications) between the different machines. All events from a single machine can be totally ordered since they reference the same physical clock. The machine level provides no details about the structure of activities within each machine.

The machine level is not strictly part of the programmer designed hierarchy (as are the process and procedure levels). The structure at the machine level can change from execution to the next, or even in a single execution as is the case in process migration [14, 15].

We include the machine level in our hierarchy for two reasons. First, in the systems that we commonly use, we can either directly specify or have explicitly visible the allocation of processes to machines. Second, the performance of a distributed program can be changed dramatically depending on this allocation. It is important to be able to make the distinction between local and remote interactions.

(C) *Process Level*

The process level represents a distributed program as a collection of communicating processes. At this level, we can view groups of processes that reside on the same machine, or we can ignore machine boundaries and view the computation as a single group of communicating processes.

If we view a group of processes that reside on the same machine, we can study the effects of the processes competing for shared local resources (such as CPU and communication channels). We can compare intra- and intermachine communication levels. We can also view the entire process population and abstract the process's behavior away from a particular machine assignment.

(D) *Procedure Level*

At the procedure level, a distributed program is represented as a sequentially executed procedure-call chain for each process. Since the procedure is the basic unit supported by most high-level programming languages, this level can give us detailed information about the execution of the program. The procedure level activities within a process are totally ordered.

The step from the process to the procedure level represents a large increase in the rate of component interactions, and a corresponding increase in the amount of information needed to record these interactions. Procedure calls typically occur at a higher frequency than message transmissions.

(E) *Primitive Activity Level*

The lowest level of the hierarchy is the collection of primitive activities that are detected to support our measurements. Our primitive activities include process blocking and unblocking by the scheduler, message send and receive, process creation and destruction, procedure entry and exit. Each event is associated with a probe in the operating system or programming language run-time that records the type of the event, machine, process, and procedure in which it occurred, a local time stamp, and event type dependent parameters. The events are listed in Table 1.

All events are monitored at the primitive activity level. These events can be associated with metrics at higher levels of the hierarchy. For example, a message send event could be mapped to the program level as part of the total message traffic in the program, to the machine level as part of the message traffic between machines, or to the process level as part of the message traffic between individual processes. More complex mappings are used for computing metrics such as parallelism or utilizations.

2.2. The Measurement Hierarchy

The program hierarchy provides a uniform framework for viewing the various levels of abstraction in a distributed program. If we wish to understand the performance of a distributed computation, we can observe its behavior at different levels of detail. We chose a measurement hierarchy whose levels correspond to the levels in our hierarchy of distributed programs. At each level of the hierarchy, we define performance metrics to describe the program's execution. For example, we may be interested in parallelism at the program level, or in message frequencies at the process level. We can look at message frequencies between processes or between groups of processes on the same machine. This selective observation permits a user to focus on areas of interests without being overwhelmed by all of the details of other unrelated activities. The hierarchical structure matches the organization of a distributed computation and its associated performance data.

Table 2 lists several of the performance metrics that can be calculated by IPS. Some of these metrics are appropriate for more than one level in the hierarchy, reflecting different levels of detail. The list in Table 2 is provided as an example of the type of metrics that can be calculated. A different model of parallel computation can define a different program hierarchy with its own set of metrics.

(A') *Program Level*

All of the metrics listed in Table 2 are valid at the program level. At this level, these metrics provide a summary of the total program behavior.

Most of the metrics are simple to compute. A few of the other metrics are more complex and can be computed in several ways. For example, utilization, ρ, can be computed as the sum of the ρ's for each machine. Alternatively, it can be derived from the parallelism (speed-up) metric, $\rho = P / N_m$ [16].

(B') *Machine Level*

The machine level provides more detail about program's behavior than at the program level. For example, the metrics for message rates (and quantities) are computed for each pair of machines. This forms a matrix whose marginal values are the total traffic into or out of an individual machine. Metrics at the machine level are computed in a similar manner as those at the program level.

(C') *Process Level*

At the process level, the metrics reflect the load generated by individual processes. Message traffic at this level is computed for each pair of processes.

(D') *Procedure Level*

The procedure level provides information to examine the performance effect of parts of a process.

t_{start}:	Process starting time	t_{end}:	Process ending time
t_{block}:	Process blocking time	t_{resume}:	Process un-blocking time
t_{enter}:	Procedure entering time	t_{exit}:	Procedure exiting time
t_{send}:	Message sending time	t_{rcv}:	Message receiving time
$t_{rcv-call}$:	Attempt to receive time	t_{queue}:	Message arrival time

Table 1: IPS Primitive Events

N_p:	Number of processes.	N_m:	Number of machines.
T:	Total execution time (real time).	T_{cpu}:	Total CPU time.
T_{wait}:	Total waiting time.	T_{wait_cpu}:	Total CPU wait time (scheduler waits)
R:	Response ratio, $R = T / T_{cpu}$.	L:	Load factor, $L = (T_{cpu} + T_{wait_cpu}) / T_{cpu}$
P:	Parallelism, $P = T_{cpu} / T$.	ρ:	Utilization
M_b:	Message traffic (bytes/sec)	C:	Procedure call counter
M_m:	Message traffic (msgs/sec)	PR:	Progress ratio, T_{cpu} / T_{wait}

T, N_p, N_m, T_{cpu}, T_{wait}, T_{wait_cpu}, R, L, ρ, M_b, M_m, and C are metrics which will be applied to different levels of the measurement hierarchy.

Table 2: Performance Metrics

3. IPS IMPLEMENTATION

This section describes the details of the initial IPS implementation on the Charlotte distributed operating system. We are also building a new version on the 4.3BSD UNIX operating system.

There are two phases in the operation of IPS — data collection and data analysis. During the first phase we execute the user program and collect the raw trace data. All necessary data are collected automatically during the execution of the program. There is no mechanism provided (or needed) for the user to specify the data to be collected. During the second phase, the programmer can interactively access the measurement results.

3.1. The Charlotte Distributed Operating System

The Charlotte operating system[8, 17] is being used for our initial implementation of IPS. Charlotte is a message-based distributed operating system designed for the Crystal multicomputer network[18], which currently connects 20 VAX-11/750 node computers and several host computers using an 80MB/sec Pronet token ring[19]. The Charlotte kernel supports the basic interprocess communication mechanisms and process management services. Other services such as memory management, file server, name server, connection server, and command shell are provided by utility processes.

3.2. Basic Structure

IPS consists of three major parts: *agent*, *data pool*, and *analyst*. Each of the three parts is distributed among the individual machines in the system. The basic structure of our measurement tool is similar to the structure of METRIC[20] and DPM[5].

The Agent is a collection of probes implanted in the operating system kernel and the language run-time routines for collecting the raw data when a predefined event happens.

The data pool is a memory area in every machine for the storage of raw data and for caching intermediate results of the analyst.

The analyst is a set of processes for analyzing the measurement results. There will be at least one *master analyst* which acts as a central coordinator to synthesize the data sent from the different *slave analysts*. The master analyst coordinates the results from one or more slave analysts and provides an interface to the user. The slave analysts sit on the individual machines for local analysis of the measurement data.

There are some major differences between our structure and the structures of METRIC and DPM. In our scheme, the raw data is kept in the data pool on the same machine where the data was generated. Slave analysts exist on each machine, instead of a single global analyst.

For some data analyses, the master analyst will make a request to a single slave analyst. This is the case e.g. when we request the message traffic between two processes that are on the same machine. Other analyses require the master analyst to coordinate multiple slaves to produce a result. This occurs for metrics computed at the program level of the hierarchy.

The local data collection and (partial) analysis has several advantages. Raw data are collected on the machine where they were generated. Local storage of raw data should incur less measurement overhead than transmitting the traces to another machine. Sending a message between machines is a relatively expensive operation. Local data collection in IPS will use no network bandwidth and little CPU time.

A second advantage to local data collection is that we can distribute the data analysis task among several slave analysts. Low level results can be processed in parallel at the individual machine and sent to the master analyst where the higher level results can be extracted. It is also possible to have the slaves cooperating in more complex ways to reduce intermachine message traffic during analyses (see Section 4).

3.3. Raw Data Collection

Local data collection requires that each machine maintain sufficient buffer space for the trace data. The question arises whether we can store enough data for a reasonable analysis. To study this, we measured the message and procedure call frequencies on several programs. These programs were run on the Charlotte or 4.2BSD UNIX operating systems. The measurement results are summarized in Table 3.

Data gathering for interprocess events are done by agents in the Charlotte kernel. Each time that an activity occurs (most of them appear as system calls), the agent in the kernel will gather related data in an event record and store it in the data pool buffer.

Procedure call events happen at a much higher frequency than interprocess events. Event tracing for procedure calls could produce an overwhelming amount of data. We see this in Table 3, with procedure call rates of over 6000/second — almost three orders of magnitude greater than interprocess events. Due to this high frequency, we use a sampling mechanism combined with modifying the procedure entry and exit code in the Charlotte implementation.

Program Name	Description	Messages	Procedure Call	System
Checkers (Master) Checkers (Mid) Checkers (Slave)	Checkers game, using α/β search	0.60/sec 0.22/sec 0.15/sec	0.67/sec 18.5/sec 1230/sec	Charlotte VAX/750
Pconnector (run 1) Pconnector (run 2)	Initially connected system processes during Charlotte OS bootstrap	9.1/sec 1.2/sec	190/sec 30/sec	Charlotte VAX/750
TSP (10 cities) TSP (20 cities)	Traveling Salesman solver.		2029/sec 6639/sec	UNIX VAX/750
Simulation (run 1) Simulation (run 2)	Resource/deadlock simulation		419/sec 113/sec	UNIX VAX/750
vmc (run 1) vmc (run 2)	Wisconsin Modula Compiler		2401/sec 4231/sec	UNIX VAX/750
make (run 1) make (run 2)	UNIX make facility		4918/sec 4658/sec	UNIX VAX/750

Table 3: Message and Procedure Call Frequencies

Because we are using sampling at the procedure level, results at this level will be approximate. Sampling techniques have been used successfully in several measurement tools for sequential programs, such as the XEROX Spy[21] and HP Sampler/3000[22].

We set a rate (ranging from 5 to 100 ms) to sample and record the current program counter (therefore the current running procedure). We also keep a call counter for each of procedure in the program[23]. Each time the program enters a procedure, the counter of that procedure is incremented. At the sampling time, a record which includes time of day, CPU time, procedure ID, and the call counter. The sampling frequency can be varied for each program execution. We are currently experimenting to determine a minimum value that will provide sufficient information.

3.4. Raw Data Analysis

Analysis programs in the slave and master analysts cooperate to summarize the raw data in respect to user queries. Intermediate Results Tables (IRT) for metrics at the process and procedure levels are kept in each slave analyst. IRT's of machine and program levels are computed and stored in master analyst. The master analyst can reside on any machine as far as communication channels among master and slave analysts can be established. In our implementation, master analyst is a process running on a host Unix system. An independent user interface part is separated from the implementation of the master analyst, so that different interfaces between user and master analyst can be easily adopted for different environments.

Three different query processing schemes are used. The first category contains queries that only need the information in the IRT at the program and machine levels. Therefore master analyst can easily handle these queries by fetching appropriate entries in the IRT. The second category contains queries that require intermediate results stored in the IRT's at slave analysts. The master analyst has to communicate with corresponding slave analysts to retrieve information in the IRT's of slave analysts. The last category of user queries needs direct access to the raw data of the slave analysts (e.g. a query for a list of the event traces in certain time interval). User queries in this category will cause the raw trace data to be scanned at the time of the query.

The processing costs for queries in various categories differ significantly. Queries in first and second categories involve only table searching in master or slave analysts. However, queries in third category are much more expensive due to processing of the large amount of raw data. The choice of data stored in the IRT's on master and slave analysts has a large affect on the costs of user query processing.

The hierarchical program and measurement model of our measurement tool provides a top-down abstraction of performance behavior of the program execution. Users can concentrate on fewer places for inquiring details of raw data information. Therefore, by appropriately selecting IRT's on master and slave analysts, most user queries will fall into first and second query categories.

4. AUTOMATIC PROGRAMMER GUIDANCE

Our performance system is based on the idea that performance tool should provide answers, not just numbers. A performance system should be able to automatically guide the programmer to locate performance problems and help users improve program efficiency. We describe some analysis techniques in this section which support this idea.

The previous sections described a system for measuring the performance of the different parts of a distributed program and at different levels of detail. A programmer can manually use this information to find performance bottlenecks. In its simplest form, the programmer starts at the top (program) level of the hierarchy. Using the available metrics, the programmer will get a general picture of the execution of the program and decides where to look in the next (machine) level of the hierarchy. The decision may be affected by choosing the machine with the smallest utilization or highest procedure call rate. At the process level, the programmer can examine the performance metrics for each process on the machine and choose the process that appears to have the largest performance effect. This descent can be continued to the procedure level.

The execution of a parallel or distributed program can be quite complex. Often, individual performance metrics will not reveal the cause of poor performance. It may be that a sequence of activities, spanning several machines or processes, is responsible of the slow execution. Consider an example from traditional procedure profiling. We might discover a procedure in our program that is responsible for 90% of the execution time. We could hide this problem by splitting the procedure into 10 sub-procedures, each responsible for only 9% of the program execution time. Our analysis techniques should be able to detect a situation where cost is dispersed among several procedures, and across process and machine boundaries.

Another difficult problem is determining the affect of contention for resources. It is possible that the scheduling or planning of activities[24] on the different machines will have a large effect on the performance of the entire program.

The following sections discuss some of the techniques that we are developing or using for automatically guiding the programmer. These techniques include (1) identifying critical resource utilization; (2) determining interaction and scheduling affects; and (3) detecting program time phase behavior.

4.1. Critical Resource Utilization

Turnaround or completion time is an important measure for parallel programs. When we use the turnaround time as our measure, speed is everything. One way to determine the cause of a program's turnaround time is to find the path through the execution history of the program that has the longest duration. This *critical path* [25] identifies where in the hierarchy we should focus our attention.

We can view a distributed program as having the following characteristics: (a) It can be broken down into a number of separate activities. (b) The time required of each activity can be estimated. (c) Certain activities must be executed serially, while other activities may be carried out in parallel. (d) Each activity requires some combination of resources as CPU's, memory spaces, and I/O devices etc. There may be more than one feasible combination of resources for an activity, and each combination is likely to result in a different estimate of activity duration.

Based on these properties of a distributed program, we can use the critical path method (CPM)[25, 26] from network analysis in our performance analysis. The critical path method commonly used in operational research for scheduling issues, and has also been used to evaluate concurrency in distributed simulations[27].

We can find the path in program's execution *history* that took the longest time to execute. Along this path, we can identify the place(s) where the execution of the program took the longest time. The knowledge of this path and the bottleneck(s) in this path will help us focus on the performance problem.

Figure 2 shows a program history graph for a distributed program with 3 processes. This graph shows the program history at the process level. The critical path (identified by the bold line) quickly shows us the parts of the program with the greatest effect on performance. Presentation of these analysis results offers some interesting problems. A program history graph may contain more than 100,000 nodes and the critical path may contain a non-trivial percentage of these nodes. We use statistical presentation techniques and display

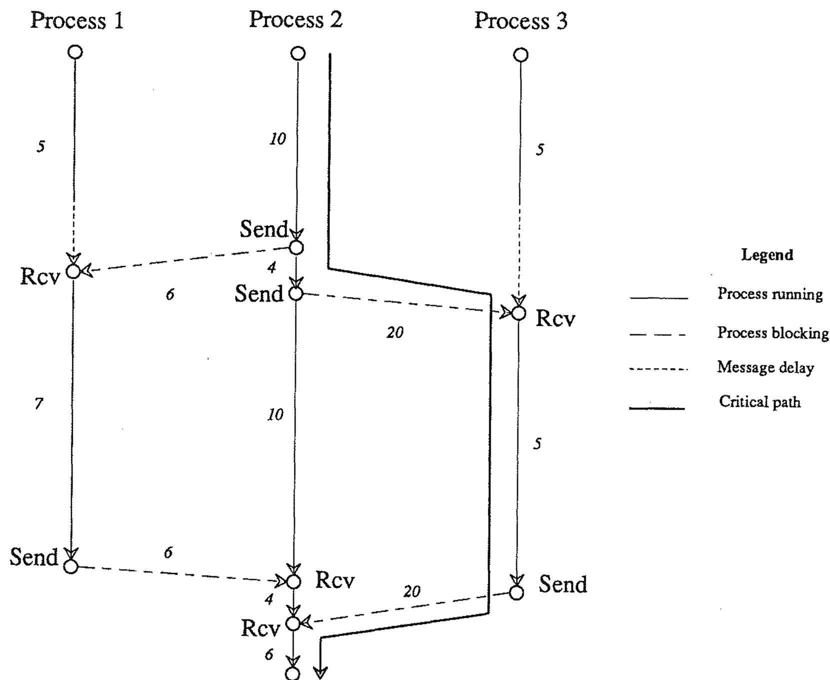

Figure 2: Example of Critical Path -- Process Level

the (time weighted) most commonly occurring nodes, and the most commonly occurring sequences in the path. We use high-level language debugging techniques to relate these events directly to source program. Observing the most commonly occurring sequences allows us to detect performance bottlenecks that span procedure, process, or machine boundaries. Performance problems that are divided among several procedures or even processes or machines will be readily apparent.

Turnaround time is not the only critical measure of parallel program performance. Often throughput is more important, e.g., in high-speed transaction systems[28]. Throughput results can be easily obtained using the statistical techniques described above. The statistical presentation techniques are applied to the entire program history graph — not just the critical path — and this would direct the programmer to those places in the program that most heavily influence the throughput.

An important side issue is how to efficiently compute the critical path information. The program history graph is directed and acyclic, so this is a simpler problem to compute than the general longest path. We are using the distributed structure of our measurement system to test and compare different algorithms for finding the critical path (the longest path) in a graph. We have implemented a central algorithm based on the PDM shortest path algorithm[29] and two variations of distributed algorithm from Chandy and Misra[30]. The preliminary results show a maximum speed-up of 2 for the distributed algorithm over the central version for graphs of about 20,000 nodes, computed on Crystal network. We are currently conducting further experiments and the results will be reported in a future paper.

4.2. Interaction and Scheduling Affects

The methods described in the previous section are based entirely on the structure of the program; they ignore the delays caused by competition for such shared resources as CPU's. In other words, the above techniques are based only on CPU time and ignore real time delays.

We are developing second class of analysis/guidance techniques that measure interactions in the program. One method is to calculate the critical path including delays caused by processes on the same machine competing for the CPU. This critical path then includes real delays due to blocking for events such as receipt of a message, and reflects the interactions and scheduling of the concurrent events in a program.

A second method is based on the correlation of the intervals where a program is blocked to the places in the source program where these intervals occur. With the knowledge of these correlations, programmers can find different idle spots in the program and look for possible improvements. Again, a statistical presentation of results is the most promising. We can present the parts of the program that most frequently remain idle.

4.3. Program Time Phase Behavior

The execution patterns of a program may change over time. For example, a parallel program may go through a period of intense interaction with little computation, then switch to a period of intense computation with little interaction among the concurrent components. If we try to analyze the program's execution behavior as a whole, we may find it difficult to correct the program.

We are currently investigating *focusing* techniques that will identify the different phases in a program's execution. These phases are identified based on critical performance metrics such as interaction (communication) frequencies, CPU usage, and combinations of these metrics. Each phase can be considered a smaller program with more uniform performance characteristics. The focusing technique is used to direct our other guidance techniques to work on the more specific and easily correctable problems.

5. SUMMARY AND STATUS

Our approach to a performance measurement system is to unify performance information into a single, regular structure. This regular structure allows easy and intuitive access for performance information, straight-forwarded mapping onto user interfaces, and the ability to easily automate reasoning about the program's behavior. Our hierarchical model is a natural way to describe distributed programs and their performance. This model provides views from many levels of abstraction, simplifies building tools that reason about a program's behavior.

The techniques to automatically guide the programmer to performance problems will ease the task of identifying performance problems. These techniques, combined with a user interface that directly relates performance results to the program source code, allow the programmer to concentrate on fixing problems rather than finding them.

As a testing prototype for our hierarchical framework we have implemented IPS on the Charlotte distributed operating system. The basic structure and the probes for raw data collection have been put in the Charlotte kernel and programming language run-time routines. Implementation of data analysis mechanisms and master/slave analysts are mostly completed. We have also provided a simple user interface and query methods for access of performance results. Different algorithms for critical path analysis technique have been developed and tested with application programs. We are currently investigating new algorithms for critical path analysis and other data analysis/guidance techniques and conducting several experimental tests on the measurement of distributed programs. Also we are testing IPS with more and varied application programs to better understand the performance system.

An initial implementation of IPS on 4.3BSD UNIX is also underway. This implementation will expand our customer base of application programs. We will also extend this implementation to a Sequent shared-memory multiprocessors. Details of further progress and test results will be presented in future reports.

Our research in the area of performance measurement and evaluation is necessary to keep up with the growing universe of parallel applications. Although the principles and techniques developed in our research are based on the loosely-coupled parallel processing, they should also be applicable to a wide range of programming systems including the shared-memory multiprocessing systems.

6. REFERENCES

[1] M. V. Marathe, "Performance Evaluation at the Hardware Architecture Level and the Operating System Kernel Design Level," Ph.D. Thesis Computer Sciences Department CMU (Dec. 1977).

[2] Ilya Gertner, "Performance Evaluation of Communicating Processes," Ph.D. Thesis Computer Science Department, University of Rochester (May 1980).

[3] Uwe Hercksen, Rainer Klar, Wolfgang Kleinoder, and Franz Kneissl, "Measuring Simultaneous Events in a Multiprocessor System," *Proceedings of 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* pp. 77-88 (August 1982).

[4] Richard Snodgrass, "Monitoring Distributed Systems: A Relational Approach," Ph.D. Thesis Computer Sciences Department CMU (1982).

[5] B. P. Miller, "DPM: A Measurement System for Distributed Programs," *IEEE Trans. on Computers,* (to appear).

[6] B. P. Miller, S. Sechrest, and C. Macrander, "A Distributed Program Monitor for Berkeley Unix," *Software – Practice & Experience* **16**(2)(February 1986). Also appears in short form in the 5th Int'l Conf. on Distributed Computing Systems, Denver (May 1985)

[7] Zary Segall and Larry Rudolph, "PIE: a Programming and Instrumentation Environment for Parallel Processing," *IEEE Software* **2**(6) pp. 22-37 (Nov. 1985).

[8] Raphael Finkel, Marvin Solomon, David DeWitt, and Lawrence Landweber, "The Charlotte Distributed Operating System -- Part IV of the First Report on the Crystal Project," Tech. Report 510 Computer Sciences Dept. , Univ. of Wisconsin-Madison (Sept. 1983).

[9] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel, "Interprocess Communication in Charlotte," *IEEE Software,* (to appear in 1987).

[10] S.J. Leffler, W.N. Joy, and M.K. McKusick, *UNIX programmer's Manual, 4.2 Berkeley Software Distribution,* Computer Science Dept. University of California at Berkeley (August 1983).

[11] M. L. Scott and R. A. Finkel, "LYNX: A Dynamic Distributed Programming Language," *Proc. of the 1984 Int'l Conf. on Parallel Processing,* pp. 395-401 (August 1984).

[12] Thomas J. LeBlanc and Stuart A Friedberg, "Hierarchical Process Composition in Distributed Operating Systems," *Proc. of the 5th Int'l Conf. on Distributed Computing Sys.,* pp. 26-34 (May 1985).

[13] C. Maples, "Analyzing Software Performance in a Multiprocessor Environment," *IEEE Software,* pp. 50-63 (July 1985).

[14] M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proc. 9th Symposium on Operating Systems Principles,* pp. 110-119 (December 1983).

[15] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemtable Remote Execution Facilities for the V-System," *Proc. of 10th ACM Symp. on Operating Systems Principles,* pp. 2-12 (December 1985).

[16] B. P. Miller, "Parallelism in Distributed Programs: Measurement and Prediction," Computer Sciences Technical Report 574, University of Wisconsin–Madison (1985).

[17] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel, "Charlotte: Design and Implementation of a Distributed Kernel," Tech. Report 554 Computer Sciences Dept. , Univ. of Wisconsin-Madison (Aug. 1984).

[18] D. DeWitt, R. Finkel, and M. Solomon, "The Crystal multicomputer: design and implementation experience," *To appear on IEEE Trans. on Software Engineering* , (1986).

[19] Proteon Associates, *Operation and Maintenance Manual for the ProNet Model p1000 Unibus.* 1982.

[20] Gene McDaniel, "METRIC: a Kernel Instrumentation System for Distributed Environments," *Proc. of th Sixth ACM Symposium on Operating System Principles,* pp. 93-99 (November 1977).

[21] Gene McDaniel, "The Mesa Spy: An Interactive Tool for Performance Debugging," *Proc. of 1982 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* pp. 68-76 (1982).

[22] Abbas Rafii, "Structure and Application of a Measurement Tool - SAMPLER/3000," *Proceedings of 1981 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 110-120 (September 1981).

[23] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, "gprof: a Call Graph Execution Profiler," *Proceedings of SIGPLAN '82 Symposium on Compiler Construction*, pp. 120-126 (1982).

[24] Bernard Lint and Tilak Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEE Transactions on Software Engineering* SE-7(2) pp. 174-188 (March 1981).

[25] K. G. Lockyer, *An Introduction to Critical Path Analysis*, Pitman Publishing Company (1967).

[26] W. E. Duckworth, A. E. Gear, and A. G. Lockett, "A Guide to Operational Research," *John Wiley & Sons, New York*, (1977).

[27] O. Berry and D. Jefferson, "Critical Path Analysis of Distributed Simulation," *Proc. of Conf. on Distributed Simulation 1985*, (January 1985).

[28] L.F. Mackert and G. M. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," Research Report, IBM Almaden Research Center (January 1986).

[29] Narsingh Deo, C. Y. Pang, and R. E. Lord, "Two Parallel Algorithms for Shortest Path Problems," *Proc. of the 1980 International Conference on Parallel Processing*, pp. 244-253 (Aug. 1980).

[30] K. M. Chandy and J. Misra, "Distributed Computation on Graphs: Shortest Path Algorithms," *Communications of the ACM* **25**(11) pp. 833-837 (November 1982).