MEASURING DISTRIBUTED PROGRAMS: A HIERARCHICAL AND INTERACTIVE APPROACH

by

Barton P. Miller and Cui-qing Yang

Computer Sciences Technical Report #613

September 1985

Measuring Distributed Programs: A Hierarchical and Interactive Approach

Barton P. Miller
Cui-qing Yang

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

Abstract

We have designed an interactive tool, called IPS, for performance measurement and analysis of distributed programs. IPS is based on a hierarchical model of distributed computation which maps the program's behavior to different levels of abstraction. The hierarchical model unifies performance data from the whole program level down to procedure and statement level. Users are able to maneuver through the hierarchy and analyze the program measurement results at various levels of details. IPS allows the programmer to interactively evaluate the performance history of a distributed program. Because of the regular structure of the hierarchy, IPS can provide information to guide the programmer to the cause of performance bottlenecks. Critical path analysis is used, in conjunction with simple performance metrics to direct the programmer in identifying bottlenecks.

Research supported by the National Science Foundation grant MCS-8105904, a Digital Equipment Corporation External Research Grant, and the Defense Advance Research Projects Agency (DoD) Contract No. N00014-82-C-2087.

1. Introduction

A program can be designed as a group of cooperating processes distributed across several (or many) machines. The reason for distributing a program may be to increase reliability, availability, or performance of the program. If increased performance is our goal for writing a distributed program, then we need a way to measure and evaluate the program's performance. We need a measurement tool that will allow us to evaluate the performance of a program and to evaluate the effect of changes that we make to the program. With this tool, we should be able to inquire about our program's performance at varying levels of detail. Furthermore, the performance tool should provide sufficient information to guide the programmer to location of performance bottlenecks.

Much of the research on performance measurement of distributed systems and programs[1, 2, 3, 4, 5, 6, 7] 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 user program. Often, we need information from several of these levels and need some way of coordinating the information received from these various levels of abstraction.

Several studies on performance measurement of distributed programs worked at the level of message communication and interprocess events[2, 4, 7]. This level of detail provides information about the complex interactions in between the processes in a distributed computation but provides little information about the internal workings of the processes. The results obtained from measurements are at a coarse-grained level, measuring such things as speed-up and communications levels. Traditional performance measurement tools provide information about individual processes (such as page fault frequencies or subroutine call frequencies)[8, 9, 10, 11] but provide no information about the interactions between the processes that constitute a distributed program.

We unify the levels of detail by describing a distributed program as a hierarchy. Our hierarchical structure extends from the whole distributed program down to the finer-grained procedure and statement level. At each level of the hierarchy, we can describe the distributed program in increasing detail.

Along with the program hierarchy, we have defined a hierarchy of metrics to describe the program's performance. At each level of the program hierarchy, there are metrics to describe the program's performance at that level of detail. The programmer can use this hierarchy of performance data to examine the different aspects of program behavior.

We have developed a measurement system, called IPS, that implements our measurement hierarchy. this system allows the programmer to interactively examine the execution history of a distributed program at various levels of detail. The programmer can traverse the hierarchy, *zooming in* to focus on more details, or *zooming out* to see larger trends.

Important goals of a performance measurement tool are to provide facilities for aiding the user to understand program behavior, to locate trouble spots, and to improve the efficiency of programs. Our efforts are devoted to integrating the performance tool with program tuning techniques. For this reason, IPS also provides guidance in the locating of performance bottlenecks. It provides information to the programmer to help determine where to look in the hierarchy.

We are considering a distributed environment with a large number (dozens to hundreds) of machines, so have chosen a decentralized structure for our measurement tool. Most of the function of the tool is distributed over individual machines in the system, attempting to maximize local processing and minimize communications overhead caused by IPS. The pilot implementation of the tool is being done on the Charlotte distributed operating system[12].

In Section 2, we will describe our hierarchical model of distributed computation and then describe the corresponding measurement hierarchy in Section 3. Section 4 presents some of the implementation details of IPS. We discuss data analysis techniques and methods for directing the programmer to performance problems in Section 5. Section 6 reports the current status of IPS and conclusions.

2. Hierarchical Program Model

A hierarchical model is useful for solving complex problems. The objects in a hierarchical model are organized in well-defined layers, and the interfaces between different layers can insulate a layer from the implementation details of the other layers. The hierarchical model provides the ability to view a complicated 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 in the hierarchy, viewing different components at the same level of abstraction.

IPS considers a distributed program to be a hierarchy. This hierarchy consists of components on different machines, and the machines are each running individual processes. A process itself consists of different levels of activities[13] such as procedures, statements and machine instructions. 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.

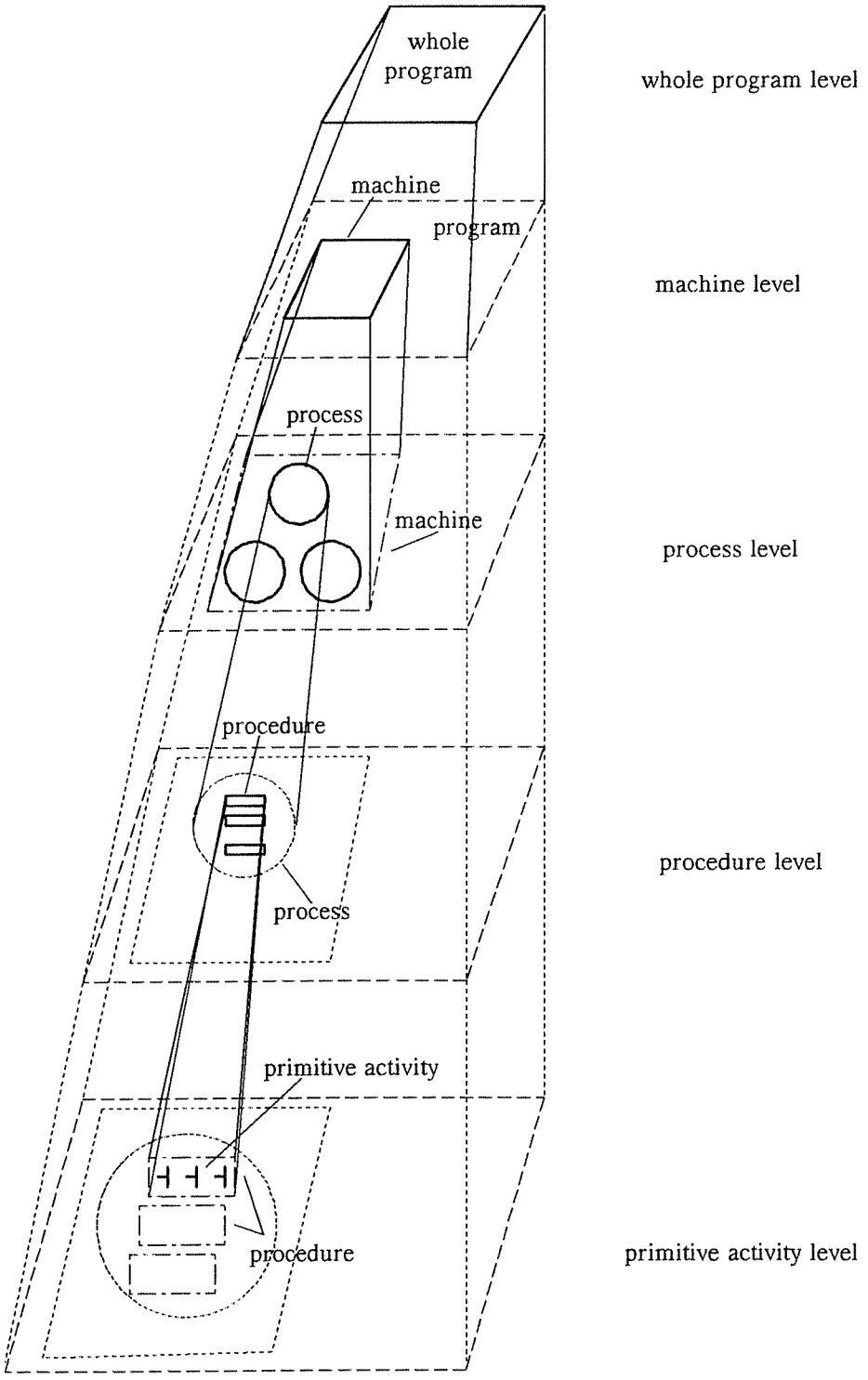

Figure 1: Overview of Computation Hierarchy

(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. 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].

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. For these reasons, we chose to incorporate machines in our hierarchy.

(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 also compare intra- and intermachine communication levels. If we ignore machine boundaries, we can 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. This can be seen by noting that procedure calls typically occur at a higher frequency than do message transmissions (see Section 4).

(E) *Primitive Activity Level*

The lowest level of the hierarchy is the collection of primitive activities in the program. We consider the activities which are reflected in the upper levels of the hierarchy. 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 includes 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.

The primitive events can be mapped to the upper 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.

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)		

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 (see Table 3).

Table 2: Performance Metrics

3. The Measurement Hierarchy

If we wish to understand the performance of a distributed computation, we can observe its behavior at different levels of detail. These 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 the 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. Table 3 summarizes the use of these 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)[15] metric, $\rho = P/N_m$. The parallelism can be computed in two ways. If our program is the only program running on the machines, we can compute T, the total execution time, to be the elapsed real time. If our program must share the machines on which it executes, we must build a graph of the program execution history (as described in [15]) and use the longest path through this graph as T.

(B') *Machine Level*

Metrics at the machine level are computed in a similar manner as those at the program level. We can compute metrics such as utilization in several ways. If we are not sharing the machine with other programs, ρ can be computed as ratio of the total CPU time to the elapsed real time. If other programs are running on the same machine, we cannot use the elapsed real time, and must again construct a program execution graph.

The machine level provides more detail about message traffic than at the program level. 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.

(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.

	Program Level	Machine Level	Process Level	Procedure Level
N_m	X			
N_p	X	X		
T	X	X	X	
T_{cpu}	X	X	X	X
T_{wait}	X	X	X	
T_{wait_cpu}	X	X	X	
L	X	X	X	
M_b	X	X	X	X
M_m	X	X	X	X
P	X			
ρ	X	X	X	
C	X	X	X	X

Table 3: Performance Metrics for Different Hierarchy Levels

We can obtain more specific information from IPS by restricting the data that it uses to calculate the performance metrics. the simplest restriction is to place time bounds on the selected data. This allows us to examine program behavior during a particular phase of program execution, to use a special workload, or to follow the performance effect of a particular transaction. More sophisticated restrictions might be used, such as those suggested in [16].

4. IPS Implementation

IPS is implemented as a distributed program. This section describes the details of the IPS implementation.

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 of the 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. The program execution and trace data collection is done under the Charlotte distributed operating system.

During the second phase, the programmer can interactively access the measurement results. The metrics described in Section 3 and the critical path data described in Section 5 are available to

the programmer.

4.1. The Charlotte Distributed Operating System

IPS is being implemented on Charlotte operating system[17, 12]. Charlotte is a message-based distributed operating system designed for the Crystal multicomputer network[18], which currently connects 20 homogeneous node computers (VAX-11/750s) and several host computers using a Pronet token ring (80MB/sec)[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. Charlotte interprocess communication primitives are non-blocking, synchronous, and unbuffered.

4.2. Basic Structure

IPS (Figure 2) consists of three major parts: *agent*, the *data pool*, and *analyst*. A distinct feature of this structure is that all three parts are distributed among the individual machines in the system. The basic structure of our measurement tool is similar to the structure of METRIC[20] and to the structure of DPM[7].

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 routines 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

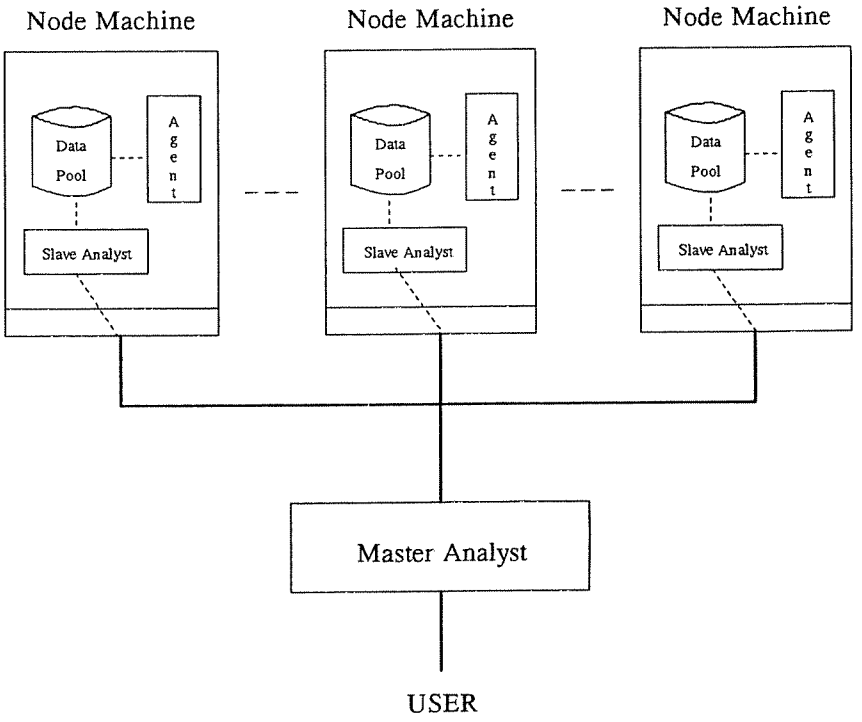

Figure 2: Basic structure of measurement tool

generated. The 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 when we request the message traffic, M_b or M_m, 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. The raw data is collected on the machine which it was generated. This has the advantage that it 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. If the number of the machines in the system becomes large, the transfer of the collected raw data will not be limited by the communication lines.

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 5).

4.3. 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 4.

Data gathering for interprocess events will be 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.

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.		1890/sec 1760/sec	UNIX VAX/780
Simulation (run 1) Simulation (run 2)	Resource/deadlock simulation		530/sec 190/sec	UNIX VAX/780
vmc (run 1) vmc (run 2)	Wisconsin Modula Compiler		2150/sec 3920/sec	UNIX VAX/780
make (run 1) make (run 2)	UNIX make facility		4380/sec 4180/sec	UNIX VAX/780

Table 4: Message and Procedure Call Frequencies

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 4, with procedure call rates of over 4000/second — three orders of magnitude greater than interprocess events. Because of this high frequency, we use a sampling mechanism combined with modifying the procedure entry and exit code. 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[11] and HP Sampler/3000[9].

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 will keep a call counter for each of procedure in the program[10]. 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.

5. Critical Path Analysis

IPS can measure the performance of the different parts of a distributed program and at different levels of detail. This information is used to guide the programmer 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 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 machine 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 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. It is also possible that the scheduling or planning of activities[21] on the different machines will have a large effect on the

performance of the entire program. It is useful to find the path through the execution history of the program that has the longest duration. This *critical path* [22] 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)[22, 23] from network analysis in our performance analysis. We can find a path in the program which takes the longest time to execute. This path and all the events along this path, identifies the place(s) where the execution of the program delays the longest time. The knowledge of this path and the bottleneck(s) in this path will help us focus on the performance problem. As noted in the previous section, results obtained at procedure level will be approximations (because of sampling).

Figure 3 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.

Each slave analyst contains a graph describing part of the program history. The vertices in a slave's subgraph correspond to the events done by processes on that slave's machine. All of these graphs together form a complete event history graph of the program. We are initially using a relatively simple method for calculating the critical path. In parallel, each slave analyst tries to reduces the size of its graph by finding connected subgraphs whose vertices and edges are all contained in a slave's machine. The reduced graphs are then sent to the master analyst for final processing.

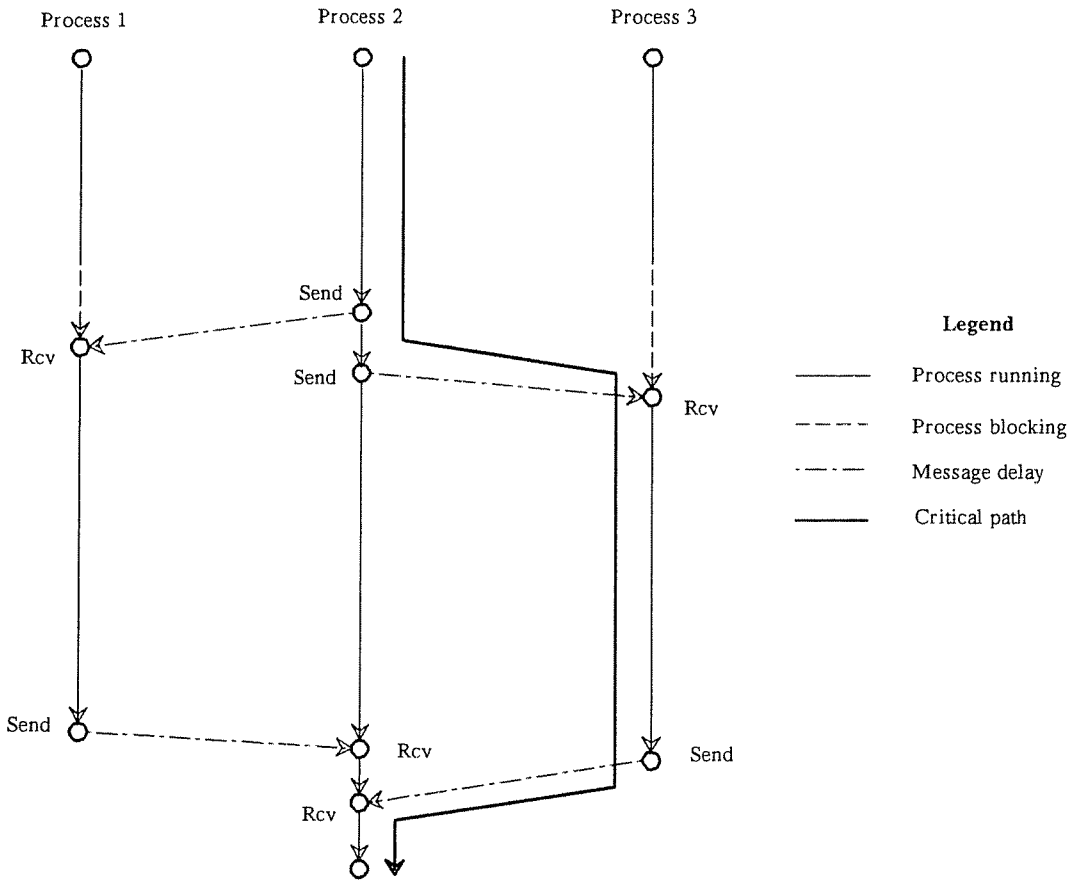

Figure 3: Example of Critical Path -- Process Level

We will experiment with algorithms that transfer graphs between slave analysts to attempt a greater reduction of vertices before the final set of graphs is sent to the master.

6. Conclusion

The hierarchical model is a natural way to describe distributed programs and their performance. This model allows us to incorporate several types (levels) of performance data into a single structure. the programmer is provided with a uniform view of program behavior.

The hierarchy also allows the programmer to vary the amount of information and the level of detail needed to solve a problem. The regular structure of the hierarchy makes it easier for the performance tool to provide guidance to the programmer.

We are currently implementing 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 the programming language run-time routines have been modified. Current efforts are concentrated on studying and developing various algorithms for different techniques of data analysis and results presentation. We are conducting several experiments on the measurement of different distributed programs using IPS.

7. 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] U. Herzog and W. Kleinorder, "Einfuhrung in die Methodik der Verkehrstheorie und ihre Anwendung bei Multiprozessor-Rechenanlagen," *Computing, Suppl. 3,* pp. 41-64 Springer-Verlag, (1981).

[4] 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).

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

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

[7] B. P. Miller, "DPM: A Measurement System for Distributed Programs," Computer Sciences Technical Report 592, University of Wisconsin-Madison (May 1985). Submitted for publication.

[8] Domenico Ferrari and Vito Minetti, "A Hybrid Measurement Tool for Minicomputers," *Experimental Computer Performance and Evaluation,* pp. 217-233 North-Holland Publishing Company, (1981).

[9] Abbas Rafii, "Structure and Application of a Measurement Tool - SAMPLER/3000," *SIGMETRICS ACM,* pp. 110-120 (September 1981).

[10] 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).

[11] Gene McDaniel, "The Mesa Spy: An Interactive Tool for Performance Debugging," *SIGMETRICS ACM,* pp. 68-76 (1982).

[12] 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).

[13] Robert L. Brown, Peter J. Denning, and Walter F. Tichy, "Advanced Operating Systems," *IEEE Computer Magazine,* pp. 173-190 (October 1984).

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

[16] Peter Bates and Jack C. Wileden, "An Approach to High-Level Debugging of Distributed Systems," *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering on High-Level Debugging,* pp. 107-111 (March 1983).

[17] 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).

[18] David J. DeWitt, Raphael Finkel, and Marvin Solomon, "The CRYSTAL Multicomputer: Design and Implementation Experience," *Tech. Report 553 Computer Sciences Dept. Univ. of Wisconsin-Madison,* (Sept. 1984).

[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," *Proceedings of Sixth ACM Symposium on Operating System Priciples,* pp. 93-99 (November 1977).

[21] Bernard Lint and Tilak Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," *IEEE Transactions on Software Engineering* **SE-7, No. 2** pp. 174-188 (March 1981).

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

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