Performance Measurement for Parallel and Distributed Programs: A Structured and Automatic Approach

CUI-QING YANG AND BARTON P. MILLER, MEMBER, IEEE

Abstract—**This paper presents our new approaches in designing performance measurement systems for parallel and distributed programs. One of the approaches involves unifying performance information into a single, regular structure that reflects the structure of programs under measurement. We have defined a hierarchical model for the execution of parallel and distributed programs as a framework for the performance measurement. A complete picture of the program's execution can be presented at different levels of detail in the hierarchy. The second approach is based upon the development of automatic guidance techniques that can direct users to the location of performance problems in the program. Guidance information from such techniques will supply not only facts about problems in the program, but also provide possible answers for the further improvement of program efficiency.**

A performance measurement system, called IPS, has been developed as a prototype of our model and design. IPS is implemented on the Charlotte distributed operating system at the University of Wisconsin—Madison. Some of the test results from IPS are also discussed in the paper to demonstrate unique features of our measurement system.

Index Terms—**Critical path, distributed computing, hierarchical model, operation systems, parallel or distributed programs, performance measurement.**

I. Introduction

A BUILDING performance measurement system for parallel and distributed programs involves the collection, interpretation, and manifestation of information concerning the interactions among concurrently executing processes. The inherent concurrency in a distributed program, the lack of total ordering for events on different machines, and the nondeterministic communication delay between peer processes in such a program add complexity to the problem of performance measurement. The conventional methods of performance measurement for sequential programs are not adequate in the distributed environment because they address only the performance of individual programs on a single processor. New techniques and tools are needed to aid in performance evaluation of parallel and distributed programs.

Two gaps are observed in the current state of performance measurement tools for parallel and distributed programs. The first gap, the *semantic gap*, is a gap between the semantics of the structures with which we build parallel or distributed programs and the semantics by which performance measurement systems to view parallel or distributed programs. On the one hand, people are using more structured methods to develop parallel and distributed programs to cope with the increased complexity. Some examples of these methods are: new constructs in operating systems, a variety of parallel and distributed programming languages, and the current trend toward object-oriented distributed systems. On the other hand, most of the existing performance measurement systems for parallel and distributed programs are built without a well-defined model. Performance measurements in these systems were treated *ad hoc.* A few approaches have defined structures in measurement systems to model the structure of programs. However, many structures of such systems do not match the structure of programs seen by a programmer. This semantic gap prevents many measurement tools from capturing the infrastructure of programs and from providing a complete picture of program's execution.

The second gap is called the *functional gap.* Users of performance measurement tools want not only to know how well their programs execute, but also to understand why the performance may not be as good as expected. The performance metrics provided by most measurement tools are good indications of what happened during a program's execution, but are of little use as guides to the improvement of program performance. A gap exists between what users need and what measurement tools can offer. Users need more help from a measurement tool to locate the performance problems and to find ways for possible improvements.

In this paper, we present our new approach to bridge these two gaps. The basis of our approach is to unify all performance information in a single, regular structure which matches the structure of programs. Such a structure allows easy and intuitive access to performance information, supports the construction of flexible user interfaces, and facilitates automatic reasoning about a program's be-

Manuscript received October 18, 1988; revised April 17, 1989. Recommended by P. A. Ng. This work was supported in part by DARPA under Contract N00014-85-K-0788 and by the National Science Foundation under Grants MCS-8105904, and CCR-8703373 to the University of Wisconsin—Madison.

C.-Q. Yang is with the Department of Computer Science, University of North Texas, Denton, TX, 76203.

B. P. Miller is with the Department of Computer Sciences, University of Wisconsin—Madison, Madison, WI 53706.

IEEE Log Number 8931158.

0098-5589/89/1200-1615$01.00 © 1989 IEEE

havior. Our performance measurement tool also provides facilities that aid users in understanding program behavior and locating trouble spots. We have developed automatic guidance techniques, such as the critical path analysis for the execution of distributed programs, for locating performance problems in programs. These techniques not only supply facts about performance behavior of the program, but also provide possible reasons for the performance problems in the program.

In the following discussion, we start with a brief overview of related research in Section II. This provides a base line for the discussion of our new approach. Section III describes our hierarchical model for distributed programs and a corresponding model for program measurement. These models unify many levels of performance data and serve as the framework of the performance measurement system. Section IV discusses automatic guidance techniques for performance analysis of execution of distributed programs. The major design considerations and some implementation details of the measurement system, IPS, are presented in Section V. Section VI describes some measurement results using IPS to show the functions and features of the new measurement tool. We conclude our paper by summarizing the key ideas and discussing directions for future research in Section VII.

II. Related Research

Much of the research on performance measurement of distributed systems and programs shows that we can describe the behavior of a distributed program at many levels of detail and abstraction. Marathe classifies the performance parameters of a multiprocessor system along the "machine cycles" axis into four separate levels: system programming, operating system design, hardware architecture, and hardware engineering [24]. The same method of classification can also be used to study performance behavior of parallel and distributed programs. The two most common levels for describing performance of distributed programs are the *intraprocess* and *interprocess* levels.

Traditional or intraprocess performance measurement tools provide information about individual process. The *gprof* facility on BSD Unix [14], the HP sampler/3000 [30], the Mesa Spy [25], and other tools [11], [17] are all based on the measurement of information about individual processes (such as subroutine call frequencies or memory access rates).

Performance tools for distributed programs have concentrated on interactions between processes. There are two main reasons for looking at process interactions. First, process interactions are subject to synchronization problems such as race conditions and protocol mismatches. Second, many I/O operations such as file accesses in a distributed environment are mapped into interactions between server and client processes. Therefore, process interactions are traced to understand the state changes of process.

Gertner [13] explicitly recognized the importance of monitoring messages as interactions among processes. He models his performance evaluation tool after finite state machines and uses traces of the process interactions (messages) to trigger the state transitions. Similarly, in a case study of the performance measurement of a distributed dining philosophers algorithm on ZMOB [1], [34], messages between processors are assigned time values and recorded upon transmission, reception, or dequeueing for application input. Miller has developed a measurement tool for distributed programs (DPM) based solely on the interprocess level [26]. The system monitors program activities at the process level and communication events among different processes. Other than message communication, some measurement systems take various resources (e.g., CPU or memories) shared by processes as interactions among them. Performance measurement of EGPA processor array [12], [18] only traces the history of process identifiers which have run on the CPU. Nevertheless, the binding of switchable memory modules with processing elements in the system MIDAS [23] was monitored for the performance analysis of data-driven operations.

A complete picture of program's execution needs information from different levels and some way to coordinate the information received from these various levels of abstraction. Research on integrated instrumentation environment for multiprocessors [32] and the PIE project [15], [33] at CMU proposes the concept of programming observability. It requires provisions for observing status and performance events at all levels (hardware level, kernel level, run-time support level, and application level) and the combination of program run-time information and development-time information. The integrated view is important to the programmer's ease in using the performance tool. The relational data model in the PIE project integrates various information at different levels and different stages of development. However, since the relational model does not fit the structure of programs, it is hard to have information from various levels form a coherent picture of a program's execution. The structure of a measurement system needs to match the structure of programs to be measured. In this way, the so-called semantic gap will be reduced.

The measurement results from most performance systems are presented in some form of performance metrics, such as the process execution time, message traffic statistics, and overall parallelism. These metrics are good for the evaluation of the performance outcome of a program's execution. They tell much about how good or bad the program's execution is but little about why. Few efforts were directed at the study of combining performance measurement tools with techniques to locate performance problems and improve the behavior of parallel and distributed programs. This leads to the so-called functional gap. A performance measurement tool should not only be a judge of the past, but also be a prophet of the future. 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 and give them guidance for possible solutions.

III. The Program and Measurement Hierarchies

A general approach to problem-solving is to decompose a big problem into a set of smaller problems and first to solve these smaller problems. This decomposition is the way that we structure programs: the original problem is divided into pieces such as modules and processes. Different data structures and individual procedures are further defined in each process. All of these objects form a hierarchy within the structure of the program.

Our approach to the performance measurement of parallel and distributed programs is based on organizing the performance measurement tool as a regular structure that matches the structure of the program being measured. We choose a hierarchical model as a framework for our performance measurement system. A hierarchical model provides multiple levels of abstraction, supports multiple views, and has a regular structure. The performance information within this hierarchy reflects the program behavior based on its internal structure, and gives a complete picture of program's execution. An interactive user interface allows users to easily traverse through the hierarchy, to zoom-in/zoom-out at different levels of abstraction, and to focus on the places in the program structure that have great impact on the performance behavior of the program.

In this section we demonstrate these ideas by presenting a sample hierarchy for distributed programs that is based on our initial implementation systems—the Charlotte Distributed Operating System [2] and 4.3 BSD Berkeley UNIX [20]. 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 hierarchical model and reflects our current implementation [27]. The hierarchy is not fixed and it is easy 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 [31] to our hierarchy with little effort. Our hierarchical structure could be also applied to systems such as HPC [19], which has a different notion of program structuring, or MIDAS [23], which has a three-level programming hierarchy.

A. 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 Fig. 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 hardware gates.

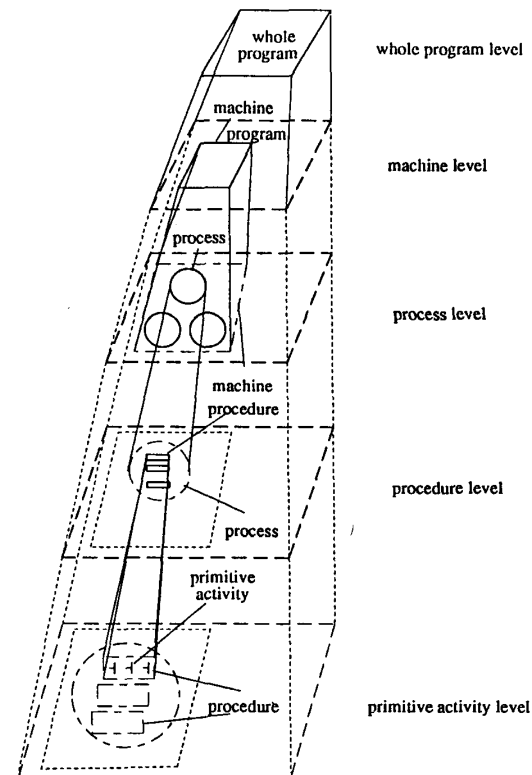

Fig. 1. Overview of computation hierarchy.

1) 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 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.

2) 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.

3) 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 inter-

machine communication levels. We can also view the entire process population and abstract the process's behavior away from a particular machine assignment.

4) 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 much higher frequency than message transmissions.

5) 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 I.

B. 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 II lists several of the performance metrics that can be calculated by the measurement system. Some of these metrics are appropriate for more than one level in the hierarchy, reflecting different levels of detail. Table III summarizes the use of these metrics at each level. The list in Table II 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.

IV. Automatic Guidance Techniques

We base our performance system on the idea that it should provide answers, not just numbers. A performance system should be able to guide the programmer in locating performance problems and should help users improve program efficiency. In this section, we discuss some analysis techniques that support our approach.

The execution of a parallel or distributed program can be quite complex. Often individual performance metrics do not reveal the cause of poor performance, because a sequence of activities, spanning several machines or processes, may be responsible for slow execution. Consider an example from traditional procedure profiling. We might discover a procedure in our program that is responsible for 90 percent of the execution time. We could hide this problem by splitting the procedure into 10 subprocedures, each responsible for 9 percent of execution time. For this reason, it is necessary to detect a situation in which cost is dispersed among several procedures, and across process and machine boundaries.

There are other problems that are difficult to detect using simple performance metrics. It may be important to determine the effect of contention for resources. For example, the scheduling or planning of activities on different machines can have a great effect on the performance of the entire program [21]. Another example is the problem caused by the execution pattern of a program changing over time. 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 its concurrent components. In such a case it will be difficult to understand the detailed behavior of the program by analyzing the program's execution as a whole.

Our strategy in designing a measurement system is to integrate automatic guidance techniques into such a system. Therefore, information from these techniques, such as that which concerns critical resource utilization, interaction and scheduling effects, and program time-phase behavior should be available to help users analyze a program's execution. In our research, we have developed one of the techniques—critical path analysis for the execution of distributed programs [35].

A. Critical Path Analysis for Execution of Distributed Programs

Turnaround or completion time is an important performance measure for parallel programs. When turnaround time is used as the measure, speed is the major concern. One way to determine the cause of a program's turnaround time is to find the event path in 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. As an example, Fig. 2 gives the execution history of a distributed program with three processes. This figure displays the program history at the process level, and the critical path (identified by the bold

TABLE I
PRIMITIVE EVENTS

t_{start}:	Process creation	t_{end}:	Process termination
$t_{block-cpu}$:	Process block for CPU	$t_{unblock-cpu}$:	Process un-block for CPU
$t_{block-syc}$:	Process block fo synch.	$t_{unblock-syc}$:	Process un-block for synch.
t_{enter}:	Procedure entry	t_{exit}:	Procedure exit
$t_{send-call}$:	Message send call	$t_{rcv-call}$:	Message receive call
t_{send}:	Message send	t_{rcv}:	Message receive

TABLE II
PERFORMANCE METRICS

N_p:	Number of processes.	N_m:	Number of machines.
T:	Total execution time.	T_{cpu}:	Total CPU time.
T_{wait}:	Total waiting time.	T_{wait_cpu}:	Total CPU wait time (scheduler waits)
R:	Response ratio, T / T_{cpu}.	L:	Load factor, $(T_{cpu} + T_{wait_cpu})$ / T_{cpu}
P:	Parallelism, T_{cpu} / T.	ρ:	Utilization, P / N_m
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 (see Table III).

TABLE III
PERFORMANCE METRICS FOR DIFFERENT HIERARCHY LEVELS

	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

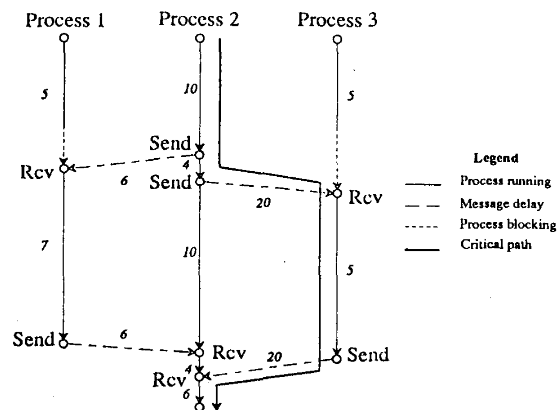

Fig. 2. Example of critical path at process level.

line) readily shows us the parts of the program with the greatest effect on performance.

We can view a distributed program as having the following characteristics:

1) It can be broken down into a number of separate activities.

2) The time required for each activity can be measured.

3) Some activities must be executed serially, while others may be carried out in parallel.

4) Each activity requires a combination of resources, e.g., CPU's, memory spaces, and I/O devices. There may be more than one feasible combination of resources for different activities, and each combination is likely to result in a different duration of execution.

Based on these properties of a distributed program, we can use the critical path method (CPM) [9], [22] to analyze a program's execution. The technique used in our critical path analysis is based on the execution history of a program. We can find the path in the 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 of the bottleneck(s) along it will help us focus on the performance problem.

B. Program Activity Graph

To calculate critical paths for the execution of distributed programs, we first need to build graphs that represent program activities during the program's execution. We call these graphs *program activity graphs* (PAG's). A PAG is defined as a weighted and directed multigraph, that represents program activities and their precedence relationship during a program's execution. Each edge represents an activity, and its weight represents the duration of the activity. The vertices represent beginnings and endings of activities and are the *events* in the program (e.g., send/receive and process creation/termination events). A *dummy activity* in a PAG is an edge with zero weight that represents only a precedence relationship and not any real

action in the program. More than one edge can exist between the same two vertices in a PAG.

The critical path for the execution of a distributed program is defined as the longest weighted path in the program activity graph. The length of a critical path is the sum of the weights of edges on that path.

A program activity graph is constructed from the data collected during a program's execution. Two classes of activity considered in our model of distributed computation are computation and communication. For computation activity, basic PAG events are computation (process) creation and scheduling events. The communication events are based on the semantics of our target system. In Charlotte, the basic communication primitives are message Send/Rcv and message Wait system calls [2]. A Send/Rcv call issues a user communication request to the system and returns to the user process without blocking. A Wait call blocks the user process and waits for the completion of previously issued Send/Rcv activity. Corresponding to these system calls are four communication events defined in the PAG: send_call, rcv_call, wait_send, and wait_rcv. These primitive events allow us to model communication activities in a program. We show, in Fig. 3, a simple PAG for message send and receive activities in a program.

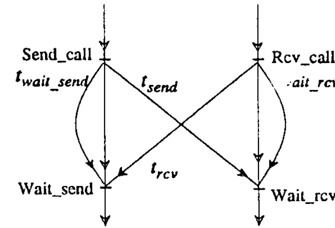

Fig. 3. Construction of simple program activity graph.

The weights of message communication edges in Fig. 3 (t_{send}, t_{rcv}, t_{wait_send}, and t_{wait_rcv}), represent the message delivery time for different activities. Message delivery time is different for local and remote messages, and is also affected by message length. A general formula for calculating message delivery times is: $t = T_1 + T_2 \times L$, where L is the message length, and T_1 and T_2 are parameters of the operating system and the network. We have conducted a series of tests to measure values of these parameters for Charlotte. We calculated average T_1 and T_2 for different message activities (intra- and intermachine sends and receives) by measuring the round trip times of intra- and intermachine messages for 10,000 messages, with message lengths from 0 to the Charlotte maximum packet size. These parameters are used to calculate the weight of edges when we construct PAG's for application programs.

C. Algorithms of Critical Path Analysis

An important side issue is how to compute the critical path information efficiently. After a PAG is created, the critical path is the longest path in the graph. Algorithms for finding such paths are well studied in graph theory. Since all edges in a PAG represent a forward progression of time, no cycles can exist in the graph. To find the longest path in such graphs is a much simpler problem than in graphs with cycles. Also, most shortest path algorithms can be easily modified to find longest paths, because of the acyclic property of our graphs. Therefore, in the following discussion, we consider those shortest-path algorithms to be applicable to our longest-path problem.

1) Diffusing Computation on Graphs: Diffusing computation on a graph, proposed by Dijkstra and Scholten [8], is a general method for solving many graph problems. All of our algorithms for the longest path problem are variations of this method.

We define a *root* vertex of a directed graph as a vertex in the graph that has only out-going edges, and a *leaf* vertex of a directed graph as a vertex in the graph that has only in-coming edges. A diffusing computation on a graph can be described as follows:

> From all root vertices in the graph, a computation (e.g., a labeling message) diffuses to all of its descendant vertices and continues diffusing until it reaches all leaf vertices in the graph.

We distinguish two variations of the diffusing computation: *synchronous execution* and *asynchronous execution.* In synchronous execution, a nonroot, nonleaf vertex will diffuse the computation to its descendant vertices only after it receives all computations diffused from all in-coming edges. In asynchronous execution, a nonroot, nonleaf vertex will diffuse the computation to its descendant vertices as soon as it receives a new computation from any one in-coming edge. Synchronous execution can deadlock in a graph with cycles. However, the computational complexity of synchronous execution is linear in the number of edges and vertices. On the other hand, asynchronous execution does not need explicit synchronization spots in its execution. Potentially, this will provide more concurrency for the computation in a distributed environment.

2) Test of Different Algorithms: We chose the PDM shortest-path algorithm as the basic for our implementation of centralized algorithm which are used for comparison with distributed algorithm [6]. An outline of the PDM algorithm are given in Appendix 1. More detailed discussion of the algorithm can be found in [6]. Our implementation of the distributed longest path algorithm is based on Chandy and Misra's distributed shortest path algorithm [4]. Every process represents a vertex in the graph in their algorithm. However, we chose to represent a subgraph instead of a single vertex in each process because the number of total processes in the Charlotte system is limited and we were testing with graphs having thousands of vertices. The algorithm is implemented in such a way that there is a process for each subgraph, and each process has a job queue for the diffusing computation (labeling the current longest length of the vertex). Messages are sent between processes for diffusing computations across subgraphs (processes). Each process keeps individual message queues to its neighbor processes. An outline of

the two versions of the distributed algorithm appears in Appendix 2. A detailed discussion of the algorithm is given by Chandy and Misra [4].

We used two application programs to generate PAGs for testing the longest path algorithms. The total number of vertices in the graphs varies from a few thousand to more than 10,000. Application 1 is a master–slave structure, and Application 2 is a pipeline structure. Both programs have adjustable parameters. By varying these parameters, we vary the size of the problem and the size of generated PAG's. All of our tests were run on VAX-11/750 machines. The centralized algorithms ran under 4.3 BSD UNIX, and the distributed algorithms ran on the Charlotte distributed operating system [2].

3) Discussion: Speed-up (S) and efficiency (E) are used to compare the performance of the distributed and centralized algorithms. Speed-up is defined as the ratio between the execution time of the centralized algorithm (T_c) to the execution time of the distributed algorithm (T_d): $S = T_c/T_d$. Efficiency is defined as the ratio of the speed-up to the number of machines used in the algorithm: $E = S/N$.

We used input graphs with different sizes and ran the centralized and distributed algorithms on up to 9 machines. Speed-up and efficiency were plotted against the number of machines. The results are shown in Figs. 4, 5, 6, and 7. We have observed a speed-up of almost 4 with 9 machines in synchronous execution of the distributed algorithm. The distributed algorithm with larger input graphs and more machines resulted in greater speed-up but less efficiency. The sequential nature of synchronous execution of diffusing computations determines that the computations in an individual machine have to wait for synchronization at each step of the diffusion. As a result, the overall concurrency in the algorithm is restricted, and the communication overhead with more machines offsets the gain of the speed-up.

The complexity of the synchronous version of the diffusing algorithms is linear with respect to the number of vertices and edges in the graph, because the diffusing computations go through each edge and vertex exactly once. In asynchronous execution, at the worst case, the computation will be proportion to the total number of all possible paths from the source to each vertex in the graph.

Asynchronous execution can increase concurrency in a distributed computation by generating more diffused computations in the job queue and releasing the synchronization requirements among executions. However, in this case we sacrifice economy of work because the asynchronous algorithm does less careful bookkeeping. Our test results indicate that the work in the asynchronous execution grows so fast that even a parallel algorithm is not viable.

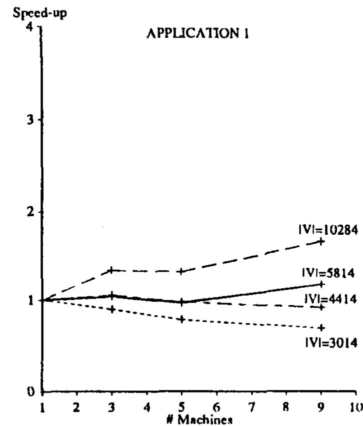

Fig. 4. Speed-up of distributed algorithm.

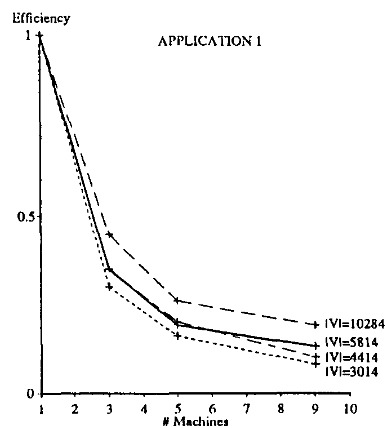

Fig. 5. Efficiency of distributed algorithm.

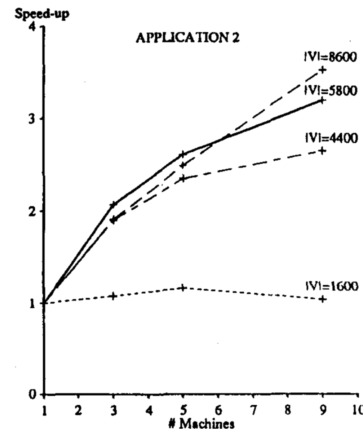

Fig. 6. Speed-up of distributed algorithm.

V. Design and Implementation of IPS Measurement Tool

We have implemented a pilot performance measurement system for distributed programs, IPS, based on our hierarchical measurement model. IPS operates in two separated phases—data collection and data analysis. During the first phase, the program is executed and trace data is collected. All necessary data are collected automatically

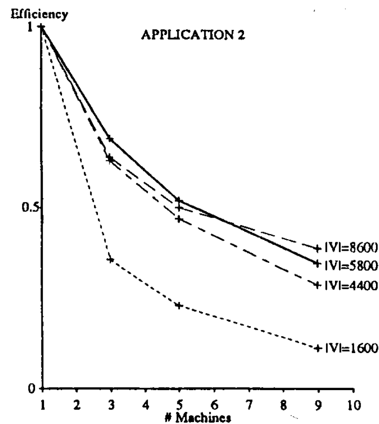

Fig. 7. Efficiency of distributed algorithm.

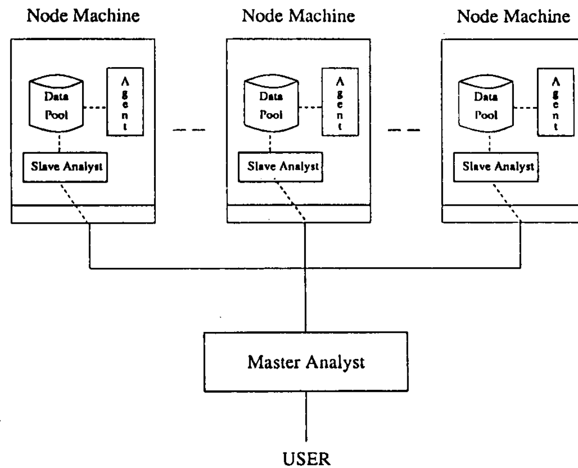

Fig. 8. Basic structure of measurement tool.

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, programmers can interactively access the measurement results. This section describes the design and implementation of IPS on the Charlotte distributed operating system.

A. The Charlotte Distributed Operating System

The Charlotte operating system was used for the initial implementation of IPS. Charlotte is a message-based distributed operating system designed for the Crystal multicomputer network [7]. Crystal consists of 18 VAX-11/750 node computers and several host computers connected with an 80MB/sec Pronet token ring [3]. The Charlotte kernel supports the basic interprocess communication mechanisms and process management services. Other service such as memory management, file server, name server, connection server, and command shell are provided by utility processes [2].

B. Basic Structure

IPS consists of three major parts: agents, data pools, and analysts (see Fig. 8). Each of the three parts is distributed among the individual machines in the system. The *agent* is the collection of probes in the operating system kernel and the language run-time routines for collecting trace data when a predefined event happens. The *data pool* is a memory area in every machine for the storage of trace data and for caching intermediate results from the analyst. The *analyst* is a set of processes for analyzing the measurement results. There is one *master analyst* that provides an interface to the user and acts as a central coordinator to synthesize the data sent from the different *slave analysts*. The slave analysts reside on the individual machines for local analysis of the measurement data.

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. The local data collection and (partial) analysis has several advantages. Trace data are collected on the machine where they were generated. Local storage of trace 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 machines 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 (such as in the critical path analysis).

C. Data Collection

Local data collection requires that each machine maintains sufficient buffer space for the trace data. The question arises whether we can store enough data for a reasonable analysis. Procedure call events happen at a much higher frequency than interprocess (message) events. Event tracing for procedure calls could produce an overwhelming amount of data. We have measured the message and procedure call frequencies of several programs running on the Charlotte and Unix operating systems. The procedure call rates are over 6000/second—almost three orders of magnitude greater than interprocess events [27]. Due to 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 [25] and HP Sampler/3000 [30].

We set a rate, ranging from 5–100 ms, to sample and record the current program counter (therefore the current running procedure). We also keep a call counter for each procedure in the program [14]. Each time the program enters a procedure, the counter of that procedure is incre-

mented. At the sampling time, a record which includes a time stamp, the current procedure PC, and the procedure call counter is saved in the trace data. The sampling frequency can be varied for each program execution. A higher sampling rate will give better precision to the sample results. However, the sampling overhead also increases with the sampling frequency.

Data gathering for interprocess events is done by agents in the Charlotte kernel. Each time that an activity occurs (most 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.

Our implementation for data collection stops at the procedure level, although the hierarchical model itself can be easily extended to include low level information such as that of per statement or per instruction execution. However, the huge amount of data involved in such low level activities prevents any simple solution. Our data collection facility also supports a multiprogramming environment. Processes under monitoring need first to register to the system (via a system call), and only those processes which have registered are under consideration for data collection.

D. Data Analysis

The general structure for data analysis in our measurement tool is shown in Fig. 9. Analysis programs in the master and slave analysts cooperate to summarize the raw data in response to user queries. The master analyst can reside on any machine as long as communication channels between master and slave analysts can be established. In our implementation, the master analyst is a process running on a host UNIX system. An independent user interface is separated from the implementation of the master analyst, so that different interfaces between the user and the master analyst can be adopted for different environments.

Since the amount of the trace data is usually quite big, it is too expensive to process all user queries directly from those data. We create a set of intermediate result tables (IRT) in master and slave analysts to store information preprocessed from the trace data.

There are three different query processing categories. The first category contains queries that only need the information in the IRT at master analyst. The master analyst can easily handle these queries by accessing 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 trace data to be scanned at the time of the query processing.

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, whereas, queries in the third category are much more expensive due to processing of the large amount of raw data. The choice of data stored in IRT's can have a large affect on the costs of user query processing.

Most user queries fall into the first two categories; however, users may occasionally need direct access to the trace data. One such example is when the needed information is not provided by the metrics, e.g., the communication patterns among different processes. Another example is when a user needs to scrutinize the details about program's execution, e.g., to check why a process is blocked during a given time period. These queries will fall into the third category and involve extra costs for processing the raw data.

The metric information about the process and procedure levels is preprocessed by slave analysts and kept in local IRT's. The metric information about the program and machine levels is preprocessed by the master analyst based on low-level information sent from all slave analysts and saved in IRT's at the master analyst. The master analyst gets queries from the user and decides how to distribute a query among slave analysts. The query processing here is similar to the query processing in a distributed database system.

The calculation of performance metrics in the IPS depends upon the ordering of different events in the system. For most metrics, the ordering only involves events occurring in the same machine (e.g., metrics for individual machine and process). For other analysis (such as the critical path analysis in Section IV), we need information about partial ordering among events in different machines. Only a few metrics depend upon the information of total ordering among events across machines. One such example arise when calculating the elapsed time of the whole program. Maintaining total ordering among events in a distributed environment requires synchronizing clocks on different machines. In our implementation, we adopted a simple approximation method based on the TEMPO algorithm [16].

VI. Measurement Tests with IPS

The IPS measurement system provides a wide range of performance information about a program's execution. This information includes performance metrics at different levels of detail, histograms of the basic metrics, and guidance information for the critical path in a program's execution. In evaluating the usefulness of our system, we have been able to obtain some interesting results in the studies we have performed. More time is needed, however, to fully evaluate the models, methods, and tools presented in this paper. This section describes a sample performance measurement session of a distributed application to show the effectiveness of the information provided by the IPS, and to show how this information helps us to better understand the performance behavior of a program.

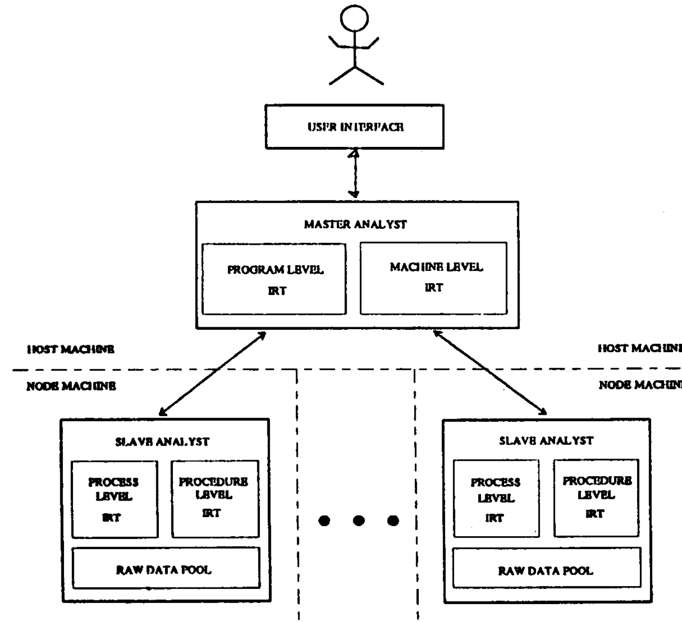

Fig. 9. General structure for measurement results analysis.

A. *Interactive Measurement Session*

The program we have chosen for measurement tests on the IPS system is an implementation of the *Simplex method* for linear programming [5]. The configuration for our test is set as follows: the input matrix size is 36 × 36, the program has a controller process and 8 calculator processes, and these processes run on 3 node machines.

IPS provides an interactive user interface for the measurement of programs. The interface supports a command menu from which users can choose appropriate actions. The menu contains two groups of commands. One group includes the commands for maneuvering through different levels of the hierarchy—for example, going up one level, going down one level, and selecting other items in the same level. Another group consists of the commands for selecting different measurement actions, such as presenting metrics and displaying histograms and critical path information.

The measurement session starts at the program level. We can select a command from the menu to display performance information at this level. Fig. 10 shows the metrics at the program level. These metrics are defined in Section III-B, and reflect various aspects of the performance behavior of the program. The information found in various histograms can help users investigate a program's behavior during a stretch of time. Fig. 11 shows a histogram of CPU time at the program level.

The performance information at the program level gives us a general characterization of the program's execution. For example, as we can see from Fig. 10, the total message waiting time (Block_sync Time) for the entire program is large, and the overall parallelism (speed-up) under this configuration is only 1.15. These results raise questions about why the speed-up of the program's execution is so limited and where the bottlenecks in the program might be.

Metrics for Program Level	
# of Machines:	3
# of Processes:	9
Elapsed Time(ms):	27670
Cpu Time(ms):	31870
Block_sync Time(ms):	203794
Block_cpu Time(ms):	9679
Message Traffic(#):	687
Message Traffic(bytes):	2220080
# of Procedure:	58
# of Procedure Calls:	938
Parallelism:	1.15
Load Factor:	1.30

Fig. 10. Metrics at program level.

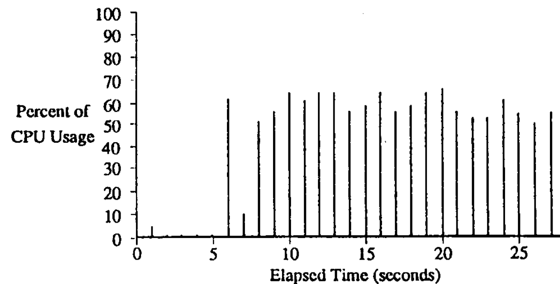

Fig. 11. Histogram of CPU time at program level.

For more detailed information, we continue our study at the machine and process levels. Fig. 12 shows the metrics at the machine level. Each machine has 3 processes.

Metrics	Machine 1	Machine 2	Machine 3
# of Process:	3	3	3
Elapsed Time(ms):	27636	27541	27500
Cpu Time(ms):	16233	7911	7726
Block_sync Time(ms):	58507	72911	72376
Block_cpu Time(ms):	7273	1034	1372
Message Traffic(#):	429	129	129
Message Traffic(bytes):	1383104	418488	418488
# of Procedures:	22	18	18
# of Procedure Calls:	626	156	156
Utilization:	0.59	0.28	0.28
Load Factor:	1.45	1.13	1.17

Fig. 12. Metrics at machine level.

Metrics	Process 3	Process 4	Process 5
Process Name:	control[1]	calcs[3]	calcs[6]
Elapsed Time(ms):	27240	27352	27421
Cpu Time(ms):	9988	3187	3058
Block_sync Time(ms):	12230	22911	23366
Block_cpu Time(ms):	5022	1254	997
Message Traffic(#):	343	43	43
Message Traffic(bytes):	1104112	139496	139496
# of Procedures:	10	6	6
# of Procedure Calls:	522	52	52
Response Ratio:	2.60	8.55	8.55
Load Factor:	1.50	1.39	1.32

Fig. 13. Metrics at process level (in Machine 1).

However, Machine 1 has performance results different from those of Machines 2 and 3. This is because Machine 1 runs with the controller process and 2 calculator processes, whereas Machines 2 and 3 run with three calculator processes each. The process level information of machine 1 is shown in Fig. 13. Two calculator processes (processes 4 and 5) in machine 1 have similar performance behavior. All other processes in machines 2 and 3 are calculator processes and have similar performance results as in machine 1. We can also check histograms of the different metrics at machine and process levels.

We can go further, to the procedure level, to investigate the behavior within each process. Fig. 14 shows this information for the controller and one of the calculator processes. The profiling information here is presented in the same format as in conventional profiling tools [25], [30]. The information about the distribution of CPU time in different procedures can help users determine which part of the program code dominates the execution.

B. Information from Critical Path Analysis

The information from various metrics and histograms gives us a general picture of the program's execution. We have learned about many aspects of the program's behavior from this information; for instance, that parallelism of the program is not high, that there is considerable communication between the controller and calculator processes, and that each calculator process has light work load and spends most of the time in waiting for messages. However, all this information is mainly applicable to individual activity in the program. It tells us little about the interactions among different parts of a program, and about how these interactions affect the overall behavior of a program's execution. Therefore, it is still difficult to discover why the parallelism is low, how much communications affect the program's execution, and which process (controller or calculator) has a bigger impact on the program's behavior. More sophisticated analysis techniques are needed for a measurement system that will help users in analyzing performance results.

The critical path analysis technique in IPS provides guidance for finding possible bottlenecks in a program's execution. The critical path information is represented by the percentages of communication and CPU time of the various parts of the program along the total length of the path. Fig. 15 gives the critical path information at the program level. We can see that the communication cost (including intermachine and intramachine messages) is more than one third of the total length of the critical path. This reflects the fact that the communication overhead in Charlotte is relatively high compared to other systems [2].

The critical path information at the process level (see Fig. 16) gives us more details about the program's execution. The execution of the controller process takes 58 percent of the whole length of the critical path, while the execution of all calculator processes take less than 5 percent of the whole length. The ratio between the execution of controller process and all calculator processes in the critical path is more than 10 : 1, whereas the measurement result in Section VI-A indicates that the ratio between the execution of controller and each calculator is only about 3 : 1. Therefore the critical path information shows more clearly the domination of the execution of the controller process. Such domination of the controller in the critical path restricts the overall concurrency of the program. This explains why the parallelism for the current configuration is so low. From the length of the critical path, we can calculate the maximum parallelism of the program [10], [28], which equals the ratio between the total CPU time and the length of the critical path. This maximum parallelism depends upon the structure and the interactions among the different parts of the program. The maximum parallelism for the program under our tests (with 8 calculators running on 3 machines) is only 1.91. The communication costs and the CPU load effects in different machines lower the real parallelism to 1.15.

Finally, we display critical path information at the procedure level in Fig. 17. This information can be useful in locating performance problems across machine and process boundaries. The top three procedures, that take 49 percent of the entire length of the critical path, are in the controller process. Procedure *MainLoop* in the calculator processes, which is in charge of communications between calculators and the controller, takes 33 percent of the entire execution time of each calculator process (see Fig. 14). However, they are much less noticeable in the critical path because of the dominance of the controller process.

C. Remarks

We have seen that the execution of the controller process dominates the performance behavior. This is be-

Procedure Name	Counter	Time(%)
MainLoop	1	44
Init	1	23
SendChild	96	20
read1	6	7
CheckWaiting	96	4
recv	96	2
main	1	*
Getarg	3	*
SetUp	1	*
ConvertNum	3	*
Total	304	100

(a) Controller

Procedure Name	Counter	Time(%)
dorowop	15	56
MainLoop	1	33
selecttherow	15	10
Initial	1	1
SetUp	1	*
main	1	*
Total	34	100

(b) Calculator

(* denotes less than 1%)

Fig. 14. Execution profiling of controller and calculator.

Entry Name	Time(ms)	%
CPU	10347	62
Inter-machine Msg	4960	30
Intra-machine Msg	1360	8
Total	16667	100

Fig. 15. Critical path information at program level.

Entry Name	Time(ms)	%
P(1,3) CPU	9740	58
P(1,3)->P(3,5) Msg	840	5
P(3,5)->P(1,3) Msg	840	5
P(1,3)->P(2,5) Msg	480	3
P(2,5)->P(1,3) Msg	480	3
P(3,4)->P(1,3) Msg	480	3
P(1,3)->P(3,4) Msg	480	3
P(1,3)->P(2,4) Msg	440	3
P(2,4)->P(1,3) Msg	440	3
P(1,3)->P(1,5) Msg	408	2
P(1,5)->P(1,3) Msg	408	2
P(1,3)->P(1,4) Msg	272	2
P(1,4)->P(1,3) Msg	272	2
P(1,3)->P(3,3) Msg	240	1
P(3,3)->P(1,3) Msg	240	1
P(3,5) CPU	159	1
P(1,5) CPU	108	1
P(2,5) CPU	88	1
P(3,4) CPU	79	*
P(2,4) CPU	67	*
P(1,4) CPU	64	*
P(3,3) CPU	42	*
Total	16667	100

(P(i,j) denotes process j in machinei, * denotes less than 1%)

Fig. 16. Critical path information at process level.

Procedure Name	Mach. Process ID	Time(%)
MainLoop	(Mach 1, Proc 3)	21
SendChild	(Mach 1, Proc 3)	17
Init	(Mach 1, Proc 3)	11
read1	(Mach1, Proc 3)	4
CheckWaiting	(Mach 1, Proc 3)	4
MainLoop	(Mach 3, Proc 4)	1
recv	(Mach 1, Proc 3)	1
MainLoop	(Mach 3, Proc 5)	1
MainLoop	(Mach 1, Proc 5)	1
MainLoop	(Mach 2, Proc 4)	1
MainLoop	(Mach 1, Proc 4)	*
MainLoop	(Mach 2, Proc 5)	*
Total CPU		62

(* denotes less than 1%)

Fig. 17. Critical path information at procedure level.

cause, in our test configuration, the controller process serves too many (8) calculator processes, but each calculator process is lightly loaded. One way to cope with the problem is to reduce the number of calculator processes in the program. We have conducted a set of measurement tests with programs having 2 to 8 calculator processes for the same 36×36 input matrix, running on 3 machines. The test results are shown in Fig. 18(a). We can observe that, to a certain extent, for this fixed initial problem, having fewer calculator processes gives a better result. The execution time has its minimum when the program runs with 3 calculators. However, if the number of calculator processes gets too small (2 in this case), each calculator has to do too much work and creates a bottleneck. Note that the test using 2 calculator processes is the best with respect to the assignment of processes to machines (only one process per machine). While in the 3 calculator case the controller process is running on the same machine as a calculator process. Therefore, the contention for CPU time among processes is not the major factor that affects the overall execution time of the program.

The critical path information for these tests (shown in Fig. 18(b)) supports our observation. For the configuration of 3 calculator processes, the controller and calculator process have the best balanced processing loads, and the lowest message overhead. This coincides with the shortest execution time in Fig. 18(a). The Simplex program has a master-slave structure. The ratio between computation times for the controller and calculator processes *on the critical path* reflects the balancing of the processing loads between the master and slaves in the program. We have observed that when the master and slave processes have evenly distributed processing loads (dynamically, not statically), the program shows the best turnaround time. Otherwise, if the master process dominates the processing, the performance suffers due to the serial execution of the master process. On the other hand, if the slave processes dominated, it would be possible to add more slaves. Appendix 3 contains a proof that supports our claim that for programs with the master–slave struc-

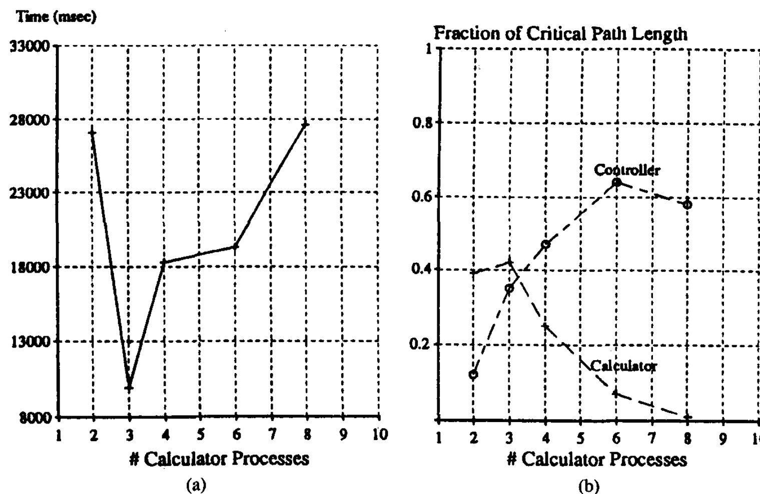

Fig. 18. Program elapsed time and components of the critical path. (a) Elapsed time. (b) Critical path components.

ture, the length of the critical path in the program's execution is at its minimum when the path length is evenly distributed between master and slave processes.

The measurement tests given here shows that the data of the critical path provides extra information for the evaluation and analysis of program's behavior in their execution. This information can be used as guidance to adjust the structure of the program for better performance. Such adjustment involves, e.g., restructuring (decomposing or combining) individual processes in the program, regrouping or replacement of processes on different processors and rearranging procedures or even statements in a process, etc. Nevertheless, because of the inherent complexity with a parallel or distributed program, such restructuring will not be an easy job, and could be application dependent. Much more study needs to be done in respect to the future direction of automatic guidance techniques for the performance measurement of parallel and distributed programs.

VII. Conclusions

We have described a structured and automatic approach to the design and implementation of a performance measurement system for parallel and distributed systems. This approach is based on a hierarchical model as a framework for the measurement system and the integration of the traditional performance metrics with automatic guidance techniques. A prototype, the IPS system, based on our approach has been built on the Charlotte distributed system. Experiments in measuring real application programs on IPS demonstrate that our measurement hierarchy intuitively maps performance information to the program's structure, and gives a multilevel view of the behavior of a program's execution. The abstraction of the hierarchy helps users easily focus on places of interest, while the technique of critical path analysis provides extra information for performance debugging and directs users to possible bottlenecks in the program.

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 loosely coupled parallel processing, they are also applicable to a wide range of programming systems, including shared-memory multiprocessing systems. Projects for the development of new version of IPS systems running on distributed UNIX systems and multiprocessor Sequent machines are currently undergoing at the University of Wisconsin—Madison.

Appendix 1
Centralized Algorithm for Critical Path Analysis

Definitions of some symbols used in the centralized algorithm and distributed algorithm are: G—input graph, u, v—vertices of graph, (u, v)—edge of graph, w_{uv}—weight of edge (u, v), SOURCE—starting vertex, Q—vertex queue in the algorithm, $P[u]$—predecessor of vertex u, $D[u]$—weight of path from SOURCE to vertex u, (Length, Pred)—messaage packet in distributed algorithm.

The following is a sketch of the centralized algorithm for critical path analysis (see Section IV).

```
for all u in G do
   D[u] := 0;
end
initialize Q to contain
SOURCE only;
while Q is not empty do
  delete Q's head vertex u;
  for each edge (u,v) starting at
u do
     if D[v] < D[u] + w_uv then
        P[v] := u;
        D[v] := D[u] + w_uv;
        if v was never in Q then
           insert v at the tail of Q;
        else
           insert v at the head of Q;
        end
     end
  end
end
```

(a): Asynchronous Version

```
for all u in G do
  Count[u] := # of in-coming edges;
  D[u] := 0;
end
initialize Q to contain
SOURCE only;
while Q is not empty do
   delete Q's head vertex u;
  for each edge (u,v) starting at
u do
     dec(Count[v]);
     if D[v] < D[u] + w_uv then
        P[v] := u;
        D[v] := D[u] + w_uv;
     end
     if Count[v] = 0 then
        insert v at the tail of Q;
     end
  end
end
```

(b): Synchronous Version

Appendix 2
Distributed Algorithm for Critical Path Analysis

The following is a sketch of the distributed algorithm for critical path analysis (see Section IV):

```
for all u in sub-graph G do
  Counter[u] := 0;
  D[u] := 0;
end
initialise Q to include SOURCE or empty;
while not termination do
  while Q is not empty and
          msg queues not full do
    delete Q's head element;
    if element = (Length, Pred) then
     if D[u] < Length then
      if Counter[u] > 0 then
        put Ack[P[u]] in Q or msg queue;
      end
      P[u] := Pred;
      D[u] := Length;
     for each edge (u,v) that starts at
       u do
        put length msg:
         (D[u]+w_uv,Pred=u) in Q
         or msg queue for v;
     end
      Counter[u] += # of out-going edges;
     if Counter[u] = 0 then
        put Ack[P[u]] in Q or msg queue;
     end
     else
       put Ack[Pred] in Q or msg queue;
     end
   else
     dec(Counter[u]);
     if Counter[u] = 0 then
       put Ack[P[u]] in Q or msg queue;
     end
   end
  end
  Send all msg queues that are not empty;
  Receive from all neighbor processes;
  if get any msg from other processes then
     processing msg and put length msg
     and acks in Q;
  end
end
```

(a): Asynchronous Version

```
for all u in sub-graph G do
   Counter[u] := # of in-coming edges;
   D[u] := 0;
end
initialise Q to include SOURCE or empty;
while not local_termination do
  while Q is not empty and
           msg queues not full do
    delete Q's head: (Length,Pred);
    dec(Counter[u]);
    if D[u] < Length then
       P[u] := Pred;
       D[u] := Length;
    end
    if Counter[u] = 0 then
       for each edge (u,v) that starts at
         u do
         put length msg:
          (D[u]+w_uv,Pred=u) in Q
           or msg queue fo v;
       end
    end
  end
  Send all msg queues that are not empty;
  Receive from all neighbor processes;
  if get any msg from other processes then
    processing msg and put length msg in Q;
  end
end
exchange msg with neighbors to get
consensus of global_termination
```

(b): Synchronous Version

Appendix 3
A Proof of the Minimum Critical Path Length

In the following discussion, we give a simple proof to support our claim that for programs with the master–slave structure, the length of the critical path in the program's execution reaches the minimum when the whole path length is evenly distributed between master and slave processes. Our proof applies the related study in Mohan's thesis [29] to the aspect of critical path length.

Assume that a general master-slave structure is represented as N slave processes working synchronously under the control of a master process (see Fig. 19). Let a computation have a total computing time of C, consisting of the time for master C_m and the time for slaves C_s (for simplicity, all times are deterministic). The computation time in the master process includes one part for a fixed processing time (e.g., initialization, result reporting time) F_m and another part of per slave service time (e.g., job allocating, partial results collecting) c_m. Therefore,

$$C_m = Nc_m + F_m.$$

Assume F_m is negligible compared to Nc_m, i.e., $F_m << Nc_m$; we have:

$$C_m = Nc_m.$$

The nature of the synchronization pattern in the master–slave structure determines that the execution of the master

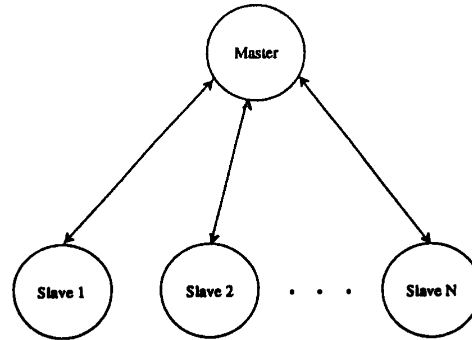

Fig. 19. Processes of master–slave structure.

process is serialized with the concurrent execution of N slave processes. Hence, the length of the critical path in the program's execution, $L_c(N)$, is:

$$L_c(N) = C_m + \frac{C_s}{N} = Nc_m + \frac{C_s}{N}.$$

To find the minimum of $L_c(N)$, we have:

$$\frac{d(L_c(N))}{dN} = c_m - \frac{C_s}{N^2} = 0,$$

and

$$N = \sqrt{\frac{C_s}{c_m}}.$$

Since $(d^2(L_c(N))/dN^2) > 0$ when $N = \sqrt{C_s/c_m}$, $L_c(N)$ has its minimum value at the point. Therefore, the minimum length of the critical path is:

$$\min\left(L_c(N)\right) = Nc_m + \frac{C_s}{N} = \sqrt{c_m C_s} + \sqrt{c_m C_s}.$$

In this equation, both master and slaves have the same amount of share ($\sqrt{c_m C_s}$) in the length of the critical path. This result indicates when the length of the critical path reaches to the minimum, the entire length is evenly distributed in master and slave processes.

Acknowledgment

Many thanks are due to M. Solomon, R. Finkel, and M. Livny for their continuous support and constructive suggestions. In the prototype implementation at University of Wisconsin—Madison, we are also grateful to the members of the Crystal and Charlotte Projects, in particular, B. Gerber, N. Hall, B. Kalsow, M. Litzkow, B. Rosenburg, M. Scott, and T. Virgilio, whose efforts make it possible for the completion of this research.

References

[1] M. Abrams and A. K. Agrawala, "Performance study of distributed resource sharing algorithms," Dep. Comput. Sci., Univ. Maryland, Tech. Rep. TR-1521, July 1985.
[2] Y. Artsy, H.-Y. Chang, and R. Finkel, "Interprocess communication in Charlotte," *IEEE Software*, vol. 4, no. 1, pp. 22–28, Jan. 1987.
[3] Proteon Associates, *Operation and Maintenance Manual for the ProNet Model p1000 Unibus*, 1982.

[4] K. M. Chandy and J. Misra, "Distributed computation on graphs: Shortest path algorithms," *Commun. ACM*, vol. 25, no. 11, pp. 833–837, Nov. 1982.
[5] G. B. Dantzig, *Linear Programming and Extensions*. Princeton, NJ: Princeton University Press, 1963.
[6] N. Deo, C. Y. Pang, and R. E. Lord, "Two parallel algorithms for shortest path problems," in *Proc. 1980 Int. Conf. Parallel Processing*, Aug. 1980, pp. 244–253.
[7] D. DeWitt, R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," *IEEE Trans. Software Eng.*, vol. SE-13, no. 8, pp. 953–967, 1987.
[8] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Inform. Processing Lett.*, vol. 11, no. 1, pp. 1–4, Aug. 1980.
[9] W. E. Duckworth, A. E. Gear, and A. G. Lockett, *A Guide to Operational Research*. New York: Wiley, 1977.
[10] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," Dep. Comput. Sci., Univ. Washington, Tech. Rep. 86-08-01, Aug. 1986.
[11] D. Ferrari and V. Minetti, "A hybrid measurement tool for minicomputers," in *Experimental Computer Performance and Evaluation*. Amsterdam, The Netherlands: North-Holland, 1981, pp. 217–233.
[12] H. Fromm, U. Hercksen, U. Herzog, K. H. John, R. Klar, and W. Kleinoder, "Experiences with performance measurement and modeling of a processor array," *IEEE Trans. Comput.*, Vol. C-32, no. 1, pp. 15–31, Jan. 1983.
[13] I. Gertner, "Performance evaluation of communicating processes," Ph.D. dissertation, Dep. Comput. Sci., Univ. Rochester, May 1980.
[14] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," in *Proc. SIGPLAN '82 Symp. Compiler Construction*, 1982, pp. 120–126.
[15] F. Gregoretti and Z. Segall, "Programming for observability support in a parallel programming environment," Dep. Comput. Sci., Carnegie-Mellon Univ., Tech. Rep. CMU-CS-85-176, Nov. 1985.
[16] R. Gusella and S. Zatti, "TEMPO: Time services for the Berkeley local network," Dep. EECS, Univ. California, Berkeley, PROGRES Rep., Dec. 1983.
[17] G. Hamilton, "Logic analyzer gives programmers real-time view of software performance," *Electronics*, pp. 117–122, May 5, 1983.
[18] R. Klar, "Hardware measurements and their application on performance evaluation in a processor-array," *Comput. Suppl.*, vol. 3, pp. 65–88, 1981.
[19] T. J. LeBlanc and S. A. Friedberg, "Hierarchical process composition in distributed operating systems," in *Proc. 5th Int. Conf. Distributed Computing Systems*, May 1985, pp. 26–34.
[20] S. J. Leffler, W. N. Joy, and M. K. McKusick, *UNIX Programmer's Manual, 4.2 Berkeley Software Distribution*, Dep. Comput. Sci., Univ. California at Berkeley, Aug. 1983.
[21] B. Lint and T. Agerwala, "Communication issues in the design and analysis of parallel algorithms," *IEEE Trans. Software Eng.*, vol. SE-7, no. 2, pp. 174–188, Mar. 1981.
[22] K. G. Lockyer, *An Introduction to Critical Path Analysis*. New York: Pitman, 1967.
[23] C. Maples, 'Analyzing software performance in a multiprocessor environment," *IEEE Software*, pp. 50–63, July 1985.
[24] M. V. Marathe, "Performance evaluation at the hardware architecture level and the operating system kernel design level," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Dec. 1977.
[25] G. McDaniel, "The Mesa Spy: An interactive tool for performance debugging," in *Proc. 1982 ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, 1982, pp. 68–76.
[26] B. P. Miller, S. Sechrest, and C. Macrander, "A distributed program monitor for Berkeley Unix," *Software—Practice & Experience*, vol. 16, no. 2, Feb. 1986; also appears in short form in the *5th Int. Conf. Distributed Computing Systems*, Denver, CO, May 1985.
[27] B. P. Miller and C.-Q. Yang, "IPS: An interactive and automatic performance measurement tool for parallel and distributed programs," in *Proc. 7th Int. Conf. Distributed Computing Systems*, IEEE Comput. Soc., Sept. 21–25, 1987, pp. 482–489.
[28] B. P. Miller, "DPM: A measurement system for distributed programs," *IEEE Trans. Comput.*, vol. 37, no. 2, pp. 245–247, Feb. 1988.
[29] J. Mohan, "Performance of parallel programs: Model and analyses," Ph.D. dissertation, Dep. Comput. Sci., Carnegie-Mellon Univ., Tech. Rep. CMU-CS-84-141, 1984.
[30] A. Rafii, "Structure and application of a measurement tool—SAMPLER/3000," in *Proc. 1981 ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, Sept. 1981, pp. 110–120.
[31] M. L. Scott and R. A. Finkel, "LYNX: A dynamic distributed programming language," in *Proc. 1984 Int. Conf. Parallel Processing*, Aug. 1984, pp. 395–401.
[32] Z. Segall, A. Singh, R. T. Snodgrass, A. K. Jones, and D. P. Siewiorek, "An integrated instrumentation environment for multiprocessors," *IEEE Trans. Computers*, vol. C-32, no. 1, pp. 4–14, Jan. 1983.
[33] Z. Segall and L. Rudolph, "PIE: A programming and instrumentation environment for parallel processing," *IEEE Software*, vol. 2, no. 6, pp. 22–37, Nov. 1985.
[34] N. Vanderlipp, J. Callahan, M. Abrams, and A. Agrawala, 'Implementation and measurement of a distributed dining philosophers algorithm on ZMOB," Dep. Comput. Sci., Univ. Maryland, Tech. Rep. TR-1530, Aug. 1985.
[35] C.-Q. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *Proc. 8th Int. Conf. Distributed Computing Systems*, June 13–17, 1988, pp. 366–373.

Cui-Qing Yang received the undergraduate degree in computer engineering from Nanjing Institute of Technology, China, and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin—Madison.

He has been an Assistant Professor in the Department of Computer Sciences at the University of North Texas, Denton, since 1987. His research interests include parallel and distributed computer systems, operating systems, performance measurement, object-oriented programming, and computer networks.

Dr. Yang is a member of the IEEE Computer Society.

Barton P. Miller (M'85) received the B.A. degree in computer science from the University of California, San Diego, in 1977, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1979 and 1984, respectively.

Since 1984, he has been an Assistant Professor in the Department of Computer Sciences at the University of Wisconsin—Madison. His research interests include parallel and distributed debugging, parallel and distributed program measurement, network management and naming services, distributed operating systems, and user interfaces.