A STRUCTURED AND AUTOMATIC APPROACH TO THE PERFORMANCE MEASUREMENT OF PARALLEL AND DISTRIBUTED PROGRAMS

by

Cui-Qing Yang

Computer Sciences Technical Report #713

August 1987

A STRUCTURED AND AUTOMATIC APPROACH TO THE PERFORMANCE MEASUREMENT OF PARALLEL AND DISTRIBUTED PROGRAMS

by

CUI-QING YANG

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1987

Abstract

We address two major issues in building performance measurement systems for parallel and distributed programs. The first issue is how a measurement system can provide a complete picture about the execution of a program, and how this information should be organized so that users can easily and intuitively access all the data without being inundated with irrelevant details. The second issue is how a measurement tool can supply more information than just performance statistics, and how such information can guide the programmer in locating performance problems and in providing possible ways to make further improvements.

Our approach to the first issue involves unifying performance information into a single, regular structure that reflects the organization of the program. We have defined an 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. Users are able to maneuver through the hierarchy, concentrating on the spots where the most interesting activities have occurred, and can interactively shift their focus.

The approach to the second issue 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 not only supply facts about problems in the program, but also provide possible answers to improve its performance. The construction of guidance techniques is facilitated by the regular structure of the program and measurement hierarchy.

A performance system, called IPS, has been implemented on the Charlotte distributed operating system as a test of our models and design. An automatic guidance technique – critical path analysis for the execution of distributed programs – is integrated in IPS. Measurement tests on IPS show that the critical path information in conjunction with hierarchically organized performance metrics provide a comprehensive picture of program's execution and help users identify the cause of performance bottlenecks.

The principles and techniques developed in this thesis are based on loosely-coupled parallel processing. They are also applicable to a wide range of computer systems, including the shared-memory multiprocessing systems.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1

Introduction

Writing parallel and distributed programs has become an important part of computer programming. A steadily increasing number of systems now support loosely-coupled or tightly-coupled parallel programming and parallel applications. Structuring a program as a group of cooperating processes distributed across several machines increases reliability, availability, and performance of the computation. This dissertation will address the topic of increasing performance.

One method of studying and evaluating the performance behavior of a parallel and distributed program is to measure its performance during program's execution. This differs from other methods of performance studies such as modeling and simulation in that the entity under measurement is exactly the entity to be studied; no artifacts are introduced from modeling or simulation (here we distinguish from these artifacts the interference of a measurement system to the behavior of the entity under measurement). Performance measurement or performance monitoring is important to understand the run-time behavior of a program executing on a real system. In addition, the information gathered from measurements is essential for providing parameters and validating performance models and simulations.

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[1, 2, 3] adds 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.

1.1. Research Goal and Approach

Two gaps exist 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 used by 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[4, 5, 6], a variety of parallel and distributed programming languages[7, 8, 9, 10], and the current trend toward object-oriented distributed systems[11, 12, 13, 14]. On the other hand, most of the existing performance measurement systems for parallel and distributed programs are built without a well-defined model of the environment and of the program structure.

Performance measurements in these systems were treated *ad hoc*. A few approaches have defined structures in measurement systems to model the structure of programs to be measured. Structures of these measurement systems, however, do not match the structure of the program as seen by the programmer. Lack of well-defined models for measurement systems and the mismatching of structures for measurement systems to structures of parallel/distributed programs leads to the semantic gap. This semantic gap prevents most existing measurement tools from capturing the infrastructure of programs and from providing a complete picture of program's execution. It also leads to the second gap.

This 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 and how they can improve program performance. 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 in the program and to have guidance for finding possible performance improvements.

Our basic goal is to bridge these two gaps by providing more effective performance tools for developing parallel and distributed programs. The main areas of research in this thesis are to design a new performance measurement system for parallel and distributed programs that supplies extensive information about the execution of user programs, and to develop algorithms and techniques that will automatically guide programmers to the location of performance problems. The basis of our approach is to unify all performance information in a single, regular structure. This regular structure matches the semantic structure of programs, allows easy and intuitive access to performance information, supports the construction of flexible user interfaces, and facilitates automatic reasoning about a program's behavior.

We choose a hierarchical model as a framework of our performance measurement system. The hierarchy is a regular structure that reflects the structure of a program. All data collected during program measurement and the information extracted from these data are organized in a hierarchy. The performance behavior of a program's execution can be presented at different levels of abstraction.

By defining a hierarchical model as a framework, the performance measurement system is structured in a way that semantically matches program structure. Thus, the performance information from this measurement system represents the performance behavior based on programs' internal structure, and gives a more complete picture of programs' 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.

To bridge the functional gap between users and measurement systems, a performance measurement tool needs to provide facilities that aid users in understanding program behavior, locating trouble spots, and improving the efficiency of programs. Our research has developed automatic guidance techniques for locating performance problems in programs. It not only supplies facts about performance behavior of the program, but also provides possible reasons for the performance problems in the program.

1.2. Terminology

A *distributed system* refers to a computer system in which the computing capacity of the entire system is distributed over many loosely-coupled machines, called *node machines*. The distinguishing properties of a distributed system are that there is no central controller and no shared memory among node machines. Each node is autonomous, and parts of a program can run simultaneously on them. Nodes communicate with each other by messages

The execution of program code on a processor is called a *process*. A process is the basic unit of a computation that is scheduled to run on a node machine of a distributed system. Every process has its own context (code, data, and machine environment) for its execution. Multiple processes can exist on each node.

A *distributed program* is built from a collection of cooperating processes executing simultaneously on various machines of a distributed system. Each process of a distributed program represents a part of the program. Therefore, the execution of a distributed program consists of the execution of all of its parts (processes).

The combination of the execution of all processes for a given distributed program within the same machine as is defined to be a ***thread of execution*** of the program. Processes within a thread compete to run sequentially on the same machine. Different threads of a program execute concurrently on different machines. As a result, the execution of a distributed program is the combination of all of its threads executing as multiple processes in different machines.

1.3. Thesis Organization

We start with an overview of previous work in Chapter 2. This provides a base line for the discussion of our research. Chapter 3 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. To test our ideas, we implemented a prototype measurement system. The main design considerations and some implementation details of the measurement system (called IPS) are presented in

Chapter 4. Chapter 5 discusses automatic guidance techniques for performance analysis of the execution of distributed programs. Chapter 6 describes a measurement session using IPS to show the functions and features of the new measurement tool. We conclude our thesis by summarizing the key ideas and discussing directions for future research in Chapter 7.

Chapter 2

Related Work

This chapter presents an overview of previous work in performance measurement of parallel and distributed systems and programs. In the following sections, we first summarize several issues which are important in the design of a performance measurement system. We then describe the various features of existing and proposed performance systems using those issues as a guideline. We conclude with a few comments on the current state of research in this area, and on the general directions that our research will follow.

2.1. Issues in Performance Measurement Systems

Performance measurement of parallel and distributed systems raises problems due to the complexity and variety of the objects to be measured. Existing measurement systems show a remarkable degree of diversity in their application and structure. There is no general consensus as to the best design or organization of such systems. Our discussion attempts to deal with this diversity by listing several important issues that create a framework for the study of performance measurement systems. The framework serves as a basis for comparing current measurement systems, and facilitates the study of new designs.

This section discusses the following major issues in the design of a performance measurement system:

- the measurement environment and the application domain,
- the measurement mechanisms,
- the measurement policies, and
- the user interface.

Measurement Environment and Application Domain

The first issue in the design of a performance measurement system concerns the system's working environment and its applications. Performance measurement on a loosely-coupled distributed system is different from that on a tightly-coupled multiprocessor. The measurement goals for application programs will be different from those for operating systems. Even within the same application domain, different measurement systems take different views of what should be measured.

A measurement system may define a model for its measurement objects. This model reflects how the measurement system views the objects to be measured and describes the measurement goals of the system. However, most current measurement systems do not provide well-defined models for their measurements.

Measurement Mechanism

The second issue inherent to a performance measurement system concerns basic mechanisms such as for event detection, data filtering and data reporting. These mechanisms are used to monitor the occurrences of significant events, to collect and process the raw data, and to organize data for further analysis. Although these basic functions are found in most measurement systems, their implementation and structure vary depending upon the working environment and application.

Measurement Policy

The third issue in the design of a measurement system lies in its measurement policy. This policy describes which data should be measured during a program's execution, how this data should be analyzed, and which results should be presented to the programmer. The main components of a system's measurement policy include the performance metrics defined in the system and the techniques for analysis of the measurement data. The performance metrics provide various measures for the performance behavior of measurement objects, and the analysis techniques determine how performance metrics are derived from the raw data. Many measurement systems have their policies and techniques predefined and built in the system; users of these systems can select from the various features available. Some systems, however, do not provide a built-in measurement policy. It is then the user's responsibility to decide what performance information is needed and how to derive this information from the measurements.

User Interface

The last issue to be considered is the interface between a user and the measurement system. The nature of this interface affects system usability. It determines how measurement jobs are submitted to the system, how users retrieve performance information, and how the system presents performance results. A basic user interface provides only separate commands to start a measurement session and to fetch the performance information. While some measurement systems have interactive interfaces so that users can interact with systems through the whole measurement session. Performance results are presented from measurement systems with many different forms such as tables, histograms, charts and other graphic representations.

2.2. Performance Measurement for Sequential Programs

The execution of traditional sequential programs involves only a single thread of computation in a machine. All the events in such a program's execution can be completely ordered. Therefore, by sequentially monitoring these events during the program's execution, a profile of the program's execution can be derived from the data recorded. This profile contains information about the distribution of total

execution time among different parts of the program, e.g. modules, procedures, and code segments in the program. The parts in the program that dominate the total execution time of the program are the possible bottlenecks in the program's execution.

Two tools, HP-Sampler/3000[15] and Mesa Spy[16], use the same technique of program counter (PC) profiling to provide information about CPU time in terms of code segments, procedures, and modules. Software sampling is used to periodically sample the PC of currently running programs. Symbolic output at the source language level is derived from the PC histogram data. Both Sampler and Spy have interactive user interfaces to provide easy access to performance information. The major difference between these two systems is that the Spy system exploits knowledge of the Mesa language and run time environment[17], whereas the Sampler/3000 takes advantage of the stack architecture of the HP-3000 system[18].

Another program profiling facility, "gprof"[19], runs on the Unix operating system[20]. Gprof also uses the PC sampling mechanism for CPU time profiles. By monitoring procedure calls at the run-time, the system provides information about the procedure-call graph and the distribution of program execution time for each procedure. In addition, gprof has the ability to account for the execution time of called procedures in the execution time of the procedures that call them. This feature allows users to consider the costs of individual routines in the different levels of the procedure-call graph and aids in the evaluation of modular programs.

The software sampling mechanism used in program profiling may slow down the system due to the measurement overhead. To minimize such interference, some measurement tools take a hardware approach in the design of their measurement mechanisms. One such example is the HP-63410A software-performance analyzer[21]. A special hardware board is plugged into the system to monitor both the occurrences and the durations of the references for blocks of memory and sections of code during a program's execution. The primary advantage of the analyzer is that it can characterize software nonintrusively as a program executes in the system. However, the hardware approach needs special facilities for the dedicated systems and can only measure events at relatively low levels in the system.

2.3. Performance Measurement for Parallel and Distributed Programs

In contrast to traditional sequential programs, parallel and distributed programs create new problems for performance measurement systems. New measurement mechanisms and policies are needed to deal with issues such as monitoring multiple processors in the system, tracing interactions among different parts of the program, and characterizing concurrency within parallel and distributed programs. In this section, we describe several measurement systems that represent the variety of approaches taken in this area.

The METRIC system[22] is an early effort to develop a basic mechanism for performance measurement of distributed software. It is implemented on a network of minicomputers connected by an Ethernet. The system consists of three parts: the object system probes, the data accountant, and the result analyst. These parts are relatively independent and connected only by the communication channel. Users insert calls to the probe procedures into the source program. Trace data are generated from these probes and sent to the accountants, which record and selectively filter the data for later analysis. The analysts are user processes that summarize and tabulate trace data according to the policies of the measurements being made.

The division of the METRIC into three cooperating parts is an elegant and effective approach to structuring measurement tools in a distributed environment. The mechanisms provided by the system allow low overhead in the operating system and flexibility in selecting and studying the data. However, users need to specify the appropriate measurement policy for collection of raw data and analysis of performance information.

Hardware monitors are also used for the performance measurement of parallel programs. Research on the performance of concurrent programs has been conducted on the hierarchically organized multiprocessor system, EGPA (Erlangen General Purpose Array)[23, 24]. A hardware monitor (ZAHLMONITOR III) as well as associated software[25] is used to record process activities of a program at individual processors during a program's execution.

The basic structure of EGPA is a pyramid that has one controlling and administering B-processor for every four working A-processors. Each B-processor has its own local memory and can access the memories of its A-processors. All A-processors in a unit share their local memories. The hardware monitor measures the active and idle states of CPU and I/O channels for each processor, and records the complete history of processes that have run on the CPU. Histograms of A-processor activities are derived from the traces of process events.

The measurement goal of the EGPA system is to monitor concurrent execution of the subtasks of a program in different processors. Only a single event (the process ID of the current running process in each A-processor) is traced as an indication of the overall parallel activity in various parts of the program. No other measurement metrics are defined in the system. Consequently, the measurement results provide limited information about the performance behavior of running programs.

The execution of a parallel/distributed program is such a complex phenomenon that single and isolated piece of information about a program's execution cannot offer a comprehensive picture of its performance behavior. To better understand the program behavior, and to provide more coherent performance information, several measurement systems have defined models for the measurement of

parallel/distributed programs based on the interactions between processes. We will discuss some of these systems and their models.

In Gertner's performance monitoring research on RIG (Rochester's Intelligent Gateway)[26, 27] and in a recent case study of the implementation and performance measurement of Dining Philosophers algorithms on the ZMOB distributed system[28, 29], the execution of distributed programs was modeled as a finite state machine (FSM). In both cases, processes of a program executed on individual processors were described by a finite state machine. Message events between processes represented the transition of FSMs and were the basic events monitored in the system.

To measure the performance behavior of a program, users specify a finite state machine that represents the all possible states and the transitions between these states of the program. The FSM of the program is described either in symbolic form in a special language (defined in Gertner's research) or in a tabular representation (given in ZMOB's experiments). Message communication activities are traced at run-time. After the execution of the program, the trace analyzing program reads the FSM representation for the program and the file of message traces, and produces the information of state transitions as well as statistics on state occupancy times of the program's execution.

Finite state machines are general and powerful tools for describing activities that can be represented as states, and the transitions between those states. Nevertheless, the execution of a real program involves such a huge number of possible states that the finite state machine model is not feasible. Also, it is not trivial to model a program's behavior as a finite state machine. Users must specify their own FSM's for each program to be monitored in the system, and the way to do this for any arbitrary program in not clear.

The work by Miller takes another view of distributed programs[30, 31, 32]. A measurement system called DPM, as implemented on 4.2BSD Unix[20] and DEMOS/MP[33] operating systems, is based on a model of distributed computations in which only two actions of a program — computation and communication are defined. The measurement system monitors program activities at the process level and communication events among different processes. The research concentrates on the analysis part of the measurement tool. Various analysis techniques are studied to extract pertinent information from the data collected. Among them, parallelism is used as a characterization of the execution of distributed programs.

The DPM system provides a coherent model for the measurement of distributed programs, and a uniform methodology for building a measurement tool. However, since the model only reflects program activities at the process level, it is difficult for users to relate the measurement results to the underlying structure of the program without further information from the lower levels. Generally, measurement results

are useful in performance evaluation, but the information gained from the measurement is of limited use in finding performance problems within the processes.

One recent, promising approach is the PIE project[34, 35]. The major goal of the PIE system is to support *programming for observability* in a parallel programming environment. The goal of programming for observability is to integrate performance and semantic information at all levels and all stages of development into a single data representation.

The monitoring mechanism in PIE is based upon the insertion of software sensors into the run-time support and into user programs. PIE includes performance data from both the intra- and interprocess levels, and the information about run-time and development time. This data is stored using a relational data model, as proposed by Snodgrass[36] (see next section), which integrates various views of the program's behavior. The PIE system supports graphic representation for different views of program's behavior and provides interactive user interface. Users define their own queries in order to retrieve performance information from the measurement data. Four predefined views on the general behavior of parallel programs (e.g., execution time, memory frame allocation) are provided for the easy access of measurement results.

The integrated view provided by PIE is important to the programmer's ease in using the performance tool[37]. The relational data model integrates data from various levels and facilitates the access of information in the system. However, the relational data model does not match the structure of programs. While PIE provides intensive information for the observability, it gives little help in integrating this information into a coherent picture of a program's execution.

2.4. Performance Measurement for Parallel and Distributed Systems

Performance measurement for parallel and distributed systems is closely related to the performance measurement for parallel and distributed programs. The working environments for them are basically the same; therefore, many mechanisms are applicable to the measurement of both systems and programs. Some measurement systems are developed without recognizing the distinction between the measurements for programs and systems. On the other hand, the special interests of performance measurement for parallel and distributed systems lead to many different approaches in the research and development of measurement systems,

Marathe investigated performance measurement and analysis of multiprocessing computer systems[38] in the development of the C.mmp multiprocessor system[39]. In contrast to the measurement systems for application programs, this research concentrated on the levels of hardware architecture and operating system kernel design. A hardware monitor was used as the primary measurement mechanism. Several experiments were performed for the measurements of instruction mix for

different application programs, multiprocessor contention for shared data in C.mmp/Hydra environment[39], and functional tracing of the operating system. There were no general metrics defined in the system for performance measurement on these levels. The experiments conducted in the research appear as special case studies.

Snodgrass took another approach to the monitoring of distributed systems for performance evaluation[36]. Treating monitoring as an information-processing activity, he proposed using the relational model as an appropriate formalization of the information processed by a distributed monitor. His research emphasized on the issue of data collection and data retrieval for a distributed monitor.

The monitor was implemented on Cm* under both the StarOS and the Medusa operating systems[40, 41]. Event records for program operations are generated by sensors in the operating system, collected by local monitors, and eventually sent to a remote monitor for further processing. The information collected by these monitors is presented to the user as a database of time-varying relations which can be manipulated by a temporal query language. It is the users' responsibility to decide what information to monitor in the system, and how to extract performance results from the measurement data.

The relational model is a good structure for storing and retrieving measurement data. However, as mentioned for PIE, the structure of this model does not match the semantic structure of the system being measured. Snodgrass did not describe what kind of general information is needed to characterize the performance behavior of a distributed system and nor did he indicate how easily this information can be retrieved with queries prepared by users.

Measurement systems are being developed to study special parallel and distributed architectures and applications for these systems. As an example, the MIDAS system[42] is a pyramid structure in which processors are organized into three levels: cluster, secondary and primary processors. Unlike the conventional computer architectures, MIDAS provides 16 independent switchable memory blocks for processors in the cluster. The interconnection network permits any switchable memory block to be dynamically attached to any processor at any time. The system supports the data flow operations by dynamically switching memory modules from processor to processor. The performance measurement of data flow operations is carried out by hardware in secondary processors. The system activity of binding between processors and switchable memories in clusters is recorded. The information about this binding and the queue in which memory modules wait for processors is used to analyze the performance behavior of the system.

The switchable memory modules are the major resource for data flow operations in the MIDAS system. The measurement of the binding between processors and switchable memories is an efficient way to monitor interactions

between processing units. No other metrics and measurement policies were defined in this system. Although the measurement goal of the system is quite limited, the performance information gleaned from the measurement is hardly enough to give a general picture of the performance behavior of the system.

2.5. Discussion

We have reviewed various research in the performance measurement of parallel and distributed systems and programs. As a conclusion, we offer some observations based on our discussion of related work in this area.

In general, we can see that there are many different levels of the detail for the characterization of the performance behavior of parallel and distributed programs and systems. Some earlier approaches focus the measurement interests upon individual levels: such as in the hardware instruction and the kernel activity level (Marathe's, MIDAS), or upon the process activity level (Gertner's, EGPA, DPM). The PIE project integrates various information at different levels and different stages of development into a single data representation. However, since the relational model in PIE does not fit the structure of programs, it is not clear how the mixture of information from various levels is able to form a coherent picture of a program's execution. The structure of a measurement system should match the structure of programs to be measured. In this way, the so-called semantic gap between measurement systems and the objects of measurement 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. The information from such metrics tells much about how good or bad the program's execution is but little about why. Few efforts were directed at the study of integrating performance measurement tools with techniques to locate performance problems and improve the behavior of parallel and distributed programs. This leads to the functional gap mentioned above. 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 to these problems.

Chapter 3

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 for distributed programs. A hierarchical model provides multiple levels of abstraction, supports multiple views of the data represented in the hierarchy, and has a regular structure. The objects in a hierarchical model are organized in well-defined layers separated by interfaces that insulate the internal details of layers. Therefore, we can view a complex problem at various levels of abstraction. We can move vertically in the hierarchy, increasing or decreasing the amount of detail that we see. We can also move horizontally, viewing different components at the same level of abstraction.

In this chapter 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[43] and 4.3BSD 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[44]. 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[45] to our hierarchy with little effort. Our hierarchical structure could be also applied to systems such as HPC[46], which has a different notion of program structuring, or MIDAS[42], which has a 3-level programming hierarchy.

3.1. The Program Hierarchy

In our sample hierarchy, a program consists of parallel activities on several machines. Machines are each running several processes. A process itself consists of the sequential execution of procedures. An overview of our computation hierarchy is illustrated in Figure 3.1. 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.

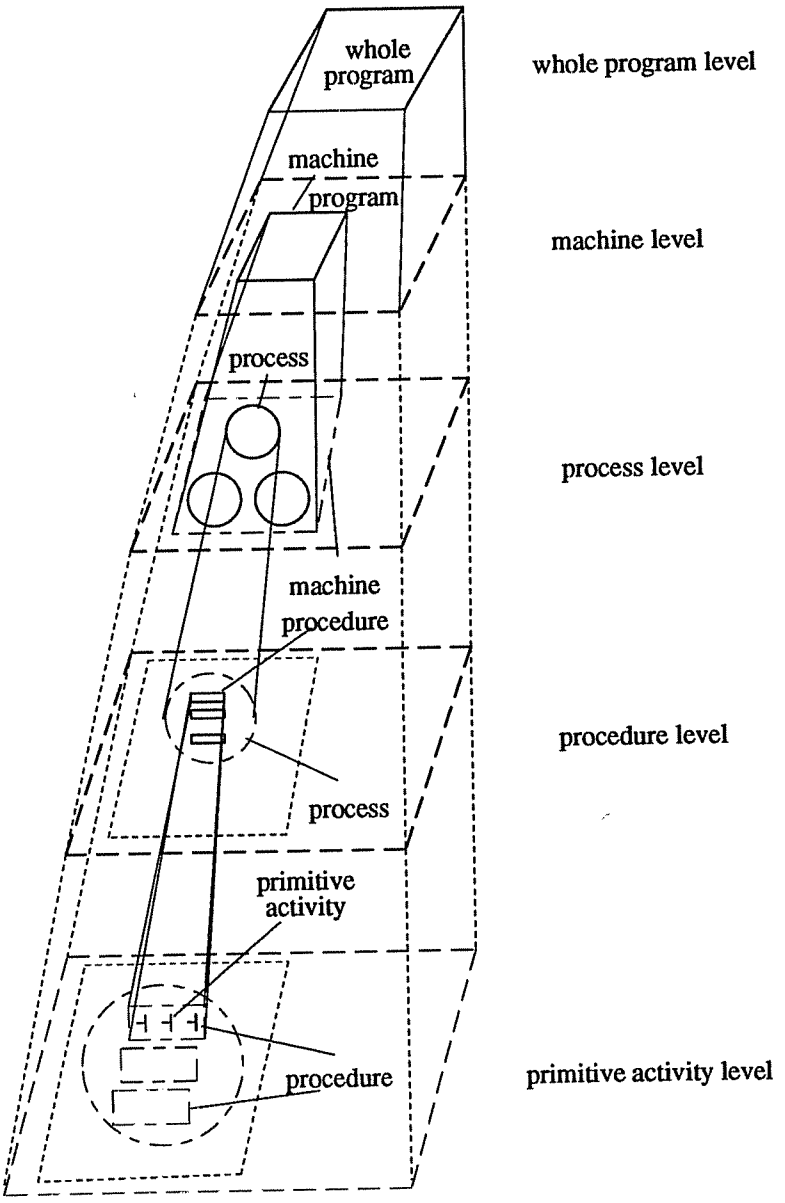


Figure 3.1: Overview of Computation Hierarchy

(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 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 one execution to the next, or even in a single execution as is the case in process migration [33, 47].

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 change dramatically depending on this allocation. It is important to be able to make the distinction between local and remote interactions.

(C) *Process Level*

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

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

(D) *Procedure Level*

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

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

(E) *Primitive Activity Level*

The lowest level of the hierarchy is the collection of primitive activities that are detected to support our measurements. Our primitive activities include process blocking and unblocking by the scheduler, message send and receive, process creation and destruction, procedure entry and exit. Each event is associated with a probe in the operating system or programming language run-time that records the type of the event, machine, process, and procedure in which it occurred, a local time stamp, and event type dependent parameters. The events are listed in Table 3.1.

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

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 3.1: IPS Primitive Events

3.2. The Measurement Hierarchy

The program hierarchy provides a uniform framework for viewing the various levels of abstraction in a distributed program. If we wish to understand the performance of a distributed computation, we can observe its behavior at different levels of detail. We chose a measurement hierarchy whose levels correspond to the levels in our hierarchy of distributed programs. At each level of the hierarchy, we define performance metrics to describe the program's execution. For example, we may be interested in parallelism at the program level, or in message frequencies at the process level. We can look at message frequencies between processes or between

groups of processes on the same machine. This selective observation permits a user to focus on areas of interests without being overwhelmed by all of the details of other unrelated activities. The hierarchical structure matches the organization of a distributed computation and its associated performance data.

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 3.3).

Table 3.2: Performance Metrics

Table 3.2 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 3.3 summarizes the use of these metrics at each level. The list in Table 3.2 is provided as an example of the type of metrics that can be calculated. A different model of parallel computation can define a different program hierarchy with its own set of metrics.

(A') *Program Level*

All of the metrics listed in Table 3.2 are valid at the program level. At this level, these metrics provide a summary of the total program behavior.

Most of the metrics are simple to compute. A few of the other metrics are more complex and can be computed in several ways. For example, utilization, ρ, can be computed as the sum of the ρ's for each machine. Alternatively, it can be derived from the parallelism (speed-up) metric, $\rho = P / N_m$ [31].

(B') *Machine Level*

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

	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 3.3: Performance Metrics for Different Hierarchy Levels

(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.

Chapter 4

Design And Implementation Of IPS Measurement Tool

IPS is a pilot implementation of a performance measurement system for distributed programs, based on our hierarchical measurement model. There are two phases in the operation of IPS — data collection and data analysis. During the first phase, the program is executed and trace data is collected. All necessary data are collected automatically during the execution of the program. There is no mechanism provided (or needed) for the user to specify the data to be collected. During the second phase, programmers can interactively access the measurement results.

This chapter describes the design and implementation of IPS on the Charlotte distributed operating system. After a brief discussion of the basic structure of IPS, we will concentrate on details of the data collection and data analysis.

4.1. 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[48]. Crystal consists of 18 VAX-11/750 node computers and several host computers connected with an 80MB/sec Pronet token ring[49]. 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[43].

4.2. Basic Structure

IPS consists of three major parts: *agent*, *data pool*, and *analyst* (see Figure 4.1) Each of the three parts is distributed among the individual machines in the system. The basic structure of our measurement tool is similar to the structure of METRIC[22] and DPM[32].

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.

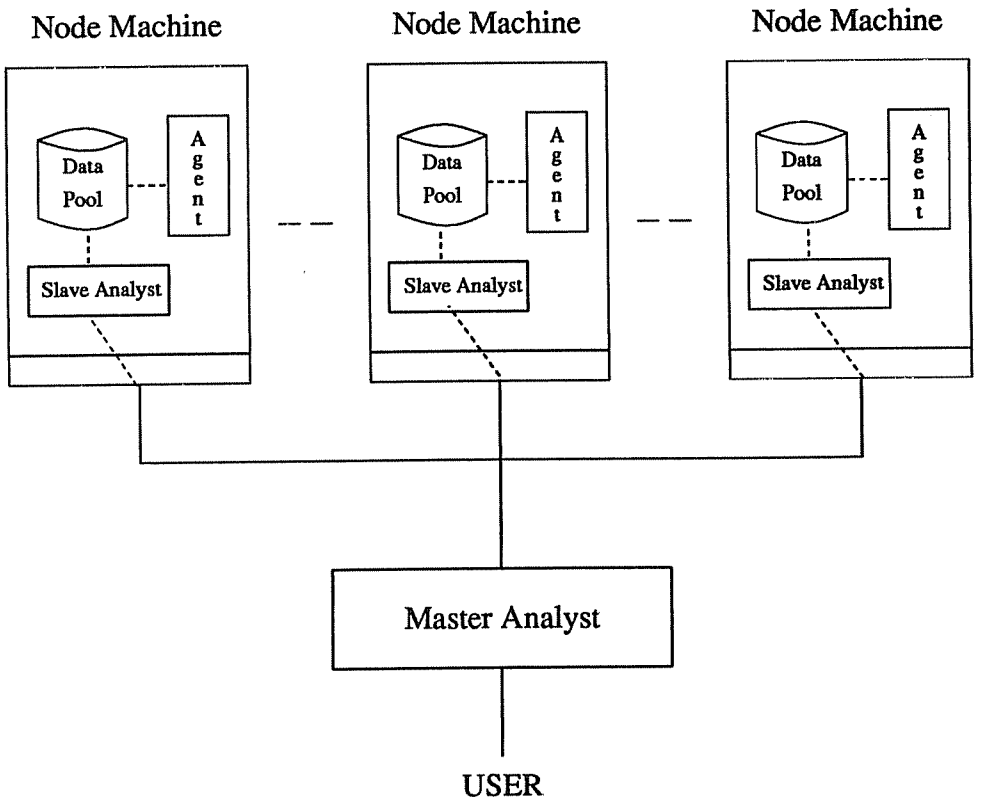


Figure 4.1: Basic Structure of Measurement Tool

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

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

The local data collection and (partial) analysis has several advantages. 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 (see Chapter 5).

4.3. Data Collection

4.3.1. Data Collection Mechanisms

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 system. The measurement results are summarized in Table 4.1.

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 connects system processes during Charlotte OS bootstrap	9.1/sec 1.2/sec	190/sec 30/sec	Charlotte VAX/750
TSP (10 cities) TSP (20 cities)	Traveling Salesman solver.		2029/sec 6639/sec	UNIX VAX/750
Simulation (run 1) Simulation (run 2)	Resource/deadlock simulation		419/sec 113/sec	UNIX VAX/750
vmc (run 1) vmc (run 2)	Wisconsin Modula Compiler		2401/sec 4231/sec	UNIX VAX/750
make (run 1) make (run 2)	UNIX make facility		4918/sec 4658/sec	UNIX VAX/750

Table 4.1: Message and Procedure Call Frequencies

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 see this in Table 4.1, with procedure call rates of over 6000/second — almost three orders of magnitude greater than interprocess events. Due to this high frequency, we use a sampling mechanism combined with modifying the procedure entry and exit code. 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[16] and HP Sampler/3000[15].

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 of procedure in the program[19]. Each time the program enters a procedure, the counter of that procedure is incremented. 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 (see next section).

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.

4.3.2. Costs for Raw Data Collection

A series of tests were conducted to measure the costs of our raw data collection facilities on Charlotte. Overhead from event tracing and overhead of data sampling were treated separately to identify the cost of each component. The overhead for event tracing comes from the extra checking and storing of the event records by various probes in the kernel. The sampling overhead is the amount of CPU time used by the sampling routine to collect the status of the interrupted program and to store the event records. The cost for a single sample includes factors such as the machine speed, the interrupt cost in kernel and the hash table searching in the sample routine.

Test programs with different process organizations (such as master-slave and client-server) were chosen to run on the unmodified Charlotte and Charlotte with measurement facilities. Various combinations of sampling frequencies and event tracing were used to test the measurement costs in different cases. The costs of the raw data collection as compared with the unmodified Charlotte kernel are summarized in Table 4.2. The overall performance penalty for a program running on modified Charlotte kernel and not using the measurement facilities is less than one percent (0.32%). Programs that are using only the event tracing cost an additional 1%. The cost of sampling varies from around 1% to 15% depending on the sampling frequencies chosen. As a result, the total costs for a program under measurement are dominated by the sampling overhead and depend mainly on the sampling precision required by the programmer.

Sampling Frequency	Charlotte Kernel w/ Measurement Facilities	
	No Event Tracing	w/ Event Tracing
No Sampling	0.32%	1.25%
2/sec	1.07%	1.55%
10/sec	2.00%	2.59%
100/sec	14.6%	15.2%

Table 4.2: Average Measurement Costs on Charlotte (Relative to Program Running on Unmodified Charlotte)

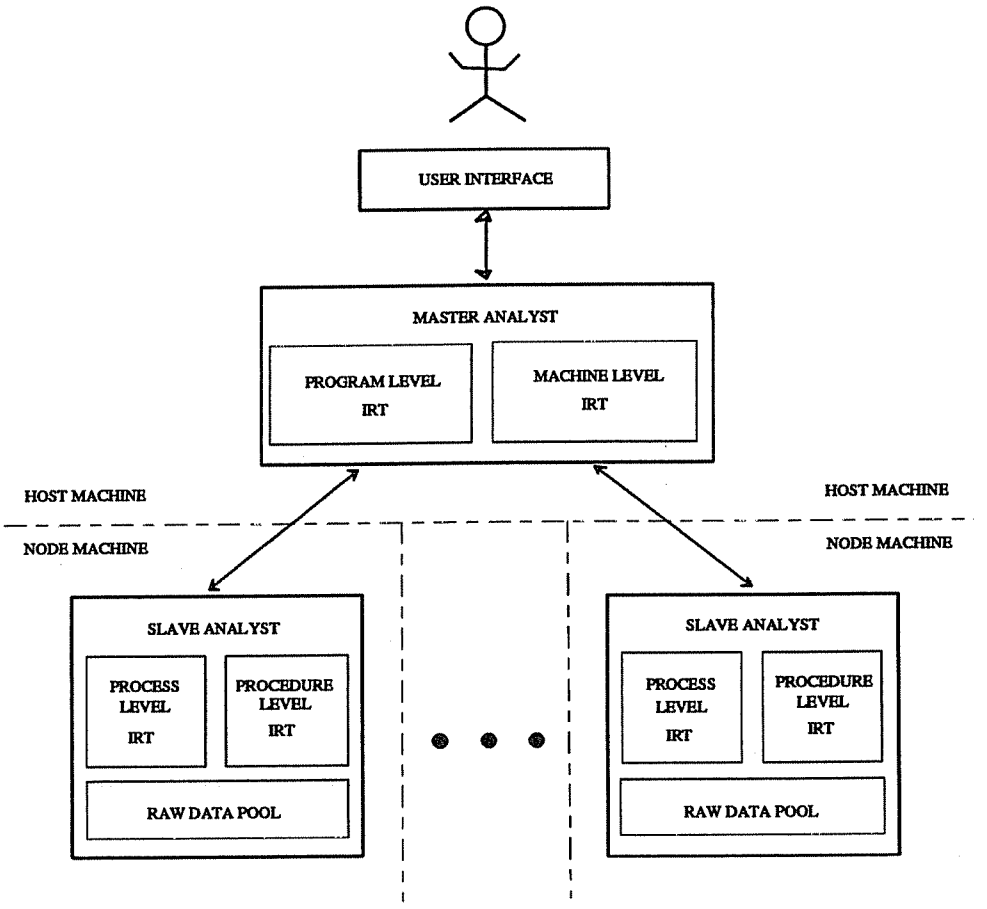


Figure 4.2: General Structure for Measurement Results Analysis

4.4. Data Analysis

4.4.1. General Structure

Figure 4.2 shows the general structure for data analysis in our measurement tool. 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 pre-processed 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 occasionaly 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.

4.4.2. Intermediate Results And User Query Processing

The interactive nature of our performance tool allows users to investigate the performance behavior of the program with a variety of user queries. If every user query required processing the entire pool of raw data, the costs for user query processing would be prohibitive. As a result, we separate the data analysis phase into two stages: intermediate result processing and interactive user query processing. In the first stage, various intermediate results are pre-processed from the raw data and saved in IRT's at master and slave analysts. In the second stage, user queries fetch information from either the intermediate result tables or from the raw data pools.

The basic results from our measurement tool are the performance metrics that are described in Chapter 3. Among these metrics are T_{cpu}, T_{wait_cpu}, T_{wait_sync}, M_n and M_b, which are used for calculating other metrics. Most of user queries retrieve information derived from these metrics. Therefore the task of pre-processing is to calculate and store intermediate results for these metrics.

In our implementation, a metric function is represented as an array. Each entry in the array is a triple: $M[i] = (y_i, t_{l_i}, t_{h_i})$. The y_i is the accumulated value of the function up to the time t_{l_i}. The low range, t_{l_i}, and the high range, t_{h_i}, define a time interval within which $dM/dt = 1$, whereas the function value is constant during the time interval from t_{h_i} to $t_{l_{i+1}}$. The following algorithm computes the value of $M(t)$ at time t:

repeat
    Search the array for element i;
until $t_{l_i} \le t < t_{l_{i+1}}$;
if $t_{h_i} \le t < t_{l_{i+1}}$ **then**
  {t is in the time interval where the function value does not change}
    $M(t) = y_i + t_{h_i} - t_{l_i}$;
else
  {t is in the time interval where the function value is changing}
    $M(t) = y_i + t - t_{l_i}$;

Since the array is sorted by time, we can use binary search to find element i for an arbitrary time t. This is more efficient than searching the raw data pool. Only one pass through the raw data records is needed for pre-processing all array representation of various metric functions. This overhead of IRT pre-processing is worthwhile in view of the gain of efficiency in user query processing.

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. However, the major differences of our query processing from a relational database system are:

(a) Many operations in the data analyses require random access to individual event records in the trace data. This type of access is typically not efficient in a relational database.

(b) All trace data for a measurement session are read-only. This simplifies the problem of concurrency control for the data consistency.

(c) There are no replicated data in our system. Data on each node machine is unique and local to the slave analyst on that machine. Consequently the general strategy for processing raw data is relatively simple — most raw data is processed locally by the slave analyst.

4.4.3. Clock Coordination

The calculation of performance metrics in the IPS system depends upon the ordering of different events in the system. For most metrics, this ordering only involves events occurring in the same machine (e.g., metrics for the individual machine and process). For other analyses (such as the critical path analysis in Chapter 5), 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 arises when calculating the elapsed time of the whole program.

Maintaining total ordering among events in a distributed environment requires synchronizing clocks on different machines, which involves extra overhead at run-time and is not trivial[50]. Clock synchronization can be only approximated, the accuracy of which depends upon the technology and the environment. In our implementation on Charlotte, clocks on node machines are reset every time a new measurement session is started. The coordination of clock time on different machines is of concern only during the execution of a measured program. This time is relatively short (most likely within several hours) compared to the lifetime of a system. With current quartz technology, we can assume that the drift of the clock frequency rate within such a short period of time is very small[50]. Therefore, we adopted a simple approximation method based on the TEMPO algorithm[50], which keeps the clocks in a local network synchronized with an accuracy comparable to the resolution of each individual clock.

The basic idea for clock coordination lies in calculating the fixed clock offsets among different machines, using the timing information collected from monitoring message communications between these machines. For each inter-machine message, we record the local time when the message leaves a machine and the local time when

the message packet arrives at another machine. This information is sent to the master analyst, where the clock offsets are computed.

The method used for our clock coordination provides accuracy in the same order of magnitude as the resolution of the clocks in individual machines (clock resolution is $1ms$ in our tests). The approximation in this small range (a few ticks of the clock) has little effect on calculations of metrics that take hundreds or thousands of clock ticks (e.g., the elapsed time of the whole program). In addition, calculations for most metrics only depend on the information of local clocks. Hence, the approximation of clock coordination has no influence on the accuracy of the results for most metrics.

A basic assumption of the TEMPO algorithm is that the distribution of transmission times can be considered the same in both directions. This condition can also hold true for a ring network (as in the case of Charlotte), since the differences in the distribution of time delays due to asymmetry in the length of the wire are negligible with respect to the other components, such as the software overhead and the buffer delays in message transmission.

Chapter 5
Automatic Guidance Techniques

5.1. Introduction

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 chapter, we shall discuss some analysis techniques that support our approach.

The previous chapters describe a system for measuring the performance of the parts of a distributed program at different levels of detail. A programmer can use this information to manually evaluate the behavior of the program. In the simplest form of this system, the programmer starts at the top (program) level of the hierarchy. Using available metrics, the programmer derives a general picture of the execution of the program and decides where to look in the next (machine) level of the hierarchy. This decision may be affected by the choice of the machine with the smallest utilization or highest procedure call rate. Next, at the process level, the programmer can examine the performance metrics of each process on the machine and choose that which appears to have the largest performance effect. This procedure can be continued down the hierarchy to the procedure level. In this way, users can obtain a picture of the program's execution and focus their attention on the points of greatest interest. However, to obtain all the information, users must proceed through various details of searching in the hierarchy. This process is time-consuming, and it is sometimes very difficult to pinpoint problems by referring only to the performance metrics of individual components in the program. A good measurement system should provide further facilities for aiding users to understand a program's behavior, to locate trouble spots, and to improve program efficiency.

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% of the execution time. We could hide this problem by splitting the procedure into 10 subprocedures, each responsible for 9% 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[51] on different machines can have a great effect on the performance of the entire program.

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. This chapter presents the implementation of this technique in IPS. Section 5.2 addresses the basic concept of critical path analysis in the execution of distributed programs. Section 5.3 provides a definition of the critical path as shown in the program activity graph and describes the construction of these graphs based on data collected from the measurement. Different algorithms for critical path analysis and the testing of these algorithms are presented in Section 5.4.

5.2. 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* [52] identifies where in the hierarchy we should focus our attention. As an example, Figure 5.1 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 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:

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

(b) The time required for each activity can be measured.

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

(d) 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.

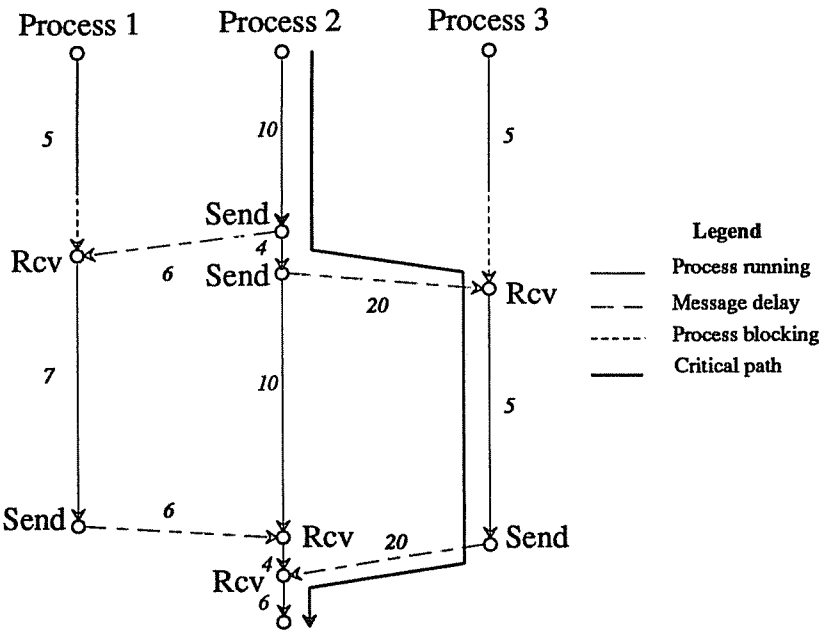


Figure 5.1: Example of Critical Path at Process Level

Based on these properties of a distributed program, we can use the critical path method (CPM)[52, 53] to analyze a program's execution. The method is commonly used in operational research for planning and scheduling, and has also been used to evaluate concurrency in distributed simulations[54].

In contrast to CPM, 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.

Turnaround time is not the only critical measure of the performance of parallel programs. Often the throughput is more important, e.g., in high-speed transaction systems[55]. Our discussion in this chapter concentrates on the issue of critical path analysis, but we also address techniques for deriving the throughput information from the trace data.

5.3. 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* (PAGs). The longest path in a program activity graph represents the critical path in the program's execution. In this section, we define the program activity graph and related ideas. We then describe how various communication primitives of distributed programs are represented in program activity graphs and how these graphs are built on the basis of information obtained from program measurement.

5.3.1. Definitions

The definition of program activity graph is similar to that of an activity network in project planning[56]. The execution of distributed programs can be divided into many nonoverlapping individual jobs, called *activities*. Each activity requires some amount of time, called the *duration*. A precedence relationship exists among the activities, such that some activities must be finished before others can start. Therefore, 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 work in the program. More than one edge can exist between the same two vertices in a PAG.

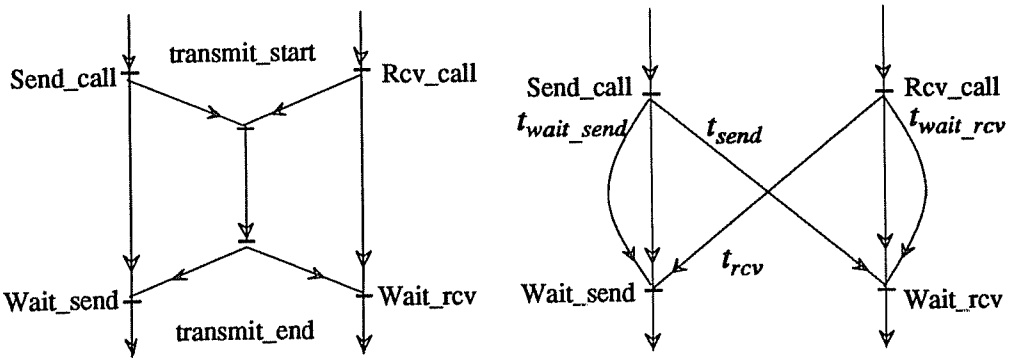
The critical path for the execution of a distributed program is defined as the longest weighted path in the program activity graph. The length of a critical path is the sum of the weights of edges on that path.

5.3.2. Construction of Program Activity Graphs

A program activity graph is constructed from the data collected during a program's execution. There are two requirements for the construction of program activity graphs: first, the activities and events represented in a PAG should be measurable in the execution of programs; second, the activities in a PAG should obey the same precedence relationship as do program activities during the execution.

Two classes of activity considered in our model of distributed computation are computation and communication. For computation activity, two basic PAG events are starting and terminating 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[43]. 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 Figure 5.2, a simple PAG for message send and receive activities in a program. Two extra events of transmit_start and transmit_end are included in Figure 5.2a to depict the underlying relationship among various communication events. They represent the actual data transmission inside the operating system. Since these extra events occur below the application program level, we do not consider them in our PAG. Therefore, we transform the graph in Figure 5.2a into that in Figure 5.2b and still preserve the precedence relationship among the basic communication events.



(a) Detailed Version **(b) Simplified Version**

Figure 5.2: Construction of Simple Program Activity Graph

The weights of message communication edges in Figure 5.2b (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 inter-machine sends and receives) by measuring the round trip times of intra- and inter-machine messages for 10000 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 PAGs for application programs.

5.4. 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. We have implemented a distributed algorithm for finding the one-to-all longest paths from a designated source vertex to all other vertices. A centralized algorithm was also tested as a standard for comparison with distributed algorithms. In this section, we describe some details of the implementation and testing of these algorithms, and provide comparisons of test results and some remarks on the algorithms.

5.4.1. Assumptions and Our Testing Environment

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 that we studied 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.

A program activity graph consists of several subgraphs that are stored in different host machines. The data to build these subgraphs are collected during the execution of application programs. We can copy subgraphs between node machines; the copying time is included in the execution time of algorithms. However, our measurements indicate that this copying time is much less than one percent of the total execution time. All subgraphs were sent to one machine to test the centralized algorithm. In testing the distributed algorithm, subgraphs were either locally processed or sent to some collection of machines to be regrouped into bigger subgraphs.

We used two application programs to generate PAGs for testing the longest path algorithms. 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. In graphs generated from Application 1, more than 50% of the total vertices were in one subgraph, while the remaining ones were evenly distributed among the other subgraphs. The vertices in the graphs from Application 2 were evenly distributed among all subgraphs.

All of our tests were run on VAX-11/750 machines. The centralized algorithms ran under 4.3BSD UNIX, and the distributed algorithms ran on the Charlotte distributed operating system[43].

5.4.2. Diffusing Computation on Graphs

Diffusing computation on a graph, proposed by Dijkstra and Scholten[57], is a general method for solving many graph problems. All of our algorithms for the longest path problem are variations of this method. Therefore, we will first give a brief description of the general structure of diffusing computation before we discuss

the details of our algorithms.

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.

5.4.3. Test of Different Algorithms

We chose the PDM shortest-path algorithm as the basis for our implementation of centralized algorithm[58]. The experiments of Denardo and Fox[59], Dial et al[60], Pape[61], and Vliet[62] show that, on the average, the PDM algorithm is faster than other shortest-path algorithms if the input graph has a low edges-to-vertices ratio (in our graphs, the ratio is about 2). An outline of the PDM algorithm and a brief proof for the correctness of the algorithm are given in Appendix A. More detailed discussion of the algorithm can be found in [58].

Our implementation of the distributed longest path algorithm is based on Chandy and Misra's distributed shortest path algorithm[63]. Every process represents a vertex in the graph in their algorithm. However, we chose to represent a sub-graph 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 sub-graph, 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 sub-graphs (processes). Each process keeps individual message queues to its neighbor processes. An outline of the two versions of the distributed algorithm and a proof of the correctness of the algorithm appear in Appendix B. A detailed discussion of the algorithm is given by Chandy and Misra[63].

We tested our algorithms with graphs derived from the measurement of Applications 1 and 2. The total number of vertices in the graphs varies from a few thousand to more than 10,000. Figure 5.3 shows the test results for the distributed algorithm. For asynchronous execution, we show only one time result in the figure for comparison with the time for synchronous execution. The execution times for asynchronous and synchronous execution differ dramatically.

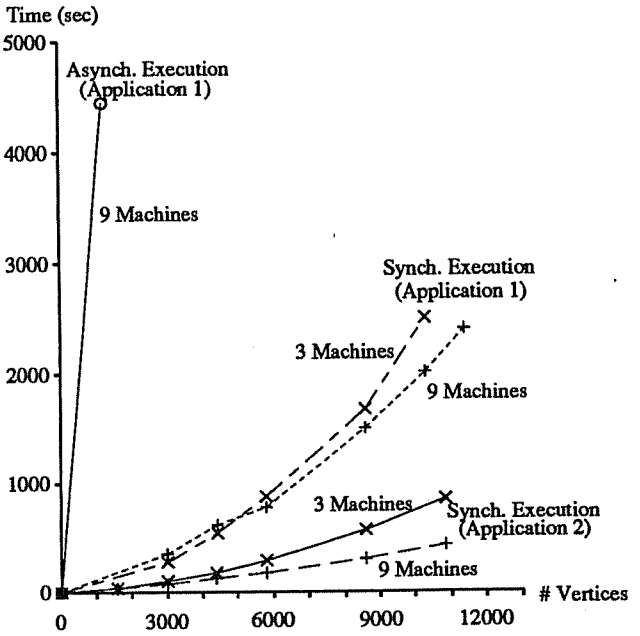


Figure 5.3: Execution Times for the Distributed Algorithm

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 Figures 5.4, 5.5, 5.6, and 5.7. We can see from these measurements that the distributed algorithm with larger input graphs and more machines resulted in greater speed-up but less efficiency.

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.

We have observed a speed-up of almost 4 with 9 machines in synchronous execution of the distributed algorithm. Speed-up increases with the size of the input graph and the number of machines participating in the algorithm. On the other hand, the efficiency of the algorithm decreases as more machines are involved in the algorithm. 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.

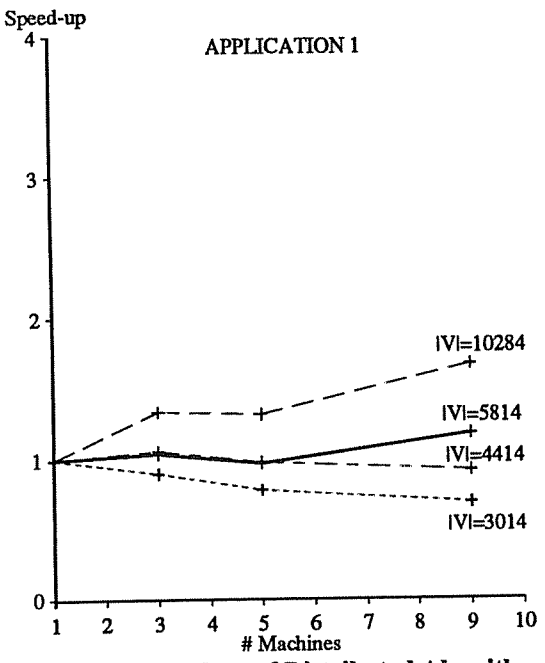


Fig. 5.4: Speed-up of Distributed Algorithm

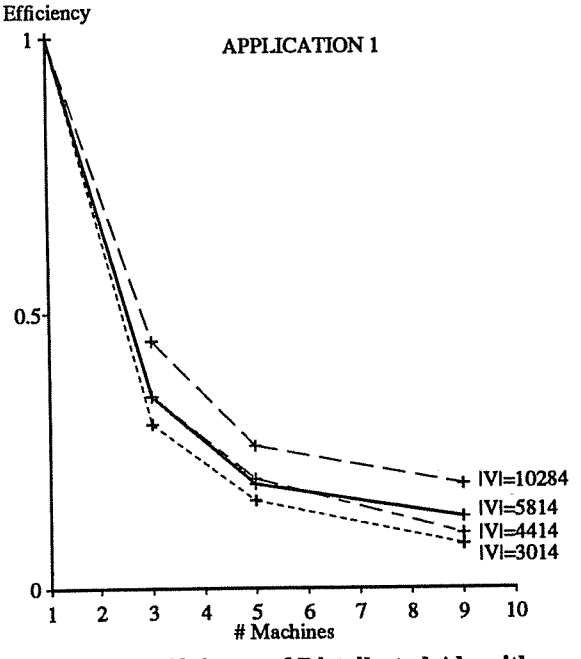


Fig. 5.5: Efficiency of Distributed Algorithm

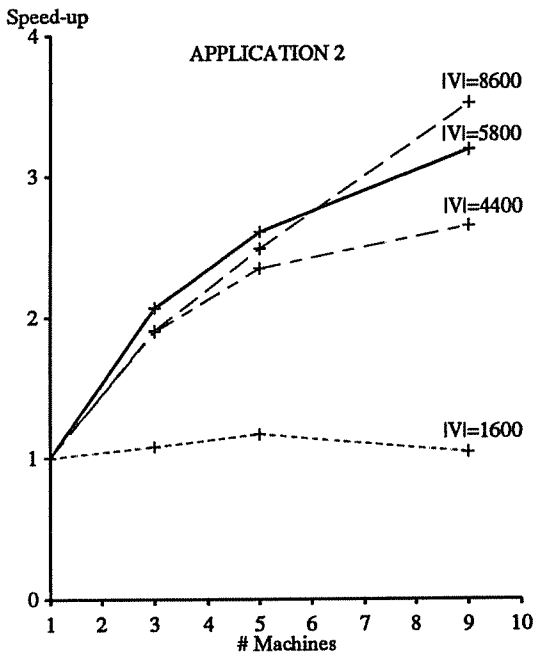


Fig. 5.6: Speed-up of Distributed Algorithm

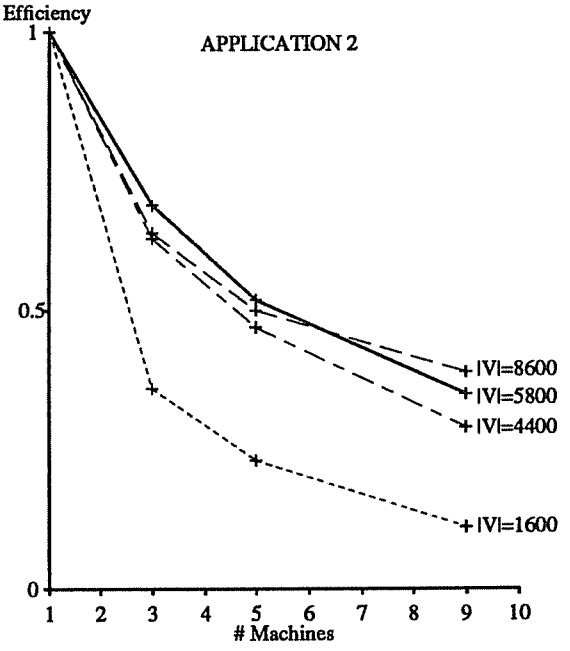


Fig. 5.7: Efficiency of Distributed Algorithm

5.5. Remarks

The technique of critical path analysis is one method that we developed to provide guidance for locating performance problems in the program. A PAG is created from the data collected during program's execution. The longest path in this graph represents the critical path in the execution of the program. Knowledge about this path helps programmers identify the possible bottlenecks in the program. We have implemented different algorithms to calculate the critical path in a PAG. Due to the acyclic nature of PAG's, these algorithms are simple and efficient. The distributed algorithm for critical path analysis attains a limited speed-up and is well fit to the structure of master-slave analysts in IPS.

Presentation of results of critical path analysis offers some interesting problems. A PAG may contain more than 100,000 nodes; the critical path may contain a nontrivial percentage of these nodes. We use statistical presentation techniques to display the (time weighted) most commonly occurring nodes, and the most commonly occurring sequences in the path. We then use high-level language debugging techniques to relate these events directly to the source program. Observing the most commonly occurring sequences allows us to detect performance bottlenecks that span procedure, process, or machine boundaries. Performance problems that are divided among several procedures, or even among processes or machines, will be readily apparent.

In next chapter, we will give a sample measurement session for a real application program to show how an automatic guidance technique can be integrated into a performance measurement system, and how the guidance information can help users find performance problems in the program.

Chapter 6

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. This chapter 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. Note that all of the tables and histograms presented in this Chapter come directly from output of the IPS system.

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 thesis.

This chapter is divided into three sections. The first section describes the sample program we have chosen to test the IPS measurement system. The second section describes the measurement session and analyses of the program's behavior. The last section is a brief summary of IPS features that are important in the performance measurement of distributed programs.

6.1. The Program for Measurement Tests

The program we have chosen for measurement tests on the IPS system is an implementation of the *Simplex method* for linear programming[64]. There are two reasons why we use this program as a sample program for measurement tests in our system. First, the Simplex method is popular as an application program in various areas of science and engineering. Second, a test program for the Simplex method has already been implemented on the Charlotte distributed system by Bhide[65]. It is more interesting to measure a real program rather than one written especially for our own measurement. The selection of a sample program for our measurement is relatively arbitrary, but our major concern is to show the use of the IPS system on a real program.

The Simplex method is an important linear programming tool[64, 53]. It is used to obtain an optimal solution for a system that has linear constraints and an objective function. This objective function is a linear combination of constrained variables. The aim of the method is to solve the system so that the objective function is maximized.

We can represent the system in the matrix form as follows:

Constraint equations: $\mathbf{Ax} = \mathbf{d}$.

Objective function: $\mathbf{cx}$.

Where

A is an $m \times n$ matrix
x is a column of n elements
d is a column of m elements
c is a row of n elements

Let the matrix **B** be constructed as follows:

$$\mathbf{B} = \begin{bmatrix} \mathbf{A} & \mathbf{d} \\ \mathbf{c} & -z \end{bmatrix}$$

The variable z denotes the value of the objective function. At the initial step, the value of z, which is a linear combination of the constrained variables, is zero, since the value for all constrained variables is assumed to be zero. In the following discussion we assume that the system has an unique solution for all constrained variables. The absence of such a solution would be easily detected.

6.1.1. Basic Serial Algorithm for Simplex Method

The basic algorithm for the Simplex method consists of repeated iterations of the following steps:

(1) Select column j such that $\mathbf{c}_j > 0$.

(2) Select row i with the smallest positive ratio $d_i/\mathbf{A}_{i,j}$.

(3) Perform row operations on matrix **B** to achieve
$\mathbf{B}_{i,j} = 1$, and $\mathbf{B}_{row,j} = 0, \quad 0 \le row \le m+1, \; row \ne i$

Failure to find a positive $\mathbf{c}_j$ in the first step implies that the objective function cannot be improved, and that the optimal solution has been found. In step 2, if no i can be found that satisfies the criterion mentioned, another column j needs to be selected in step 1. Inability to find such an i for all columns with $\mathbf{c}_i > 0$ means that the optimal solution for the system has been found.

6.1.2. Columnwise Distribution Algorithm of Simplex Method

One way to distribute the work of the serial algorithm for the Simplex method is to divide columns of the B matrix into several groups, and to have individual processes conduct calculations within each group. We call this the *columnwise distribution algorithm.*

Assume that the job is distributed among q calculator processes such that q divides n. Each calculator process is allocated n/q columns of matrix **B**, such that process 1 has columns 1 through n/q, process 2 has columns $n/q+1$ through $2n/q$, and so on. Each process also has a copy of column vector **d**, and there is a controller process that coordinates all calculator processes.

The columnwise distribution algorithm allows several iterations to be completed before any communication is needed. We call the intervals between communications *rounds* to distinguish them from iterations. Two parameters in the

algorithm, k and s, are used to select the number of columns commonly distributed among calculator processes within each round. At the start of each round, every calculator process chooses k columns from its set of columns that have a positive c value (the last row in matrix **B**), and sends these k columns to the controller process. The controller selects s columns from the total kq columns sent by all calculator processes, and broadcasts them to all calculators. Each calculator then has its own individual n/q columns, plus the globally-known s columns, at most k of which are duplicates of its individual ones.

For each iteration of the round, all calculators choose the same globally-known column (i.e. the one with the highest **c** value), determine the appropriate row on which to base a row operation, and then perform the row operations on the columns they possess. All calculators make the same choices, since they share globally-known columns, and they modify those columns identically without need of further communication. A round continues either until a fixed limit on the number of iterations has been reached, or until no globally-known column can be used (i.e., all **c** values are nonpositive). This calculation is shown in Appendix C in greater detail.

6.2. Measurement Session of the Test Program

In the following discussion, we describe an IPS measurement session of the Simplex program. 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.

6.2.1. Interactive Measurement Session

IPS provides an interactive user interface for the measurement of programs. The interface supports a command menu from which users can choose appropriate actions. Figure 6.1 shows the general format of a command menu. 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.

Metrics	Histogram	Event Path	Select Member	Down Level	Up Level	Quit

Figure 6.1: Format of a Command Menu

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

Figure 6.2: Metrics at Program Level

The measurement session starts at the program level. We can select a command from the menu to display performance information at this level. Figure 6.2 shows the metrics at the program level. These metrics are defined in Section 3.2, 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. Figure 6.3 shows a histogram of CPU time at the program level.

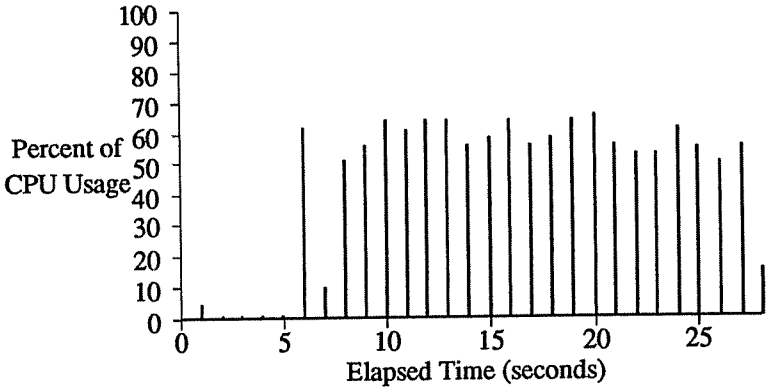


Figure 6.3: Histogram of CPU Time at Program Level

The performance information at the program level gives us a general characterization of the program's execution. For example, we note in Figure 6.3 that there is a quiet period at the beginning of the program's execution. We would be interested in finding which part of the program causes this idle period. As we can see from Figure 6.2, 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	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

Figure 6.4: Metrics at Machine Level

For more detailed information, we continue our study at the machine and process levels. Figure 6.4 shows the metrics at the machine level. Each machine has 3 processes. 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 Figure 6.5. Two calculator processes (processes 4 and 5) in machine 1 have similar performance behavior. We show only one set of process data for machine 2 (see Figure 6.6), since all other processes in machines 2 and 3 are calculator processes and have similar performance results. We can also check histograms of the different metrics at machine and process levels. By comparing histograms in different machines and processes, we find that the quiet period in Figure 6.3 corresponds to the time period when only the controller process is busy, generating the initial problem (the matrix **A**, and elements **c, d**); all the calculator processes wait idly.

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

Figure 6.5: Metrics at Process Level (in Machine 1)

Machine and process level information helps us identify the performance behavior of individual items in the program. We find only a slight difference between the performance of calculator processes running in Machine 1 and those in Machine 2 or 3. On the other hand, the controller process spends 34% of its time in computing, 47% of its time in waiting for messages, and 19% of its time in waiting for the CPU, while the calculator processes spend 11% of their time in computing, 85% of their time in waiting for messages, and 4% of their time in waiting for the CPU. These facts demonstrate that there is no major contention for computing resources among processes since each process spends little time in waiting for the CPU. However, interactions (appearing as messages) between controller process and all calculator processes are heavy, and waiting for communication dominates the execution of each calculator process.

Metrics	Process 3
Process Name:	calcs[2]
Elapsed Time(ms):	27167
Cpu Time(ms):	2589
Block_sync Time(ms):	23996
Block_cpu Time(ms):	611
Message Traffic(#):	43
Message Traffic(bytes):	139496
# of Procedures:	6
# of Procedure Calls:	52
Response Ratio:	8.94
Load Factor:	1.32

Figure 6.6: Metrics at Process Level (in Machine 2)

We can go further, to the procedure level, to investigate the behavior within each process. Figure 6.7 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[16, 15]. The information about the distribution of CPU time in different procedures can help users determine which part of the program code dominates the execution. As discussed above, the procedure *Init* in the controller process, which is in charge of generating the initial problem, takes 23% of the controller's CPU time and creates the quiet period in Figure 6.3.

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%)

Figure 6.7: Execution Profiling of Controller and Calculator

6.2.2. 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 items 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. Figure 6.8 gives the critical path

information at the program level. We can see that the communication cost (including inter-machine and intra-machine 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[43].

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

Figure 6.8: 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 machine i, * denotes less than 1%)

Figure 6.9: Critical Path Information at Process Level

The critical path information at the process level (see Figure 6.9) gives us more details about the program's execution. The execution of the controller process takes 58% of the whole length of the critical path, while the execution of all calculator processes take less that 5% of the whole length. The domination of the controller process 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[32, 66], 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.

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%)

Figure 6.10: Critical Path Information at Procedure Level

Finally, we display critical path information at the procedure level in Figure 6.10. This information can be useful in locating performance problems across machine and process boundaries. The top three procedures, that take 49% of the entire length of the critical path, are in the controller process (again, we see the procedure *Init*). Procedure *MainLoop* in the calculator processes, which is in charge of communications between calculators and the controller, takes 33% of the entire execution time of each calculator process (see Figure 6.7). However, they are much less noticeable in the critical path because of the dominance of the controller process.

6.2.3. Discussion

Previous sections demonstrate how a measurement session proceeds interactively in the IPS system and how the critical path information is useful in discovering performance problems in the program.

We have seen that the execution of the controller process dominates the performance behavior. This is because, 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 Figure 6.11. 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 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.

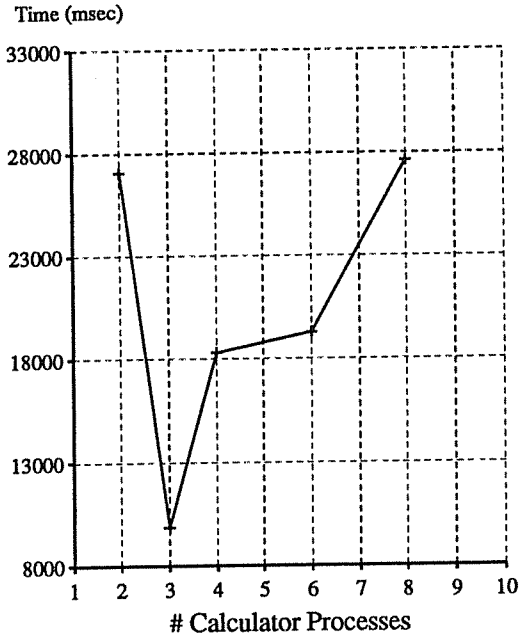
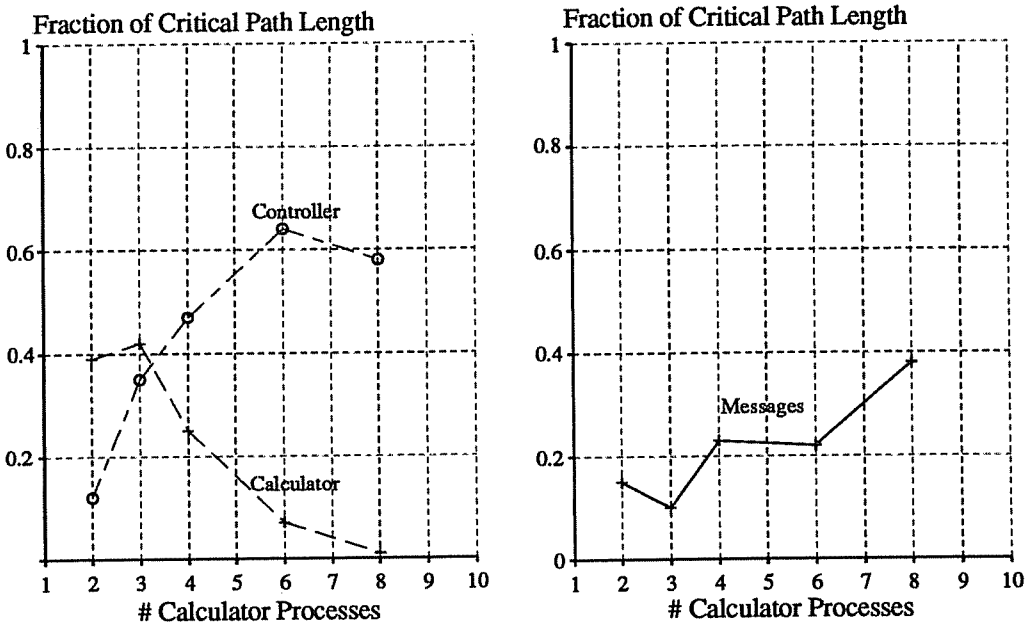


Figure 6.11: Program Elapsed Time in Different Configurations

The critical path information for these tests (shown in Figure 6.12) supports our observation. For the configuration of 3 calculator processes, the controller and calculator processes have the best balanced processing loads, and the lowest message overhead. This coincides with the shortest execution time in Figure 6.11. 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 D contains a proof that supports our claim that for programs with the master-slave structure, 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.



(a) Process Components **(b) Message Components**

Figure 6.12: Components of the Critical Path

6.3. Summary

The measurements presented in this chapter show a wide variety of uses for the IPS measurement system. Performance information in the IPS is well-organized and easy to use. A complete and clear picture of the performance behavior can be obtained from various metrics and histograms at different levels of the hierarchy. The layered abstraction of the hierarchical structure helps users to focus their interests. We have also demonstrated the power of guidance techniques in a measurement system. The critical path information guides users in locating performance problems in the program. In addition, it allows us to predict the behavior a computation will show under different configurations. It is possible to accommodate other guidance techniques in IPS; this is an area of current research.

The total processing time for a measurement session depends on the range of the execution time for the program under measurement. It is within several minutes for most of our test programs.

Chapter 7

Conclusions

Our research in performance measurement tools for parallel and distributed programs focuses on two major issues:

(1) How a performance measurement system can provide a complete picture of the execution of a program, and how this information should be organized so that users can easily and intuitively access all the data without being inundated with irrelevant details.

(2) How a performance measurement tool can supply more information than just performance statistics, and how such a tool can guide the programmer in locating performance problems and in indicating possible ways to make further improvements.

Our approach to the first issue starts with a hierarchical model of distributed programs. The hierarchical model is a natural way to describe distributed programs and their performance. It provides a single, regular structure to describe a program, from the statement level up to the entire program. This regular structure provides views from many levels of abstraction, allows easy and intuitive access to performance information, and simplifies the construction of tools that reason about a program's behavior.

The approach to the second issue is based on the development of automatic guidance techniques that can direct users to the location of performance problems in the program. The construction of guidance techniques is facilitated by the regular structure of the program and measurement hierarchy. The guidance techniques, combined with a user interface that directly relates performance results to the program source code, allow the programmer to concentrate on fixing problems rather than on finding them.

A prototype, the IPS system, has been built on the Charlotte distributed operating 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 multi-level 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.

Future Research

Our research as described in this thesis suggests several directions for future work. Some of these directions are outlined below:

- Different hierarchies: The hierarchical model is a flexible model that can be targeted to many different application domains. The hierarchy implemented in our IPS system is based on the specific requirements of our application environment. The hierarchy is chosen to match the structure of objects being measured. For example, to measure the performance behavior of communication protocols in a long-haul network, we could choose a measurement hierarchy that matches with the hierarchy of ISO 7-level model[68]; whereas, for a database system, we might choose levels corresponding to queries, transactions, atomic operations, and physical device access[69].
- Other automatic guiding techniques: A performance system should be able automatically to guide the programmer in locating performance problems and in helping users improve program efficiency. Developing automatic guiding techniques in performance measurement systems opens a new research area. It requires a combined knowledge of disciplines such as performance measurement, program semantics, and algorithm design. One example of such guiding techniques involves detecting time phase behavior during a program's execution. With this information, users can identify different phases in a program's execution and concentrate on those phases that hold the most interest. The use of knowledge-based debugging and diagnostic systems[70,71,72] suggests another way to extract guiding information from the measurement data. The predefined analysis techniques of a measurement system can be stored as rules in the knowledge bases. Acting as an expert system, the measurement tool might be able to incorporate more complex guidance strategies.
- Measurement systems for a shared memory environment: The basic ideas in our research are also applicable to measurement in shared memory environments. However, new questions are raised for the performance measurement of shared memory systems. In these systems, the granularity of concurrency in a program is smaller. Instead of using messages, the basic synchronization mechanisms will be based on access to shared memory spaces (such as semaphores and monitors). Therefore, the primitive events that such a measurement system needs to monitor will occur with much higher frequency.
- User interface: The user interface is an important aspect in the design of a performance measurement system. The current technology of human-machine interfaces (such as window systems, mouse input, color and graphic displays) provides many options for the design of better user interfaces in performance measurement systems. Design issues include how graphic representation of program structures can be adopted for displaying a program's behavior, and how different representations of multidimensional, statistical information, such

as Chernoff's face representation[73, 74, 75, 76], can be used for the presentation of time-varying performance data in measurement systems.

Appendix A

Centralized Algorithm for Critical Path Analysis

Following is a sketch of the centralized algorithm for critical path analysis (see Chapter 5).

```
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

Based on the discussion in Deo's paper[58], we now give an informal proof of the correctness of our algorithm. We only prove the correctness of the asynchronous version of our algorithm, since the synchronous version of the algorithm is a special case of the asynchronous version in which the diffusion of the length information in the graph follows a special pattern – the information about the longest path to a vertex arrives first at each vertex.

During the execution of the algorithm, the label $D[u]$ is always updated to be the currently known longest length from SOURCE to u, and $P[u]$ is always updated to be the predecessor vertex of u on the currently known longest path from SOURCE to u. Since each insertion of a vertex u into Q is preceded by an increment of $D[u]$, and for a finite graph without cycles this $D[u]$ bounds to be finite, this algorithm is guaranteed to terminate.

To see that the $D[u]$'s do indeed converge to the longest length, we first note that at termination $D[v] \geq D[u] + l(u,v)$ holds for every edge (u,v). Suppose the vertex sequent (SOURCE $\equiv u_0, u_1, ..., u_k \equiv u$) is a path from SOURCE to u, then its path length is given by
$l(u_0,u_1) + ... + l(u_{k-1},u_k) \leq (-D[u_0] + D[u_1]) + ... + (-D[u_{k-1}] + D[u_k]) = -D[\text{SOURCE}] + D[u] = D[u]$.

Thus, $D[u]$ is the longest length from SOURCE to u, and the vertex sequence, SOURCE $\equiv P[...P[u]...], ... , P[[u]], P[u], u$ is the longest path from SOURCE to u as obtained from our algorithm.

Appendix B

Distributed Algorithm for Critical Path Analysis

Following is a sketch of the distributed algorithm for critical path analysis (see Chapter 5):

```
for all u in sub-graph G do
   Counter[u] := 0;
   D[u] := 0;
end
initialize 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
initialize 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

In our implementation, each process has a job queue, Q, and a set of queues for messages to neighbor processes. There are two kinds of messages used in the algorithm. One is the Length message, which contains the length of the longest path currently known from SOURCE to a vertex and the predecessor on that path. Another one is Ack message, which is used to send a response to the predecessor denoting that the length sent by the predecessor has been (or will be) taken into consideration by all vertices reachable from the vertex. A length message or an Ack generated during the diffusion is either put into Q or into the corresponding message queue depending on whether the destination is local or remote. Length messages and Acks received from other processes will be unpacked and put into the local job queue.

Three local variables are maintained for each vertex u: $D[u]$ is the length of the path, $P[u]$ is the predecessor vertex of u on the longest path currently known from SOURCE to u, and Counter[u] is the number of unacknowledged messages, that is, the number of messages sent from this vertex for which no Ack has been received so far. Note, if Counter[u] > 0 at any time, then a vertex has exactly one message to which it has not sent an Ack, and this Ack should go to $P[u]$. At initialization, Q in each process is either empty or includes only the SOURCE, depending on which sub-graph the SOURCE belongs to. Also Counter[SOURCE] is set to the value of the number of out-going edges from SOURCE. The termination condition for the process including the SOURCE vertex is when Counter[SOURCE] equals to zero, and other processes will then receive special termination messages from this process.

We now give an informal proof of the correctness of our algorithm based on the discussion in Chandy-Misra's paper[63]. We only prove the correctness of the asynchronous version of our algorithm, since the synchronous version of the algorithm is a special case of the asynchronous version in which the diffusion of the length information in the graph follows a special pattern – the information about the longest path to a vertex arrives first at each vertex.

We assume our graphs are finite and connected, and there are no cycles in the graph. Therefore, the longest path from SOURCE to any vertex u, $L[u]$, must also be finite.

Lemma 1. *For any* u, $L[u] \geq D[u]$.

Proof. We observe that every $D[u]$ is the length of some path from SOURCE to u.

Lemma 2. *If there is a path of length* $D'[u]$ *to* u, *then from some point onward in the computation* $D[u] \geq D'[u]$, *if the algorithm does not terminate.*

Proof. Proof is by induction on the number of edges on the path. Lemma 2 is trivial when the number of edges in the path is zero. Now assume Lemma 2 holds for all paths with k or few edges. Consider a path with k+1 edges from SOURCE to u in which v is the penultimate vertex and the path length to v is $D'[v] = D'[u] - w_{vu}$.

From the induction hypothesis, eventually $D[v] \geq D'[v] = D'[u] - w_{vu}$; therefore u will eventually receive the length message of $D[v] + w_{vu}$ which guarantees that $D[u] = D[v] + w_{vu} \geq D'[u]$. It follows from the algorithm that $D[u]$ can never decrease. Therefore, $D[u] \geq D'[u]$ from that point onward in the computation.

Lemma 3. *If the algorithm does not terminate, then from some point onward in the computation, all vertices in the graph eventually form a rooted directed tree where $P[u]$ is the parent of u, and SOURCE is the root.*

Proof. From Lemmas 1 and 2, if the algorithm does not terminate then eventually every vertex u will have $D[u] = L[u]$ and $P[u]$ will be the prefinal vertex on this path. Therefore, all vertices form a rooted directed tree where $P[u]$ is the parent of u, and SOURCE is the root.

Theorem 1. *The algorithm terminates.*

Proof. Assume the algorithm never terminates. Then $D[u] = L[u]$ for every vertex u from some point in computation and hence no vertex sends a length message from then on. Since all vertices form a rooted directed tree (Lemma 3), a leaf vertex w in this tree cannot be the predecessor of any vertex. Therefore, eventually Counter(w) = 0 and w will send an Ack to $P[w]$. Induct on the height of the tree to show that every vertex will eventually have Counter = 0. When Counter[SOURCE] = 0, the algorithm will then terminate.

Theorem 2. *At the termination of the algorithm, for any vertex u, $D[u] = L[u]$ and Counter[u] = 0.*

Proof. For a vertex u, we define $E[u]$ to be the number of edges on a longest path from SOURCE to u. The result follows by induction on all vertices u with $E[u] \leq k$, for $k = 0, 1, 2,...$

Appendix C

Process Description of Simplex Method

Following are skeleton process structures for the columnwise distributed algorithm of Simplex Method (see Chapter 6 for details):

```
procedure RowOperations;
begin
        loop -- each time through is one Simplex iteration
                pick a globally known column with c>0; if none exit;
                select row such that ratio (d[row]/A[row,col]) is the least positive;
                if none then select another column;
                perform row operations;
                for row := 1 to m+1 do
                        ratio := B[row,j]/B[i,j];
                        for col := 1 to n+1 do
                                if row <> i then
                                        B[row,col] := B[row,col] - ratio × B[i,col]
                                else
                                        B[row,col] := B[row,col] / B[i,col]
                                end;
                        end;
                end;
        end
end RowOperations;

process Calculator;
begin
        receive n/q columns and the d vector;
        loop -- each iteration is one round
                select k positive elements of vector c;
                send the k corresponding columns to the controller;
                receive s global columns from controller;
                if nothing received then exit;
                RowOperations; -- perform all iterations of this round
        end
end Calculator;
```

(a) Description of Calculator Process

```
process Controller;
begin
        send n/q columns and the d vector to each calculator;
        loop
                receive columns from each calculator;
                if no columns received exit;
                pick up s columns from columns received;
                send s columns to each calculator;
        end
end Controller;
```

(b) Description of Controller Process

Appendix D

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[67] to the aspect of critical path length.

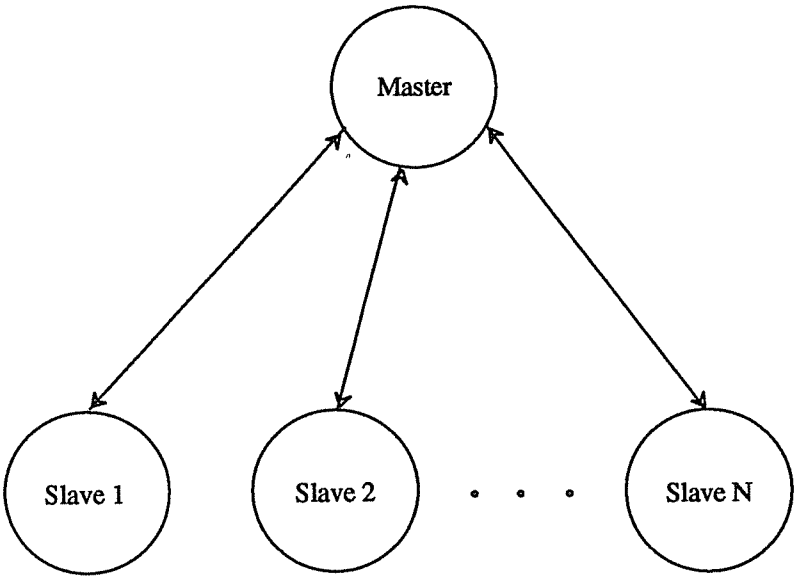


Figure D.1: Master-Slave Structure

Assume that a general master-slave structure is represented as N slave processes working synchronously under the control of a master process (see Figure D.1). 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, and communication times with slaves in the program of the Simplex method), c_m. Therefore,

$$C_m = N c_m + F_m .$$

Assume F_m is negligible compared to $N c_m$, i.e., $F_m >> N c_m$; we have:

$$C_m = N c_m .$$

The nature of the synchronization pattern in the master-slave structure determines that the execution of the master 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 $\frac{d^2(L_c(N))}{dN^2} > 0$ when $N = \sqrt{\frac{C_s}{c_m}}$, $L_c(N)$ has its minimum value at the point. Therefore, the minimum length of the critical path is:

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

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 that the length of the critical path reaches to the minimum when the entire length is evenly distributed in master and slave processes.

REFERENCES

[1] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM* **21**(7) pp. 558-565 (July 1978).

[2] U. Herzog and W. Hoffmann, "Synchronization Problems in Hierarchically Organized multiprocessor Computer Systems," *Performance of Computer Installations*, pp. 29-48 North-Holland Publishing Company, (1979).

[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] B. Liskov, "Primitives for Distributed Computing," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pp. 33-42 (December 1979).

[5] B. J. Nelson, "Remote Procedure Call," Ph. D. Thesis, Technical Report CMU-CS-81-119, Carnegie-Mellon University (1981).

[6] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS* **5**(3) pp. 381-404 (July 1983).

[7] United States Department of Defense, "Reference Manual for the Ada Programming Language," (ANSI/MIL-STD-1815A-1983) (17 Feb. 1983).

[8] R. E. Strom and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pp. 73-82 (27-29 June 1983). *ACM SIGPLAN Notices* 18:6 (June 1983)

[9] D. Gelern, "Generative Communication in Linda," *ACM TOPLAS* **7**(1) pp. 80-112 (January 1985).

[10] Michael Scott, "LYNX: A Dynamic Distributed Programming Language," Ph.D. Thesis, Computer Sciences Dept. University of Wisconsin-Madison (May 1985).

[11] G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe, "The Eden System: A Technical Review," *IEEE Trans. on Software Eng.* **SE-11**(1) pp. 43-59 (January 1985).

[12] P. Dasgupta, R. LeBlanc, and E. Spafford, "The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System," Technical Report Git-Ics-85/29, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA (1985).

[13] Mike Dean and Rick Schantz, "The Cronous Distributed Operating System," *IEEE Workshop on Design Principles for Experimental Distributed Systems*, (October 16-17, 1986).

[14] Anand Tripathi, "An Overview of the Nexus Distributed Operating System Design (Extended Abstract)," *IEEE Workshop on Design Principles for Experimental Distributed Systems*, (October 16-17, 1986).

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

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

[17] J. Wick and R. Johnsson, "An Overview of the Mesa Processor Architecture," *Proc. of the Symposium on Architectural Support for Programming Languages and Operating Systems*, (1982).

[18] R. P. Blake, "Exploring Stack Architecture," *IEEE Computer* **10**(5)(May 1977).

[19] 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).

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

[21] Gail Hamilton, "Logic Analyzer Gives Programmers Real-Time View of Software Performance," *Electronics*, pp. 117-122 (May 5, 1983).

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

[23] R. Klar, "Hardware Measurements and Their Application on Performance Evaluation in a Processor-Array," *Computing, Suppl. 3*, pp. 65-88 Springer-Verlag, (1981).

[24] Hansjorg Fromm, Uwe Hercksen, Ulrith Herzog, Karl-Heinz John, Rainer Klar, and Wolfgang Kleinoder, "Experiences with Performance Measurement and Modeling of a Processor Array," *IEEE Transactions on*

Computers **C-32**(1) pp. 15-31 (Jan. 1983).

[25] 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).

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

[27] Keith A. Lantz, Klaus D. Gradischnig, Jerome A. Feldman, and Richard F. Rashid, "Rochester's Intelligent Gateway," *IEEE Computer*, pp. 54-68 (1982).

[28] Neal Vanderlipp, John Callahan, Marc Abrams, and Ashok Agrawala, "Implementation and Measurement of a Distributed Dining Philosophers Algorithms on ZMOB," Tech. Report TR-1530, Computer Science Dept. University of Maryland (August 1985).

[29] Marc Abrams and Ashok K. Agrawala, "Performance Study of Distributed Resource Sharing Algorithms," Tech. Report TR-1521, Computer Science Dept. University of Maryland (July 1985).

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

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

[32] B.P. Miller, "DPM: A Measurement System for Distributed Programs," *IEEE Transactions on Computers*, (to appear 1987).

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

[34] Francesco Gregoretti and Zary Segall, "Programming for Observability Support in a Parallel Programming Environment," Tech. Report CMU-CS-85-176, Dept. of Computer Science, Carnegie Mellon University (November 1985).

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

(November 1985).

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

[37] Zary Segall, Ajay Singh, Richard T. Snodgrass, Anita K. Jones, and Daneil P. Siewiorek, "An Integrated Instrumentation Environment for Multiprocessors," *IEEE Transactions on Computers* C-32(1) pp. 4-14 (January 1983).

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

[39] William A. Wulf, Roy Levin, and Samuel P. Harbison, *HYDRA/C.mmp: An Experimental Computer System,* McGraw-Hill Book Company, New York (1981).

[40] Anita K. Jones, Robert J. Chansler, Jr., Ivor Durham, Peter Feiler, and Karsten Schwans, "STAROS -- A Multiprocessor Operating System for Implementing Task Forces," *Proc. of the 7th Symposium on Operating System Principles*, ACM-SIGOPS, (1979).

[41] J. K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu, "Medusa: an Experiment in Distributed Operating System Structure," *Proceedings of the 7th Annual Symposium on Operating System Principles*, (November 1979).

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

[43] Yeshayahu Artsy, Hung-Yang Chang, and Raphael Finkel, "Interprocess Communication in Charlotte," *IEEE Software* 4(1) pp. 22-28 (January 1987).

[44] Barton P. Miller and Cui-qing Yang, "IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs," To appear in Proc. of the 7th International Conference on Distributed Computing Systems, IEEE Computer Society, Berlin, FRG (September 21-25, 1987).

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

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

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

[48] David J. DeWitt, Raphael Finkel, and Marvin Solomon, "The CRYSTAL Multicomputer: Design and Implementation Experience," *IEEE Trans. on Software Engineering* **SE-13**(8) pp. 953-956 (August 1987).

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

[50] Riccardo Gusella and Stefano Zatti, "TEMPO: Time Services for the Berkeley Local Network," PROGRES Report, EECS University of California Berkeley (December 1983).

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

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

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

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

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

[56] Narsingh Deo, *Graph Theory with Applications to Engineering and Computer Science,* Prentice-Hall, Inc., Englewood Cliffs, N. J. (1974).

[57] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations," *Inf. Process. Lett* **11**(1) pp. 1-4 (August 1980).

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

[59] E.V. Denardo and B.L. Fox, "Shortest-Route Methods: 1. Reaching, Pruning, and Buckets," *Operations Research* **27**(1) pp. 161-186 (Jan.- Feb. 1979).

[60] R.B. Dial, F. Glover, D Karney, and D. Klingman, "A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees," *Networks* 9(3) pp. 215-248 (Fall 1979).

[61] U. Pape, "Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problems -- a Review," *Math. Programming* 7(2) pp. 212-222 (October 1974).

[62] D. Van Vliet, "Improved Shortest Path Algorithm for Transportation Networks," *Transportation Research* 12(1) pp. 7-20 (February 1978).

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

[64] G. B. Dantzig, *Linear Programming and Extensions,* Princeton University Press, Princeton, NJ (1963).

[65] Raphael Finkel, Bahman Barzideh, Chandreshekhar W. Bhide, Man-On Lam, Donald Nelson, Ramesh Polisetty, Sriram Rajaraman, Igor Steinberg, and G. A. Venkatesh, "Experience with Crystal, Charlotte and Lynx (Second Report)," Tech. Report #649, Computer Sciences Dept., University of Wisconsin-Madison (July 1986).

[66] Derek L. Eager, John Zahorjan, and Edward D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," Tech. Report 86-08-01, Dept. of Computer Science, University of Washington (August 1986).

[67] J. Mohan, "Performance of parallel programs: model and analyses.," CMU-CS-84-141, Ph.D. Thesis, Carnegie Mellon U. Computer Science Dept. (1984.).

[68] A. S. Tanenbaum, *Computer Networks,* Prentice-Hall (1981).

[69] James Gary, "Notes On Data Base Operating System," Computer Science Research Report, RJ2188(30001), IBM Research Laboratory, San Jose, California (February, 1978).

[70] Mark A. Linton, "Knowledge-Based Debugging (Session Summary)," *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugger*, pp. 23-24 (March, 1983).

[71] Robert L. Sedlmeyer, William B. Thompson, and Paul E. Johnson, "Knowledge-Based Fault Localization in Debugging," *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugger*, pp. 25-31 (March, 1983).

[72] David S. Snowden, "A Knowledge-Based Diagnostic System for Ada Semantic Errors (Position Statement)," *Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High Level Debugger*, (March, 1983).

[73] H. Chernoff, "The Use of Faces to Represent Points in n-Dimensional Space Graphically," Tech. Report No. 71, Dept. of Statistics, Stanford University (1971).

[74] Peter C. C. Wang, *Information Linkage Between Applied Mathematics and Industry*, Academic Press (1979).

[75] Edward R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Connecticut (1983).

[76] Philip Stein, David Coleman, and Bert Gunter, "Graphics for Display of Statistical Data," *Supplement to RCA Engineer* **30**(3)(May/June 1985).