Performance Characterization of Distributed Programs

by

Barton Paul Miller

Performance Characterization of Distributed Programs

Barton Paul Miller

Ph.D.

Computer Science Division
Department of Electrical Engineering
and Computer Sciences

Sponsors
National Science Foundation
State of California
Defense Advanced Research Projects Agency

Michael L. Powell
Committee Chairman

ABSTRACT

Writing distributed programs is difficult for at least two reasons. The first reason is that distributed computing environments present new problems caused by asynchrony, independent time bases, and communications delays. The second reason is that there is a lack of tools available to help the programmer understand the program he/she has written. The tools we use for single machine environments do not easily generalize to a distributed environment. There has been only limited success with previous systems that have tried to help the programmer in developing, debugging, and measuring distributed programs.

To better understand distributed programs we have: specified a model for distributed computations, developed a measurement methodology from this model, constructed tools to implement the measurements, and developed data analysis techniques to obtain useful results from the measurements. The most important feature of the models, methodology, and tools is consistency between the programmer's view, the computation model, the measurement methodology, and the analysis.

This consistency has resulted in several benefits. The first is a simplicity of structure throughout the measurement and analysis tools. The second benefit is the ease of obtaining useful information about a program's behavior.

The model of distributed programs defines the two basic actions of a program to be computation and communication. Our research focuses on the communications performed by a program. The measurement model is based on the monitoring of communications between the parts of a program. Given our definition of a program, monitoring communications completely encapsulates the behavior of a computation. From the measurement model, we have constructed tools to measure distributed programs for two working operating systems, UNIX and DEMOS/MP. These measurement tools provide data on the interactions between the parts of a distributed program.

We have developed a number of analysis techniques to provide information from the data collected. We can report communications statistics on message counts, queue lengths, and message waiting times. We can perform more complex analyses, such as measuring the amount of parallelism in the execution of a distributed program. The analyses also include detecting paths of causality through the parts of a distributed programs. The measurement tools and analyses can be structured so that results can be fed back into the operating system to help with scheduling decisions.

ACKNOWLEDGEMENTS

There are a number of people whose help, guidance, and support made the completion of this thesis possible.

First and foremost, I would like to thank Mike Powell. As a research advisor, he always had time for discussion and offering of constructive criticism. Mike has a seemingly limitless supply of common sense.

I am also greatly indebted to Domenico Ferrari. I learned a lot from Domenico, both technically and in how to work with people. He was a much appreciate source of ideas, criticism, and support.

I would like to thank Charles Stone for reading this thesis and for participating in my Qualifying Exam Committee.

This acknowledgement would be incomplete with a special thanks to Dave Presotto. He is a respected colleague, good friend, and fellow pilot. I can think of no one with whom I would more gladly share a research project or a cockpit. Dave shared the burden of the DEMOS/MP development and trials and tribulations of our respective researches.

I was fortunate to play composer and conductor to a superb orchestra. Stuart Sechrest and Katie Macrander deserve special thanks for their work on the UNIX implementation of the performance tools used in this thesis. Stuart implemented the kernel structures for the UNIX measurement system, and Katie the user interface and process control facilities. I would also like to thank Nick Lai for his work on the Traveling Salesman program used in some of our measurements. He was always willing to run "just one more test".

Michelle Arden suffered through proof reading this thesis, and that was greatly appreciated. I would like to thank the members of the OSMOSIS and Progres research groups for their suggestions and support, and also for their tolerance.

Finally, to my family – a place I always found support and encouragement. And to Karen – reading what I wrote, always being there, and always interested and caring.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

The Problem

1.1. Introduction

Understanding a program is a matter of having enough information. The problem of understanding a program can affect the programmer who writes the program, the user who runs the program, and the administrator of the computer system on which the program is run. Information about the program is needed during its development and during its use. Most computer systems today provide some tools to help with the understanding of programs and how they execute. The increase in the use of networks and in programs exploiting those networks has made the task of gathering information about programs more difficult. While there are many potential benefits from using distributed systems, there are also new problems created by this new environment. Models of computing and the program development tools that are provided have not kept pace with the steady increase in distributed computing and new complexities caused by distributed computing.

There are two areas that have traditionally addressed the problem of understanding program execution. These two areas reflect our view of the current state of the program. In the case where we suspect or know the program is executing incorrectly, we refer to the activity of observing the program and attempting to correct its behavior as "debugging". In the case where we are confident the program generates correct results, but we wish to better understand the method by which it generates these results, presumably to improve the method, is referred to as "performance evaluation".

Performance evaluation and debugging can usually be distinguished by the amount of participation by the programmer in the execution of the program. When a program's performance is being monitored, no change is made to its execution, except for whatever (typically small) unavoidable costs are caused by the measurement tool. Debugging is often thought of as an interactive activity. The programmer can arrange to have the program stop at various points in its execution. These stopping points are used as opportunities to examine and modify the state of the program. The programmer has control over the point at which the program continues execution – continuing either from its stopping point, or from some other point

in the program.

The separation between debugging and performance evaluation is not always clear. There are cases where a performance study discovers a previously unknown error in the program's execution. Likewise, there are cases where the act of debugging results in finding a performance problem. The main emphasis of this thesis is in the performance aspects of understanding a program's execution. Because of the overlap between these areas, we may occasionally trespass into the territory of debugging.

Performance tools exist for many types of programs and programming environments. There is a substantial increase of activity in the area of distributed or concurrent programming. Several systems have been designed to provide facilities for writing distributed programs[Almes *et al* 83], and systems have been or are being extended to include semantics to support the writing of distributed programs[Joy *et al* 83, Cheriton *et al* 79, Baskett *et al* 77]. Some of the motivations for building such systems are to increase reliability, availability, and performance. The construction of distributed systems is an increasingly common endeavor. Our goal is to make this task easier.

1.1.1. What is the Purpose of this Thesis?

Most systems being designed today support some level of concurrent programming and multiple machine environments[Almes *et al* 83, Cheriton & Zwaenepoel 83]. In spite of the increased activity in distributed systems, there is no agreed upon model of computation, nor is there a model of program performance monitoring.

When we monitor a traditional, single process program, we observe such well defined metrics as paging activity or frequency of subroutine calls. For distributed programs, there are no such standards. Applying the traditional metrics to a distributed program does not provide a complete solution. Distributed programs involve a new level of complexity. The factors that contribute to this complexity are *asynchrony*, *time*, and *delay*.

Asynchrony occurs when more than one piece of the distributed program is executing simultaneously on different machines. A program that runs on a single machine, while having the illusion of parallel execution, really executes a single instruction at a time. When a program runs in a distributed environment, we can have many instructions in the program executing at the same time. Synchronization of the different pieces of a computation becomes more important.

Time is a factor because we cannot provide a universal time base for all the machines. Time can be synchronized in a relative sense between processors[Lamport 78], but a complete ordering (full synchronization) of events is not possible. There have been algorithms designed to keep an approximation of a universal time[Gusella & Zatti 83], but even algorithms that work well cannot guarantee exactly and perpetually synchronized clocks.

The delay factor is due to the finite and non-deterministic delay for communications between machines. No action affecting a part of a computation on one machine can instantly affect other parts of the computation on different machines. This causes two problems. The first problem is that there is no way of obtaining an instantaneous picture, or *snapshot*, of the state of an entire computation. The second problem is that there is no way of causing an event to happen immediately for all parts of a computation on different machines. In addition, it not possible to arrange for an action to happen in the future at an agreed on time, since we cannot keep the clocks fully synchronized.

Traditional performance and debugging tools do not provide enough information to deal with the problems of a distributed environment. The concept of a computation consisting of numerous, concurrently executing parts and the interactions between those parts, is missing. Of course, these problems are only significant if we have a distributed computation with interactions between the various parts. The primary goal of this thesis is to provide some of the tools and methodologies needed to deal with the problems associated with developing distributed computations.

1.1.2. Organization of this Thesis

The remainder of this chapter describes the basic terms and definitions that will be used throughout the thesis. From the definitions and terms will emerge a more concrete definition of the problem we are trying to solve. We briefly describe our measurement model to provide an overview of our approach to understanding distributed programs. Chapter 2 examines previous and ongoing work in areas related to the development of distributed programs. Key ideas that have contributed to this research are identified.

The basic measurement model and methodology are presented in Chapter 3. That description includes a commentary on our measurement philosophy, and a description of the components and structure of the model. The measurement methodology has been implemented as a tool. Chapter 4 describes the two implementations of this tool, and the operating systems underlying those implementations.

Whereas Chapters 3 and 4 describe the means for acquiring information about a distributed program, Chapter 5 provides techniques for evaluating this information. In addition to the analysis techniques, several analyses are described. Chapter 6 summarizes the key ideas presented in the thesis, discussing their general applicability, and describes ideas for future work. Appendix A lists the details of various algorithms used in the data analyses.

1.2. What is a Distributed Program? And Other Definitions

We define a *distributed program* to be a collection of processes cooperating to perform some computation. The component processes are not constrained to run on the same machine. No assumptions are made about the locations of the processes. The two extremes are the case where all processes run on the same machine, and the case where each process runs on its own machine. The tools and methodologies that we are describing do not depend on how the program is distributed.

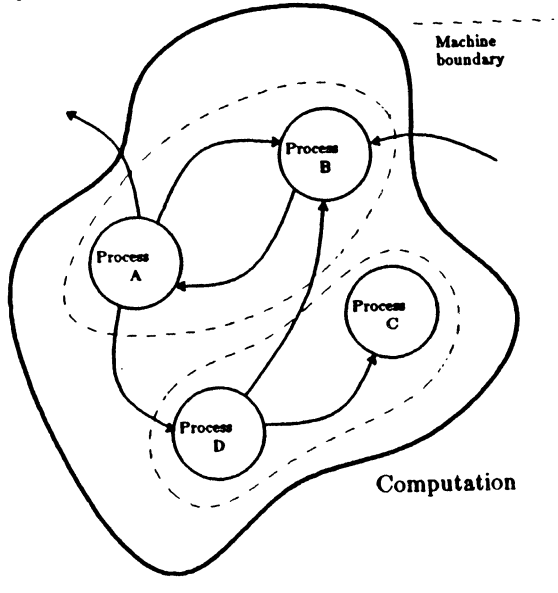

Figure 1.1: A Distributed Computation

A distributed program (more simply called a *computation*) is made up of *processes*. A process is the basic building block of a computation. It consists of an address space containing code and data, and an execution stream. Each process has access only to its own address space. Processes do two things: compute and communicate. Computing is the normal execution of instructions. Communication is the means by which a process will interact with other processes and the operating system. An *interaction* is an activity that involves more than one process, or a process and a part of the operating system. The complexities of the distributed environment become apparent when a process in a computation interacts with another part of the computation. A computation is illustrated in Figure 1.1.

Communication is based on messages. A message allows the copying of part of one process's address space into that of another process. A message is an interaction involving exactly two processes: the process originating the data (the *sender*) and the process consuming the data (the *receiver*). We make no restrictions on the structure of the message delivery. The communications path may be unidirectional or bidirectional. The message passing operations may be synchronous or asynchronous. Message delivery may or may not be guaranteed or required to preserve message order. Message paths may be dynamically or statically created and destroyed, and the processes in the computation may be dynamically created and destroyed. We make no assumptions about the network or facility underlying the communications mechanism. The model of communications is general enough to be applicable over a wide range of systems.

Our model of computation does not include systems that have processes with shared address spaces. Conceptually, a shared memory system can be modeled as a message based system (and vice versa)[Lauer & Needham 79]. In practice, the interactions in a message based system are easier to detect than in a shared memory system, and therefore easier to monitor. In Chapter 6 of this thesis, we discuss possible strategies for monitoring shared memory systems.

Processes execute on *machines*. A machine consists of a central processor (*CPU*), memory, and peripheral devices. Machines do not have direct access to each other's memories. Each machine has a portion of the operating system running on it to support process execution, communications, memory management, and device management. The communication functions supplied by the operating system provide for interprocess communications both within and between machines.

1.3. Introduction to Measurement Model and Methodology

Understanding the interactions between the processes in a computation is critical to understanding the behavior of the computation. It is in the process interactions that we see the difficulties caused by asynchrony, time, and delay. The interactions are the focus of our measurement model.

Our model of measurement follows the basic philosophy of "look, but don't touch" with respect to the program that is being studied. The goal is minimal disturbance of the execution of the program. This means that the computation being measured should not execute more slowly or achieve different results because it is being measured. The programmer of the computation should not have to insert special code into the program to produce trace data, and the program should not have to be recompiled to include special libraries to provide tracing. It should be possible to make performance measurements on any program (compatible with protection policies) without previous special work to accommodate such measurements. The goal of minimal disturbance also means that the measurement process should minimally degrade the performance of the executing computation. If the cost of measurement is high, then the act of measuring a computation could substantially change its execution behavior.

The measurement model in combination with the system that implements that model specifies a description of the events in which we are interested, a means of collecting data about these events, a selection process on that data, and methods for analyzing the data. The events that we measure should correspond to the level of interface at which the programmer (and the computation) work. If we are measuring the sending of a message, we should see it at the message send level, and not in all the details required to implement the lower level protocols. The level of abstraction presented by the measurements should correspond to that used by the author of the computation.

We define an *event* to be an action performed by a process in a computation. The data recorded about the occurrence of an event is called a *trace record* of the event (or more simply, a trace). Events are divided into two categories, those of *internal* and *external* events. An internal event is an action by a process that has no effect outside of the process. Internal events change only the state of the executing process. Internal events can be detected by traditional debugging tools and event-driven program monitors. Execution breakpoints and examining the state of a variable in process are examples of a debugger monitoring internal events. An external event is an action by a process that affects the state of another process. Therefore, external events

involve communications between processes. Our research is based on external events and the information that can derived from these events.

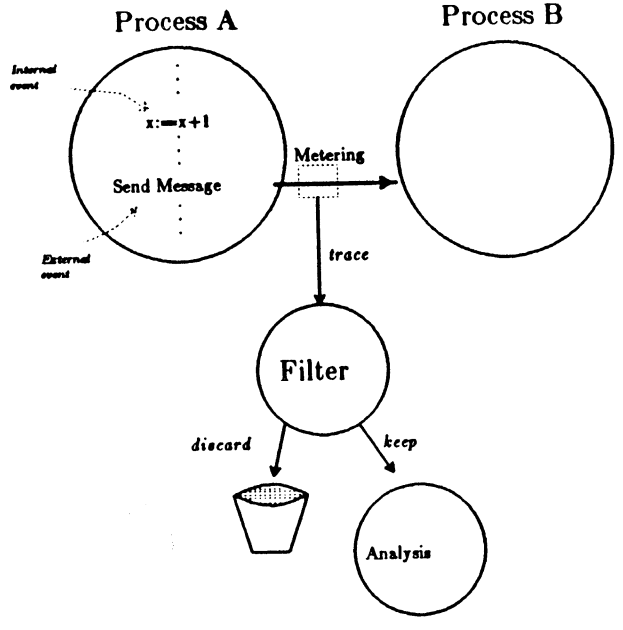

Figure 1.2: Events and the Measurement Model

Figure 1.2 gives an overview of event detection and our measurement model. Internal events are not visible from outside of a process and are therefore not detected. In our measurement model, the detection of external events is referred to as *metering*. A trace is produced for each event that is detected. After the trace is produced, a decision is made whether or not to keep the trace. The selection decision is called *filtering*. If the trace is kept, it can then be processed to provide results that may be used to understand the behavior of the process (and the overall computation). We call the processing of the traces *analysis*.

The metering stage of measurement will lie within the kernel of the operating system because of the desire not to change the program itself. The facility should be simple, so as to make the necessary modifications as simple as possible. Changes to an operating system kernel are typically much more difficult than those to parts of the system outside the kernel.

The filtering stage provides for a flexible set of rules to perform data reduction. We provide a facility that allows easy change to the selection criteria, and easy adapting to new or changed trace types.

The analysis of the data provides a summary of execution of the computation. It is at the analysis stage that useful information is provided about the computation. The goal of the measurements dictate the type of analysis being performed and the overall structure of the measurement system.

1.4. Goals of this Research

To support the goal of providing tools and methodologies for developing distributed computations, we must achieve several intermediate goals. The first goal is to develop a data measurement model that provides useful information about the execution of a computation. The measure of the usefulness of the data will be the success in providing useful analyses. The second goal is to show the feasibility of the model by designing a methodology from the model, and to construct the data collection tools from the methodology. We strengthen this demonstration by providing an implementation of the tools on different operating systems. Finally, we show the validity of the entire system by developing analysis techniques that provide useful results from the collected data. This last part is developed in two steps. The first is to develop the models and algorithms for the particular analyses of the data. The second step is to test these analyses on trial problems.

There are other indirect goals. We would like the tools and techniques to be applicable to a wide range of systems. The overall organization of the measurement system should be simple and consistent. Finally, developing distributed programs should, in some way, be easier because of the existence of the measurement tools developed here.

Chapter 2

Related Work

2.1. Introduction

This thesis addresses the problem of understanding the execution of distributed programs. Understanding program execution includes both debugging and performance evaluation, and these two areas have many concepts in common. In studying both of these areas we find aspects of both controlling a computation and monitoring a computation. We also find, in both debugging and performance evaluation, that after data about a computation is acquired (monitoring), an analysis of the data must be made. The analysis could lead to a change in the program, or a change in debugging strategy. The analysis in performance evaluation would provide a summary description of the behavior of processes. Often, the evaluator of the data is the programmer. The debugging or measurement tool collects the data and presents it to the programmer. Sometimes we find limited assistance for the analysis from the tool that is provided.

There is much overlap between debugging and performance evaluation. We divide our discussion of related research into these two categories based on the main emphasis of each particular work. We identify ideas that have contributed to this thesis and point out weaknesses in the models or designs.

There is a third area for research about understanding distributed programs. This is the area of models and simulations of distributed computations. These models and simulations address many of the same issues that are addressed in our data analyses. The body of literature on models of parallel computation is large, and we only present works that are of particular interest to measurement studies.

2.2. Performance Monitoring

When examining the body of previous works on the monitoring of distributed programs, the METRIC[McDaniel 75] system stands out as an early milestone. METRIC was built to be a performance monitor for a distributed environment that exploited the architecture of the Ethernet[Metcalfe & Boggs 76]. It is based on a model of programs communicating over a local network.

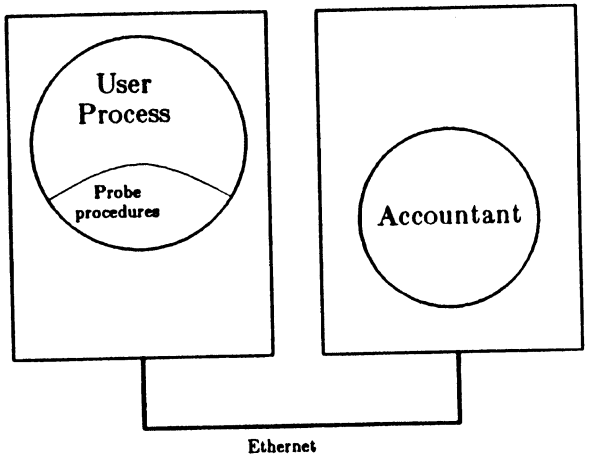

Figure 2.1: Overview of METRIC

The network provided a fast communications medium with minimal overhead. The protocols were simple, with most of the responsibility for error detection and recovery given to the user. METRIC is structured into three parts (see Figure 2.1). The first is the *probe*, which is a procedure call to a system procedure on the machine on which the monitored program is being executed. The systems running METRIC did not have a hard protection boundary between the operating system and user programs. As a result, a "procedure call" is used, rather than a system call or a trap. The programmer generates performance traces by inserting procedure calls to the probe procedure into the source program. The probe procedure takes a trace event type and trace data and sends it over the network. The second part of METRIC consists of the recipients of the trace data. These recipients are called *accountants*. There can be one or more of these, residing on different machines. The accountants record data that is sent to them for later analysis. An accountant may record all data, or selectively filter them according to policies associated with the type of data being measured. The third part of METRIC is the collection of *analysts*, processes that summarize and tabulate the trace data. These are dependent on the types of measurements being made.

The separation of event trace generation, data selection, and data analysis provides a powerful measurement structure for a distributed system. It allows minimal overhead in the operating system kernel and flexibility in selecting and studying the data. METRIC requires programmer intervention to select and insert the event probes. Changes to the measurements require recompilation of the program being monitored. To avoid recompilation, a facility for patching the object code of a program to insert new probes was proposed. For an existing program, data selection lies entirely in the accountant.

For the environment in which METRIC was developed, it was a useful tool. To be useful in other network environments, modifications would have to be made. For example, METRIC explicitly depends on a broadcast network. When a new accountant starts, it simply listens to a particular network address. There is no protocol for establishing connections between the programs being measured and the accountant(s). On a network with a more complex communication protocol structure, there would need to be an explicit connection protocol.

Several research groups have been involved in the organization of trace data after it is produced. These methods of organization involve database techniques for handling the data, or various predicate languages for describing correct or interesting input.

Snodgrass[Snodgrass 82] presents a model based on events defined around strongly typed objects, and stores the event traces in a relational database. Snodgrass's system models data collection on typed objects. This could be an operating system supporting typed objects[Wulf *et al* 81] or a programming language[Ada 83,Morris 80] that supports typed objects and type abstractions that define the operations over these types. The type abstraction is responsible for generating trace events for significant activities that occur while performing operations on objects of the corresponding abstract type. The model of computation is based on operations on objects. For each type of object in the system, the designer of the abstraction must decide what events are interesting enough that traces should be generated. The designer explicitly makes internal events visible outside of the program.

Snodgrass uses a relational database to store the event traces. To access these traces, a database query language that includes the notion of time is used. This query language can be used to retrieve information about the execution of the program. It is somewhat disappointing that Snodgrass does not present guidelines for the use of this database and query language, other than to assert that this is a powerful mechanism to obtain information about a

program.

Bates and Wileden[Bates & Wileden 83] present a methodology for evaluating event traces from a distributed system. The methodology is called *behavior abstraction* (BA). The basic idea of BA is to be able to identify higher level abstraction from the basic trace events. The *event description language* (EDL)[Bates & Wileden 82] allows the specification of which basic events form higher-level events. Event traces can also be prevented by filtering. The eventual plan for BA is to allow fully hierarchical specification of abstractions, meaning that a trace group can itself be listed as a component of another group. The current implementation allows only a single level of grouping. For each group, the entire body of traces is examined. Even though a trace is included in a group, it is not eliminated from being included in other groups. Not eliminating a trace inclusion in other groups was a simplification their analysis programs made for early implementations of EDL. Trace grouping is illustrated in Figure 2.2.

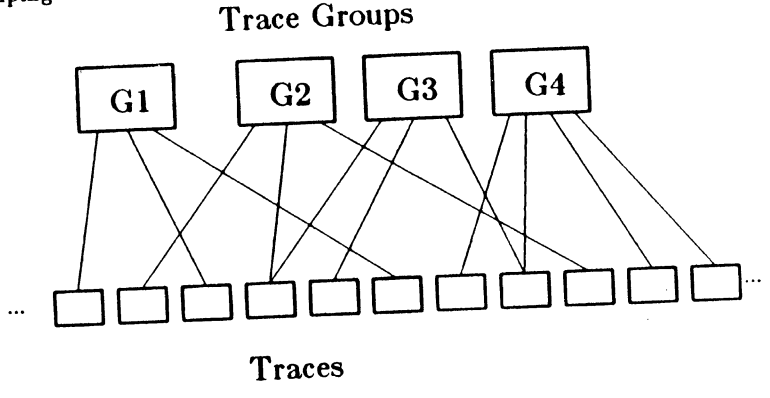

Figure 2.2: Trace Grouping in BA

The motivation for BA is to specify a high level description of a program's execution, and then compare it to the high level trace events to determine whether the execution was correct. For complex programs it may be difficult to specify execution descriptions to detect and extract higher level events. As with Snodgrass's system, thus far, there are no complete examples of how the hierarchical grouping of traces might be used.

2.3. Debugging

The development of distributed debugging has been a progression from traditional single process debugging tools to those that deal with more complex environments. The added complexity comes from two factors. The first factor is the existence of multiple machine environments. Multiple machine debuggers allow the programmer to run a collection of programs on different machines and to debug these simultaneously. The second factor is having interactions between processes. The earlier works in this area considered the problem of multiple processes cooperating in a computation, but residing on a single processor. The most recent work in distributed debugging combines the two factors. Debugging takes place over many machines for a collection of cooperating processes.

Two interesting examples of multiple machine debuggers are the COPILOT system[Swinehart 74] and a system developed at TymShare Corporation[Vickers 76].

COPILOT is based on a model of interaction between the programmer and the machine. The basic idea is that a user's behavior and activities while developing a program should be matched by the system that is being used. The result of this view is a system that provides a high level language interface, an interactive environment, and the ability to handle multiple activities. It is the last item in which we are interested.

The COPILOT view of multiple activities is that of multiple programs (processes) that are being developed simultaneously by a single programmer. The programmer needs the ability to switch his/her attention from one program to another, and be provided with enough information to switch back again. COPILOT addresses the issue of the programmer-program interface, and not that of the interfaces (if any) between the programs being debugged. COPILOT allows the programmer to simultaneously debug multiple programs residing on a machine, but does not consider, and therefore, does not help with, the problem of interacting processes. In terms of our measurement model, COPILOT provides access to the internal events of many processes, no external events are visible.

The TymShare system was designed to allow debugging on multiple machines and to allow debugging of programs written in many different languages. To allow debugging on many machines, the part of the debugger which is on the machine where the program is working communicates with modules of the debugger residing on the remote machines. The programmer can start, stop, or insert break points in the programs being debugged, and examine or modify the memory of these programs. There is no explicit

consideration of interactions between the programs being debugged, so the issues of asynchrony, delay, and time are not addressed.

At the University of Rochester[Smith 81], the interactions between processes cooperating on a computation have been studied and debugging techniques for multiple process computations have been developed. This research addressed the problem of debugging a multiple process computation executing on a single machine. Event traces are generated for system calls, message transmissions, and debugging commands.

The system call events allow tracing of the operation of sending or trying to receive a message. Message events trace the occurrence of *border crossings*. A border crossing occurs when a message enters or leaves a communications object (called a *port*) or a process. Message events coincide with the external events of our measurement model. Debugging command events are caused by actions performed by the debugger. Events are reported to the debugger, and the programmer can specify how events are to be handled. Events caused by the execution of a program can be changed, deleted, or new ones inserted. The programmer can be required to examine each event before it takes place (e.g., before a message is delivered), or the events can be handed to debugging *demons*. Demons are activated by the debugger when a predicate based on a sequence of event occurrences associated with the demon is satisfied. When the demons execute, they can do the same control operations on events that can be done by the programmer.

The Rochester system provides the programmer with a substantial amount of information. The demons are provided as a method of coping with the large volume of detailed trace data. It is possible to construct demons to deal with some of this detail, and it might be possible to construct demons to do some commonly used functions. As with the BA system, it is not obvious how to construct demons that will significantly help with the debugging of a complex program.

The Rochester proposal was designed for a single machine environment. The ability to watch events at such a low level, delay or modify message operations, and start and stop processes is much more difficult to provide in a distributed environment. There is no simple generalization of this debugging system to multiple machines.

Garcia-Molina, Germano, and Kohler[Garcia-Molina *et al* 81] proposed a system for distributed debugging based on process interactions. Their system incorporates the features of traditional single process debugging practices on the individual processes, within a debugger of higher-level functions that involve the interactions between processes. Debugging is to proceed in two

steps in a bottom-up fashion. First, each individual process is debugged using a traditional program debugger. Interactions with other processes would be simulated by auxiliary test processes. After each individual process has been tested, the computation is tested as a whole.

The concept of testing each individual process is appealing. We would like to have confidence in the components in a computation before we test the interactions. The problem lies in the design and implementation of the auxiliary test processes. The construction and testing of these could be as difficult as constructing the original computation.

The debugging of interactions is based on traces. Traces are generated on the occurrence of interesting events in the life of the computation. There are two forms of events. The first are system generated, and reflect what Garcia-Molina, *et al*, believe to be common operations. These include such things as message send and receive, transaction begin, end, commit, and abort, deadlock detect, etc. Traces could also be generated by having the programmer insert explicit traces. There is no explicit separation of internal and external events. Traces are recorded in files. They propose viewing the traces as relations in a relational database and to use a query language to extract information about the traces.

The last part of their proposal describes controlling the execution of the computation in much the same way as we would control a single process. Features such as starting, stopping, and break points are to be supported.

A great weakness with this proposal is the lack of a coherent model of computation. Lacking that model, it is not clear what type of events the authors wish to measure. The list of desired events is long and reflects a grab bag approach to the problem. The data analysis has a similar problem. Putting the traces in a relational database allows for taking several different views of the data. The problem lies in that there may be, for performance reasons, structures for which this is inappropriate. Building arbitrary graphs, and rapid traversal of these structures may be difficult in a relational database.

The description of controlled execution ignores the problems of time and delay. Without solving these problems, it is unclear whether a computation can be controlled with enough precision to get reasonable results. The changes caused by process control could so distort execution of the computation that the results would be significantly changed.

The MuTEAM project[Baiardi *et al* 83] is another example of using a debugger to match an anticipated view of a program's execution with its

actual execution. The MuTEAM language is based on Hoare's CSP[Hoare 78]. The CSP definition has been extended to include asynchronous (buffered) communications, and the specification of multiple senders for receive (read) commands is allowed. The debugger is given a formal specification of the execution of a program. This is given as a collection of state definitions and events (communication functions) between processes that cause a new state to be entered. This information describes all possible transitions. When a program performs an interprocess communication function, the run time library communicates this fact to the debugger, which then compares the event to the formal description. If the event fits the description, the execution of the program is not interrupted. If the recorded events do not match the program description, then the computation is stopped by the debugger and the programmer is notified.

This system is similar to the BA system in that there is no indication that it will work for more than simple examples, and no indication of more complex uses is given. A significant problem with the formal specification is answering the question about how one creates such a specification, and once created, how to verify that it is correct.

Two papers deal with the issue of manipulation of the interactions between processes, much as we would manipulate execution through a single process using traditional debugging tools. Both deal with the concepts of watching the message streams between processes, controlling these streams, changing them, and controlling the execution of the component processes.

The first system is a proposal called Black Flag[Philips 82]. Black Flag is a message debugger for use with the Accent[Rashid & Robertson 81] operating system kernel. The main idea in Black Flag is to interpose the debugger in the communications path between the sender and receiver. All messages go through the debugger, and the processes involved in the communication see no difference in the message passing operations. The debugger can display messages, halt message delivery, and (in future implementations) allow editing of the message communications by modifying, adding, or deleting messages. A facility is also proposed to provide *breakpoints* that allow the user to specify some predicate (presumably based on the intercepted messages) that, when satisfied, causes Black Flag to perform one of the above commands.

The Black Flag design is extremely simple, and therefore allows for easy implementation. The design makes no attempt to deal with the problem of the modified message delivery times. Messages can be arbitrarily delayed at the debugger causing significant performance changes. Even in the best case

in which messages are not held by the debugger, but are immediately forwarded to the original recipient, the resulting time to deliver the message could be significantly different from the delivery time without the debugger.

The second system was done at MIT[Schiffenbauer 81]. It was implemented on a collection of Xerox Alto personal computers connected with an Ethernet. Schiffenbauer takes a more sophisticated view towards message debugging, and considers the problem of the affect of the debugger on message order. All messages are intercepted at the machine on which the message recipient resides and then forwarded to the debugger. The delay time for the message is calculated, and is used when the message is delivered. To handle the problem of the debugger causing delays by holding the messages, the clock on a machine that has a process waiting for a message is stopped until a message is ready for delivery. This is done by having a high priority process take over the machine until the debugger forwards a message destined for the waiting process. When the message is finally delivered, the machine updates its clock by the amount of time previously calculated from the message delivery delay.

There are several problems with the MIT system. The first is the possibility for deadlock. Since an entire node is suspended until a receive request for a single process is made, the message that would satisfy the receive may never be sent. The same is possible for a collection of processes communicating in a circular chain. A second problem is that freezing execution of an entire machine may not always be practical (or even possible). On a shared computer system some other mechanism would have to be developed to manipulate a process's view of time, without affecting the other users of the machine.

2.4. Models and Simulation Studies

Models and simulations offer an opportunity to understand distributed programs before building a complete program. Models have been based on various constructions such as finite state machines, program graphs, and probability distributions. We can divide the models of distributed programs into two categories. This first category contains the *static* models. These models describe the structure of the computation. A static model must be studied analytically (as opposed to using simulation). The second category contains the *dynamic* (or execution) models. The dynamic models describe the execution of the computation. A dynamic model for a distributed program requires a definition of the initial state of the program, and how the program proceeds through subsequent states. Dynamic models often lend themselves to

simulation. Simulations typically provide a simpler, better controlled environment to run a program. Simplicity is required to make the simulation run in acceptable time. The simpler environment can also be used to eliminate complexities that detract from the understanding of the program being developed.

The static models of distributed programs share the common characteristic that they are not easily able to handle complexity. There are two common restrictions. If a model contains enough information to accurately reflect real computations, the complexity of analyzing the model may be prohibitive for anything except trivial computations. If the model is tractable for complex computations, it often does not contain enough detail to be useful.

Augmented program flow graphs[Taylor & Osterweil 79] are described as a method of representing a distributed or parallel computation. These graphs are built from the static information available at the source code level of the program. These graphs can be used to obtain correctness information about such phenomena as non-initialization, unreachable code, or deadlocks. The authors' analysis assumes that communications connections between parts of the computation are statically determined. They showed that such computations for dynamic connectivity are intractable (NP-complete).

A similar graph representation of programs is described in [Reif 78]. Reif developed algorithms for analyzing data flow and reachability in communicating processes. Given a program flow graph, he is able to compute spanning graphs of control and determine whether portions of the computation will ever be executed (are reachable). The algorithm is extended to consider the case where the connection specifications are dynamic (i.e., the specification of communications channels are functions changing over time, rather than constants). Information on the existence of paths is available (e.g., does A ever talk to B?), but not on the nature of the paths (e.g., bandwidth or order of data over a path).

A model of the execution of the processes in a distributed computation based on finite state machines (FSM) was developed by [Gertner 79]. This model was used to collect data from the RIG network[Ball *et al* 76]. Each process is modeled as a FSM in order to help reduce the large quantity of data collected. As data is received, the processes change state to tabulate results, and to indicate the ongoing activities. Two major problems are associated with this approach. The first problem is that as details of the execution are added to FSM model, the cost of evaluating the model increases rapidly. The second problem is that the FSM descriptions of the processes proved difficult

to program.

Downey[Downey 82] presents a technique for evaluating concurrent execution of programs, based on a stochastic model of execution. The basic assumption is that message delivery time and the process execution times are random variables. Because these are random variables and we can measure the distribution of the variables, we can obtain insight into the execution of the program. This analysis is possible for the cases where we understand the interactions of the processes (i.e., where we can derive joint distributions from the distributions for each individual process). Unfortunately, for anything but the most trivial cases, the mathematical statement of the joint densities becomes too complex to be of any use.

Simulation of an execution environment is an approach that can provide useful information in the development of distributed programs. The SIMON simulator[Fujimoto 83] provides a simulated environment for distributed programs. SIMON includes basic interprocess communication functions and assignment of processes to multiple machines. SIMON also includes a facility for evaluating various configurations of machine interconnection topologies and structures. Programs are written using the SIMON primitives, and execute completely within the SIMON environment. The simulator can be used to analyze how a distributed algorithm executes in various configurations. Since the execution environment is a simulator, a wide range of measurement data can be obtained.

The major restrictions in SIMON are that 1) communications paths and processes are statically created, and cannot change for the life of the computation and 2) only one process can be assigned to a given machine. This restricts the class of computations that can be measured. Programs developed under SIMON use the SIMON interprocess communication, and therefore must be modified to use the actual system's IPC mechanism when run on a real system.

Simulators such as SIMON find their greatest usefulness in the early stages of developing distributed programs. Various algorithms can be more easily tested and understood in a simulator environment. Once the basic algorithms are understood, the complete program is developed in the actual execution environment.

Chandy and Misra[Chandy & Misra 78] examine the implementation of queuing simulations by a collection of cooperating processes. Each process in the simulator represents one node in the queuing model. Messages between processes represent customers in the simulation. The collection of cooperating processes is presented as an example of a distributed computation that can be

studied. They show that their distributed simulation will never deadlock, and that the simulation correctly models the behavior of the program being simulated.

2.5. Summary

The behavior of distributed computations has been studied from many perspectives. The analytical models and simulations provide a valuable starting point to help in constructing computations, but the restrictions in the level of detail in these models prevent us from obtaining a more accurate view of actual computation. There are still surprises to be found when the modeled or simulated computation is really built and executed.

The debugging approach to distributed programs seems to be the most intuitively attractive. As programmers, we are familiar with the process of interactively understanding what a program is doing. Unfortunately, a distributed environment presents a number of obstacles to using the debugging paradigm. The problems of time, delay, and asynchrony make it difficult not to interfere with the pattern of execution of the program that is being debugged. Schiffenbauer attempted to solve many of these problems. The results of his solution were a great degree of detail, a substantial slowing in the execution of the programs, and only partial success in not disturbing the pattern of execution of the program. Monitoring a computation without attempting to control its execution offers the best opportunity to understand its behavior.

In this chapter we have described proposals for various strategies for monitoring the behavior of a computation. A major problem with those proposals is the lack of a reasonable model of distributed computation. Often, there is no model at all. For example, the work by Garcia-Molina, Germano, and Kohler[Garcia-Molina *et al* 81] describes many interesting ideas about measurement, but these ideas are unguided without a supporting model, while the model in [Smith 81] is too detailed and system specific to be generally useful in the understanding of distributed programs.

In the area of monitoring, the METRIC system[McDaniel 75] is a landmark work. METRIC used a consistent view of the world. The view is that of programs running on personal computers, interacting by means of a simple, fast, non-guaranteed message delivery system. This consistency allowed METRIC to be a simple yet powerful tool. The structure of METRIC (that of probe, accountant, analyst) provided a good division of function for a distributed measurement tool.

Some work has been done in the area of processing the information obtained from monitoring distributed programs. We refer in particular to: [Snodgrass 82] and, to a less extent [Garcia-Molina *et al* 81]. The flexibility of using a relational database to store and retrieve information has many advantages. There are two problems with the current proposals. The first is that, even though they specify how to store and access information in the databases, there is very little direction and experience on how to apply this to real measurement problems. The second problem is that, currently, relational databases are too slow. Many of the analysis techniques that would be used on the data require random access, and disk access time causes excessive delays.

Chapter 3

The Measurement Model and Methodology

3.1. Measurement Model

The model of distributed programs presented in Chapter 1 provides a simple and uniform way to view computations. The components of a computation (processes) calculate and communicate. If processes do not communicate, our computation can be considered a collection of traditional programs to which the traditional debugging and performance monitoring tools apply. When processes communicate, understanding the behavior of the computation becomes more difficult. When this communication takes place between machines, understanding the behavior of the computation becomes again more difficult. The key idea is that the difficulties in understanding stem from the fact that processes interact.

The model of processes cooperating in a computation via messages provides a uniform structuring of a computation. In this model, message communications are the only mechanism available to a process to communicate with other processes and the operating system. Processes are then fully encapsulated by their communications. If we understand the communications in a computation, we will understand a large part of the behavior of the computation.

The communications in a computation consist of the processes performing the communication functions, and the objects being communicated. This dictates the type of events we will need to monitor to understand the behavior of a computation. First, we must monitor the process level activities. Events such as processes being created or destroyed, or starting or stopping execution are of interest. Second, communications activities must be monitored. These events would be those associated with sending, delivering, and receiving a message. Events describing the creation or destruction of communications paths are also in this category.

There are two implicit assumptions in the above description of the measurement model. These assumptions are *transparency* and *consistency*. Transparency means that, when we talk of measuring events associated with

the execution of a computation, we are making the assumption that when we measure these events we will do nothing (or at least little) to change how the events occur. This is in contrast to the distributed, message debugger approach, as in [Schiffenbauer 81, Philips 82].

Consistency is providing a uniform view of the computation being measured. The programmer uses a certain set of primitive functions to build a distributed computation. The view of computation and communication as seen by the programmer is based on these primitives. The events defined in our measurement model should be consistent with the view as seen by the programmer. This is shown in the example of monitoring a message sent from one process to another. The programmer would see send and receive operations. The actual communications may be implemented using a protocol that involves acknowledgements, division of the message into fixed packets, routing decisions, etc. Viewing the communications at this level of more detailed semantics would obscure message delivery in unnecessary detail. Likewise, adding another level of semantics onto the measurement events can also be less useful to the programmer. For example, if we used a formal description language that described the correct sequence of communications, the problem would become determining if the description is correct, before we can determine if the program is correct.

3.2. Measurement Methodology

The measurement methodology is a description of how we can build program measurement tools for programs based on the model of computation presented in Chapter 1. The methodology must embody the characteristics of the measurement model described in the previous section. In particular, our measurement methodology must adhere to the principles of transparency and consistency.

The measurements must be done in a transparent manner. This means that, to measure a program, we require no *a priori* knowledge of the computation. No special action by the programmer of the computation must be necessary, and we must be able to measure any program. The program must be unaware that it is being monitored, and none of the interfaces to the operating system need to be changed. The measurements should disturb the actual computation as little as possible. They will cause some degradation of the computation's performance, but this degradation is to be kept as small as possible.

Transparency also dictates that the measurements be done passively. Except for the degraded performance, no changes must be made to the events

being monitored. This is explicitly not interactive debugging. By this, we mean that actions such as redirection of messages, breakpoints, and modifications of the message streams are not allowed. A measurement facility must be an observer of the computation, and not in any way a participant.

We require the measurements to be consistent with the programmer's view. The measurements are based on the recording of interesting actions occurring during the life of processes within a computation. These actions, called *meter events*, consist primarily of activities that reflect interactions between processes (such as a message being sent and received). Other events related to communications are also recorded. This group of events consists of actions that effect the creation, modification, and destruction of communications paths. The last group of events that are recorded pertain to the state of the processes in the computation. The basic events are the creation of a process, the starting and stopping of its execution, and the destruction (termination) of the process.

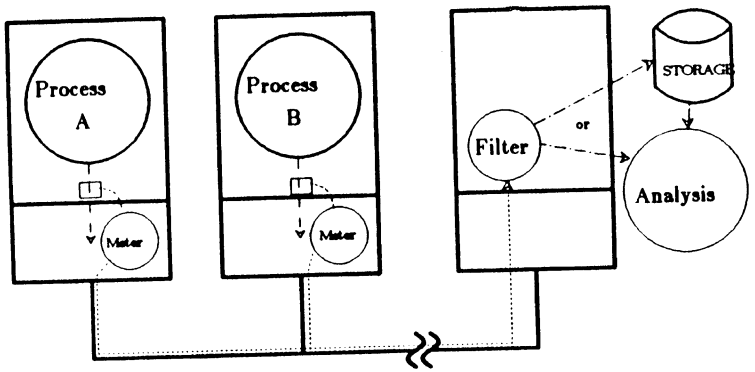

Figure 3.1: Overview of Measurement Facility

The overall structure of the measurement facility is shown in Figure 3.1. The three stages of measurement are *metering*, *filtering*, and *analysis*. The metering is the extraction of the data from the operating system for processing outside the operating system. We refer to the operating system *kernel* as that part of the system that implements the basic objects and communications

primitives of the systems. In some systems (e.g., DEMOS/MP), this includes only a small part of the total operating system, while in other systems (e.g., UNIX), most of the operating system is included in the kernel. The *host kernel* is the kernel for the machine on which a particular process currently resides. Note that this implies that each machine has a kernel or portion of a kernel resident. The *host operating system* is the operating system of which the host kernel forms a part.

Filtering is a selection and reduction process on the data. This stage records or passes the data on for analysis. The analysis is the extraction of information of interest from the collected data.

3.2.1. The Meter Events

There is a set of meter events that reflect the basic operations as seen by the programmer of a distributed computation. The structure of the metering stage is very simple due to the small set of meter events (currently less than 10). These event types are, for the most part, the same across the different operating systems supporting the measurement facility. Depending on the system from which the measurements are being extracted, there may be slight variations in the details of the data collected with each event type. A complete discussion of the system specific variations is found in Chapter 4.

Included with each event trace is a standard header describing the trace. The header of an event trace contains the following fields:

MACHINEID — The machine from which the trace came.

PROCTIME — The amount of CPU time used by this process up to the time this trace was generated. PROCTIME is independent of the load on the host system.

TRACETYPE — The type of event described by this trace.

The basic trace events are:

SendMessage — The action of a process sending a message over a communications path. Enough information to be able to determine the sender and receiver is included with this trace.

ReceiveCall	The action of a process *requesting* the operating system to receive a message.
ReceiveMessage	The actual receipt of the data. This records the action of the message being passed to the process so that it can continue execution. Enough information to be able to determine the sender and receiver is included with this trace.
MessageQueued	The placing of a message in a queue for the receiving process.
CreatePath	The creation of a communications path. For datagram type communications, a CreatePath event will not necessarily occur.
DestroyPath	The destruction of a communications path.
CreateProcess	The creation of a new process as part of the computation being measured.
StartProcess	The change in the state of a process from not executing to executing. This event occurs when the process's state is explicitly changed; not as a result of, for example, becoming unblocked after a message receipt.
StopProcess	The change in state of a process from executing to not executing. This event occurs when the process's state is explicitly changed; similar to StartProcess.
DestroyProcess	The termination of a process.

3.2.2. Metering

Metering is the activity that takes place within the operating system kernel to extract the events of interest. This portion of the measurement facility is the only change required to the supporting operating system. Figure 3.2 shows the structure of the metering of an external event. There are two basic parts. The first part is the collection of the information that is needed to form a trace. The second part is a communications path over which to pass the trace information.

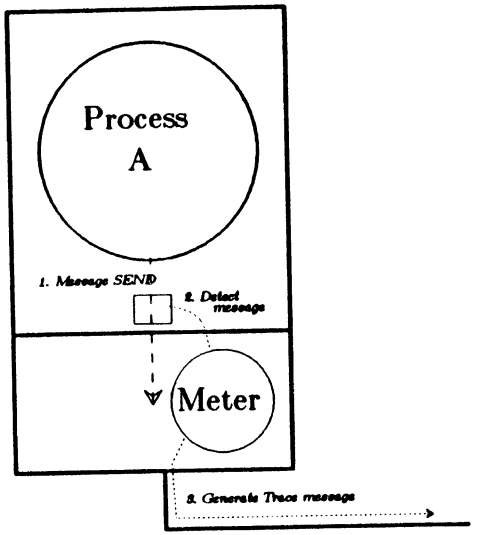

Figure 3.2: Metering a Process

The point in the operating system kernel where the necessary information is available must be located to meter the specified events. At these points, we insert *meter probes* in the code of the kernel. These probes are procedure calls to a software module (*meter module*) that is responsible for passing the traces to the filter. The parameters particular to the specific trace being generated are passed to the meter module with the procedure call. These values, along with the standard format header, are passed to the filter.

A communications path, called the *meter path*, must be available to the metering routines for sending traces to the filter stage. This meter path should be reliable in the sense that messages sent are eventually delivered. Messages should not be lost or duplicated. It is possible that later stages in the measurement system (the analysis stage) might be able to detect such anomalies in the data, but it is a good policy to try to reduce these problems as much as possible. There is no constraint on message ordering. Each trace includes the local machine time; hence the analysis stage is able to restore (partial) message order.

Metering, to the extent possible, should not increase the complexity of the host kernel. In fact, a few procedure calls are added to the existing host kernel code. The number of these calls inserted in the kernel is about the same as the number of different types of traces. In the same spirit of minimizing complexity, the metering stage uses the same message facility as already exists in the host operating system for its communication channel to the filter stage.

There must be a method for allowing the programmer to specify the message path to the filter stage for processes being metered. This requires the addition of a new system function (system call) allowing a message channel to be specified.

3.2.3. Filtering

Filtering is the data selection and reduction stage in the measurement system. It reduces both the size and number of traces as they are produced. Data is received from the metering stage, filtered, and then passed on to the analysis stage or stored for later analysis. There are many possible schemes for data selection[Bates & Wileden 83]. The scheme currently used in the measurement system is based on a general, one level, pattern matching algorithm. This filter also allows for the specification of trace record formats, so that the filter can produce standard format traces coming from different systems.

The location of the metered processes and the location and number of filter processes are a source of flexibility in the measurement system. The choice of how to attach processes to filters provides the ability to do many types of analyses.

3.2.3.1. Trace Description and Selection

The filter receives three types of input. These are the *descriptions* of the trace record formats, the *selection rules* for filtering, and the trace records themselves. These are similar to the facilities provided in statistical packages such as SAS[SAS 82]. The descriptions and the selection rules are processed at the time the filter begins its execution. These stay in effect while the trace records are processed.


```
header 0,
            machine,0,4,16
            traceType,4,4,10
            time,8,4,10
            procTime,12,4,10;
SENDMESSAGE 1,
            fromTask,0,4,16
            toTask,4,4,16
            channel,8,4,10
            code,12,4,10
            sendLink,16,4,10
            passLink,20,4,10;
```

Figure 3.3: Trace Description

A trace record consists of a header, standard for all types of traces, and an operation dependent portion. The trace descriptions consist of a description of the header format, and a description of the operation dependent part of each trace. Figure 3.3 gives a sample of a description for the header and a SENDMESSAGE trace (for the DEMOS/MP version). The first part of the trace description gives the name (in this example "header" or "SENDMESSAGE"), and the value identifying that trace type. Following this is an entry for each field in the trace. Each entry contains the name of the field (to correspond with those that will be specified in the selection rules), its offset into the message, the length of the field, and a default number base for displaying the field.

Using the descriptions, it is possible to take traces from different systems, in different formats, and process them in a uniform manner. As the filter receives each trace, it compares the trace against a set of selection rules. Each rule is a pattern that, if matched, specifies that the current trace is to be accepted. A rule is a list of selection fields which specify the conditions for acceptance of each field. A selection field is a field name, a selection condition (comparison operator), and a value. A selection field is satisfied if the evaluated condition is true. If all selection fields in a selection rule are true, the trace is accepted. The possible conditions are $>$, $<$, $=$, $\neq$, $\geq$, and $\leq$. Figure 3.4 shows a simple set of selection rules.

```
machine=3, traceType=1, time>10000;
machine=1, type=1, fromTask=20003, toTask=30003 channel=0;
```

Figure 3.4: Simple Selection Rules

The value specified with each selection field has several options. The value may be either a simple value, a *wildcard* "*" (a value that matches any value), or a field name. In the case where the value is a field name, the value of the field specified for the selection field is compared to the named field (also contained within this record). Any value may be prefixed with the the *discard* indicator "#", so that if the trace is accepted, this field is eliminated from the trace. This allows the size of the traces to be reduced. Figure 3.5 shows a more interesting set of selection rules.

```
machine=#*, traceType=1, time=#*, procTime=#*, code=#*;
type=8, readTask=writeTask, size≥1024;
```

Figure 3.5: Selection Rules

3.2.3.2. Early Filtering

A crude level of filtering is done before the trace messages leave the kernel (metering stage). The motivation for this is to reduce the volume of trace messages if the tracing paradigm permits. The metering should not cause a significant performance decrease for the process being metered or for the host kernel. The early filtering is a simple selection.

For each operation that can be traced (e.g., SENDMESSAGE), there is a flag which indicates whether that operation is to be metered. If this flag is not set, no traces of this type are generated for the filter. If an analysis required only the events of a message being sent and received, then all other trace types could be ignored, reducing the effects of the measurements on system performance.

The selection flags are combined into a bit mask and stored with the description of the meter path. There must be a system function (system call) to allow the selection flags to be specified.

3.2.3.3. Configurations

Figure 3.6 shows a sample configuration for metering user processes. Several processes forming a computation are sending traces to a single filter, which is selecting and storing the data for later analysis. We would expect this structure to be useful to a programmer evaluating a new program.

Different configurations provide for the ability to apply the measurement system to different problems. For example, the configuration in Figure 3.6 could be extended to have the filter collect data on all communications activities within a single machine. This type of configuration could allow measurement of message quantity and frequency, queue lengths, and process

scheduling. Our measurement system can gather these different types of information which traditionally required specialized tools to be built. It takes no extra work to extend this type of measurement to a collection of machines, or to the entire system. Only the processes in the computation(s) that is being measured will generate traces.

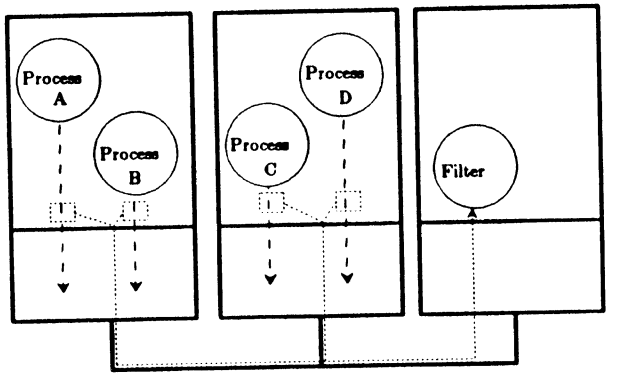

Figure 3.6: Computation with Single Filter

An alternative to the post mortem processing of the traces is to process them as they are produced and selected by the filter (in *real time*). This allows the data produced by the measurement system to be used as feedback information to the host operating system. This is discussed further in Section 5.2.6.

3.2.4. Analysis

The variety of analyses that can be performed on the collected data, and the extent of the information that can be extracted is the measure of the worth of this measurement system. A complete discussion of the data analyses is in Chapter 5. There are two major directions for the analyses. The first direction deals with the structure of the measurement system. The different measurement configurations allow a wide variety in the character and application of the analyses. Applications from user program debugging to system performance evaluation are possible.

The type of trace data collected allows for a wide variety of analysis techniques. The first level of this is simple statistical data. This takes the form of event counting and time/frequency data. More advanced analysis can be done by treating the data as a sequence of events whose order provides interesting information. Two analyses that fall into this class are the study of parallelism in the execution of a distributed program, and the study of tracking paths of causality in the program's execution.

3.3. Summary

The measurement model presented in this chapter provides a guide to the events that are important to the understanding of the behavior of a computation. The feature that distinguishes distributed computations is the interaction between the asynchronous and possibly remote components of the computation. The activities that are to be measured are defined to be consistent with the programmer's view of the activities of the computation. This provides information that is directly useful for understanding the execution of the computation. To develop a model that would allow measurement of a computation without substantially affecting its execution, the collection of measurement data was required to be transparent. Transparency also has the benefit that advanced knowledge of the structure of the computation is not required.

Starting from our measurement model, we have developed a methodology for collecting the data necessary for event traces. The measurement model dictates what must be measured, and the methodology described how the measurement will be carried out. The measurement methodology is flexible enough that it has a wide variety of applications in addition to simple measurements. It is also flexible enough that it should be able to be implemented on systems that support the model of computation presented in Chapter 1.

The next chapter describes two sample implementations of the measurement methodology, and how they relate to the underlying operating systems.

Chapter 4

The Implementations and Underlying Systems

4.1. Requisites for a Metering Implementation

In Chapter 1 we presented a simple model of distributed computation. Based on that model, in Chapter 3 we presented a model and a method for measuring these computation. To verify the validity of these concepts it is important to test them in a running operating system. By implementing them, we translate the concepts into tools that programmers and systems managers can use to understand programs. The models and methods can be evaluated by evaluating the usefulness of the tools that incorporate them.

The first step in verifying our model and methods is to construct these tools in some operating system. If this is done successfully and the tools produce useful results, then this is a strong argument for the validity of our approach to understanding distributed computations. But constructing the measurement tools on a single operating system is only a partial demonstration of the validity of the measurement model. The model is based on a general definition of distributed computation. To verify that our models are generally applicable, we need to implement our tools on more than one operating system. Each system on which we can implement the measurement tools (and obtain useful results) provides added support for our approach to understanding distributed programs.

The measurement tools were first implemented on the DEMOS/MP operating system[Baskett *et al* 77, Powell 77, Powell & Miller 83]. DEMOS/MP was a good choice for the first implementation for several reasons. The first is that its model of computation closely follows our model of distributed computation presented in Chapter 1. DEMOS/MP provides processes using a simple interface for communications. There are no parts of DEMOS/MP that are incompatable with our computation and measurement model. The second reason for choosing DEMOS/MP is that its current implementation is a research system. It does not support a computing community which depends on it for their day to day computing needs. This has several advantages. Changes can happen more frequently and therefore new ideas can be tested

more quickly. Interfaces to critical system components can change without concern for the effects on a large population. An important advantage of a research system is the ease in testing extreme cases, such as lightly loaded or heavily loaded systems.

The second implementation of the measurement tools was on the 4.2BSD version of the UNIX operating system[Ritchie & Thompson 78, Joy *et al* 83]. There were two main reasons motivating the second implementation. The first reason was as stated above – to verify the applicability of the models and methods by another implementation of the measurement tools. The second reason was to implement the tools on a system with a regular user community. This would allow the measurement methods to be tested by other people.

The two operating systems used for the implementations differ substantially in structure, but there are enough similarities in the semantics that each can support the measurement system paradigm.

The foundation of the measurement system is the ability to collect the appropriate data. There are two requirements for a working implementation of the measurement system. The first is that the host operating system must provide an acceptable model of computation. The second requirement is that the additions or modifications to the host operating system to collect the necessary data be possible without substantially changing the host kernel.

The model of computation that we are using was presented in Chapter 1. The general requirement is that the host operating system provide processes and a message passing interface. Ideally, processes should not share memory and the message interface should be the only interface provided to a process. This insures that the metered events are the only events that occur; i.e, no communications takes place that cannot be detected by the measurement system.

The second requirement means that the system will accommodate the changes and additions needed to provide for data collection. Fundamental to accomplishing this is the ability of the host kernel to use a message interface identical to that used by the processes in a computation. If the message interface can be used for the meter path, then the work needed to add the meter stage is manageable.

The performance of the measurement tools is an important consideration. The first goal for our measurement tools is that processes that are not being metered do not have their execution affected by the presence of the measurement features added to the host system. Thus, if no processes are being measured, then there should be no performance penalty due to the

measurement tools. This goal translates into the requirement that any tests inside the host kernel for the need to perform a metering activity must be fast and simple.

The second performance goal for the measurement tools is that the cost of metering a computation be minimum. Ideally, we would like the metering to cause no degradation in system performance. If the measurements could be collected by listening to the communications on an Ethernet or a ring[Wolf & Liu 78], then the data collection could be done passively on a separate machine on the network. A structure such as that of a passive monitor is suggested in [Presotto 83]. There are three problems with this for program measurements. The first problem is that activities between processes on the same machine are not necessarily visible to the network. Secondly, the data on the network may be at a considerably lower semantic level than that seen by the programmer. At the network level, there could be clustering of short messages into a single packet, splitting of long messages into shorter packets, and a different name domain. The third problem is that there are other activities, such as process creation, that do not appear as explicit communications.

These reasons dictate that the host kernel must participate in the collection of the data. This participation has a performance cost. This cost can be reduced by doing verry little computation when generating a measurement message. If we assume that the filter process resides on a separate machine, then the only cost for a measurement is to send the trace to the filter. The cost of the host kernel generating a meter trace is the cost of sending one message in the host operating system. The cost of sending a message is about a third to a half less expensive then if a process had done a similar message send. This is because once inside the kernel, the overhead of changing protection domains and performing error checking on parameters is not necessary. In addition, meter traces can be buffered, 20 to 50 at a time, before they are sent to the filter. This allows a substantial reduction in message traffic. Since, typically, the cost of sending a one byte message is not significantly different from that of sending a 1000 byte message, this message traffic reduction results in a reduction of the work done by the host kernel.

4.2. The Underlying Systems

The following sections describe the features of the DEMOS/MP and 4.2BSD UNIX operating systems that are relevant to the measurement model. In particular, the following features are described:

(1) The process model, including process creation, destruction, and control.

(2) Process addressing.

(3) The interprocess communication functions, including those necessary to create and destroy message paths.

4.2.1. DEMOS/MP

The DEMOS operating system was first written at Los Alamos Scientific Laboratories for the Cray-1 computer. The system was designed as a message based operating system using communications its fundamental mechanism. DEMOS was ported to a DEC VAX[DEC 81] running under the UNIX operating system. DEMOS was extended to run on multiple machines and ported to a collection of Zilog Z8000 processors[Rapoport 81,Kolovson 81,Carter 81]. The current version of DEMOS/MP runs on both the VAX and Z8000 implementation, though all measurements for this thesis were performed on the VAX because of memory space limitations on the Z8000 processors.

DEMOS/MP is a kernel based operating system. The operating system kernel(s) implement the basic objects of the system (processes, message paths, messages), and the fundamental operations on those objects (process creation, dispatching, message delivery). Most of the functionality of DEMOS/MP is in system processes outside the kernel. These include the file system, memory and process management, user interface (shell), timer process, and name service. DEMOS was originally designed as a message based system, with no semantics for explicit memory sharing between processes. The transition from a uniprocessor to a multiprocessor operating system did not require any substantial change of semantics.

A DEMOS/MP process is shown in Figure 4.1. A process consists of the program being executed, along with the program's data, stack, and state. The state consists of the execution status, dispatch information, incoming message queue, memory tables, and the process's message path table. The message paths are the only connections a process has to the operating system, system resources, and other processes. Thus, this table provides a complete encapsulation of the execution of the process.

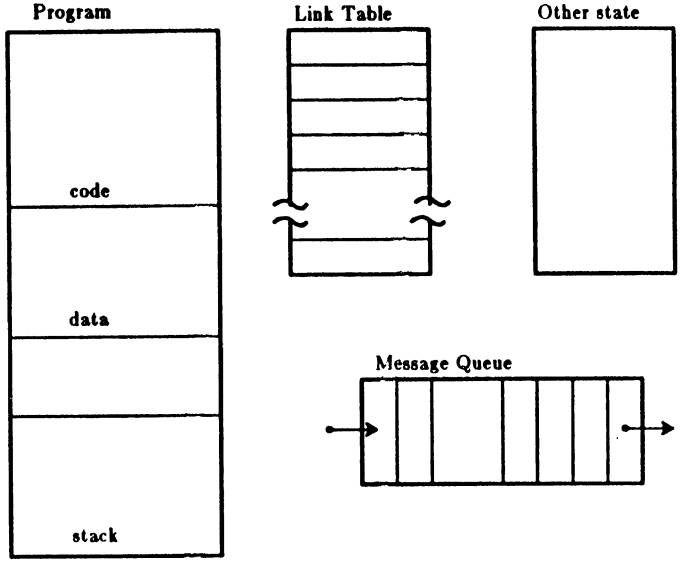

Figure 4.1: Structure of a DEMOS/MP Process

Processes are created and manipulated by requests to the process manager. Process creation in DEMOS/MP is done by specifying a program (a file containing executable code) to run, and creating a new address space in which to execute that program. The new process is then given its initial communication connections. These connections are typically to the name server, file system, and process manager. After the initialization, the creating process requests the process manager to start the new process. During the execution of the process, it may be stopped, resumed, or terminated.

Message paths in DEMOS/MP are called *links*. Links can be thought of as one-way message channels, but are essentially protected and mapped process addresses. Links may be created, duplicated, passed to other processes, or destroyed. Links are manipulated much like capabilities; that is, the kernel participates in all link operations, but the logical control of a link is vested in the process that the link addresses (which is always the process that created it). Addresses in links are context-independent; if a link is passed to a different process, it will still point to the same destination process. A link may

also point to a kernel.

In addition to providing a message path, a link may also provide access to a memory area in another process. When a process creates a link, it may specify read or write access to some part of its address space in that link. This memory is accessed using kernel calls to transfer data to or from the link data area. This is the mechanism for large data transfers, such as file accesses or data transfer in process migration. The kernel sends a sequence of messages containing the data to be transferred in a move data operation.

Processes in DEMOS/MP can do two things: compute and communicate. By computing, we mean the normal execution of instructions by the process. Communication is the means by which the process accesses any state or requests activity external to itself. The process communicates with other processes in the user's computation by using the same communications mechanism as is used to communicate with system processes (e.g., the file system) or the operating system kernels. The kernels use the same communications mechanism to communicate amongst themselves.

DEMOS/MP supplies a set of primitive functions, called *kernel calls*, to allow a process to communicate. The basic operations include:

CREATELINK	A new link is created, that points to the process that creates it. The link is placed in the process's link table. A data area in the creating process may be optionally specified.
SENDMESSAGE	A message is sent over the specified link. The message contains data and, optionally, a link to be passed to the recipient. The sending process continues execution immediately.
RECEIVEMESSAGE	The process waits for a message to be delivered. When a message arrives, or if one is already waiting, the message is copied into the user's address space. If a link was passed with the message, it is added to the receiving process's address space.
ARECEIVEMESSAGE	The same as a RECEIVEMESSAGE, except that it does not wait if no message is available.

CALL	A combination of a SENDMESSAGE and a RECEIVEMESSAGE. A reply link is constructed and passed with the send portion of the call.
DESTROYLINK	The specified link is removed from the process's link table.
MOVEDATA	Two links with data areas and a data area length are specified. Data is transferred from the data area of the first link to the data area of the second link. This is the preferred method of transferring large blocks of data between processes.
STOPTASK	The process making this kernel call is terminated without error.

The important thing to notice about the above list of kernel calls is that it is concise. These functions supply the basic set of operations needed by a process. Other (presumably higher level) functions needed by a process can be requested by sending messages to the appropriate part of the system, using these basic functions.

4.2.2. 4.2BSD UNIX

UNIX was originally designed as a simple, small system to run on a minicomputer. The operating system has since been ported to many different machines and expanded substantially in its functionality. UNIX was designed as a uniprocessor system, with only a restricted facility for cooperation between processes. A major contribution of the 4.2BSD version was the addition of a more general interprocess communication facility. The 4.2BSD version of UNIX used in this study runs on DEC VAX computers. This version was derived from the Western Electric 32V version (which in turn was derived from the Version 7 for the PDP-11).

UNIX is a monolithic operating system. By this we mean that the majority of operating system functionality is contained within the system kernel. The kernel contains process scheduling and dispatching, the file system, network, name service, and memory management. The 4.2BSD version of UNIX has about 150 different system calls.

A process in UNIX is made up of a program and its data, stack, and state. The state consists of the process's execution state, descriptor table, and system stack. The descriptor table provides the program's access to files, devices, and

communication paths. It is important to note that message queues and other information associated with communications are not stored with the process. 4.2BSD UNIX does not provide for processes to explicitly share address spaces (though, the system may have different processes share a pure code segment). There is no implicit or explicit mechanism for processes to interact with each other by means of their address spaces.

Process creation, destruction, and manipulation are implemented by the UNIX kernel. Process creation is done by the *fork* system call. When a process performs a fork, its entire state, including the descriptor table, is duplicated to form a new process. Only files and communication paths are shared between processes. Typically, after a process performs a fork, the forking (*parent*) process continues executing, or waits for the new (*child*) process to complete. The child process then uses the *exec* kernel call to overlay itself with a new program, and then starts executing that program. Between the time when the fork and exec calls are performed, the parent process is still logically in control (even though there are two independent paths of executions). In the child process, this time may be used to initialize the child's state before executing the new program. The initialization may take the form of opening or closing files, or establishing or terminating communication paths. The fork and exec calls are only valid within a single machine in 4.2BSD. There are UNIX systems that have been modified so that the normal UNIX semantics work for process control between machines[Popek *et al* 81, Walker *et al* 83].

In contrast to DEMOS/MP, 4.2BSD UNIX has several different ways of addressing a process. These are by process identifier, communications descriptor, or external naming domain. The process identifier (PID) is an integer value that is assigned to the process when it is created. This value is unique within a given machine. The PID is used in controlling the process, to stop, resume, or terminate it.

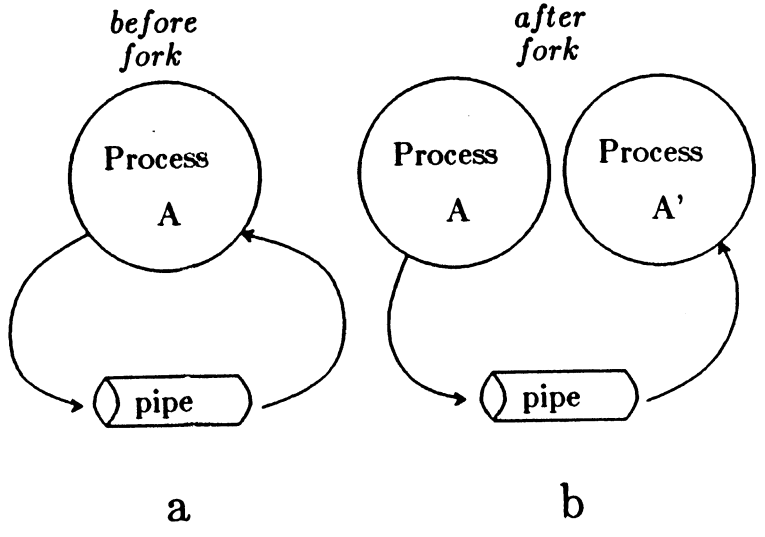

Figure 4.2: Establishing a Communications Pipe

A process may be named by a communications descriptor. A communications path is established in a single process which then performs a fork. This path is called a *pipe*. After the fork, both processes have access to the descriptors for the same pipe and use it to communicate. A pipe is a one-way communications path which has two descriptors; one for read and one for write. The descriptor for the pipe names the object on the other end. This is illustrated in Figure 4.2ab. Note that communications descriptors do not uniquely name processes. Since processes can fork and then share descriptors, a descriptor may refer to any number of processes. Communications descriptors really name communications objects and not processes. Processes can also be named by using external naming domains supported by the interprocess communications. 4.2BSD currently supports two external naming domains. The first, called the *UNIX domain*, is used only within a single machine. Communications objects (*sockets*) are created, and then bound into a name space supported by the file system. Since the file system in 4.2BSD is not accessible between machines, this naming is not valid for processes on different machines.

Independent processes may communicate by agreeing on a name, and using it to establish a communications path. Once established, these paths look similar to pipes. The most significant difference is that socket paths are bidirectional.

The second naming domain is called the *INTERNET domain*. This domain is based on the standard used on the ARPANET[Padlipsky 82]. Sockets are created and bound into an address space made up of machine names (*hosts*) and a list of connections within a machine (*ports*). The INTERNET domain allows processes on different machines to establish communications paths. Once the path is established, it appears the same as a UNIX domain connection.

4.2BSD also provides a facility for sending datagram messages. Datagram messages require the domain address to be specified on each message send. The address specified is a communications object (socket). The current datagram facility provides no guarantee of delivery, or message order.

4.2BSD provides the basic functions to create communications paths, send and receive messages over these connections, and terminate communications paths. Functions to send and receive datagram messages also exist.

4.2BSD UNIX provides two similar methods of sending and receiving messages. Communications descriptors appear the same as file descriptors to the process. This means that the standard *read* and *write* system calls can be used. There are also two system calls for sending and receiving messages, *recv* and *send*. These calls work the same as read and write, with two additions. The first addition is that priority data (called *out of band data*) may be sent over a communications path, and will be delivered ahead of any queued messages. The second addition is the ability of the recv call to look ahead at the contents of the first pending message without really receiving the message. Under normal circumstances, read and recv cause the receiving process to wait until a message is available.

The message stream in 4.2BSD is a continuous stream of bytes. A message send to a particular socket delivers a sequence of bytes. If there are previous messages waiting, the new sequence of bytes is appended to the existing sequence. Message boundaries cannot be detected. Message receives specify a maximum byte count for the message to be received. The use of byte stream communications means that there may not always be a one to one correspondence between message sends and message receives.

To terminate a communications path, its descriptor is *closed*, in the same manner as a file descriptor would be closed.

4.3. The Tool Implementation

The structures of the implementations of our measurement system on the two host operating systems (DEMOS/MP and 4.2BSD UNIX) are similar. The structure of the measurement system outside the metering stage (the kernel resident part) is identical. This section presents the implementation of the overall measurement structure, and then concentrates on the structure of the metering portion for each host system.

The user interface used to start computations and specify the measurements to be collected is described in [Miller *et al* 84].

4.3.1. Common Structures

The metering stage of the measurement system is resident in the host kernel and activated as part of the particular action that is being metered. The metering stage consists of two parts. The first part is the mechanism that allows the user to specify that a given process or collection of processes is to be metered, and which meter events will be traced. It also includes specification of the communication path over which trace messages will flow. This specification is typically a system or kernel call added to the host operating system. The second part is the set of probes added to the various parts of the host kernel that are used to trigger the meter traces. Each of these probes is a procedure call to the meter module. The meter module uses a common set of procedures to form trace messages and send them over the metering path (to the filter stage).

The filter stage is a process. When the filter process starts, it first processes the trace descriptions and selection rules for accepting traces. The filter then creates a communications path over which it will accept traces, and makes this available to processes that are to be metered. The filter spends the remainder of its life receiving and processing traces. If a trace is accepted by the filter, it may be logged for future analysis, or forwarded directly to an analysis process. The choice of logging or forwarding is dependent on the type of analysis being performed.

The analysis stage is a process or a set of programs. For the case where the analysis stage is receiving traces directly from the filter, as in the feedback scheduler (see Section 5.2.6), the analysis stage processes the messages and performs some action affecting the operation of the host system. When the filter logs the traces, the analysis stage is a collection of programs that processes the trace data to extract information about the computation being metered (as in the analyses in sections 5.2.2 through 5.2.4).

4.3.2. DEMOS/MP Implementation

The process and memory management in DEMOS/MP provide a uniform view of the collection of machines on which it runs. Processes (and any other resource) can be created on any machine. Except for the ability to specify the machine on which the process is to be created, there are no semantic differences between processes on the same machine and processes on different machines.

The DEMOS/MP process manager has the ability to move a link into a process's link table. This function was extended to allow the specified link to be inserted as the meter communication path for the process. A function was added to the process manager to allow the specification of the selection flags. For a process that is to be metered, process creation takes the following steps:

1. **Create process.**
2. **Move in standard links (file system, name service, process management, etc.).**
3. **Move in meter link.**
4. **Specify selection flags.**
5. **Start process execution.**

DEMOS/MP provides a one-to-one mapping between communications objects and processes. A link is the only mechanism for specifying communications, and a link always points to a single process (or perhaps a kernel). When a message is sent, both the sender and receiver can be identified. This is because links can be considered a naming abstraction for processes. The implication of this is that the meter traces for all message operations can include the identity of both the sender and receiver.

4.3.3. 4.2BSD UNIX Implementation

4.2BSD UNIX does not provide as clean a view of a collection of machines as does DEMOS/MP. While communication using the INTERNET domain naming can be used within or between machines transparently, 4.2BSD is not a distributed system. It is a collection of machines interconnected with network facilities. Process creation and manipulation is limited, for the most part, to the machine on which the creator is currently residing. In order for the measurement system to be effective, and to allow computations across machines to be constructed easily, a layer of software (a collection of cooperating processes) was implemented to allow the process control functions to span machine boundaries.

Process creation for a metered computation looks much like the DEMOS/MP example, with the addition of extra software to allow operation across machines. The view provided the program developer is the same as that in DEMOS/MP.

4.2BSD UNIX contains both communications objects and processes. There does not necessarily have to be a one to one mapping between these two. When a message is sent, the sending process can be identified. The receiver of a message is only known as the connection to which the message is addressed. Likewise, when a message is received, the receiving process is known and only the connection on which the message was sent is known. This restricts the amount of information that can be given in a meter trace. Extra work is required in the analysis stage to map from the data collected from the traces:

Sending-process to Connection Connection to Receiving-process

to the information needed for the analysis:

Sending-process to Receiving-process.

4.4. Comparison of the Approaches

While much of the structure of the measurement system is the same for both the DEMOS/MP and 4.2BSD implementations, differences in the semantics of the two operating systems cause non-trivial differences in the data that can be collected. These differences are easily understandable in light of the historical development of the two systems. It is important to note the differences in the number of types of interfaces provided by each operating system.

Process naming is a major difference between the two systems. DEMOS/MP originated as a system based on process cooperation, and the extensions to DEMOS were not in the area of new functions but in that of applications to more varied environments. UNIX was originally designed with only the most basic features for process cooperation. The interprocess communication later added was not well integrated into the UNIX model. DEMOS/MP has a simpler model of naming, and a correspondingly simpler structure to the data that can be collected. This becomes most apparent in analyses that attempt to associate message sends with their corresponding receives. 4.2BSD UNIX added communications as a separate abstraction with its own name space. This requires analyses to provide the extra level of mapping from process name space to communications name space and back to process name space. UNIX has been evolving and there have been attempts to unify many of the interfaces. An example of this can be seen in that the

interprocess communication descriptors in UNIX look much like file descriptors. There have been proposals[Cooper 82,Miller & Gusella 84] to extend the descriptors to cover process control and other interfaces, moving closer to the structure of DEMOS links.

An important question is: how much of a process's behavior is captured by the meter measurements? Processes in DEMOS/MP have only one interface to the world, that is, links, and the messages that flow over them. Given that the communications are the basis for the measurement system, we have completely captured all external events from a process. In UNIX, there are many interfaces. In addition to the communications interface, there are interfaces for the file system and process control. These other interfaces can also cause communications. For example, the *signal* and *kill* system calls can change the execution of another process. The debugging system call, *ptrace*, can also change the execution of another process. It is possible to provide meter tracing for all the different interfaces, but this would greatly increase the complexity of the data collection. In an attempt to unify our view of a UNIX process's activities, it would be possible to model such things as I/O activity as message transmissions (reads would be considered message receives and writes would be message sends). Though data collection would be more complicated, analysis of the processes' behavior would still be possible. The details of the analyses appears in the next chapter.

Chapter 5

The Analyses

5.1. Overview

A collection of data needs some form of interpretation to have some meaning. A basic tenet of this thesis is that the measurement model and techniques, and the associated tools, can provide useful data. To demonstrate this, we describe several approaches for the analysis of the trace data generated by our measurement system.

The analyses presented here serve two purposes. First, they serve to show the usefulness of the measurement methods and tools. Each of the analyses requires certain data to be collected. The ease of extracting that data from the measurement system is a measure of the usefulness of the traces. A second purpose of the analyses is to provide a vehicle for solving interesting problems in distributed computing. It is a test of the measurement system to see if it provides a reasonable (or even possible) solution to these problems.

This chapter is divided into two sections. The first section is a collection of scenarios for the analysis of the trace data. Each scenario consists of the motivation for the analysis, the description of the data collection paradigm, and the algorithms for analysis. The motivation is a description of the problem that is being solved or of information that the analysis would provide. The problem may be collecting data to provide simple statistics, or extracting complex structural information about the program. The data collection (measurement) procedure is then explained. This includes the organization of the components of the measurement system and the type of filtering performed. Lastly, the algorithms for data analysis are described.

The second part of this chapter describes several sample analyses that were performed using the measurement facility and the techniques described in the preceding section. The sample analyses include studies of programs that already exist, as well as programs that were constructed specifically to test the usefulness of these tools.

5.2. The Analysis Techniques

5.2.1. The Trivial Case

This first technique is informal and is not really considered an analysis technique. It is included for the completeness of this discussion. This technique is to merely look at the meter traces from a computation and extract information by using human reasoning and pattern recognition abilities. This is conceptually uninteresting, but is considered as a last resort when attempting to understand the execution of a computation. We mention it to insure that our design does not preclude the possibility.

5.2.2. Basic Communications Statistics

We have defined a computation to be a collection of cooperating processes. The processes cooperate, and the cooperation is based on some communications mechanism. It is reasonable then to want to know the nature of the communications between processes. Several basic questions come to mind. Who is talking to whom? (Which processes are talking to which other processes?) What is the volume (total message traffic) of the communications? How frequent (time density) are the communications? How large are messages?

In addition to these basic questions, a few more interesting queries come to mind. Given information about the arrival and consumption of messages, we can derive information about the message queues. It is possible to gather statistics such as the maximum, and average queue lengths for each process. With the same information we can obtain the minimum, maximum, and average time that a message waits in the incoming message queue before it is consumed by the process. However, we are not restricted to minimum, maximum and average. We can collect data to record the distribution of the various measurements.

There are several reasons why these message statistics are useful. When first studying a distributed program (or the traces of a distributed program's execution), it is useful to get an overview of its behavior. The message quantity and density statistics give a first view of the interactions in the computation. This gives the programmer an initial indication of the behavior of the program. Information such as local (intramachine) and remote (intermachine) message levels can also be obtained.

The message queue length information can help in structuring the program. Suppose we have a program that provides some service by receiving request messages, performing some action, and then replying to the requesting process. If many requests are being handled at one time, then the server program may involve several processes working together. If we know that the message queues for awaiting requests are large, then we should structure the server to handle requests differently, or allocate more resources (e.g., machines) to the server.

Traces are needed for the acts of sending a message, delivering (queuing) a message, starting a message receive operation, and the actual receiving of the message. There must be enough information within the message to identify: (1) the sender and receiver of the message, (2) a time local to the machine on which the trace was collected, and (3) the locations (machines) of the sender and receiver. The only operations necessary to trace for simple message statistics are the message sends and message receives. The selection rules (for the DEMOS/MP implementation) are be:

```
traceType=Send, channel=#*, code=#*, sendLink=#*, passLink=#*;
traceType=Receive, channel=#*, code=#*, passLink=#*;
```

These rules specify selection only on the basis of the knowledge that a trace type is either a Send or Receive. The fields *channel*, *code*, *sendLink*, and *passLink* are to be discarded (not logged with the trace record).

The message queue information requires two more trace types to be accepted by the filter. The selection rules for this case are:

```
traceType=Send, channel=#*, code=#*, sendLink=#*, passLink=#*;
traceType=Receive, channel=#*, code=#*, passLink=#*;
traceType=ReceiveCall;
traceType=MessageQueued, channel=#*, code=#*;
```

The time at which the message arrives for a process and is queued (MessageQueued) and the time at which a process starts a receive operation (ReceiveCall) are needed for the additional analysis.

The organization for collecting these message statistics is based on the development of a single distributed program. A collection of processes executes, each with the meter path directed at a common filter process. The filter process records the traces for later processing.

An alternative organization is one that lets the programmer observe the program while it executes. The filter process passes the selected data to an analysis process. The analysis process summarizes the data as it is received and then displays the summary data. The summary of message statistics could be presented in a tabular format, or could use graphics to represent the processes and their communications paths.

The computations necessary to derive the basic message statistics involve only counting and arithmetic means. The only timing information that is necessary is the local time base that is kept on each machine. The quantity of message traffic is calculated by counting messages to or from each process. The time density of message traffic is the quantity of traffic divided over the period of time during which the process executes. The information about message sizes can be kept as a running average or grouped by sizes to provide information about the distribution of sizes.

The message queue information is obtained by recording the incoming message queue size for each process on each message operation. Each time a message is delivered to the process, the size of the queue length increases by one, and for each message receive operation it decreases by one. Observing these operations over the life of a process allows the maximum and average queue sizes to be computed. The time that a message waits in the message queue is computed in a manner similar to that used for the queue length. Each message delivery is paired with the corresponding message receive operation, and the time difference is computed to give the time that the message waited in the queue. Minimum, maximum, and average waiting times are derived from these measurements.

5.2.3. Measuring Parallelism in a Computation

One motivation for writing a distributing computation is to achieve an increase in its speed of execution. This performance increase is obtained by means of parallel execution of the processes within the computation. Once we construct a distributed program, the problem becomes: how do we measure the amount of parallelism in the execution of our program? In addition, it would be useful to determine the optimal distribution of the computation's processes among the available machines. There are two extremes in distributing processes to machines. There first case is when each process executes on its own machine. Maximal independent computation can be achieved by each process. A process does not share its resources with other processes and is only slowed by other processes when it waits for a message. The second case is when all processes run on the same machine. Communication between processes can be fast (relative to intermachine communications), but there can be contention among the processes for the CPU. Typically, the optimal solution lies somewhere between these two extremes.

The model and algorithm for the analysis of parallelism are based on traces obtained from a particular execution or executions of a computation. The results obtained will vary according to type of work done by the computation. As described below, the results should not be affected by factors such as system loading or the configuration of the computation (assignment of processes to machines) when the measurements are made.

The information needed to measure parallel execution can be obtained from the trace messages generated by the measurement system. The only trace types needed are message sends and message receives. There must be enough information within a message to identify the sender and the receiver of the message, and the amount of CPU time consumed by the process up to the time that trace event occurred. The selection rules for this analysis would be:

```
traceType=Send, sendLink=#*, passLink=#*;
traceType=Receive, passLink=#*;
```

These rules specify that only Sends and Receives are to be selected. The fields *sendLink*, and *passLink* are to be discarded (not logged with the trace record). The fields that are kept (fields not listed in the selection rule are ignored in the selection process) and used are PROCTIME, the amount of CPU time used by the process, and MACHINE, the machine identifier for the machine on which the process generating the trace executed.

As with the basic message statistics, the organization for collecting the trace data centers around the development of a single distributed computation. The collection of processes is executed, each process with the meter path directed at a common filter process. The filter process records the traces for later processing. The data is then available for post mortem analysis.

To measure parallelism in a computation, we first need to define parallel execution. This definition will be in terms of the trace events that we measure. Each executing process consumes machine cycles. We define *process time* (t_{proc}) to be the number of time units of CPU time that a process consumes over its lifetime. The amount of process time for process i is t_{proc_i}. The *total time* (T) for a computation is the sum of the process times for all the processes in the computation:

$$T = \sum_i t_{proc_i}$$

A process execution is a sequence of *events*. Between each pair of successive events, the process expends some quantity of its process time. We can represent a process as a list of events, with the events as nodes and directed arcs joining the nodes showing the flow of execution. The arcs are labeled with the amount of time expended between the two events. Figure 5.1 shows a computation consisting of three processes with events and execution

times labeled.

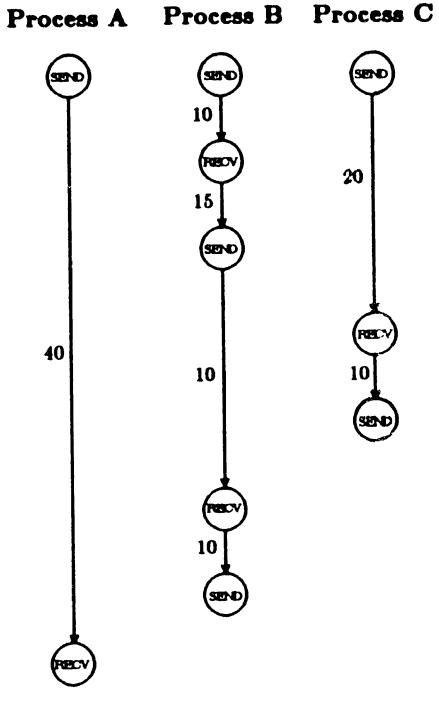

Figure 5.1: A Computation and its Events

There are interactions between processes in a computation. These interactions take the form of messages between the processes. The message interactions are added to the list representation of the computation to form a directed graph. Each message is represented as an arc from the send event to the corresponding receive event (providing both events are present in the graph). The arc is labeled with the time required to deliver the message. Figure 5.2 shows the sample computation with the message arcs added. Each node on the graph has an in-degree[Aho *et al* 74] of at most two, and an out-degree of at most two. Each message has exactly one sender and one receiver. The computation graph is directed and acyclic. It is important to note that the computation graph does not describe the program itself, but the history of one execution of the program. Each node in the graph represents an external event in the execution of the program. Following the directed arcs through

the graph reflects the passage of time. Therefore, the graph could never have a cycle.

The computation graphs can be used to determine whether a process performs synchronous or asynchronous communications. By synchronous communications, we mean that a process sends a message and immediately blocks waiting for a response. Asynchronous communication means that a process may continue computing after sending a message. The computation in Figure 5.2 shows processes performing asynchronous communications. The process times on the arcs between the send nodes and the receive nodes for each process are greater than zero. If the arcs between the sends and receives (of an individual process) were labeled with zero (or near zero) values, that would suggest synchronous communications.

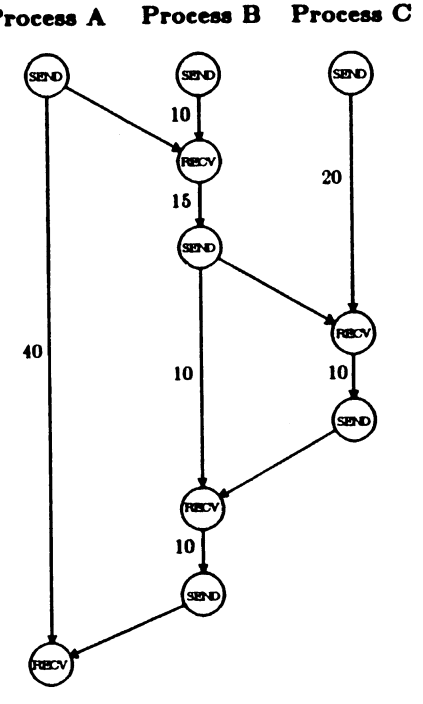

Figure 5.2: Computation with Message Arcs

In a graph for a computation with n processes, there are exactly n nodes with in-degree of zero. These nodes represent the start of computation for each process and are called *initial nodes*. The graph also contains at most n nodes with out-degree of zero. These nodes represent the termination of computation and are called *termination nodes*. We define the *maximum path* for process i to be the directed path from the initial node for process i to the termination node that gives the greatest weighted path length. The value of this greatest weighted path is t_{max_i}. The absolute bounds on t_{max_i} are

$$t_{proc_i} \leq t_{max_i} \leq T$$

The maximum path time for the entire computation (t_{max}) is the maximum over all the t_{max_i} for the component processes. The absolute bounds on t_{max} are

$$t_{proc_j} \leq t_{max} \leq T \qquad \text{for all processes } j.$$

For a given execution of a computation, if T equals t_{max}, then there has been no parallel execution. This would be the case if the entire computation were run on a single machine; the machine could execute the computation in no less time than the total process time used (T). If t_{proc_i} equals t_{max}, for all i, then we are achieving 100% parallel execution; the computation has been split into uniform size pieces (processes) and all the pieces are executing concurrently.

We define the parallelism factor, **P**, to be:

$$\mathbf{P} = \frac{T}{t_{max}}$$

The case where there is no parallel execution gives a **P** of 1, and the case of 100% parallel execution gives a **P** of n, for a computation consisting of n processes. If all the processes in a computation were executing on a single machine, and there were no interactions between the processes, we would have a **P** of 1. It is likely that if the processes were in the same computation, there would be some interaction. Given interaction between the processes, execution on a single machine, and non-zero message delivery time, it is possible to have a **P** of less than 1.

Computations are not always self-sufficient. Processes in a computation may need to interact with other processes or parts of the host system that are not part of the computation that is being studied. There is no way for us to analyze activities that happen outside the available trace data. The problem becomes important when some process in the computation must wait for a

response from an external process (or part of the host system). Looking at Figure 5.3A, the time through the external process could be longer than the direct path from the SEND to the RECV. If the communication with the external process is a synchronous request, the process time between the SEND and RECV would be close to zero, and the time taken in the outside process becomes important. We can model this problem by modifying the computation graph as in Figure 5.3B. The label of the new arc would be the difference in elapsed time (not process time) between the send and receive event.

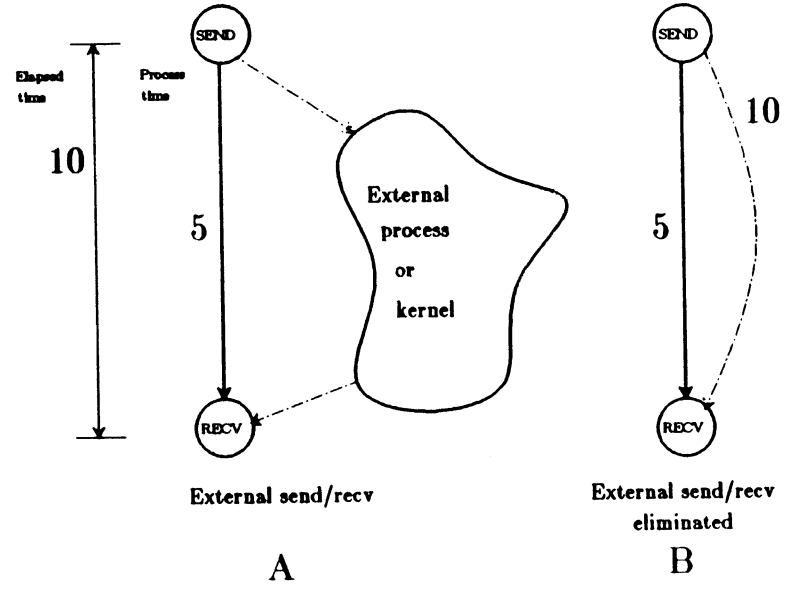

Figure 5.3: Computation Graph with External Event

The analysis of parallelism proceeds in three steps. In the first step, we assume that there are unlimited machines (as many as there are processes), and that communications are infinitely fast (the weight on all message arcs is zero). By computing **P** we determine the theoretical upper bound that could be achieved. Without a change to the computation or to the system on which we are running the computation, this is the best we can do.

Communications between processes are not infinitely fast. The second step is to factor in communications costs for some assignment of processes to

machines. Processes are still considered to have unlimited computing resources. By computing **P** in this configuration, we can isolate the effect of communications delays on execution. Estimates of the delay time for a message delivery can be obtained by sample measurements on the system where the traces were obtained. The measurements must be made for both the local case and the remote case. Also, if the message delivery time varies depending on which machines are involved in the message exchange, this information can be used in the computations. The length of the message can also affect message delivery time. Measurements for varying lengths of messages can be used to give a more accurate analysis of the computation. The meter traces include the message length for both send and receive operations.

It is not possible to obtain the message delivery time from the message traces. If a message was sent from a process residing on one machine to a process on another machine, the send and receive times would come from independent clocks. These clocks cannot be synchronized closely enough so that the times on the two machines can be compared.

The last step in the parallelism analysis is considering that processes residing on the same machine are competing for machine cycles, and therefore are each decreasing the other's execution speed. This last step gives a realistic picture of the execution of the computation. When more than one process executes on a machine, there is competition for use of the CPU. The number of processes competing for the CPU varies over the execution of the computation. Several factors determine how many processes are running. The first factor is how many processes are currently assigned to the machine. Other factors are process creation, termination, and migration from their current location. The number of processes currently assigned to a machine forms the upper bound for the number that are able to run at any given instant.

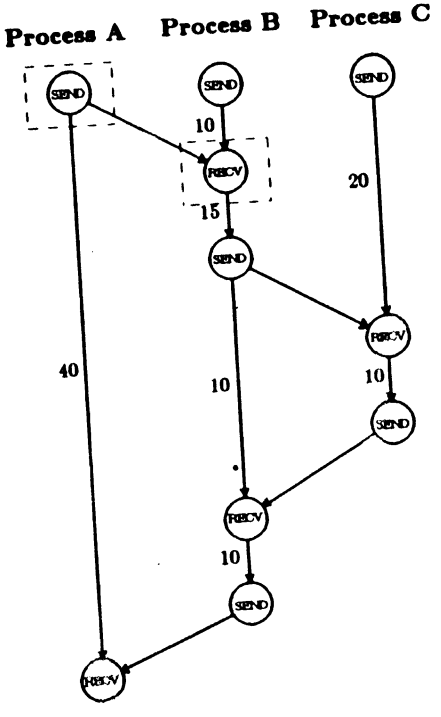

Figure 5.4: Computation with SEND/RECV event marked

Given the number of processes that could be run (the number currently on a particular machine), the problem is to determine when a process is running and when it is blocked from execution. The model of computation that we are using states that a process can only compute and communicate. The only blocking activity in this model is a message receive. A process is blocked from execution when it is attempting to receive a message that has not yet arrived. Figure 5.4 shows the sample trace of a computation. Before the time at which the indicated SEND operation for process A occurs, process B cannot continue past the indicated RECV. If process B reaches this event before the send, process B is blocked, thus reducing by one the number of processes contending for the CPU. The time delay for sending the message has been ignored in this example.

The algorithm that factors in the CPU sharing relabels the process time arcs with a new weight calculated from the number of processes currently able to run. Figure 5.5 shows the sample computation graph redrawn with concurrent time intervals (*slices*) for the three processes. Consider the case where all processes execute on the same machine. For a given interval of t units of time, with k processes currently able to run, the time interval is relabeled with a value of $k \times t$. Figure 5.6 shows the sample graph after the relabeling has been completed. The complete algorithm for relabeling the graph is given in Appendix A.

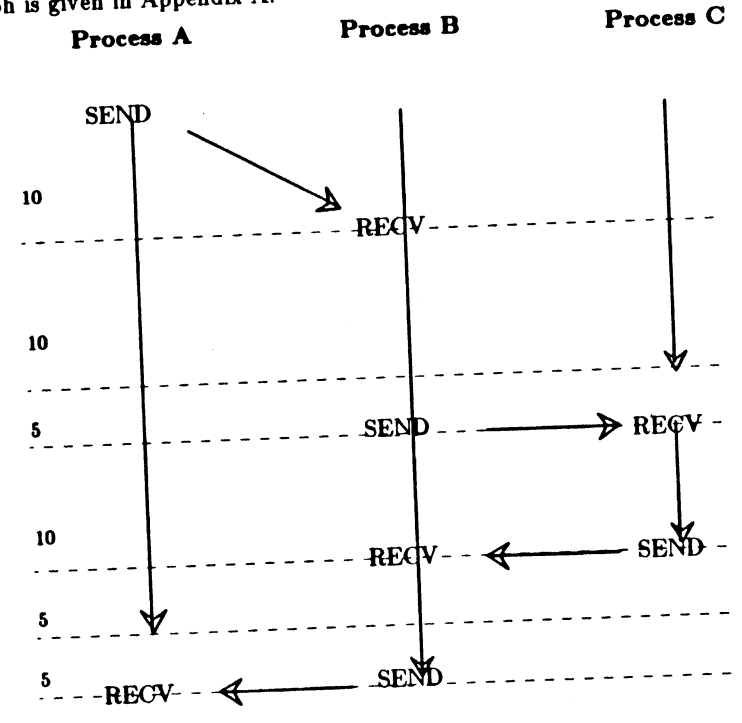

Figure 5.5: Computation Showing Intervals of CPU Sharing

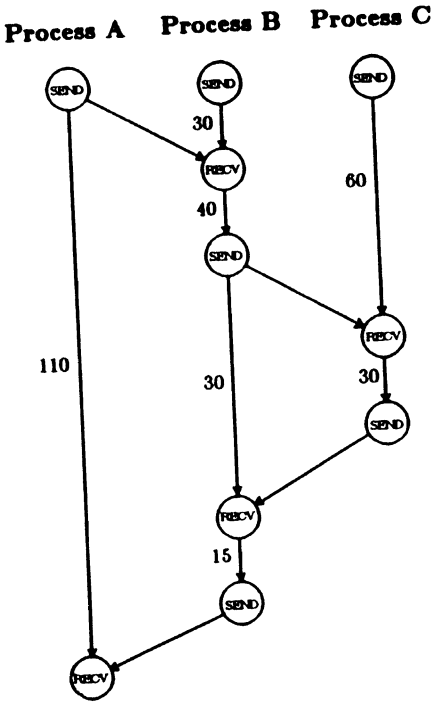

Figure 5.6: Computation with Relabeled Arcs

By varying the assignment of processes to machines, we can determine the best placement of the processes. Of course, this placement is best for the particular execution represented by the traces being studied. The trace data also identifies the machine on which the processes executed. This information can be used as an initial assignment for analysis and comparison. The trace should be over an extended execution of the computation (i.e., a larger sequence of requests for a server, or a large problem size for a numerical or seminumerical computation) for the parallelism results to be more accurate; or the analysis should be performed for many different traces of the computation's execution.

There is the implicit assumption in the preceding discussion on analyzing parallelism, that the reassignment of processes to machines will not change the execution order of events. We must make this assumption because our results come from analyzing program executions, rather than the programs

themselves. This assumption is valid if the order of interactions between the processes in the computation is deterministic. This is not always the case. A computation structured as a server, receiving requests from many independent sources, would not fit the deterministic assumption.

We can make a slightly weaker assumption about our computations. That is, that each process in the computation is deterministic with respect to its inputs. This means that a process will behave the same way independent of when it receives a given input. When processes are reassigned (in the parallelism analysis) to different machines than they were really executed, the order of execution of events in the computation could change. Given our weaker assumption about determinism in a process, this change of execution order should not have a large effect on the overall computation.

This weaker assumption is not always true. The order of interaction can affect such things as the search time in data structures. It is possible to construct pathological cases that behave much differently, given a different machine assignment. But the overall effect of these factors should be small enough in actual computations that the results for reassignment will be valid. Section 5.3.1 provides results from the DEMOS/MP file system to support this argument.

The parallelism analysis does not consider that the computation we are measuring may be sharing a machine with processes that are not part of the computation. The load on a machine caused by other computations could cause less accurate results when trying to determine the execution speed of our computation. This affect of other computations can be added to the parallelism analysis by including machine load information in the trace message header for each meter trace. As the load on each machine changes over time, this effect can be factored into the analysis.

5.2.4. Detecting Paths of Causality

When we write a computation consisting of several processes, we specify the order and frequency of the communications in the computation. We specify this information for each interaction between processes. We establish rules and protocols to provide for the correct execution of the program. But when the entire computation is executing, the overall interactions are more complex than this static picture of the computation suggests. The increased complexity comes from parallel execution within the computation and from the fact that several partially completed requests maybe be simultaneously active within the computation.

The model of computation in which we are interested for this analysis is that of a *server*. A server is a computation that receives requests from processes outside the computation, computes a result (involving one or more of the processes within the computation), and then returns the result to the requesting process. There are several questions to be answered about the behavior of a server. One questions is: given a request message received by the server, what message paths within the server are most commonly traveled? This question can be translated to: what sequences of interprocess communications occur most frequently? For sequences of length two, we can derive this information from the basic message statistics (see Section 5.2.2). The message statistics cannot provide information about longer sequences of interactions. Related to the longer sequences of interactions is a second question: given a process that has just received a message, where will that process next send a message? This question can also be viewed as determining a branching probability[Ferrari *et al* 1983], given a specific input to a process.

The basic strategy for causality analysis is to identify each request to the server, and follow the sequence of interactions within the server caused by that request. The data we need for this analysis is the same as was needed for parallelism analysis (see Section 5.2.3). Send and receive message traces are collected, and the graph of the events in the processes is constructed, with arcs in the graph between each corresponding send and receive (see Figure 5.2). This graph represents the complete collection of interactions between processes during the life of the computation.

To reduce the complexity of analyzing such a large quantity of data, we convert the problem to one of manipulating character strings. Each process in the computation is assigned a single letter designator. The two events corresponding to a message being sent and received are designated by the letter for the sending process, followed by the letter for the receiving process. For example, the send and receive pair marked in Figure 5.4 would be the string:

AB

We create a list of strings, where each string represents one request to the server and the subsequent activity within the server. These strings are called *causality strings*.

There are three types of processes that are visible during a causality analysis. The first type of process is the *server* process. This process is contained within the computation that is providing service. The second type of process is the *requestor* process. Requestor processes are the customers of the server. They make requests and receive results. The last type of process

is the *system* process. System processes are those processes to which the server makes requests. The system processes may be other servers, or perhaps a host kernel. Messages received from a requestor indicate a request for service from the server computation. Messages to system processes from the server are ignored (as are the responses), as we are interested in tracing the flow of control through the server, and not through external processes.

The algorithm for building the causality strings traverses the entire computation graph. The list of causality strings is built by:

(1) searching for each message receive event from a requestor process;

(2) for each such receive, the message sends immediately following the receive are identified;

(3) the message receives corresponding to sends in (2) are identified (i.e., the message arcs are followed), and steps (2) and (3) are repeated for each receive.

A causality string is initially a single character, which is the process performing a message receive that was detected in step (1). Each time a send is followed (i.e., a message arc is traversed) to its corresponding receive (step (2)), an additional letter (identifying the receiving process) is added to the causality string. Events associated with system processes are ignored in this algorithm. For example, the causality strings for Figure 5.7 are:

ABA
ABCBA

Once we have the causality strings, there are several results that we can obtain from them. The first result is identifying the most commonly traveled paths through the server. We take each causality string and enumerate all of its proper substrings of length 2 and longer, up to and including the causality string itself. These strings are stored in lexicographical order with a value indicating the number of times that the string has occurred in the computation. This list of substrings identifies the most commonly occurring message sequences.

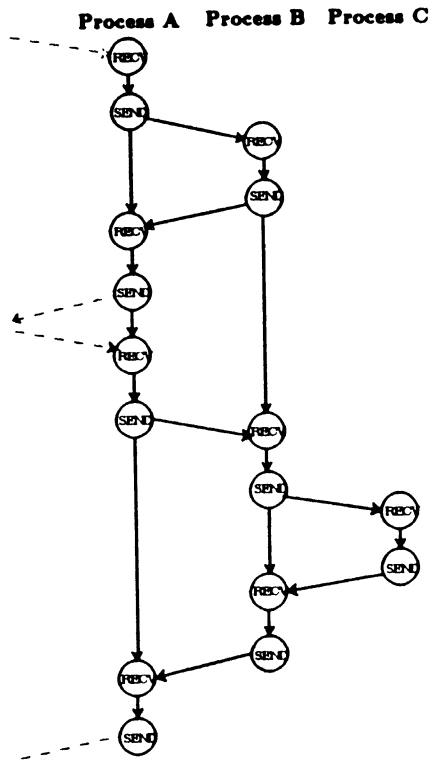

Figure 5.7: Sample Computation Graph for Causality Analysis

Given three processes, A, B, and C, we can also use the causality strings to compute the probability that process A, having just received a message from process B, will next send a message to process C. This information is similar to the branching probabilities in a queuing model or graph model simulation. The branching probabilities allow the programmer to understand how the interactions between pairs of processes are affected by the overall execution of the computation. These probabilities cannot be calculated from the simple message statistics. These statistics do not correlate message receives with the corresponding message sends.

A programmer starts with a program that defines the rules for interaction. This is a static definition. When the program executes, interactions between many processes in the program, and asynchronous

execution make the program's behavior more complex than the static view. Causality analysis provides a facility to extract dynamic execution information that can be related to the programmer's static view of the program. An example of this analysis using the DEMOS file system is given in section 5.3.2.

5.2.5. System Measurement: Communications

The measurement system can be used by the host operating system to provide information on systemwide performance. In particular, it can be used to monitor communication levels for the various machines in the distributed system. We can include all processes executing on a particular machine or collection of machines. These measurements could apply to the entire system (though their performance implications, with respect to the amount of trace data generated, must be considered).

The measurements for this analysis are done at the level of the interprocess communications interface. Details of the protocols implementing the various physical communications paths are not visible at this level. Measurements for the lower level protocols are within the domain of a network monitor.

The problem of systemwide measurements is one of organization, rather than of determining clever methods for evaluating the data. The measurement facility is initiated by the host system instead of by explicit commands from the users of the system. A decision is made by the system administrator about the scope of the measurement; programs, as they are started, have the data automatically collected. Conceptually, for the entire portion of the system being measured, a single filter process is needed.

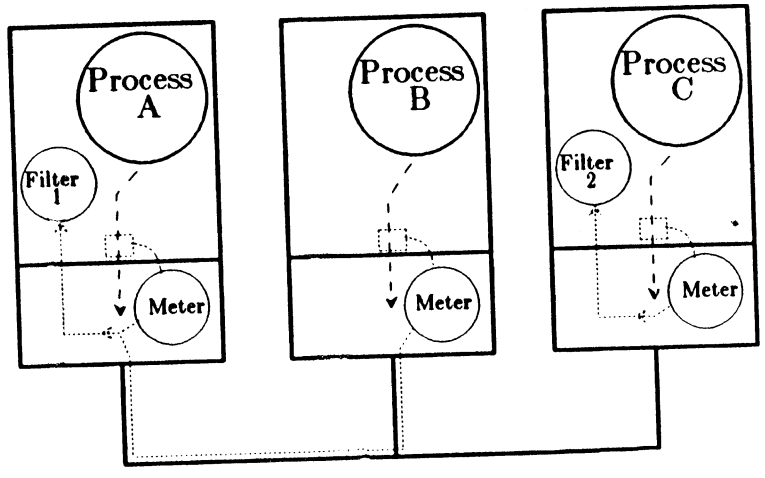

Figure 5.8: Meter Trace Collection with Multiple Filters

It may no longer be practical to have a single filter process when the number of machines included in the measurement becomes large. As activity (message traffic and process creation/destruction) in the measured portion of the system increases, so does the quantity of trace messages generated. The increasing communications traffic (from the trace data collection) impacts the performance of the remainder of the system. A more substantial influence is the increased load on one (or more) machine caused by an increasingly busy filter process. For large measurements, it would be necessary to dedicate a single machine to recording messages. This is similar to the *recorder node* in Published Communications[Presotto 83]. Alternatively, it may be desirable to distribute the cost of collecting and filtering the data. Several filter processes can be used (see Figure 5.8), each collecting data from a particular part of the system. A natural organization would be to provide one filter process per machine. This would work as long as each machine had a mass storage device. Periodically, data could be gathered from the various machines, merged, and then evaluated. Alternatively, data could be summarized at each machine,

and this summary data evaluated at a central site.

To study communications levels, it is necessary to meter only those events that generate communications traffic. If both the sender of the message and the recipient can be identified from either the send trace or receive trace (as happens in DEMOS/MP), only one of the two types of traces needs to be collected. If an intermediate communications object (such as a socket in 4.2BSD UNIX) is used, then both the send trace and the receive trace are necessary. The meter traces should contain a field indicating the time at which they were collected. This is useful when one tries to approximate the order of events by merging separately collected trace streams. A time stamp is also useful in computing the frequency of events. The trace times only need to be compared to other times from the same machine for computing event frequencies. This means that the clocks on the different machines need not be well synchronized. The accuracy of the total ordering of events improves as relative accuracy of the clocks on the different machines improves.

The analysis techniques for the data collected according to this scheme are the same as those for the basic message statistics of a single program. The number of processes would be larger when measuring an entire system, but the type of computations would be the same.

5.2.6. System Measurement: Feedback Scheduling

The analyses discussed thus far are intended to provide the programmer or system manager with information. This information may result in the programmer restructuring the computation being studied or changing the environment in which the computation executes. In both cases, a human being is part of the feedback loop.

It is possible to return the results of some of the data analyses directly to the host system. After the meter traces have been analyzed and reduced to some reasonable statistic, this information can be passed directly to the host system to be used in scheduling decisions. The trace data becomes direct feedback to the host operating system.

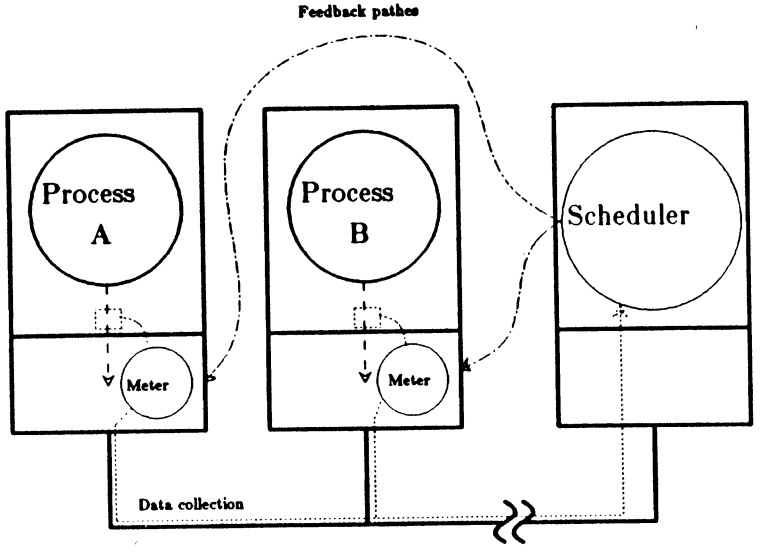

Figure 5.9: Measurement System for Feedback Scheduling

The data collection for feedback scheduling looks similar to that described in the previous section on system communications measurements. The same type of organization for metering and filtering could be used. The filters would not store the selected data. After filtering, the selected data would be passed on to the host system to provide data for scheduling. This data would then be used to modify the behavior of the host system, changing the execution environment for the programs being metered. This is illustrated in Figure 5.9.

The type of resource scheduling that can make use of the meter data is limited by the frequency with which the data needs to be collected. Activities that occur with about the same frequency as interprocess communications, or with lower frequencies, would be reasonable to measure since the meter traces themselves are messages. Activities more frequent than these would overly degrade system performance. The measurement system can provide

information that is gathered from other sources. These other sources could be more traditional performance tools gathering data on machine loading, memory usage, or paging activity. The meter message would be the medium that would carry periodic summaries of these other activities.

5.3. Sample Analyses

The following sections include several experiments using the measurement methods and system described in Chapters 3 and 4, combined with the analysis techniques introduced earlier in this chapter. The evaluation of a system such as that presented in this thesis should be based on its usefulness. We have been able to obtain useful and interesting results in the limited studies we have performed. More time than can be dedicated to a dissertation is needed to fully evaluate the models, methods, and tools.

The measurements for the various studies were performed on the DEMOS/MP and 4.2BSD UNIX operating systems. The DEMOS/MP measurements for these analyses were collected on the VAX implementation (described in Chapter 4). The UNIX measurements were run on a collection of VAX computers, connected by a 3 megabit Ethernet[Metcalfe & Boggs 76]. The machines on the network were VAX 11/780's and VAX 11/750's with varying amounts of memory and peripherals.

5.3.1. The DEMOS/MP File System

The DEMOS file system[Powell 77] provides an interesting case study for several reasons. The first reason is that it is a complex, multiprocess computation. It consists of over 20,000 lines of high-level language code organized into four processes. Secondly, it performs a complex function used by a significant portion of the system. The file system interacts with both user (non-system) processes, and with the operating system kernel. The third reason is that it is long lived, since it is available to be studied for as long as the operating system is running. The fourth reason is that, in our experiment, the file system was measured by a person other than its author. This provides a test of what can be discovered about a program without of an intimate knowledge of its structure. This discussion follows the steps that we took in analyzing the DEMOS file system.

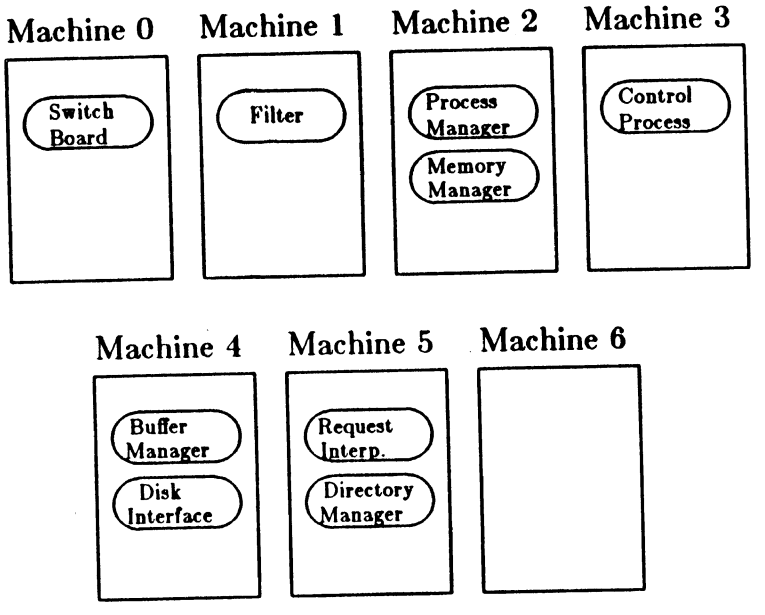

Figure 5.10: Layout of System Processes for File System Test

The measurement environment is the DEMOS/MP operating system running on seven machines. Figure 5.10 illustrates the sample layout of system processes (including the file system) on the machines. The elements of the file systems are the Request Interpreter, Directory Manager, Buffer Manager, and Disk Interface. Measurements start at the beginning of the file system's execution. This means that the file system's initialization activity of is included. User processes that will make requests to the file system were active on the various machines over the period of the test. The test was complete when all user processes terminated. The user processes are test programs that come in two classes. Each of these classes performs a different mix of computation and file I/O.

The first class of user processes performs continuous sequential accesses through a file; first writing the entire file, then reading it. The second class of user processes do continuous random accesses to a file. These accesses consist of both reading and writing. The user processes start at the beginning of the test and terminate when they have completed their task.

Two types of analyses were used to study the file system: basic communications statistics (section 5.2.1) and parallel execution behavior (section 5.2.2). A progression of measurements and analyses were performed to refine our understanding of the structure of the file system.

The first step was to get an initial feeling for the computation (i.e., the file system). The file system was run with the collection of user processes making requests, and parallelism analysis was then performed on the trace data from the file system. Traces from approximately 10,000 communication events associated with the file system were collected. The initial step was to compute **P** for the theoretical maximum parallelism (assuming zero message delay time and no CPU contention) for the four file system processes. We have written programs to automate the parallelism analysis. This involves building the computation lists, associating corresponding send and receive events, and calculating the longest paths. The result of this analysis was a **P** of 1.65 (out of 4), or 41%.

Message size (bytes)	Elapsed time per message (seconds) Local	Remote
1	.0042	.013
10	.0043	.013
100	.0080	.016
1000	.0128	.035

Figure 5.11: Message Delivery Times by Message Size

The next step in the analysis was to assign the processes in the file system to machines and evaluate **P** taking the message delivery times into consideration. Message delivery times were calculated by performing repeated send-receive-send-receive sequences between two processes. These processes repeated the sequence many times (typically more than 10,000) and the elapsed time to send these messages was measured. From this time, we can compute the time to send a single message. The measurements were made first between two processes on the same machine, and then between two processes, each on different machines. The machines were running no user processes other than the two processes involved in the test. The test was repeated for various message sizes. Since we are interested in the processes' view of message communications, we want to measure the end-to-end delivery time. The results of these measurements are summarized in Figure 5.11.

The initial assignment (for the analysis) of file system processes to machines was taken from the measurement data (traces). This data contains the configuration in which the DEMOS file system processes actually executed. The actual configuration had the Buffer Manager and Disk Interface running on one machine and the Directory Manager and Request Interpreter on another machine. The trace data was analyzed, accounting for message delivery time but ignoring any possibility of CPU contention between the processes. Parallelism analysis produced a **P** of 1.43 (or 35%). This shows a 6% loss of parallel activity due to communication delays for the chosen configuration.

The final step in parallelism analysis is to consider the effect of CPU contention on processes executing on the same machine. We repeat the parallelism analysis, keeping the same assignment of processes to machines as above. The results of this analysis will include the decrease in execution speed caused by processes sharing CPUs. We evaluated **P** for this case and obtained a value of 1.21, or 30%. CPU sharing is responsible for an additional 5% loss in parallelism for the file system.

At this point it was natural to ask if there are any other configurations that would provide a level of parallelism closer to the ideal. Parallelism analysis was repeated for two more cases: (1) each process on its own machine, and (2) all file system processes on the same machine. When each file system process was run on its own machine, **P** fell to 1.07 (26%), which is only marginally better than serial execution. This indicates that the larger communications delays of intermachine messages severely affected file system throughput. The parallelism factor was also calculated for the case where all processes were on the same machine. In this configuration, **P** was 0.95, or 23%. **P** was less than one because serial execution was further degraded by local communication delays. If the entire file system had to be run on the same machine, it might make sense to combine it into a single process.

From Disk Interface To:				
Total		Switch-board	Kernel	Buffer Mgr
% CPU	Msgs Bytes	Msgs Bytes	Msgs Bytes	Msgs Bytes
30%	27% 40%	1% †	40% 34%	58% 65%

From Buffer Mgr To:					
Total		Switch-board	Disk Interface	Request Interp.	Users
% CPU	Msgs Bytes	Msgs Bytes	Msgs Bytes	Msgs Bytes	Msgs Bytes
51%	33% 42%	3% †	44% 45%	36% 32%	16% 22%

From Directory Mgr To:					
Total		Switch-board	Request Interp.	Buffer Mgr	Users
% CPU	Msgs Bytes	Msgs Bytes	Msgs Bytes	Msgs Bytes	Msgs Bytes
1%	2% †	5% 5%	80% 67%	5% 11%	10% 21%

From Request Interpreter To:					
Total		Switch-board	Buffer Mgr	Directory Mgr	Users
% CPU	Msgs Bytes	Msgs Bytes	Msgs Bytes	Msgs Bytes	Msgs Bytes
18%	38% 17%	1% †	49% 93%	5% 1%	44% 6%

(† indicates less than 1%)

Figure 5.12: Basic CPU and Communication Statistics

Where is the time spent? To get a better understanding of the file system, we combined basic message statistics analysis with parallelism analysis. Message statistic analysis was performed on the same performance traces that were used for parallelism analysis. This allows us to further examine the file system's performance, using our standard analysis tools, without taking more measurements. Figure 5.12 summarizes the communications and CPU usage for the execution of the file system. The percentages of CPU time for each process (out of the total for the entire computation) are listed for each file system process. Also listed is the percentage of messages sent, and the percentage of message bytes sent by each process (out of the total for the entire computation). The tables also show, for each file system process, the percentage of messages and message bytes that were sent to other processes (out of the total for the individual process).

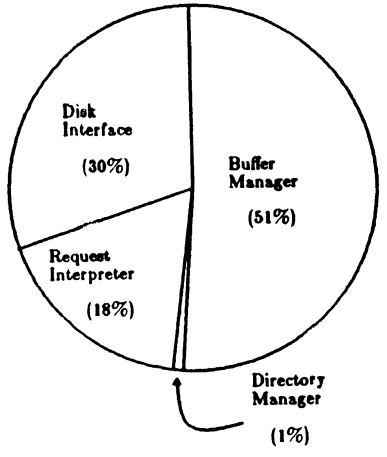

Figure 5.13: Percent CPU Time for each File System Process

The first thing to observe in this table is the percentage of CPU usage for each file system process (shown in Figure 5.13). Most noticeable is that the Directory Manager uses only 1% of the total CPU time. In addition, it is responsible for only 2% of the total messages sent by the file system and less than 1% of the total bytes transferred. This result seems reasonable given that the user test programs performed many more reads and writes than opens and closes. Since the Directory Manager is an insubstantial part of the computation, it is reasonable to eliminate it from the parallelism analysis and to recompute **P** for the various configurations. This was done, and the values for **P** were very close to the values based on all four file system processes.

The main difference is that the parallelism factor is now computed from three processes instead of four. For example, the **P** of 1.21 we obtained for the file system would be 40% and not 30%. This is because the maximum attainable **P** is three and not four.

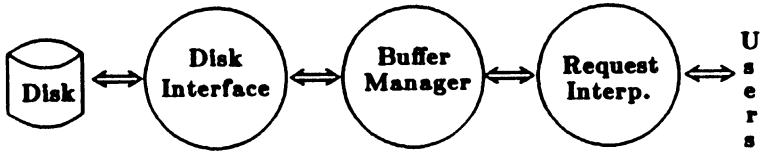

Figure 5.14: Major Message Paths Through the File System

Even without the Directory Manager in the computation, the 40% result is not as high as we would like. By looking at the amount of communications between the various parts of the file system (from Figure 5.12), we can get a better understanding of its structure. The largest volume of communications traffic proceeds from user processes, through the file system, to the disk (kernel) in a pipeline fashion. This is illustrated in Figure 5.14. With the pipeline arrangement, the amount of parallelism is determined by the number of file system requests that can simultaneously travel through the pipe to the disk (and back).

A set of measurements were made to see the effect that the number of processes simultaneously making requests had on the parallelism in the file system. File system activity was measured for the cases where one through seven processes were simultaneously making requests. The results are summarized in Figure 5.15. The file system reaches peak performance at four processes. If we could increase the amount of parallelism in the file system (that is, the number of processes active in the file system), the waiting time for each process could be reduced, and throughput increased. This statement is confirmed by the length of the message queue for the Request Interpreter process (plotted in Figure 5.15). The queue length stays constant up to four processes, then grows linearly with each additional user test process.

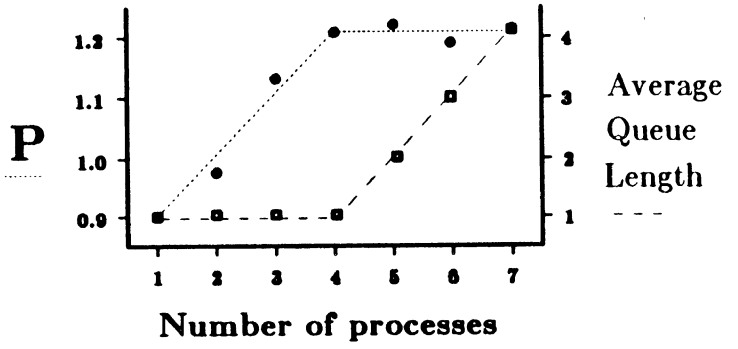

Figure 5.15: P and Queue Length vs. # of User Processes

An additional measurement was performed with the DEMOS/MP file system. This was to measure the accuracy of the parallelism factor, **P**, as a predictor of performance. When we measure a computation, traces are collected from processes that execute on particular machines. Parallelism analysis gives us the ability to compute the amount of parallelism in a program for some assignment of processes to machines. Usually, we start by assigning (in the analysis) the processes to the machines on which they actually ran. This gives us the **P** for the *measured* configuration. Using the same trace data, we can assign (in the analysis) processes to different machines and recompute **P**. This gives us the *predicted* value for **P** if we had actually run the processes in this new configuration. The question is how well do the predicted values of **P** match reality.

Data was collected for the file system running in each of three configurations (shown in Figure 5.16ABC). The measurements were made and the parallelism factor was calculated for each of the three configurations. Then the data for each configuration was used for two more analyses. For example, the data from the configuration in Figure 5.16A was analyzed (and the value of **P** obtained) for the machine assignment from the configurations in Figure 5.16B and C to obtain the predicted value for **P**. The same was done for the other two configurations. The result was nine values for **P** in a three by three matrix. Figure 5.17 shows the percent differences between the predicted values of **P** and the values calculated for the measured configuration. For example, data was collected for configuration A, and **P** was calculated from this data based on the measured configuration. **P** was also calculated from data collected for configurations B and C, but assigning (in the

analysis) the processes to machines as in configuration A. The results were differences of -3% and 2% between the measured and predicted values. Similar results were obtained for all the analyses. These variations show no excessive loss of accuracy due to analyzing the trace data in configurations different from that in which it was collected.

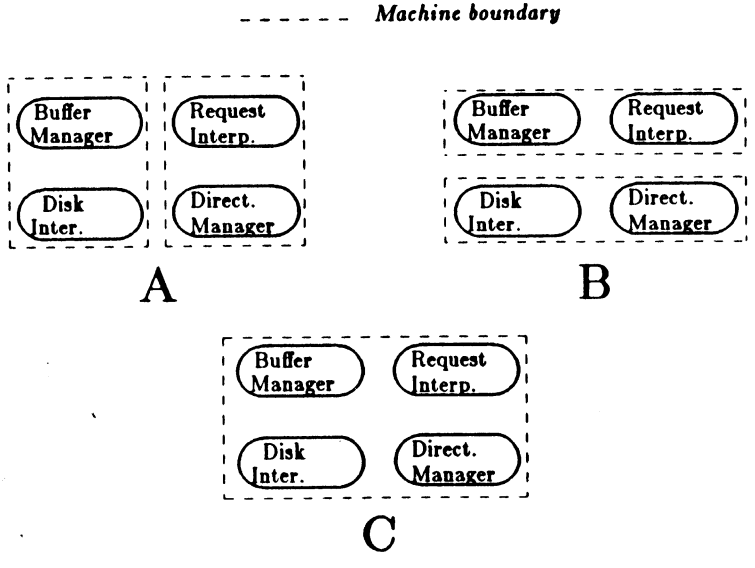

Figure 5.16: Three File System Configurations

		Predicted Configuration		
		A	B	C
Measured Configuration	A	-	-3%	2%
	B	4%	-	3%
	C	2%	-2%	-

Figure 5.17: Variation in P for the Three Configurations

5.3.2. More File System Results: Paths of Causality

We now take the analysis of the DEMOS file system one step further. Causality analysis provides us with a means to obtain a more detailed view of the way in which the processes interact within the file system. The analysis of causality is used to find the most commonly used message paths, and also to study the relationship between the processes in the file system. The measurements collected for the analysis in the previous section will be used for this analysis as well.

We obtain the causality strings from the DEMOS file system traces, and from these strings we compute the probability of one process interacting with another process. These interactions are characterized by the branching probabilities described in section 5.2.4. To compute the branching probabilities, we must first record the number of messages sent between processes. Figure 5.18 contains tables showing the message counts for the DEMOS file system processes. For example, the first table (for the Disk Interface) says that there were 1086 occurrences of the Disk Interface receiving a message from the Buffer Manager and next sending a message back to the Buffer Manager.

Disk Interface:						
Receives from:	Then sends to:	Disk Interface	Buffer Mgr	Directory Mgr	Request Interp	Users
Disk Interface		0	0	0	0	0
Buffer Mgr		0	1086	0	0	0
Directory Mgr		0	0	0	0	0
Request Interp		0	0	0	0	0
Users		0	0	0	0	0

Buffer Manager:						
Receives from:	Then sends to:	Disk Interface	Buffer Mgr	Directory Mgr	Request Interp	Users
Disk Interface		162	0	0	849	0
Buffer Mgr		0	0	0	0	0
Directory Mgr		0	0	0	0	0
Request Interp		981	0	0	2553	0
Users		33	0	0	0	0

Directory Manager:						
Receives from:	Then sends to:	Disk Interface	Buffer Mgr	Directory Mgr	Request Interp	Users
Disk Interface		0	0	0	0	0
Buffer Mgr		0	0	0	0	0
Directory Mgr		0	0	0	0	0
Request Interp		0	0	0	21	0
Users		0	0	0	0	24

Request Interpreter:						
Receives from:	Then sends to:	Disk Interface	Buffer Mgr	Directory Mgr	Request Interp	Users
Disk Interface		0	0	0	0	0
Buffer Mgr		0	9	0	0	3390
Directory Mgr		0	27	12	0	0
Request Interp		0	0	0	0	0
Users		0	3390	0	0	0

Figure 5.18: Counts of Message Sequences – DEMOS File System

Given the message sequence counts in Figure 5.18, we can construct a diagram of the interactions between the file system processes (see Figure 5.19).

This diagram can be thought of as a state diagram, representing the state of a request in the file system. The transition arcs represent message deliveries, and the states represent the process currently performing the request. Examination of the graph in Figure 5.19 shows that the most likely path to be followed is the same as the path shown in Figure 5.14.

Figure 5.19 provides additional explanation of the value of **P** obtained for the file system measurements. In the previous section we noted that the file system consisted of three active processes, with the Directory Manager contributing little to the overall computation. Figure 5.19 verifies this result. In addition, we see that when a request is made to the Buffer Manager, only 27% of the time does it result in a request to the Disk Interface. The remainder of the time (probably because of the data already being available), a response is sent directly back to the Request Interpreter. The branching probability from the Buffer Manager gives us a measure of the success of the buffering strategies used by the file system. This also shows that most requests involve only two file system processes. The measured value for **P** of 1.2 for the file system seems more reasonable with this additional information.

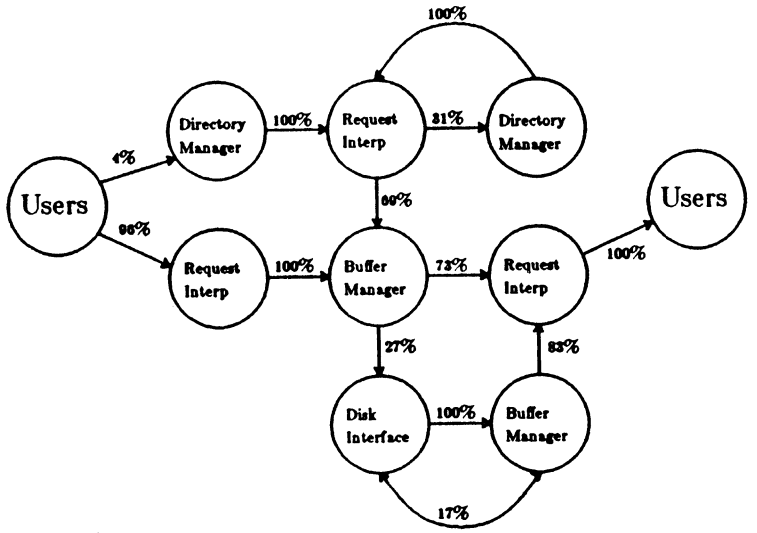

Figure 5.19: File System State Diagram

5.3.3. TSP: A Seminumerical Computation

One use of distributed computation is to achieve a performance increase in numerical or seminumerical computation. This analysis is intended to evaluate how well a particular algorithm performs at varying levels of concurrency. The Traveling Salesman Problem (TSP)[Christofides 79] is a well studied problem and can be decomposed into parts that can be computed concurrently.

Briefly described, TSP is the problem of finding the minimum cost of visiting each of N cities exactly once, given the costs of traveling from one city to another. If we consider the cities to be the n vertices, V, in a graph, and the paths between cities (with associated travel costs) to be the m edges, E, then the problem is described by the graph $G = (V, E)$. For a complete graph, $m = (n-1)^2$. The solution to TSP is the Hamiltonian circuit with the minimum cost.

TSP has been the object of other measurement studies for distributed systems[Mohan 82]. Mohan measured the performance of TSP on the Cm* multiprocessor system[Swan 78]. Cm* is a shared memory multiprocessor, using a hierarchical switch to provide access to the entire memory space for all processors. Mohan measured the amount of concurrency achieved as a function of the number of component processes (and machines) used in the computation.

We will measure a TSP program running on 4.2BSD UNIX operating system. The computation will be structured as a collection of processes executing concurrently and communicating via messages. We will use the parallelism factor, **P**, to compute the upper-bound on parallelism, the amount of parallelism actually achieved, and the amount of parallelism for various assignments of processes to machines. The object of this study is to use our measurement system and analysis tools to evaluate the TSP computation. For the remainder of this section, we will describe the structure of the computation being measured, the measurement and analysis strategy, and the results of the measurement and analysis.

5.3.3.1. The TSP Implementation

The TSP implementation we are using is similar to the one used in [Mohan 82]. The minimum path is found by dividing the set of possible paths into smaller sets of paths. The path that appears to contain the best solution is picked from this smaller set of paths. If this choice proves not the be the best, then the algorithm back tracks and tries another choice. This procedure is repeated until a solution is obtained. The cost for a single choice within a set of paths can be computed independently. The computation is performed by p processes, where $p \leq N$. This means that up to p simultaneous computations can be performed.

If there is no backtracking in the computation (i.e., if our choice of the best solution was always correct), then the computation is first divided into $N-1$ pieces and the $N-1$ pieces are computed concurrently. The computation is then divided in $N-2$ pieces and the procedure is repeated. The entire operation continues until a complete path is found.

The main difference between our computation and the Cm* computation is that the Cm* implementation used shared memory. Our computation is structured into a *controller* process and up to p *server* processes. The controller is responsible for dividing the computation into pieces to be executed by the servers. The main data structure used by the TSP computation is an $N \times N$ matrix. When the computation starts, a copy of this matrix is sent to each server. After this initial distribution, the only communications between the controller and the servers are messages from the controller instructing the servers to compute some part of the problem (a subset of the matrix), and the messages from the servers to the controller indicating completion of their assigned computation.

5.3.3.2. Measurement and Analysis Strategy

We will vary the size of the problems (number of vertices in the graph, or equivalently, the size of the matrix) and the number of processes used to compute the solution. This provides a measure of the problem's size and of the amount of available computing power. The problem is computed for problem sizes from 8, 12, 16, and 20 vertices. For each size, we compute the solution using 4, 8, 12, and 16 server processes. The computation is run with message send and receive operations being metered.

As the matrix size increases, the amount of work done by each server for each request increases. This would lead to the expectation that effective parallelism would increase for larger matrices. The amount of parallelism should also increase as the number of server processes increases. This should not be expected to be a linear increase for a number of reasons. The first reason is that communication delays cause some serialization as the pieces are distributed to the servers. The second reason is that each successive step in the computation uses smaller matrices, and therefore a potentially smaller number of server processes. This effect should be especially apparent for solutions with p (processes) close to N.

5.3.3.3. Measurement and Analysis Results

The first result that we obtained from measuring our TSP implementation turns out not be a performance result but a debugging result. The TSP program was written and tested before being used in the experiments described in the previous section. When the program was tested, the solutions produced were correct. During the first measurement of the algorithm for parallelism, it was noticed that the controller process for the calculation created only a single server process to do the computation. The answers were correct, but were only being computed in a single process. The problem was fixed, hence, the controller process produced the correct number of processes, but, when measured, it still made use of a single server. This problem was also detected and fixed.

These two bugs were (without overstating) instantly apparent from the measurement traces from our system. While we have not directly addressed the debugging aspects of the measurement tools in this thesis, results such as this one lead us to believe that the measurement tools will prove generally useful in distributed debugging.

The TSP program was run in the cases described above. Parallelism analysis was performed on the traces collected from the program executions. Three different **P** values were computed for each run of the TSP program. These are: (1) the theoretical maximum value, assuming no CPU contention and message delivery time of zero, (2) the value for one server process on each machine (n machines), and (3) the value for two server processes on each machine ($n/2$ machines). These values are summarized in Figure 5.20.

# of Processes		Matrix Size 4	8	12	16	20
4	max	1.4	2.1	2.4	2.5	2.6
	1/machine	0.3	0.8	1.3	1.6	1.9
	2/machine	0.3	0.7	1.0	1.2	1.3
8	max		2.7	3.6	3.8	3.9
	1/machine	———	1.1	2.0	2.6	3.0
	2/machine		1.0	1.6	1.9	2.1
12	max			3.7	4.1	4.5
	1/machine	———	———	2.2	3.0	3.6
	2/machine			1.8	2.3	2.7
16	max				4.4	5.0
	1/machine	———	———	———	3.2	4.1
	2/machine				2.5	3.1

Figure 5.20: Values of P for the TSP Program

Matrix size	Total CPU time (milliseconds)	% Due to controller
4	70	14%
8	1060	16%
12	10620	12%
16	80930	15%
20	19150	10%

Figure 5.21: CPU Times (in mS) for the TSP Program

The amount of parallelism increased as the size of the matrix increased (illustrated in Figure 5.22, Figure 5.23, and Figure 5.24). This is because the amount of computation done between messages increased with the size matrix. For the smaller matrices, the communication time dominated the time spent computing. Figure 5.21 shows the total amount of CPU time used by the TSP program for the different executions. From this table it is easy to see that computations that use small quantities of CPU should be run on a single machine. As the matrix size increases and the amount of CPU time needed for each server process becomes large (as compared with communications time), it becomes more important for each process to run on its own machine. By

comparing the graphs in Figure 5.23 and Figure 5.24, we can see the differences in **P** (between the two graphs) increase as the matrix size increases.

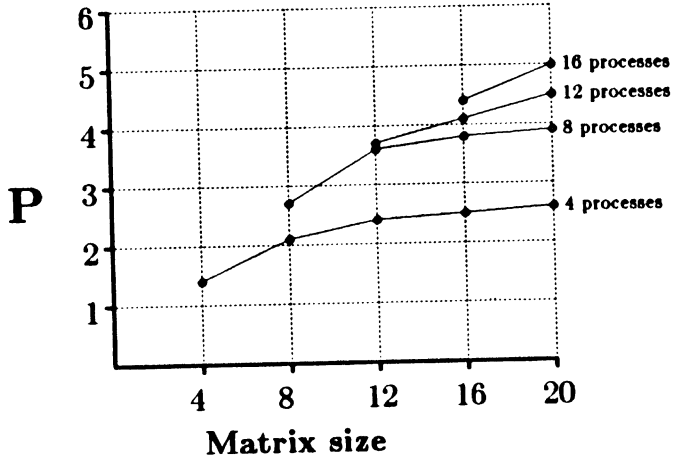

Figure 5.22: P for Maximum Parallelism (upper bound)

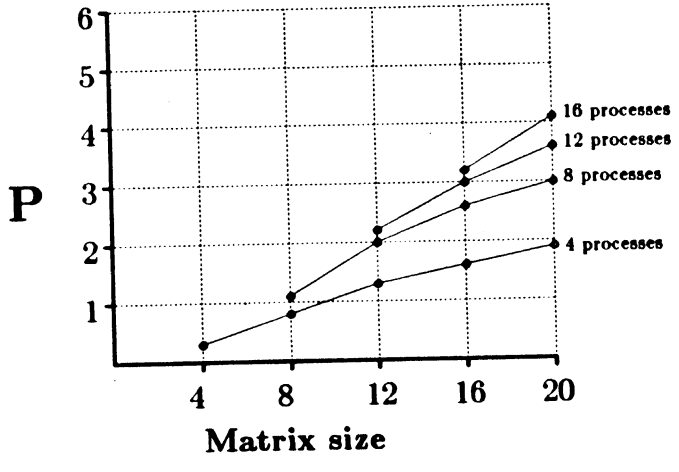

Figure 5.23: P for 1 Server per Machine

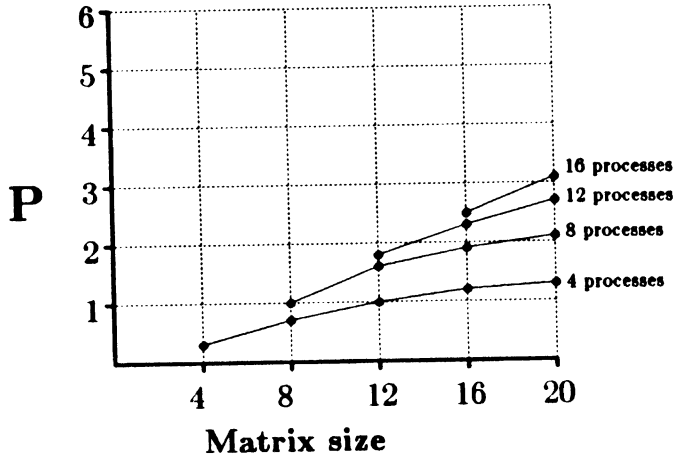

Figure 5.24: P for 2 Servers per Machine

The amount of parallelism also increased with the number of server processes (see Figure 5.25, Figure 5.26, and Figure 5.27). An optimal solution to this problem would achieve a linear increase in parallelism with respect to the number of machines. Our TSP implementation does not achieve a linear speed increase. One factor to which we can attribute this is the time needed to decompose the problem and make the branching decisions. By looking at the theoretical maximum **P** values, it is apparent that even with the cost of communications being zero, the computational cost of decomposing the problem causes periods of serial execution. We can see the same results in Figure 5.21. A significant percentage of time is being spent in the controller process. The large amount of time is caused by the TSP program recalculating values for branching decisions. It is possible for the program to retain a cache of previous calculated paths that appear to have smallest cost.

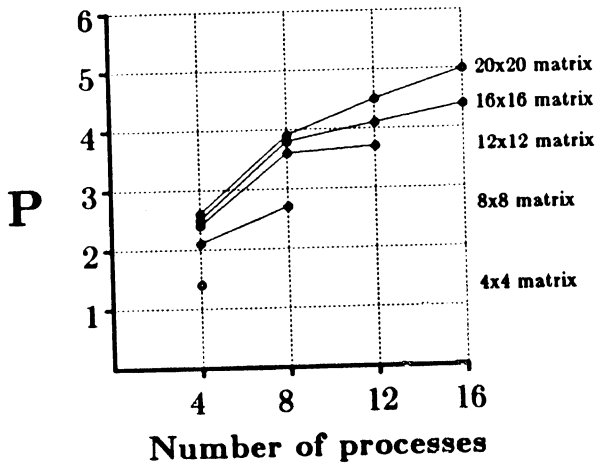

Figure 5.25: P for Maximum Parallelism (upper bound)

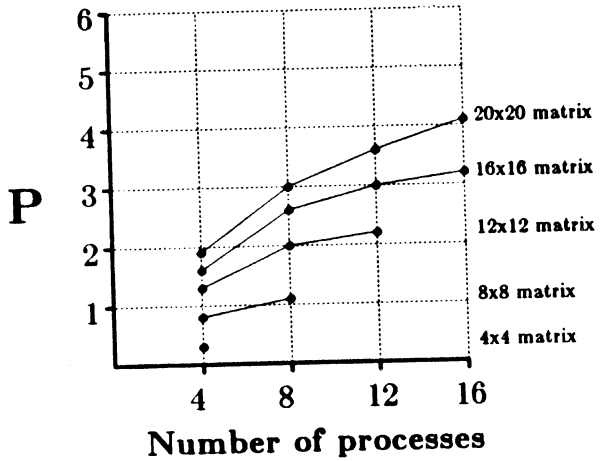

Figure 5.26: P for 1 Server per Machine

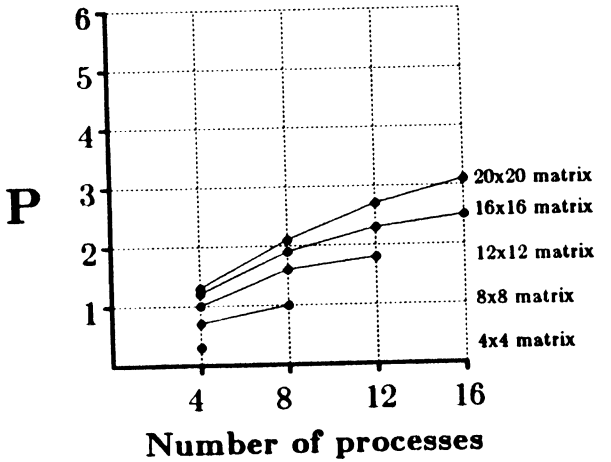

Figure 5.27: P for 2 Servers per Machine

The TSP program was modified to keep a cache of paths. We will refer to the modified program as *TSP version 2* (more simply, version 2). Using the version 2 of the program, the results were computed for matrices of size 16. The results are shown in Figure 5.28.

The new version of the TSP program was an improvement for two reasons. The first reason is the improvement in the amount of parallelism in the algorithm. We can see that the curves for the new version show a higher **P**. The second reason can be seen by looking at the slope of the curves for the new algorithm. The slopes for the new algorithm decrease less quickly than the curves for the original version. This implies that the new version of the TSP program will provide better speed improvements as more processes are added to the computation - though we still have not reached a linear increase in speed.

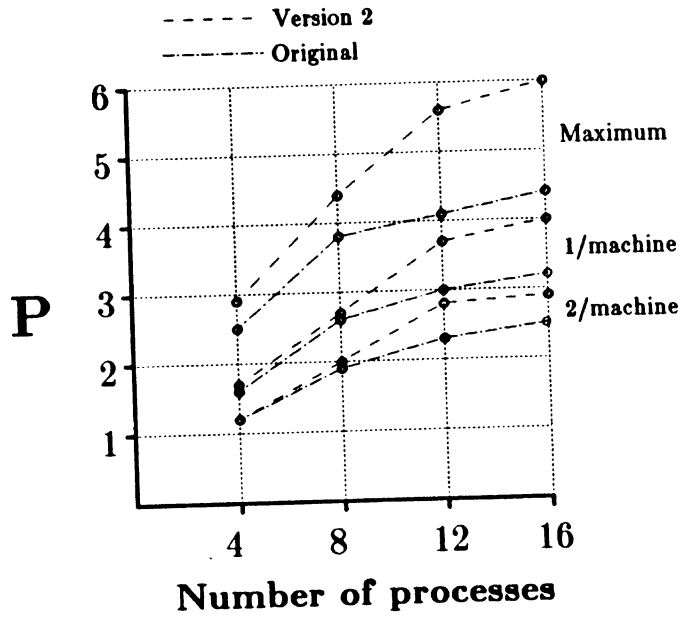

Figure 5.28: TSP: Original and Version 2 (16x16 Matrix)

The original version of the TSP program and version 2 of the program have a common problem. The problem is that the interactions between the controller and any individual server are synchronous. The controller sends a request to the server to be computed. When the server completes the computation, it returns the result to the controller and waits for the next request. The controller receives the response from the server and then computes the next request to send to the server.

We modified version 2 of the TSP program to reduce the amount of time that a server process spends waiting for requests. This is accomplished by always having a request waiting in the queue for each server process. Initially, each process is sent two requests, and as each result is returned to the controller process another is sent. This situation continues until a synchronization point is reached. The modified TSP program is referred to as *version 3*.

Using version 3 of the program, the results were again computed for matrices of size 16. The results for version 3, along with the original version

and version 2, are shown in Figure 5.29. The maximum possible **P** for versions 2 and 3 differ very little. This is because the analysis for the maximum **P** assumes that communication costs are zero, and the improvement for version 3 is based on communications delays. We see a more dramatic increase in the case where we assign one server process to each machine. The overlap of requests to each server provides a increase in the amount of parallelism in the computation. The increase is most pronounced for the smaller number of processes. This is because the measurements were performed for a problem of size 16. A solution for a problem of size 16 never divides the computation into more than 16 pieces at any stage. For a TSP program with four server processes, there is opportunity to overlap requests to each server process. When the number of server processes is increased to 16, requests cannot be overlapped to a server process (for the problem of size 16). We would expect to see the version 3 program perform better than the version 2 program for cases where $n \geq kp$, where n is the problem size, p is the number of server processes, k is an integer greater than one.

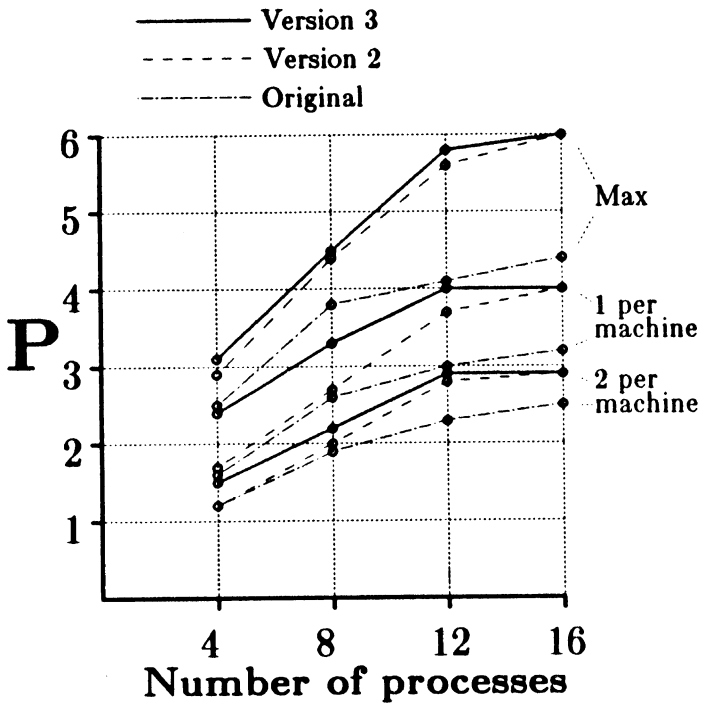

Figure 5.29: TSP: Original and Versions 2 & 3 (16x16 Matrix)

The measurement system and the analysis of parallelism provided the information needed to evaluate the performance of a parallel algorithm. We could easily detect such bottlenecks as the excessive waiting time for the server to process requests and the synchronous interactions between the controller and a server. These results led to substantial improvement in our parallel program. The same tools were also useful for debugging.

5.4. Summary

The analyses presented in this chapter show a wide variety of uses for our measurement system. Communications statistics analysis provides information that would typically be provided by one or more specialized measurement tools. Parallelism analysis provides a useful performance metric for concurrent programs. Our parallelism factor is unique in that it can be computed from data collected on a system that is being shared with other users. In addition,

parallelism analysis allows us to predict the behavior of a computation for different assignments of processes to machines. The paths of causality analysis provides us with a concrete picture of the structure of a complex program. We are able to construct the state diagram of a computation given only the meter traces. The current causality analysis uses paths of length three for its computation. It is possible to extend this analysis to provide a more detailed state diagram by using longer paths.

The true test of these analyses was in their application to computations. The various analyses, when used in concert to understand a computation, provided much useful information. The DEMOS file system provided a major test bed for the analyses. Our studies resulted in a much better understanding of the structure of the file system. The TSP program was written to evaluate the usefulness of the measurement system and of the analysis tools for programmers. The measurement system and analysis tools provided not only valuable performance data on the TSP program, but also provided valuable assistance to help debug and improve the algorithm.

Chapter 6

Conclusions

6.1. Overview

This research was intended to be a demonstration of measurement concepts and techniques. We started with a simple model of distributed computation. This model of distributed computation, in turn, allowed us to easily define a model of measurement. From this measurement model, we produced a measurement methodology which was then implemented as a measurement system in two host operating systems. Last, we developed a collection of analysis procedures to extract useful information from the data. The overall goal of this research was to show that, given our approach to measuring distributed computations, we could better understand these computations. We conclude this discussion by summarizing the high points of this thesis. These points fall into two areas. The first area is the set of concepts and world views that we have formed to allow us to develop our measurement system. The second area is a summation of results we have obtained from the measurement system.

The last section of this dissertation proposes areas for future research.

6.2. The High Points

There are several major concepts that have guided the direction of this research. In this section we briefly summarize each of these. As a result of this research, we have also developed a collection of measurement tools that produce useful results. We summarize the more interesting of these results.

6.2.1. World Views

6.2.1.1. Simple Model of Computation

The starting point of this research was a simple but general model of distributed computation. The model was based on processes that compute and communicate. The communications in this model did not use shared memory. Non-shared memory models apply to many contemporary operating systems. This implies that our measurement model and tools should be appropriate for

a wide variety of operating systems. As a demonstration of the concepts that we have presented, two separate implementations (DEMOS/MP and 4.2BSD UNIX) of the measurement tools were made. In neither case was an exceptional effort required to add the necessary functions to the host operating system. While our results are not conclusive, it appears that similar results should be attainable in other operating systems that fit our model.

6.2.1.2. Consistency

A large part of the success of the measurement tools that were developed in this research is owed to a consistent view of the programs, measurements, and analyses. Programs (or computations) were defined as a collection of communicating processes. The communication operations were described at the level seen by the programmer of the computation. The definition of the computation dictated what events were to be monitored. Given a reasonably small collection of communications primitives, the number of events to be measured is small. If we can monitor the communications events, then we can monitor all the interactions of a process with other processes (hence, we can monitor the entire computation).

The result of a consistent programmer's view, computation model, and measurement model is that the results derived from the measurement data are understandable and useful to the programmer. The results provided by the various analyses that we have done are directly applicable to the debugging, development, and maintenance of the program being measured.

There are two possible types of deviations from a consistent view. The first is to view a computation from a lower semantic level. An example of this can be seen in monitoring a message send. The programmer, typically, does not want to see this at the level of a transport protocol used on the network. For example, the concept of packets, retransmissions, and flow control are not interesting when observing the behavior of a computation; there is too much detail. A second deviation from a consistent view is adding semantic layers to the measurements. For instance, if a computation is described in terms of a language that defines correct execution behavior, we must first debug this description of the computation before using it to debug the computation.

6.2.1.3. Non-intrusive: performance

A goal for any measurement system is to minimize the effect of the measurements on the object being measured. This general goal produced, in our work, three more specific goals. These are to 1) minimize the performance effect of any measurement tools that we build; 2) not disturb the flow of

message communications; and 3) not disturb the flow of execution of the processes in a computation.

The first goal addresses the efficiency of our implementations. This is discussed in Section 4.3, so we summarize the main points. The first item was the overall design of the measurement system (inspired by the design of METRIC). By separating the data collection from selection, the amount of work done by a host kernel is reduced. All work different than collection can be done on other machines. The second item was the buffering of trace messages. This substantially reduces the number of message communications done by the host kernel. In our current implementations, this produced more than a 20:1 reduction in message operations. The third item was early filtering. Early filtering provides a simple and fast method of trace selection at the meter level.

The second and third goals for minimizing measurement effects should be discussed together. These goals reflect our view of distributed program development. There have been proposals for interactive debugging tools for for distributed programs[Philips 82,Schiffenbauer 81]. Those tools require substantial modification of the communications between processes to intercept and selectively stop messages. This can change the nature of the computation. Making this change invisible is very difficult, and the results have not been completely successful. Starting and stopping the processes within a computation has a similar problem. Stopping a process in a distributed computation can change the computation. Also, communicating the commands to the various machines that are executing a computation involves non-deterministic delays, making it difficult to have reproducible behavior.

The approach we have taken in this research for monitoring a process avoids the problems of changing the execution of the computation. In spite of our bias for performance results, we have been able to obtain reasonable debugging and performance information about the computation that we measure. The more general question of whether an interactive approach to understanding the execution of distributed computations will produce better results still remains to be answered.

6.2.1.4. Non-intrusive: transparency

Our goal was to have the programmer minimally affected by the measurement of the computation. It would significantly reduce the flexibility of the measurement tools if it had to be predetermined (before program execution) what type of measurements, if any, could be made.

The programmer should not have to design a program with the knowledge that the program will be measured. This means that the programmer should not have to insert explicit commands into the program to generate traces. It is not always possible to modify the programs being measured. The source code for the program may be unavailable, or it may be too expensive to modify and recompile the program. We want to avoid even re-compiling or -linking the program to add or change measurements. The result of making these assumptions is that existing programs can be measured without having to take the time to modify them for measurement (or change measurements). In general, a measurement tool should not require any knowledge about the internal structure of the programs.

These conditions resulted in a measurement system with a wide range of uses. Programs that were just being developed could use it, as well as programs that had been running for a long time.

6.2.2. Interesting Results

The value of our models and methods is shown in the results that we were able to obtain from our measurement tools. These results, taken as a collection (see Chapter 5 for a complete description), show a wide variety of interesting and practical uses for the tools.

6.2.2.1. Measure of Parallelism

Several metrics of parallelism in a computation have been proposed[Kuck 78,Chandy & Misra 78]. The unique characteristics of the parallelism factor, **P**, that we defined in Section 5.2.3, are that it can be derived from the execution trace of a program, and that, given the execution trace of a program, it can be computed for configurations other than that in which the program was executed. No knowledge about the structure of the computation is needed to compute **P**. The data needed to compute the parallelism factor can be collected during the normal execution of a computation. This is because the parallelism factor is based on the CPU execution time of each process in a computation; which makes the measure independent of the load on the machine that is running the process.

6.2.2.2. Feedback Scheduling

The measurement system that we have built shows its flexibility in the variety of its uses. An example of this is the use of the measurement system as a feedback scheduler for distributed system control (see Section 5.2.6). The measurement system provides data collection and selection. It also has

provisions for the analysis of the data as it is collected. If the process that is performing the data analysis is given the ability to control a portion of the host system, then we have formed a feedback loop. The most natural use of this structure is for those things directly involved with the distributed environment. This would include such things as a load balancing and process migration scheduler. Data could also be collected about the load (CPU time, request queue lengths, communications traffic) for distributed servers, to help decide how to allocate requests.

6.3. The Future

The research described in this thesis has only touched on a large field of study. There are many issues that need to be examined in greater depth, and many issues that have not been explored at all. Also, more experience is needed with the existing tools that we have built to evaluate them and determine future research directions. This section lists various issues whose study is suggested by this research. A significant question is what an ideal environment for developing and debugging distributed computations would look like. We speculate on this topic in the last section.

6.3.1. Things Still to be Done

To be completed, this research needs a larger number and variety of experiences in the areas it has addressed. We have a strong indication that directions we have chosen are reasonable. But it is only through more experience that we will be able to determine the most effective ways to use our tools for developing distributed programs. One great difficulty is the dearth of distributed programs. This is a chicken and egg problem: without good tools, few people tend to write distributed programs. Without a population of programmers for distributed programs, there is little demand for such tools. Our research should contribute towards the breaking of this cycle.

An important demonstration of the viability of our ideas was the implementation of the measurement system in two different host operating systems. This showed that the models of computation and measurement were not inexorably entwined in that of a particular operating system. This gives us hope that we have a general framework for developing and measuring distributed programs. A natural step would be to have the measurement tools implemented in other operating systems. Each new system would give us additional experience in developing distributed computations. Each new system would also add support for our model of distributed computation.

We have assumed, in the measurement tools that we have built, that all interactions between the elements of a computation are via messages. We have only briefly discussed, both in our model of machines and of processes, systems that use memory sharing. But there are several interesting systems that allow the sharing of memory between processes[Ousterhout 80, Jones *et al* 79, Cheriton *et al* 79]. The question is how to measure computation in this type of system. The answer lies in our basic model of computation. Processes compute and communicate. In this type of system, the processes communicate by using shared data.

Our measurement model says that the events we need to monitor are the interactions (or communications) between processes. We must assume that a mechanism is provided by the host operating system for synchronizing accesses to shared memory. This could done be with synchronization messages, semaphores[Dijkstra 72], monitors[Hoare 77], or other similar mechanisms. With an explicit synchronization of data accesses, we can monitor communications that occur through a shared memory. Instead of recording events such as message sends and receives, we might record semaphore P and V operation, or monitor entry, exit, blocking and unblocking from a condition. An unanswered question in this situation refers to the frequency of shared memory synchronizations compared to the frequency of message operations.

Dataflow-in-the-large systems use processes communicating via messages. The definition of a computation in such a system falls within our definition of a computation. It is reasonable to believe that the measurement tools we have defined and built would provide in those systems the same type of debugging and performance information that we provided in more traditional operating systems.

An area of active interest is that of performance modeling and simulation of distributed systems[Chandy & Misra 78, Lee 84]. Simulations need some form of work load specification. The measurement tools that we have built provide traces of the execution of distributed computations. The traces of distributed computations could be used to establish workloads for simulations of distributed systems, in a manner similar to the way address traces are used to drive demand paging simulations[Denning 70]. These simulations could be part of studies of load balancing for process scheduling or migration, or of communications scheduling policies.

6.3.2. Ideal Distributed Debugging

The final question we shall discuss is the more philosophical question of what is ideally needed to understand the execution of distributed programs. A

basic premise of our research is that we can obtain the information necessary to understand our computations by monitoring the interactions between the processes in the computations. The results we have obtained show that this strategy does provide enough information to give us a greater understanding of our computations. But we must ask "Is this enough?"

We might provide two additional facilities that could add to the information we can obtain about our programs. The first is the ability to view the contents of messages. We have assumed, in this dissertation, that knowing a message was sent, the identities of the processes involved, time, and other parameters was enough to understand a program's execution. How useful would it be to be able to know the contents of the messages? The second facility that we might provide would be the interactive control of the processes and messages in a computation. This is the ability to start and stop processes and messages during the execution of the computation. We have seen (in Chapter 2) examples where this has been tried, with only limited success. Is this a useful feature?

There are cases where knowledge about the contents of a message may help in debugging a distributed computation. For example, when debugging a concurrency control protocol in a distributed database, it would be useful to know whether the last message sent between the processes in the database system was a "commit" or "abort". Without examining the contents of the message, it might not be possible to tell what was happening that caused the problem that was being debugged. Often, knowing the order of message operations and the participants in the operations will provide enough information. The question then becomes: "How *often* would message content information help understanding a distributing computation?"

In our current measurement system, adding content information to the traces would be simple. The additional data to be sent through the network to the filter would cause extra overhead. Is this information valuable enough to offset the effects of performance degradation? (See [Presotto 83] for discussion of the performance problem in the context of fault recovery.)

Do we need the ability to start and stop processes and messages in a distributed computation? Such a mechanism would have to operate without causing changes in the results from the execution of the computation. Given this ability, is providing these functions useful? The complexity of the execution of a distributed computation argues against trying to control a computation at the level of individual interactions. The problem becomes one of how to present the state of each process in a computation, and the state of messages. Watching the flow of messages is a reasonable and understandable

concept. The harder question is how to present the state (at what level of detail) of the processes in the computation. It would be easy to quickly reach the point where there is too much information for the programmer to assimilate.

The models, methodologies, and tools that we developed for this research provide a step towards understanding the execution of distributed computations. More experience with the current tools is needed before we can address many of the questions we posed in this last section.

Appendix A

The Reweighting Algorithm

The reweighting algorithm is used by the parallelism analysis (section 5.2.3) to evaluate the execution delays caused by processes that execute on the same machine. The basic idea is that processes executing on the same machine will, at various time, compete for use of the CPU. This competition will cause each process to execute more slowly.

This appendix presents the code and data structures for the reweighting algorithm. The programming language used in the algorithm is Model[Morris 80].

There are two sets of basic data structures used by the reweighting algorithm. The first set of data structures are used to build the computation graph. Each process in the computation is described by a **Process** record. These are linked together in a list. Each event in the computation is stored in a **Node** record. The events for each process are linked in a list whose head is in the **Process** record. The links between events in a single process, and the links between corresponding send and receive events are described by a **Connection** record.

The second set of data structures is used for various bookkeeping activities during the reweighting of the graph. The **Send** records list the send events that have been seen, but whose corresponding receive events have not yet been seen. The **Delay** records keep track of the time delays associated with message delivery.

A last note: while these algorithms work, they are not as efficient as they might be. This lack of efficiency is apparent during very large data analyses. New versions of the algorithms are currently being tested.


```
$-------------------------------------------------------------------
$
$   Data structures describing analysis data structures.
$
$-------------------------------------------------------------------

$---Each process in computation has a "Process" record:

type Process is recursive record
{
    processID:      Integer;
    nodeList:       Node;           $ List of events for this process.
    machine:        Integer;        $ Current location of this process.

    base:           Node;           $ These two pointers are used to as
    curr:           Node;           $ ...place keepers for the reweighting.
    running:        Boolean;        $ T if process running, F if blocked.
    done:           Boolean;        $ Done w/analysis for this process.
    nextProcess:    Process;
};

$----Each event in the graph has a "Node" record:

type Node is recursive record
{
    eventType:      Integer;        $ SEND, RECEIVE, etc.
    time:           Integer;        $ Process time of event.
    connects:       array {Connection, [1...2]};
    newWeight:      Integer;        $ New value for reweighting arcs.
    marked:         Boolean;        $ T means have seen the send for this
                                    $ ...receive.
};

$----There is a "Connection" record for each arc in the graph.  Each Node
$    has a connection to the next node in a list for a Process.  Send event
$    Nodes also have a link to the corresponding Receive event Node.

type Connection is record
{
    weight:         Integer;        $ arc distance or weight (time)
    messageSize:    Integer;        $ # bytes in message.
    node:           Node;
};
```

```
$----List of pending Send events, for which we have not yet seen the
$    corresponding Receive event.

type Send is recursive record
{
    node:           Node;           $ Event node for pending send.
    nextSend:       Send;
};

$----Each "Delay" record is used to count down the time delay for a
$    message to be delivered.

type Delay is recursive record
{
    delay:          Integer;        $ Amount of delay.
    node:           Node;           $ Send event for this delay.
    nextDelay:      Delay;
};
```



```
$       +-------------------------------
$       |
$       |       ReweightInit
$       |
$       +-------------------------------
$

ReweightInit is beginproc

    Process proc;

    $----Initialization: set up pointers, and initial running procs:
    proc := processList;
    repeat while proc /= nil do
        proc.base := proc.nodeList;
        proc.curr := proc.nodeList;
        if proc.curr.eventType /= METERRECEIVE then
            proc.running := TRUE;
            IncrRunables (proc.machine);
        fi;
        proc := proc.nextProcess;
    od;

    delayList := nil;
endproc; $ ReweightInit
```

```
$       +---------------------------------
$       |
$       |       ReweightGraph
$       |
$       +---------------------------------
$

ReweightGraph is beginproc

    Integer decr;
    Process p;
    Delay d, lastd, saved;

    $----Initialize pointers and run counts.
    ReweightInit;

    $----repeat until everything stops
    repeat
        $----If any Send Delays have gone to zero, take them off the
        $    delay list, and marked the corresponding Receives.
        d := delayList;
        lastd := delayList;
        repeat until d = nil do
            if d.delay <= 0 then
                MarkReceive (d.node);
                if d = delayList then
                    delayList := d.nextDelay;
                    lastd := d.nextDelay;
                else
                    lastd.nextDelay := d.nextDelay;
                fi;
                saved := d;
                d := d.nextDelay;
                dispose (saved);
            else
                lastd := d;
                d := d.nextDelay;
            fi;
        od;
```



```
        $----For each new Send that has a non-zero message delivery time,
        $    make a Delay record.  Else, just mark the Receive.
        p := processList;
        repeat until p = nil do
            if (p.base = p.curr) and (p.base /= nil) then
                if p.curr.eventType = METERSEND then
                    if p.curr.connects[2].weight > 0 then
                        d := MakeDelay (p.curr, p.curr.connects[2].weight);
                        AddDelay (delayList, d);
                    else
                        MarkReceive (p.curr);
                    fi;
                fi;
            fi;
            p := p.nextProcess;
        od;

        $----For each Receive that we come across, if it is not marked,
        $    then indicate that the receiving process is blocked.  Else,
        $    just skip the event.
        p := processList;
        repeat until p = nil do
            if (p.base = p.curr) and (p.base /= nil) then
                if (p.curr.eventType = METERRECEIVE) and
                   p.curr.marked and
                   not p.running
                then
                    IncrRunables (p.machine);
                    p.running := TRUE;
                fi;
                if p.running then
                    p.curr := NextNode (p.curr);
                fi;
            fi;
            p := p.nextProcess;
        od;

until Completed do
```

```
    $----Figure out what the current time interval is, and increment
    $    all running (non-blocked) processes by this value times the
    $    number of running processes.
    decr := FindMinWeight;
    p := processList;
    repeat until p = nil do
        if p.running then
            p.base.connects[1].weight := p.base.connects[1].weight
                    - decr;
            p.base.newWeight := p.base.newWeight
                    + decr*runables[p.machine];
        fi;
        p := p.nextProcess;
    od;

    $----Decrement all the Delays by same time interval as above.
    d := delayList;
    repeat until d = nil do
        d.delay := d.delay - decr;
        d := d.nextDelay;
    od;

    $----Decrement the of runable procs for each blocked Receive.
    p := processList;
    repeat until p = nil do
        if p.running then
            if p.base.connects[1].weight = 0 then
                p.base := NextNode (p.base);
                if (if p.base /= nil then
                        (p.base.eventType = METERRECEIVE) and
                        (not p.base.marked)
                    else TRUE
                    fi)
                then
                    DecrRunables(p.machine);
                    p.running := FALSE;
                fi;
            fi;
        fi;
        p := p.nextProcess;
    od;
od;
endproc; $ ReweightGraph
```



```
$       +--------------------------------
$       |
$       |       MarkReceive
$       |
$       +--------------------------------
$
$----Mark the Receive event to indicate that the corresponding Send
$    event has been seen.
$
MarkReceive is beginproc
    formal
        Node sendNode;
    noresult;

    Node recv;

    recv := sendNode.connects[2].node;
    if recv /= nil then
        recv.marked := TRUE;
    fi;
endproc; $ MarkReceive
```

```
$       +--------------------------------
$       |
$       |       FindMinWeight
$       |
$       +--------------------------------
$
$----Look through not only the current nodes for the processes, but
$       for the delay times on messages.
$

FindMinWeight is beginproc
    result Integer;

    Process p;
    Delay d;
    Integer minWeight;

    minWeight := 2^30;
    p := processList;
    repeat until p = nil do
        if p.running then
            if p.base.connects[1].weight < minWeight then
                minWeight := p.base.connects[1].weight;
            fi;
        fi;
        p := p.nextProcess;
    od;

    d := delayList;
    repeat until d = nil do
        if d.delay < minWeight then
            minWeight := d.delay;
        fi;
        d := d.nextDelay;
    od;
    return minWeight;
endproc; $ FindMinWeight
```



```
$       +--------------------------------
$       |
$       |       Completed
$       |
$       +--------------------------------
$
$----Done when all "base" pointers go to nil (return TRUE).

Completed is beginproc
    result Boolean;

    Process p;
    Boolean done;

    done := TRUE;
    p := processList;
    repeat while done and (p /= nil) do
        done := p.base = nil;
        p := p.nextProcess;
    od;
    return done;
endproc; $ Completed
```

BIBLIOGRAPHY

[SAS 82]
SAS Users Guide: Basics, SAS Institute, Inc., Cary, North Carolina (1982).

[Ada 83]
"Military Standard - Ada Programming Language," Gov't order no. ANSI/MIL-STD-1815A, American National Standards Institute, Washington, D.C. (January 22, 1983).

[Aho *et al* 74]
A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Menlo Park (1974).

[Almes *et al* 83]
G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review," Technical Report 83-10-05, University of Washington (October 1983).

[Baiardi *et al* 83]
F. Baiardi, N. de Francesco, E. Matteoli, S. Stefanini, and G. Vaglini, "Development of a Debugger for a Concurrent Language," *Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symp. on High-Level Debugging*, pp. 98-106 Pacific Grove, Calif., (March 1983).

[Ball *et al* 76]
E. Ball, J. Feldman, J. Low, R. Rashid, and P. Rovner, "RIG, Rochester's Intelligent Gateway: System Overview," *IEEE Trans. on Software Engineering* **SE-2**(4) pp. 321-328 (December 1976).

[Baskett *et al* 77]
F. Baskett, J. H. Howard, and J. T. Montague, "Task Communcations in DEMOS," *Proc. of the Sixth Symp. on Operating Sys. Principles*, Purdue, (November 1977).

[Bates & Wileden 82]
P. Bates and J. C. Wileden, "EDL: A Basis for Distributed System Debugging Tools," *Proc. of the 15th Hawaii Int'l Conf. on System Sciences*, pp. 86-93 (1982).

[Bates & Wileden 83]
P. Bates and J. C. Wileden, "An Approach to High-Level Debugging of Distributed Systems," *Proc. of the ACM SIGSOFT/SIGPLAN Symp. on High-Level Debugging*, pp. 23-32 Asilomar, Calif., (March 1983).

[Carter 81]
F. H. Carter, "The OSMOSIS Project: A Control System for the Dual Processor Architecture," M.S. Report, University of California, Berkeley (December 1981).

[Chandy & Misra 78]
K. M. Chandy and J. Misra, "Specification, Synthesis, Verification, and Performance Analysis of Distributed Programs; A Case Study: Distributed Simulation," Technical Report TR-86, University of Texas, Austin (November 1978).

[Cheriton *et al* 79]
D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager, "Thoth, A Portable Real-Time Operating System," *Comm. of the ACM* **22**(2) pp. 105-115 (February 1979).

[Cheriton & Zwaenepoel 83]
D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proc. of the Ninth Symp. on Operating Sys. Principles*, pp. 128-139 Bretton Woods, N.H., (October 1983).

[Christofides 79]
N. Christofides, "The Traveling Salesman Problem: The Development of a Distributed Computation," pp. 131-149 in *Combinatorial Optimization*, ed. Christofides, Miugozzi, Topf, and Saudi, Wiley (1979).

[Cooper 82]
E. C. Cooper, "Better Process Control for UNIX," , University of California (February 1982).

[DEC 81]
Digital Equipment Corporation, *VAX Architecture Handbook*. 1981.

[Denning 70]
P. J. Denning, "Virtual Memory," *Computing Surveys* **2**(3) pp. 153-189 (September 1970).

[Dijkstra 72]
E. W. Dijkstra, "Hierarchical Ordering of Sequencing Processes," pp. 72-93 in *Operating System Techniques*, ed. C.A.R. Hoare & R.H Perrot, Academic Press, New York (1972).

[Downey 82]
Peter J. Downey, "Stochastic Analysis of Concurrency," Technical Report TR 82-15, University of Arizona, Tucson (October 20, 1982).

[Ferrari *et al* 1983]
D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and Tuning of Computer Systems*, Prentice-Hall, Englewood Cliffs, NJ (1983).

[Fujimoto 83]
Richard M. Fujimoto, "SIMON: A Simulator of Multicomputer Networks," Technical Report UCB/CSD 83-140, University of California, Berkeley (September 1983).

[Garcia-Molina *et al* 81]
H. Garcia-Molina, F. Germano, and W. H. Kohler, "Debugging a Distributed System," EECS Technical Report #287, Princeton University (August 1981).

[Gertner 79]
I. Gertner, "Performance Evaluation of Communicating Processes," *Proc. of the Conf. on Simulation, Modeling, and Measurement*, pp. 241-248 Boulder, Colo., (August 1979).

[Gusella & Zatti 83]
Riccardo Gusella and Stefano Zatti, "TEMPO: A Network Time Controller for a Distributed Berkeley UNIX System," Technical Report UCB/CSD 83-163, University of California, Berkeley (December 1983).

[Hoare 77]
C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Comm. of the ACM* **17**(10) pp. 549-557 (October 1974).

[Hoare 78]
C. A. R. Hoare, "Communicating Sequential Processes," *Comm. of the ACM* **21**(8) pp. 666-677 (Auguest 1978).

[Jones *et al* 79]
A. K. Jones, R. J. Chansler, I. Durham, K. Schwans, and S. R. Vegdahl, "StarOS, a Multiprocessor Operating System for the Support of Task Forces," *Proc. of the 7th Symp. on Operating Sys. Principles*, Asilomar, Calif., (November 1979).

[Joy *et al* 83]
W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, "4.2BSD System Manual," Computer Systems Research Group Technical Report, University of California, Berkeley (July 1983).

[Kolovson 81]
C. P. Kolovson, "The OSMOSIS Project: An Input/Output Facility for DEMOS/MP," M.S. Report, University of California, Berkeley (October 1981).

[Kuck 78]
D. J. Kuck, *The Structure of Computers and Computations*, John Wiley & Sons, New York (1978).

[Lamport 78]
L. Lamport, "Time, clocks, and the ordering of events in distributed systems," *Comm. of the ACM* **21**(7) pp. 558-565 (July 1978).

[Lauer & Needham 79]
H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," *Operating Systems Review* **13**(2) pp. 3-19 (April 1979).

[Lee 84]
T. P. Lee, "Configuring Local Area Network-based Distributed Systems," Ph.D. Dissertation, University of California, Berkeley (June 1984).

[McDaniel 75]
G. McDaniel, "METRIC: a kernel instrumentation system for distributed environments," *Proc. of the Sixth Symp. on Operating Sys. Principles*, pp. 93-99 Purdue University, (November 1975).

[Metcalfe & Boggs 76]
R. M. Metcalfe and D. R. Boggs, "Ethernet: distributed packet switching for local computer networks," *Comm. of the ACM* **19**(7) pp. 395-404 (July 1976).

[Miller & Gusella 84]
B. P. Miller and R. Gusella, "UNIX Process Control in a Distributed Environment," in preparation ().

[Miller *et al* 84]
B. P. Miller, S. Sechrest, and C. Macrander, "The Metering Tools on 4.2BSD UNIX," Technical Report UCB/CSD, University of California, Berkeley (May 1984).

[Mohan 82]
J. Mohan, "A Study in Parallel Computation -- the Traveling Salesman Problem," Technical Report CMU-CS-82-136, Carnegie-Mellon University (August 1982).

[Morris 80]
J. B. Morris, *A Manual for the Model Programming Language*, Los Alamos Scientific Laboratory, Los Alamos, New Mexico (February 1980).

[Ousterhout 80]
J. K. Ousterhout, "Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa," Ph.D. Dissertation, Carnegie-Mellon University (April 1980).

[Padlipsky 82]
M. A. Padlipsky, "A Perspective on the ARPANET Reference Model," RFC 871, Mitre Corporation (September 1982).

[Philips 82]
D. Philips, "Black Flag," Technical Report, Carnegie-Mellon University (June 1982).

[Popek *et al* 81]
G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Theil, "LOCUS: A Network Transparent, High Reliability Distributed System," *Proc. of the Eighth Symp. on Operating Sys. Principles*, pp. 169-177 Asilomar, Calif., (December 1981).

[Powell 77]
M. L. Powell, "The DEMOS File System," *Proc. of the Sixth Symp. on Operating Sys. Principles*, pp. 33-42 Purdue University, (November 1977).

[Powell & Miller 83]
M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proc. of the Ninth Symp. on Operating Sys. Principles*, pp. 110-119 Bretton Woods, N.H., (October 1983).

[Presotto 83]
D. L. Presotto, "PUBLISHING: A Reliable Broadcast Communication Mechanism," Ph.D. Dissertation, University of California, Berkeley (December 1983).

[Rapoport 81]
M. A. Rapoport, "Porting DEMOS to the VAX and Beyond," M.S. Report, University of California, Berkeley (August 1980).

[Rashid & Robertson 81]
R. F. Rashid and G. G. Robertson, "Accent: A communication oriented network operating system kernel," *Proc. of the Eighth Symp. on Operating Sys. Principles'*, pp. 64-75 Asilomar, Calif., (December 1981).

[Reif 78]
J. H. Reif, "Analysis of Commucating Processes," Computer Science Technical Report TR30, University of Rochester (May 1978).

[Ritchie & Thompson 78]
D. M. Ritchie and K. Thompson, "UNIX Time-Sharing System," *Bell System Technical Journal* **57**(6) pp. 1905-1929 (1978).

[Schiffenbauer 81]
R. D. Schiffenbauer, "Interactive Debugging in a distributed computational environment," Technical Report MIT/LCS/TR-264, MIT (September 1981).

[Smith 81]
Edward T. Smith, "Debugging Techniques for Communicating, Loosely-Coupled Processes," Ph.D. Dissertation, Computer Science Technical Report TR-9, University of Rochester (December 1981).

[Snodgrass 82]
R. Snodgrass, "Monitoring Distributed Systems: A Relational Approach," Ph.D. Dissertation, Carnegie-Mellon University (December 1982).

[Swan 78]
R. J. Swan, "The Switching Structure and Addressing Architecture of an Extensible Multiprocessor, Cm*," Ph.D. Dissertation, Carnegie-Mellon University (August 1978).

[Swinehart 74]
D. C. Swinehart, "COPILOT: A Multiple Process Approach to Interactive Programming Systems," Ph.D. Dissertation, Stanford University (July 1974).

[Taylor & Osterweil 79]
R. N. Taylor and L. J. Osterweil, "Anomaly detection in concurrent software by static flow analysis," Technical Report CV-CS-152-79, Computer Science Dept., University Colorado, Boulder (April 1979).

[Vickers 76]
K. E. Vickers, "The Design and Implementation of a Multi-Process, Multi-Machine, Multi-Language, Debugger," Technical Report #27399, Tymshare Corp., Cupertino, California (January 1976).

[Walker *et al* 83]
B. Walker, G. Popek, R. English, C. Kline, and G. Theil, "The LOCUS Distributed Operating System," *Proc. of the Eighth Symp. on Operating Sys. Principles*, pp. 49-70 Bretton Woods, N.H., (October 1983).

[Wolf & Liu 78]
J. Wolf and M. Liu, "A Distributed Double-Loop Computer Network (DDLCN)," *Proc. Seventh Texas Conference on Computing Systems*, pp. 6.19-6.34 (1978).

[Wulf *et al* 81]
W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA/C.mmp: An Experimental Computer System*, McGraw-Hill, New York (1981).