Performance Monitoring in Multi-transputer Networks

by
Jie Cheng Jiang[1]

Technical Report 90-32
October 1990

Department of Computer Science
The University of British Columbia
Vancouver, B.C. V6T 1W5
Canada

[1]Currently with MPR Teltech Ltd., Burnaby, B.C., Canada. Email: jiang@handel.mpr.ca

Abstract

Parallel architectures, like the transputer-based multicomputer network, offer potentially enormous computational power at modest cost. However, writing programs on a multicomputer to exploit parallelism is very difficult due to the lack of tools to help users understand the run-time behavior of the parallel system and detect performance bottlenecks in their programs. This thesis examines the performance characteristics of parallel programs in a multicomputer network, and describes the design and implementation of a real-time performance monitoring tool on transputers.

We started with a simple graph theoretical model in which a parallel computation is represented as a weighted directed acyclic graph, called the *execution graph*. This model allows us to easily derive a variety of performance metrics for parallel programs, such as program execution time, speedup, efficiency, etc. From this model, we also developed a new analysis method called *weighted critical path analysis*(WCPA), which incorporates the notion of parallelism into critical path analysis and helps users identify the program activities which have the most impact on performance. Based on these ideas, the design of a real-time performance monitoring tool was proposed and implemented on a 74-node transputer-based multicomputer. Major problems in parallel and distributed monitoring addressed in this thesis are: global state and global clock, minimization of monitoring overhead, and the presentation of meaningful data. New techniques and novel approaches to these problems have been investigated and implemented in our tool. Lastly, benchmarks are used to measure the accuracy and the overhead of our monitoring tool. We also demonstrate how this tool was used to improve the performance of an actual parallel application by more than 50%.

Acknowledgement

First of all and above all, I would like to thank my supervisors Dr. Samuel Chanson and Dr. Alan Wagner for their patience, support and understanding. The advice I received from Sam has gone far beyond academic research. And Alan, who never runs out of creative ideas, always gave me friendly guidance when my work seemed to be at an impasse.

I would like to extend special thanks to my project partner Hilde Larsen for doing an excellent job in programming the graphical user interface. Also thanks to Ola Siksik for donating her programs as benchmarks for our monitoring tool. Many other people in the Department of Computer Science at UBC have also contributed to this thesis. Ming Lau, hardware technician of the department, built the hardware circuit to generate global interrupts in the transputer network. Don Acton helped me to set up the system. Norm Goldstein and H.V. Sreekantaswamy proofread the final draft of this thesis. I am also grateful to the people of the Trollius project, especially Greg Burns of Ohio State University and Jim Beers of Cornell Theory Center, for their help and suggestion in the installation and instrumentation of Trollius Operating System.

Lastly, I am indebted to my family for their constant support and encouragement. This thesis is dedicated to my grandparents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The transputer-based multicomputer network is a new and promising class of highly parallel computer system because it not only offers potentially enormous computational power at modest cost, but also serves as testbeds for research experiments in the field of parallel processing. The focus of this thesis is instrumentation, modelling and performance analysis of parallel programs in multicomputer networks. New instrumentation techniques are explored, and the design and implementation of a real-time performance monitoring tool is presented.

1.1 The Problems

Monitoring a computer system relies on dynamically extracting information about the execution of a program at run-time, storing it and presenting it to the user in a useful format. The information collected by the monitor depends on what the user wants to know about the behavior of his program. Two traditional areas of studying the execution of a program are *debugging* and *performance analysis* [Miller84]. Debugging is concerned with the correctness of a program, while performance analysis chiefly addresses the efficiency of the program. Performance analysis includes performance measurement and performance tuning. Performance tools are invaluable to the application programmer since they not only provide performance measurement results but also help users optimize the performance of the program. The underlying instrumentation mechanisms used in debugging and performance analysis are similar. The major distinction is that debugging can control the execution of the program, while a performance monitor simply observes rather than participates in the computation. For the purpose of measuring the efficiency of a program, monitoring a computation without attempting to control its execution offers the best opportunity to understand its behavior. The emphasis of this thesis is in the performance aspect of understanding the execution of a program, though the methods and tools we have developed are also useful in uncovering bugs in seemingly correct programs.

Performance monitoring in uniprocessor computer systems has been studied extensively over the past 20 years and is well-understood; however, research in developing methods and tools for monitoring, debugging, and measuring parallel systems lags behind the technological advances in parallel architectures, distributed operating systems and parallel programming languages. Uniprocessor instrumentation techniques do not generalize to a parallel and distributed environment. Multicomputer networks feature asynchronous concurrent activities, nondeterministic and nonreproducible behaviors caused by unpredictable communication delays, and the lack of central control and accurate global time [HaWy90]. All these complicate the task of measuring and monitoring programs in parallel and distributed systems.

1.1.1 Global State and Global Clock

A multicomputer network consists of a large number of computing nodes which run asynchronously and interact with one another by passing messages. In order to obtain precise global states of the entire system, the measurement tasks have to be performed simultaneously on different nodes. Results however must be collected at a central workstation for analysis and display. Unpredictable communication delays and the lack of a central control mechanism make it difficult to guarantee that the measurement of tasks are performed at the same time and the information collected from the different nodes reflects a consistent global view of the system. A related problem is the difficulty in obtaining global clock in the multicomputer network where each node has its own physical clock and the drift between them is unpredictable. In parallel monitoring, an accurate global clock is not only useful for ordering asynchronous events on different nodes but is essential for measuring the elapsed time of message transmission. The logical clock approach [Lamport78] has been widely used for ordering events in asynchronous environments. However, it is difficult to derive absolute elapsed time using logical clocks since the differences of logical timestamps are not comparable to each other. Therefore, the logical clock approach is inadequate and inefficient to measure the performance of parallel programs in multicomputer networks.

The solution used in this thesis is a global interrupt approach in which a master node interrupts all other nodes in the multicomputer networks to perform the measurement tasks almost simultaneously, giving us an accurate snapshot of the system. Only minimal hardware support was needed to implement this scheme on the transputer network and it can be easily extended to other closely coupled multicomputer architectures.

1.1.2 Nonintrusive Monitoring

One of the most desirable properties of any monitoring tool is that it should incur minimal overhead and cause minimum interference to a monitored application. In parallel systems, stopping or slowing down a process may alter the behavior of the entire system and even produce different results. Unlike monitoring a centralized system, the presence of the monitor in a multicomputer network may not only cause severe degradation in the performance of the monitored application, but also distort the execution of the program yielding invalid results. It is impossible for any software monitoring tools to be totally nonintrusive since the monitoring software has to share the system resources with the application. Hardware monitors can be designed to have little or no effect on the host system, but they only provide limited, low level information about the activities of the host system. It is also difficult to map low level events to the source level program. The installation of extra hardware device requires skill and thorough knowledge of the host system, and can affect the hardware design and its expected performance. In addition, hardware instrumentation is expensive and impractical in most cases. On the contrary, software monitors can present information in an application-oriented manner and are easy to install. But if the performance results are to accurately reflect the behavior of the *unmonitored* application, the monitoring overhead must be within an acceptable range. [Reed89] suggests that a less than 15% performance penalty is acceptable for a software monitor.

The overhead introduced by a software monitor comes from the following sources:

- CPU time to run the monitoring software;
- memory space to store the monitoring data;
- communication bandwidth to report monitoring results to the host;
- extra context switches between monitoring processes and user processes.

In a multicomputer network where local memory available on each node is very limited, it is impossible to store all information collected by the monitor locally until the application computation terminates. Experimental results show that when the frequency of reporting is high, up to 80% of the slowdown of the monitored application is attributable to the communication overhead of the parallel monitor. Therefore, an important issue is how to minimize the interference of the monitoring messages to normal communications of the application program. Most existing systems fail to address this problem. See Chapter 2.

The approach investigated in this thesis is an adaptive reporting scheme in which the monitor tries to avoid jamming the network traffic by sending out monitoring data only when the network is lightly loaded. A pre-defined threshold function based on empirical data is used to determine whether the node is currently overloaded and whether the monitoring data should be sent. Limitations of this approach are also discussed in Chapter 4.

1.1.3 Automatic Performance Tuning

A performance tool is useful only if it can help to tune the performance of an application. It is a matter of how to present the performance data collected by the monitor to the user. Since the amount of trace data collected from all nodes in a multicomputer network is very large, it is important to present the information in a meaningful format so that the user will not be overwhelmed. Ideally the performance tool should supply users with solutions to a performance problem rather than statistical numbers. It is necessary to define a few simple metrics which can characterize the performance of the parallel program. Unfortunately, there is no generally agreed upon model for parallel computation, nor a model for the performance of these systems. Uniprocessor analysis techniques cannot handle the drastically increased number of parameters in parallel systems. New methods for analyzing the performance of parallel programs are at best underdeveloped. Most existing systems only supply users with statistical summaries of the execution of their programs.

In this thesis, we have developed a new performance analysis method for the multicomputer network, called *weighted critical path analysis*(WCPA). It is based on a simple parallel computation model in which the computation is formalized as an execution graph constructed using a minimal set of process events. Common performance metrics like program execution time, speedup, efficiency and granularity can be easily measured using the proposed method. By incorporating parallelism into the critical path analysis technique, WCPA helps users identify the program activities which have the most impact on the performance of their applications.

1.2 Motivation

The chief motivation of this thesis is the lack of tools to help users understand the run-time behavior of the parallel system and detect performance bottlenecks in their applications. Though progress has been made in developing parallel operating systems [Burns88] [Parasoft88] and parallel programming languages [Inmos83] [Zenith90] on transputers, most existing systems do not provide adequate support for users to measure and analyze the performance of their applications. It is not unusual for the application programmer to write special code and insert it into the application in order to obtain even the simplest time measurements of the program. It is almost impossible to trace the execution of a parallel program on the transputers by printing diagnosis messages from various places within the program, as most people usually do to their sequential programs. In a parallel system like the transputer-based multicomputer where most nodes in the network do not have direct access to external devices, diagnosis messages have to be routed through intermediate nodes to reach the host in order to appear on the user's terminal. Moreover, messages from different nodes will appear in some arbitrary order. Therefore it is highly desirable to provide support in the underlying operating system to capture these interesting events, collect and reorder them, and present to the user in a meaningful format.

Experimental results show that initial implementation of a parallel program typically yields disappointing performance [AnLa89]. The effort required to tune a parallel program, and the level of performance improvement that is eventually achieved depend heavily on the quality of the instrumentation that is available to the programmer. Since a parallel program typically consists of many components running concurrently on asynchronous nodes, and the interaction among different components of the parallel program can be quantitatively overwhelming and qualitatively complicated, it is difficult for the programmer to identify which part of the program contributes most to the performance of the entire program. It is desirable to provide analysis tools to appropriately direct the attention of the programmer by efficiently measuring those factors that characterize the performance of the entire program.

The successful development of performance monitoring tools relies on a good understanding of the performance characterization of the target system. Existing monitoring tools on transputers only provide simple statistical measures such as processor and link utilization on individual nodes during the execution of the whole program (see Section 2.2). There is a pressing need for new monitoring tools which can measure the overall performance of parallel applications and help users tune the performance of their programs. Previous work has concentrated on instrumentation techniques or implementation tricks on the transputer rather than performance modelling itself. We feel that to build effective performance tools on the transputer, the first step is to define a simple model which can capture the performance behavior of parallel programs on multicomputer networks. This model is described in the first part of this thesis (Chapter 3. The second part of the thesis is dedicated to the designing a parallel performance monitor on transputers based on the model we define.

1.3 Objectives and Goals

In designing a performance monitoring tool, the following are the primary goals we want to achieve:

- *Functionality*: The tool should provide users with enough information for performance studies of their program. In addition to measuring resource utilization in the system, it should have the ability to trace system and user-defined events.
- *Extensibility*: The instrumentation should not require substantial changes to the host system, both in hardware and software. Also, the monitoring system should be flexible and allow a wide range of user interfaces and analysis packages to be incorporated into the tool. This requires a separation of data collection and selection from data display and analysis and a well-defined interface between them.
- *Transparency*: the instrumentation should be transparent to the application programmer. The user should not be required to modify his program in order to monitor it. The only exception to this is the case of user-defined events, which may be application dependent.
- *Efficiency*: The overhead introduced by the monitor should be within an acceptable range.
- *Accuracy*: The performance results reported by the monitor should reflect the behavior of the unmonitored application. The behavior of the program should be the same when running with or without the monitor.
- *User-friendliness*: The monitor should be easy to use and the resulting data should be easy to read. A graphical interface is necessary to display the data in a user-friendly

manner. The monitoring tool should be flexible so that it can be turned *ON* and *OFF* interactively, either by the user from the host, or from within the program running on a multicomputer node.

There are other secondary goals. We would like the tools to be applicable to a wide range of systems rather than the instrumentation of a specific hardware architecture (the transputer) or a specific target operating system. The approaches suggested in this thesis should be generally applicable to other closely-coupled multicomputer architectures.

1.4 Thesis Outline

This section gives a brief description of the contents of the following chapters.

Chapter 2 is a literature survey of previous work in areas related to parallel and distributed monitoring. Key ideas which contributed to this thesis are identified.

Chapter 3 presents a performance model for parallel programs on the multicomputer network. We give a definition of a multicomputer network and then give a simple model of computation on the multicomputer network. Based on this computation model, we derive a set of performance metrics used to characterize the performance behavior of a parallel program. Finally, we propose a new method for measuring and analyzing the performance of parallel programs on the multicomputer network. Applicability and limitations of this method is also discussed in Chapter 3.

Chapter 4 describes the design and implementation of a parallel performance monitor on the transputers. It begins with a brief overview of the hardware and software instrumentation environment, followed by the description of the design of the monitoring system. Various techniques applied in the parallel monitor are described in detail, with new approaches to the problems discussed in section 1.1 and their implementation highlighted. The design of the graphical user interface is briefly described at the end of chapter 4.

Chapter 5 presents the testing and verification results. The accuracy of the resource utilization results measured by the monitor is validated by comparing against artificial load programs. Measurement of monitoring overhead is discussed, and a comparison is made between our clock synchronization technique with other reported software clock synchronization algorithms for transputers.

Chapter 6 shows an example of how the performance monitoring tool is used to tune the performance of a real parallel application. It demonstrates how it helps to discover a serious bug in a seemingly correct parallel program.

Chapter 7 concludes the thesis by summarizing key ideas presented in the previous chapters and suggests future enhancements of the monitoring tool.

Appendix A is a detailed description of the architecture of the transputer-based multicomputer network. Appendix B contains a list of changes made to the target software system, namely the *Trollius* Operating System. An up-to-date bibliography on parallel and distributed monitoring is included at the end of the thesis.

Chapter 2

Related Work

The problem of monitoring the execution of a program in a parallel and distributed system has attracted much attention among researchers in recent years. Prototypes of monitoring tools have been developed on a wide range of parallel architectures, with emphasis on either debugging or performance analysis [Joyce87]. These systems apply different techniques and achieve different degree of success in dealing with the problems presented in Section 1.1. In this chapter, we first make a general survey of tools developed in other distributed and parallel environment. Second we give a brief review of existing monitoring tools on transputers. Since the body of literature on parallel and distributed monitoring is large, we only present works that are of particular interest to performance studies and have had the most influence to the design of our tools. We also identify ideas that have contributed to this thesis and point out deficiencies in the model or design of existing systems.

2.1 Parallel Performance Monitoring

Among the existing tools to monitor the performance of distributed and parallel programs, the following systems have the most influence to the design of our tool.

IPS [MiYa87][Miller90] is a performance measurement system for parallel and distributed programs developed at the University of Wisconsin-Madison. IPS is based on the ideas proposed in Miller's Ph.D thesis [Miller84] and its predecessor DPM [Miller88]. IPS uses a hierarchical model as the framework for performance measurement. The behavior of a program is described at multiple levels of abstraction. *Program level* is the top level of the hierarchy and it describes the general behavior of the whole program, such as program execution time and speedup. The next level below is the *machine level*, which records summary information for each node and the interaction between them, such memory and CPU utilization of each machine. The *process level* ignores the machine boundary and views the distributed computation as a single group of communicating processes. At *procedure level*, a distributed program is represented as a collection of sequentially executed procedure call chain for each process. The lowest level of the hierarchy is the ***primitive activities level***, which is a collection of primitive activities that are detected to support upper level measurement. Performance metrics are defined for each level in the hierarchy and allow the the behavior of the program to be viewed at different level of detail. IPS applies different techniques to measure events at different level. Data for process, machine and program level are collected using event tracing, while data for procedure and primitive activities level are collected using periodic sampling. IPS is designed for loosely-coupled, message-based distributed environment and has been implemented under the Charlotte Distributed Operating System as well as the 4.3BSD Unix systems. The initial version of IPS [MiYa87] only supplies a simple textual user interface. The second generation of the tool, IPS-2 [Miller90], extends the old system with an interactive graphical user interface, which allows the programmer to display metric in tabular or graphical form and use the analysis tools interactively. IPS uses the instrumentation strategy of modifying the run-time library provided by the underlying operating system. Hooks are automatically inserted into the application by selecting a compiler option.

IPS provides automatic guidance techniques for performance tuning. The most important tool it provides is to find the path that consumes the most time through a graph of the program execution history, known as *critical path analysis*(CPA). In this thesis, we develop a new variation of this which we call weighted critical path analysis(WCPA). WCPA incorporates the notion of parallelism into CPA, in order to precisely reflect the relative importance of program elements to performance.(See Section 3.4) An analysis technique called *phase behavior analysis* which tries to automatically detect different phases in the parallel computation. is being investigated in IPS-2.

IPS does not address the problem of global state and global clock. It assumes that the clocks supplied by the underlying operating system are already synchronized among different machines. Also it does not address nonintrusive monitoring, especially the overhead of transferring large amount of trace data over the network. The overhead of IPS-2 [Miller90] ranges from 10-45%. Another disadvantage is that IPS is a post-mortem tool. Performance results cannot be viewed by the user in real-time, which makes it inappropriate for long computations.

Quartz [AnLa89], developed at the University of Washington, is a tool for tuning parallel program performance on a shared memory multiprocessor. The principle metric used by Quartz is the total processor time spent in each section of code along with the number of other processors that are concurrently busy when the section of code is being executed. When tied to the logical structure of the program, this correlation provides a "smoking gun" pointing at those areas of the program most likely responsible for poor performance. Quartz is implemented on the shared memory Sequent Symmetry Multiprocessor. Nonintrusiveness is achieved in Quartz by using a dedicated processor statistically checkpointing to shared memory the number of busy processors and the state of each processor. Each procedure in the application is assigned a weight as the total processor time of each procedure divided by the number of concurrently busy processors during the execution of the procedure. To focus the programmer's attention on the program segments that have the greatest impact on performance, Quartz presents a list of procedures sorted by its weight plus the weight of work done on its behalf. The WCPA method proposed in this thesis was inspired by Quartz. However, while Quartz incorporates the notion of parallelism into the sequential UNIX tool gprof, we incorporate the notion of parallelism into our own critical path analysis.

Another interesting tool is the TMP monitoring system developed by Haban and Wybranietz for the INCAS experimental multicomputer environment [HaWy90]. TMP is a hybrid monitor which is designed to benefit from the advantage of both hardware and software monitors while overcoming their deficiencies. A special hardware support, which consists of a test and measurement processor(TMP), is designed and attached to each node in the multicomputer. TMPs are used to collect and process event trace data generated by the instrumented application. All TMPs are connected via a separate network to a central station, thereby avoiding any interference of transferring trace data to the host system. Since monitoring data are collected, processed and transferred using extra hardware devices, the operations of TMPs are completely transparent. The overhead introduced by the monitor is minimal (less than 0.1%). Moreover, since events are generated by software, using the semantic information about the program structure provided by the compiler, the monitoring software is able to present data in an application-oriented manner. In TMP, probes to trigger events are placed in the operating system kernel so that it is not necessary to recompile the user's program. The probe routines write a trace entry to a special memory location which is then read by the TMP hardware. TMP also provides a graphical user interface to display performance results. Although TMP achieves a very attractive degree of transparency, the degree of hardware support it requires makes it expensive and unportable to most multicomputer systems. The global interrupt approach proposed in this thesis is partly inspired by TMP. We follow the principle of using minimal, affordable hardware support to achieve performance beyond the scope of any pure software monitoring tools. Several different approaches have been investigated in TMP to solve the problem of global state and global clock.

1. A kind of logical clock algorithm [Lamport78] has been implemented to preserve the causality relationship of events which occur on different nodes.

2. A software solution similar to the TEMPO algorithm [GuLa84] has been implemented to synchronize the clocks on different machines.

3. The TMP hardware offers the use of a central physical clock which triggers the local time counter on each TMP.

The current implementation of TMP only supports (1) and (2) and is able to synchronize the clocks in the order of $100\mu sec$.

In summary the major drawback of their system is the need for extensive hardware support and the lack of advanced tools for analyzing the performance data.

2.2 Transputer Monitoring Tools

The research and development of monitoring tools on transputers dates back to Capon and West's program transformation technique to monitor channel communications in Occam programs [CaWe88]. In their system, efforts are made to insert monitoring processes and additional communication channels between two communicating processes without changing the semantics of interprocess communication in Occam. It is a source level instrumentation technique and programmers are required to manually transform their program before they can be monitored. Recently Cai and Turner [CaTu89] extended this approach to monitor real-time Occam programs. The emphasis of their work is to use a logical clock to minimize the interference and achieve high transparency, in particular, to satisfy the real-time constraints in some applications. It is based on program transformation and requires manually modification of the original program. All of this work is specific to the Occam language. Neither system addresses the problem of global clock and reporting overhead.

A third transputer monitor is the one developed at Hong Kong University [HoLa89]. It measures the processor utilization and channel communications on an individual node. Three different methods are used to measure the utilization of each processor: periodic probing, idle counting and process profiling. Monitoring overhead is reduced by using assembly transputer instructions and careful code optimization. Their tool is rather simple in functionality. No advanced analysis is made of the data. Only statistical summaries are supplied by their tool.

The Victor project [Shea89] at IBM provides hardware support for nonintrusive monitoring in transputer-based multiprocessor. Monitoring is achieved with a separate hardware status bus which is independent of the regular transputer links and is connected to a dedicated PS/2 monitor system. In each node there is a scan register and a scan bus through the system that is controlled by the coprocessor adapter in the PS/2 coprocessor adapter for real-time acquisition of status data. The information collected for each node includes link activity, host id, memory activity, and state of user programmable LEDs. Although the Victor hardware monitor achieves a high degree of transparency, it has the same problem as most other pure hardware monitors. It can only be used to monitor low level activities of the system and is incapable of providing users with views of the system in an application oriented manner.

One recent work in transputer monitoring is GRAVIDAL [VoZe90], a graphical visualization environment for Occam programs on arbitrary transputer networks. It provides animated user defined views of the algorithm during run-time. The user has to manually place special statements into his source code and GRAVIDAL will generate visualized version of his algorithm. GRAVIDAL displays CPU load and link load as well as user-defined events on each node. A logical clock algorithm has been implemented in GRAVIDAL to order events on different nodes. GRAVIDAL does not provide an analysis tool for performance tuning since its emphasis is on graphical animation rather than performance studies of parallel programs.

Chapter 3

Performance Model

A parallel computation can be characterized by the way different components of the parallel program interact. There are two main streams in parallel processor design: shared memory architecture and distributed memory architecture. Processes in a shared memory system communicate via global shared variables, while processes on a distributed memory machine communicate by message passing. The multicomputer network is a class of distributed memory, MIMD parallel architecture. This chapter discusses the performance characterization of parallel programs on a multicomputer network.

3.1 Definition of Multicomputer Networks

A *multicomputer network* is a locally concentrated set of loosely coupled autonomous nodes interconnected in some topology, each with a microprocssor, local memory and hardware support for internode communication. Since hardware costs usually limit the number of connections on each node to a small number and the multicomputer network is only sparsely connected, messages must often be routed through a sequence of intermediate nodes to reach their destinations [ReFu87].

The multicomputer network has the following characteristics which distinguish itself from other parallel architectures:

- *Scalability*: Computing nodes can be easily added to a multicomputer network to obtain extra processing power. Multicomputer networks of a large number of nodes have shown to have very impressive peak performance.
- *Message-based communication*: Multicomputer nodes can only communicate via message passing over the interconnection network. This distinguishes it from tightly coupled shared memory architectures.

- *Geographical concentration*: Unlike loosely coupled systems which consists of nodes over a wide area, multicomputer nodes are usually packaged into a few boxes in the same room.
- *Communication locality*: In contrast to LAN-based environments where communication is unreliable and delays are measured in milliseconds, the communication in the multicomputer network is considered reliable and nearest neighbour communication is usually measured in microseconds.

Recent development in VLSI technology has paved the way for the development of multicomputer networks. General purpose building blocks have been proposed to simplify the multicomputer design and construction. The Inmos transputer is among the most successful in the commercial market [Inmos89]. The IMS T800 transputer is a single chip with a 32-bit processor, 4 Kbytes of on-chip memory, a floating point unit(FPU), four bidirectional bit-serial communication links, and a simple interface to memory and I/O devices. Both message passing and process scheduling are supported in hardware, yielding a highly efficient implementation. A multicomputer network can be easily constructed using IMS transputer boards. Appendix A of this thesis will contain more detailed information about the transputer architecture and construction of transputer-based multicomputer networks.

3.2 Graph Representation of Parallel Computation

A parallel program is composed of many concurrent processes running on asynchronous multicomputer nodes, interacting with one another by message passing. From the programmer's point of view, basic process activities include: process creation, process destruction and interprocess communications. The execution of a process can be viewed as a sequence of primitive process events. Interprocess communication can be synchronous or asynchronous. In this thesis, we mainly discuss a so called *semi-synchronous* interprocess communication paradigm which is supported by most operating systems on multicomputer networks. It is possible to extend this model to systems which support strictly synchronous and asynchronous interprocess communications. In the semi-asynchronous scheme, the sending process unblocks as soon as the message is sent, while the receiving process blocks until the expected message has arrived. Three types of primitive events are defined for interprocess communication activities: *message send*, *receive call* and *message arrive*. The process is suspended between a *receive call* event and the subsequent *message arrive* event.

Based on the previous discussion, a parallel computation on a multicomputer network can be formalized as a directed acyclic graph (DAG), called the *execution graph* $G =< V, E >$ where V is the set of nodes and E is the set of edges. A node in the graph represents a process event. It is one of the primitive events or a user-defined event. The following is a minimal set of primitive events for constructing the execution graph: process creation(*proc_init*), process destruction(*proc_exit*), message send(*msg_send*), receive call(*recv_call*), and message arrive(*msg_arr*). There are two types of edges in the graph, which defines a partial order over the set of all the nodes. A vertical edge represents the computation activities between two consecutive events of the same process. The direction of the edge represents the temporal ordering of the two events. A diagonal edge represents the communication between two processes. There is always an edge from a *msg_send* event to a corresponding *msg_arr* event in the graph. No edge exists between a *recv_call* event and the following *msg_arr* event because the process is suspended and there is no computation between these two events. The execution graph has the following properties:

- In a parallel computation with n processes, there are exactly n nodes with in degree zero, representing the incarnation of the processes.
- Each node in the graph has maximum in-degree 2.
- The maximum out-degree for each node is n if multicast is supported; 2 otherwise.

Each node in the execution graph can be tagged with the global timestamp of the corresponding process event. The elapsed time between any two events can be calculated by comparing the two timestamps. Figure 3.1 shows the execution graph of a parallel computation with and without multicast.

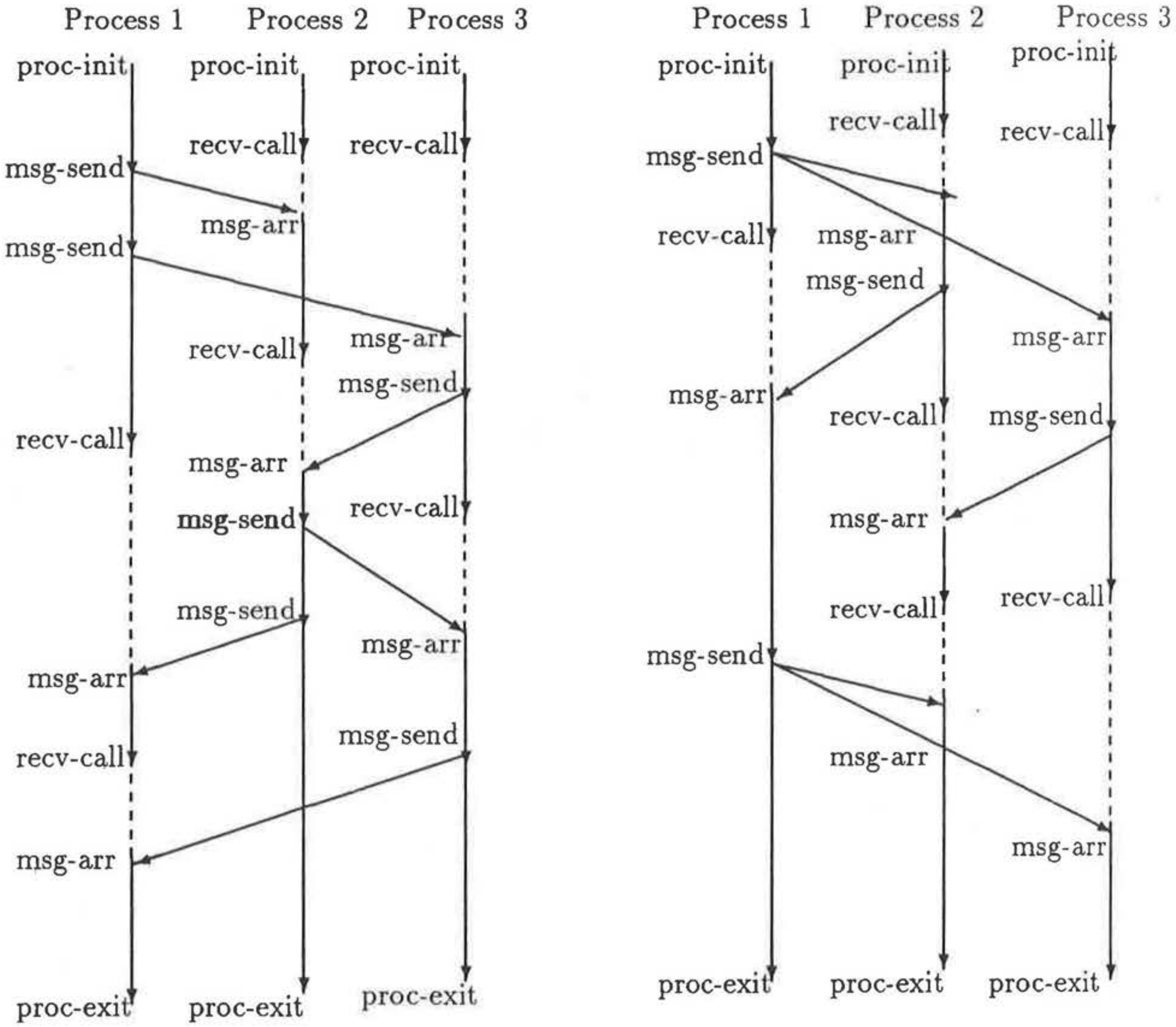
3.3 Performance Metrics

In this section, we derive performance metrics for this system based on the parallel computation model defined in the last section.

As in the performance analysis of sequential programs, the overall performance of a parallel program can be measured by the *program execution time*. We assign a weight to each edge in the execution graph equal to the elapsed time between its source event and its destination event. The program execution time is given by the length of the longest path of the execution graph. Figure 3.2(A) shows the weighted execution graph for a parallel computation with three processes on two processors. Processor 0 timeslices between the two processes. The longest path, or the *critical path* [YaMi88], is highlighted in the graph. The program execution time is the sum of the weights of all the edges on the longest path, i.e. $25+8+5+10+3+3+15 = 69$.

Two other important metrics for parallel programs are *speedup* and *efficiency*. Let $T(n,k)$ denotes the program execution time of a parallel computation with k processes on n processors, speedup is defined as $S(n,k) = T(1,k)/T(n,k)$ and efficiency is defined as $E(n,k) = S(n,k)/n$. Speedup is bounded by the number of processors, i.e. $S(n,k) \leq n$. In the execution graph, let C_i denotes the total amount of time process i spends in computation. Assuming that the same amount of work is done it follows that $\sum_{i=1}^{k} C_i = T(1,k)$. Substituting this in for $T(n,k)$ in $S(n,k)$, we obtain speedup as the ratio of total computation time to program execution time:

$$S(n,k) = \frac{\sum_{i=1}^{k} C_i}{T(n,k)}$$



(A) Parallel computation without multicast

(B) Parallel computation with multicast

Figure 3.1: An Example of Execution Graphs

. If there is no multitasking on the same processor, then C_i is the sum of the weights of all the vertical edges that belong to process i. In a parallel computation where some processors are timesliced among multiple processes, the calculation of C_i is more complicated. The execution graph has to be relabelled by assigning CPU time rather than elapsed time as the weight to the vertical edges in the graph. The *CPU time* of a vertical edge is the time the process is active computing between its source and destination event. Let P denote the CPU time and E denote the elapsed time between the two events. $P = E$ if there is no timeslicing. Let Δ_j be a time interval between the two events and the number of active processes on the processor during an interval of time m_j. Suppose there are l such intervals between the two events. The CPU time is:

$$P = E - \sum_{j=1}^{l} \frac{(m_j - 1)\Delta_j}{m_j}$$

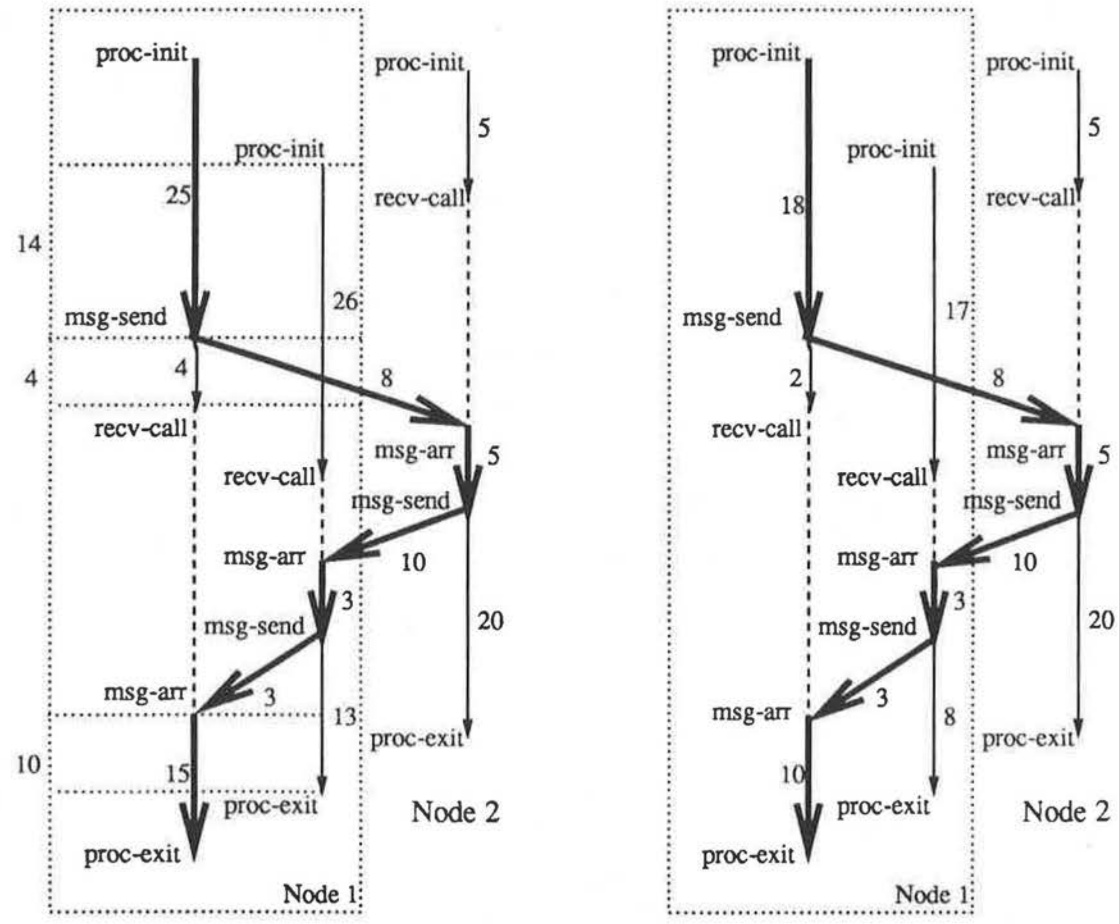
The total computation time for a process can be computed by the sum of the CPU time of all vertical edges that belong to that process. Figure 3.2(B) shows that the execution graph in (A) with its vertical edges relabelled by the processor time. The total computation time of the execution graph is: $\sum_{i=1}^{3} C_i = (18 + 2 + 10) + (17 + 3 + 8) + (5 + 5 + 20) = 85$. The speedup of the program on 2 processors is 1.23 and its efficiency is about 62%.

The *granularity* of a parallel program can be defined as the amount of time it spends in communication routines as compared to the total amount of computation. Let M denote the sum of the weights of all the diagonal edges in the execution graph. This is the total communication time of the program. Since the total computation time of the program is $\sum_{i=1}^{k} C_i$, the computation to communication ratio is $(\sum_{i=1}^{k} C_i) : M$. This ratio for the program in Figure 3.2 is 80 : 20.

In addition to the overall performance metrics for the whole program, we are also interested in the resource utilization on individual nodes over a given period of time. In a multicomputer network, the two most important resources are processors and communication channels. Given a time interval Δt, the degree of parallelism achieved in the system during Δt can be derived from the processor utilization of each individual node U_{cpu_i}. Given n processors, the *parallelism* of the system is calculated by: $P_{\Delta t} = (\sum_{i=1}^{n} U_{cpu_i})/n$. If $\Delta t = T$, then parallelism is equal to the efficiency of the parallel program, i.e. $P_{\Delta t} = E(n)$. Similarly, we can define *traffic load* of the network during Δt as: $L_{\Delta t} = (\sum_{i=1}^{m} U_{link_i})/m$ where U_{link_i} denotes the utilization of link i during Δt and m is the total number of links in the network.

All of the above metrics are defined at the program level. That is, they reflect the performance of the entire program. From the execution graph, it is also possible to derive performance metrics at the node level and process level, such as communication frequency between two nodes. These are simply statistical summaries and their calculation is straightforward.

3.4 Weighted Critical Path Analysis (WCPA)



(A) An execution graph with multitasking

(B) Relabelling the execution graph in (A)

Figure 3.2: Assigning Weight to An Extended Execution Graph

The performance measures described in Section 3.3 will supply users with answers to how efficient their programs run. It does not answer questions the efficiency of their programs or the locations of performance bottlenecks. The critical path analysis technique proposed in [YaMi88] tries to focus the user's attention to the sequence of program activities which take the longest time to execute. It is hoped that knowledge of the critical path of a program's execution helps the user identify performance problems and better understand the behavior of their program. While the critical path is useful in measuring the program execution time of a parallel program (Section 3.3), the question we would like to answer is: does the sequence of program activities that take the longest time to execute accurately reflect the activities which contribute most to the performance of the program, or are several parts of the program which take equal time to execute on one node equally important to the overall performance of the parallel program? A positive answer seems to be intuitive for those who are used to programming in a uniprocessor environment. However, in a parallel system, the degree of parallelism achieved has a dramatic impacts on the overall performance of the program. For instance, executing a segment of code on one node with all other nodes busy is not equivalent to executing for the same period of time with all other nodes idle. The latter indicates a potential sequential bottleneck in the parallel application and thus has a more significant effect on the performance of the program. Generally speaking there are two ways to deal with sequential bottlenecks in parallel programs. One is to re-structure the program to remove the sequential component. This requires substantial changes to the application and may not always be possible since many parallel applications have an inherently sequential component. If f is the fraction of computation which has to be executed sequentially, the upperbound for speedup on n processors is given by Amdahl's law [EaZaLa89]:

$$S(n,k) < \frac{1}{f + (1-f)/n}$$

Another approach is to optimize the code that has to be executed sequentially, thus reducing the fraction of sequential computation f. Therefore, it is essential to identify the sequential bottlenecks in the application. Consider the portion of a critical path shown in Figure 3.3. Suppose the number of processor is 100. A conventional critical path analysis tool would assign a weight,the elapsed time, to each edge. The elapsed time between event A and event B is 100 msec, while the elapsed time between B and C is 500 msec. It appears that the computation activities between event B and event C have a more significant effect on the performance of the program since they need longer time to execute. Since $P_{\Delta t} = 0$ between A and B, which means all other nodes are idle during that time interval, improving the execution time between A and B by 50% would reduce the program execution time by 50 msec. On the other hand, since all other nodes are busy between B and C, reducing the execution time between B and C has little or no effect on the performance of the program unless the execution time on all other nodes is also improved. The critical path of a parallel computation consists of a large number of events and it is difficult for the user to determine the relative importance of computation activities on the critical path to the performance of the program. The above example shows that the elapsed time alone is insufficient to capture the relative importance of concurrent program activities.

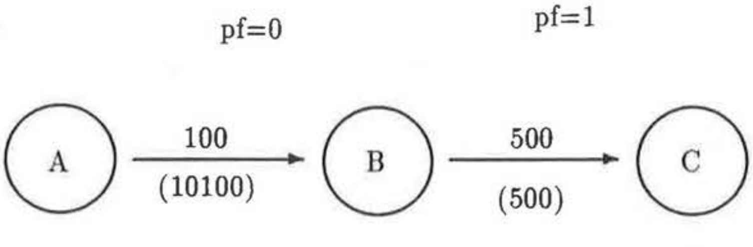


Figure 3.3: A Portion of the Critical Path

Based on the above observation, we present an analysis method, called *weighted critical path analysis* (WCPA), which incorporates the notion of parallelism into the critical path analysis. The purpose of WCPA is to the identify sequential components and activities with low degree of parallelism on the critical path. It is similar to the performance measurement technique used in Quartz [AnLa89] on shared memory machines(see Chapter 2). In WCPA, we apply the notion of parallelism to process activities on the critical path rather than to procedure activities in Quartz due to the unacceptable overhead of monitoring procedure level events in multicomputer networks. In the WCPA approach, an edge in the execution graph is weighted by two factors: the elapsed time between the two events and the degree of parallelism during that period. Let $P_{\Delta t}$ denote the parallel factor during time interval Δt where Δt is the elapsed time between the two events and n is the number of processors in the system, the weight assigned to the edge is computed by:

$$w = \Delta t + (1 - P_{\Delta t})(n - 1)\Delta t \quad (0 \leq P_{\Delta t} \leq 1)$$

When $P_{\Delta t} = 1$, i.e. maximum degree of parallelism is achieved, the weight assigned to an edge is equal to the elapsed time Δt; when $P_{\Delta t} = 0$, i.e. there is no parallelism in the system, the weight is maximized at $n\Delta t$. An interpretation of this is that when there is no parallelism, the execution on one node is wasting the resources on all other nodes, virtually consuming the resources of the entire system. Now, the longest weighted path in the execution graph represents the sequence of program activities which have the most significant effects on the overall performance of the parallel program. In Figure 3.3, the weight assigned to edges by the WCPA method is shown in brackets. Note that the weight for edge $< A, B >$ is now 10100. far more than the weight of edge $< B, C >$ 500. This correctly reflects our intuition about the relative importance of these program activities. A good metric to measure the relative importance would be the percentage of each portion on the weighted critical path out of the total weight of the whole path. The computation activities which weight the most on the critical path represents "the hottest of hot spots" in the program. Optimization of these components is expected to result in substantial improvement of the performance of the program.

3.5 Summary

In this chapter, we introduced a graph theoretical model of parallel computation on multicomputer networks, which we called an execution graph. We show that a sufficient set of five primitive events: *proc_init*, *proc_exit*, *msg_send*, *recv_call* and *msg_arr* are adequate to construct the execution graph for any parallel computation. Various performance metrics can be derived from the execution graph. Based on this model, we also developed a method to diagnose performance problems in parallel applications. This was based on the critical path analysis technique but incorporated the notion of parallelism in locating performance bottlenecks of the program. The method we proposed is shown to be able to reflect the relative importance of program activities to the overall performance more accurately than the conventional critical path analysis technique. The model and methods proposed in this chapter can be adapted to other message-based parallel and distributed environment with minor modifications.

Chapter 4

Design of the Parallel Monitor

This chapter describes the design of a parallel performance monitor and its implementation on transputers. In Chapter 1, we discussed the major issues in monitoring parallel and distributed systems and possible solutions to these problems. In this chapter the techniques and approaches used to overcome these problems are described in detail.

4.1 Environment

We begin with a brief description of the underlying instrumentation environment. One of our design goals is that the instrumentation should require minimal changes to the target hardware and software system.

4.1.1 Hardware Architecture

The parallel monitor is currently implemented on a 74-node transputer-based multicomputer in the Department of Computer Science at UBC. The multicomputer consists of a Sun 4 workstation as the host and 74 IMS T800 transputers, each containing 4 Kbytes on-chip RAM, 4 bidirectional serial links, and 1 Mbytes or 2 Mbytes local memory. The 74 transputer nodes are interconnected through 10 programmable crossbar switches. Detailed description of the hardware architecture of the T800 transputer and the C004 crossbar switches and their physical connection can be found in Appendix A of this thesis. The transputers in the network are connected to the host Sun workstation by a VME bus interface. There are currently seven connections between the host and the transputers, Nodes which do not have direct connection with the host can only communicate with the host through intermediate nodes.

The interconnection topology of the transputer network can be dynamically reconfigured by software running on the Sun which sends switch setting commands to the crossbar switches. Figure 4.1 shows a multicomputer network with 72 transputers configured as a 8x9 2-dimensional cylinder.

4.1.2 Underlying Operating System

The target software system is the *Trollius* Operating System [Burns88], a parallel operating system developed jointly at Cornell University and Ohio State University for distributed memory multicomputers and ported to the transputer-based multicomputer at UBC. Trollius provides a cross-development environment for parallel programming on transputers. It consists of two parts, one part which runs on the host and the second part running on transputer nodes. Trollius executes on top of UNIX on the host and provides a user command interface to boot the node, download programs to transputers, kill processes, etc. The most important tool provided by Trollius is message passing between processes. There are two levels of message passing in Trollius. The kernel level allows communication between processes on the same node: the network level allows communication between processes on different nodes, as well as on the same node. A Trollius process sending a message does not directly specify the process to receive the message, or vice versa. Instead, each process specifies an event type in the header of the message. If the event type specified by the sending and receiving process match, the message will be passed from the sender to the receiver. In network level message passing, the sender also has to specify the destination node of the message. Since the recipient of the message does not have to specify the source node, it can receive messages from a variety of senders. Trollius supports both an asynchronous and semi-synchronous interprocess communication paradigm as described in Chapter 3. Multicast facility is also supported in Trollius. Other tools include library routines for process creation, process destruction, signal handling, and access to remote file systems. For a detailed description of the Trollius Operating System, readers are referred to [Burns88].

4.2 System Structure

Figure 4.2 shows the basic structure of the parallel monitor. There are three major components: data generating and collection, global control, data analysis and display. One transputer in the network is distinguished as the master node. It is capable of interrupting all nodes in the system to perform measurement tasks simultaneously.

The monitoring software running on the master node includes an *interface* that accepts monitor command from the user, and a *controller* that generates global interrupt signals to synchronize the monitoring activities on all slave nodes. The data generation and collection mechanism include:

- Event *probes* inserted into the application running on slave nodes used to generate trace data, and a *meter* process to collect the event traces as they occur;

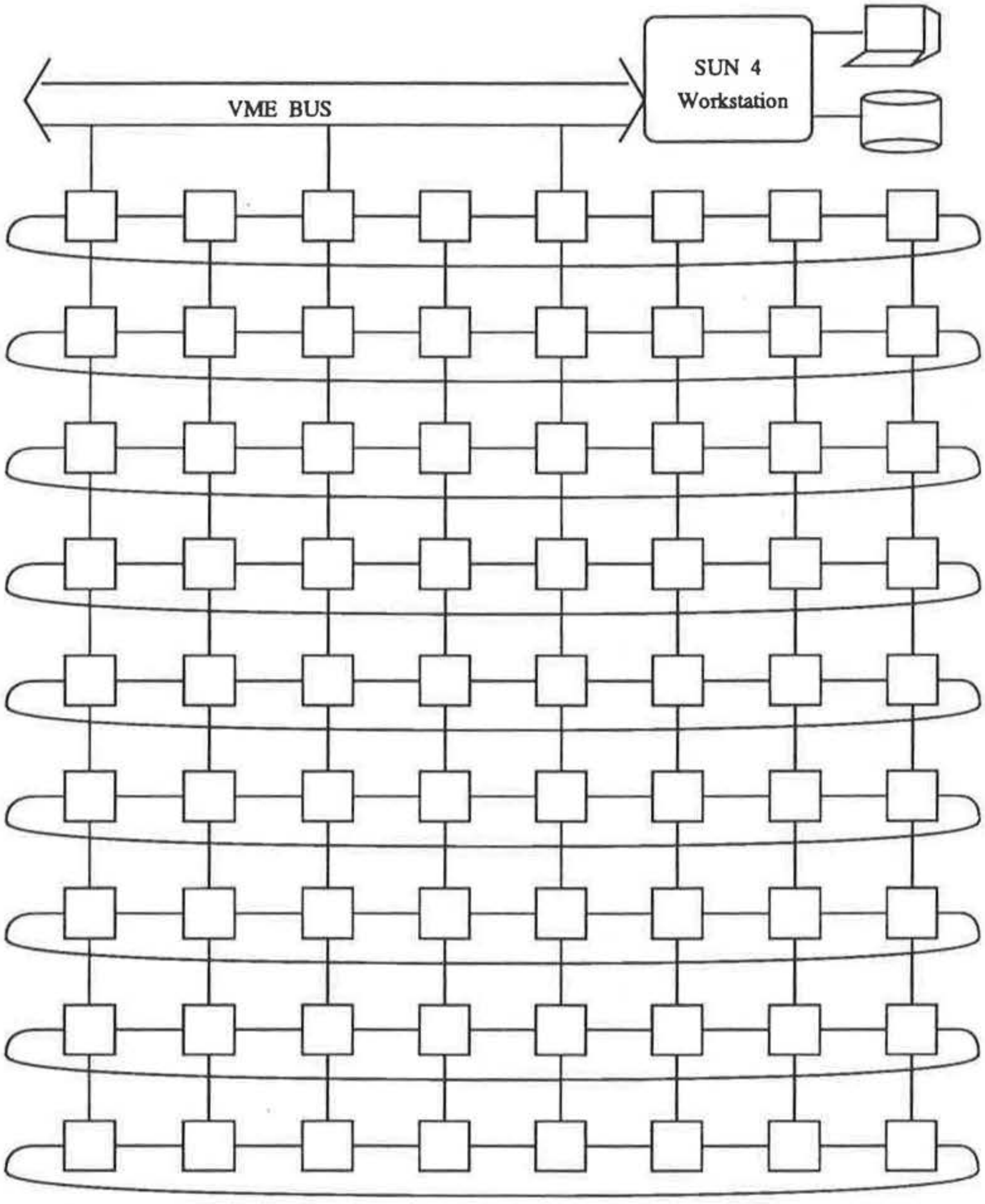


Figure 4.1: A Multi-Transputer Network Configured as a 8x9 2-D Cylinder

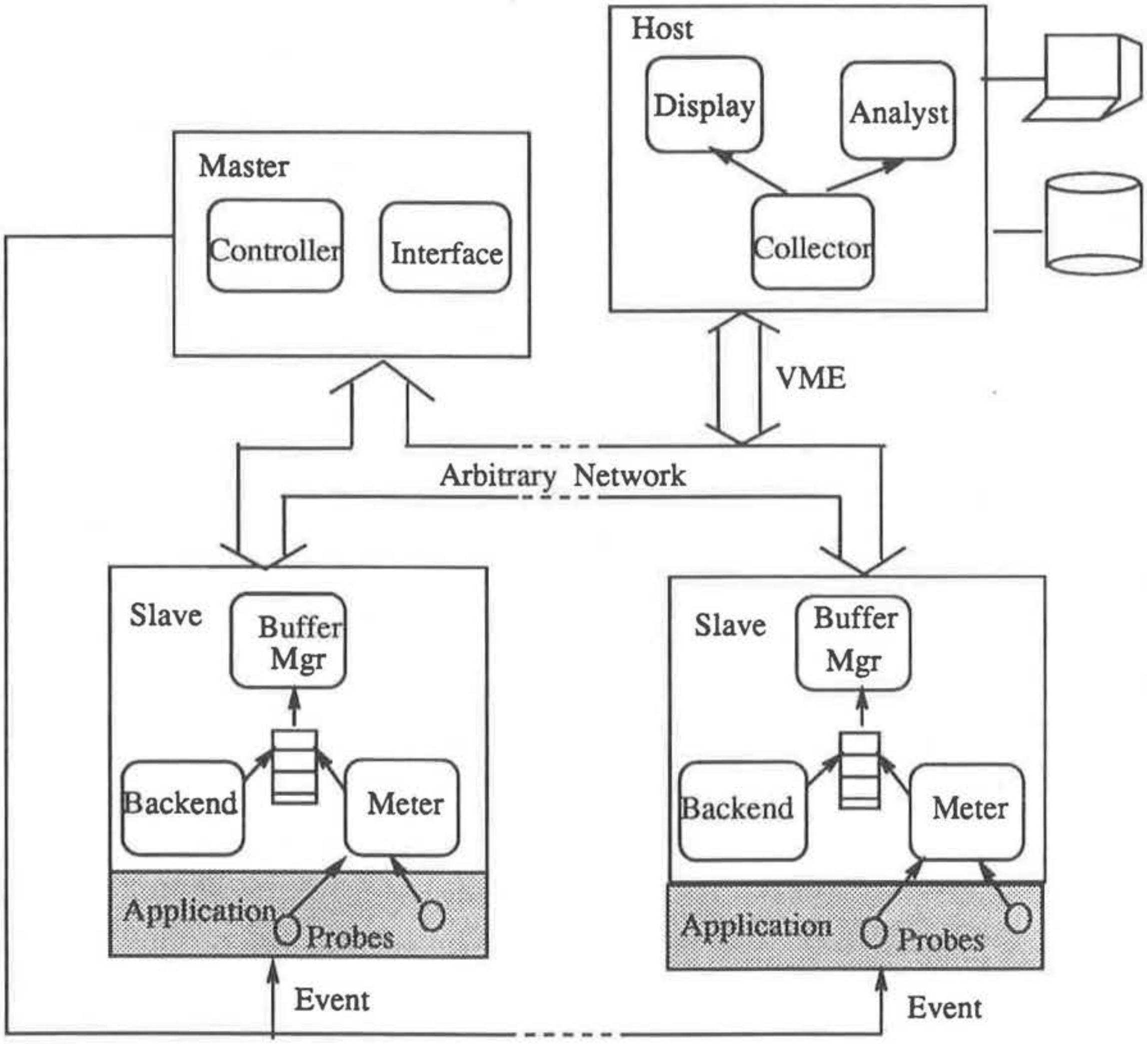


Figure 4.2: Basic Structure of the Parallel Monitor

- A *backend* process on each slave node that performs sampling and clock synchronization on the arrival of global interrupt signals.
- A *buffer pool* on each slave node to store the trace data and cache intermediate results from the meter process and the backend process. The *buffer manager* flushes the buffers when they are full to make room for new trace entries. The trace data are sent back to the host for analysis using the message passing mechanism provided by the underlying operating system.
- The *collector* on the host collects trace data from all of the slave nodes.

The host collector sends the data to the *data display* which displays the performance results graphically to the user in real time on the frontend host station. The data are also dumped to trace files for input to the *data analysis* packages.

4.3 Basic Instrumentation Techniques

There are two traditional ways of monitoring a computer system: event sampling and event tracing.

4.3.1 Event Sampling

Event sampling is a statistical approach to obtain an accurate estimation of the behavior of the computer system. The measurement task is performed at a pre-specified time interval for a long period of time. The main advantage of event sampling is that the amount of data it generates is small as compared to other approaches. This both reduces the monitoring overhead and simplifies the analysis.

In order for the data collected by sampling to be representative, sample size should be large and the sampling interval should be short so that the distribution of workload is homogeneous [Chan87]. Event sampling has proved to be the most economical and effective way of measuring resource utilization of the system. In a multicomputer network, sampling can be used to measure the utilization of processor and communication channels on each node with minimal overhead. However, unlike event sampling in uniprocessor systems, the sampling activities on different nodes must be coordinated to obtain results that reflect a consistent global view of the entire system.

4.3.2 Event Tracing

Unlike event sampling, event tracing measures events as they occur. Special software probes are inserted into strategic locations in the application programs or in the operating system kernel to trigger the recording of interesting events. Event traces are captured, buffered, and analyzed for display to the user. A major drawback with event tracing is that it is expensive when the frequency of the occurrence of the events to be traced is high.

In multicomputer networks, the volume of events generated on all node during a parallel computation can be enormous, however, the buffer space available on each node is very limited and the cost of transferring large amount of data across the network is extremely high. Therefore, event tracing is only suitable for measuring high-level events in systems with these characteristics.

Another problem with event tracing in a parallel system is that though events that occur on the same node can be totally ordered, events from different nodes may arrive at the host in unpredictable order. A single clock is needed to re-order these asynchronous events on the host. This requires the local clocks on different node be synchronized.

4.3.3 Hybrid Monitor

The parallel monitor we designed was a combination of the sampling and event tracing. It uses sampling to measure the resource utilization on each node, but uses event tracing to monitor the process events defined in Chapter 3. The process events collected are used by the analysis tool to reconstruct the complete execution history of the parallel program and to provide insight into the run-time behavior of the program.

4.4 Global Control

4.4.1 The Global Interrupt Approach

In order to obtain precise global state and synchronized global clock in the multicomputer network, we used a global interrupt approach, in which a master node interrupts all other nodes in the system to perform the measurement tasks almost simultaneously. A basic assumption is that the time required to respond to a global interrupt signal is negligible. The global interrupt approach can be used to start or stop a computation on all the nodes in the system. By generating periodic global interrupt signal, measurement tasks can be performed at some predefined time interval on system-wide basis.

It is generally not always feasible to implement the global interrupt scheme in a loosely coupled distributed systems. However, the multicomputer network features geographical concentration and communication locality, it is usually easy to extend such system to support global interrupt. Only minimal hardware support is needed to implement the global interrupt in a transputer-based multicomputer network.

4.4.2 Hardware Extension

Hardware requirements for implementing global interrupt scheme in a multicomputer network are:

1. A mechanism on each multicomputer node to accept interrupt signal and transfer control of the processor to the interrupt handling routine without delay.
2. A mechanism to deliver the external interrupt signal to all nodes in the network.
3. A mechanism to generate the interrupt signal, either from a multicomputer node or from any other external source.

The IMS T800 transputer provides an event channel in addition to the four data channels on each board. When the input of the event channel is held high, the process waiting for the event signal is scheduled. If the process blocking on the event channel is a high priority one and no other high priority process is running, the latency is at most 58 processor cycles [Inmos89]. Since the processor speed of the T800 transputer is 20 MHz, the delay in responding is less than three microseconds.

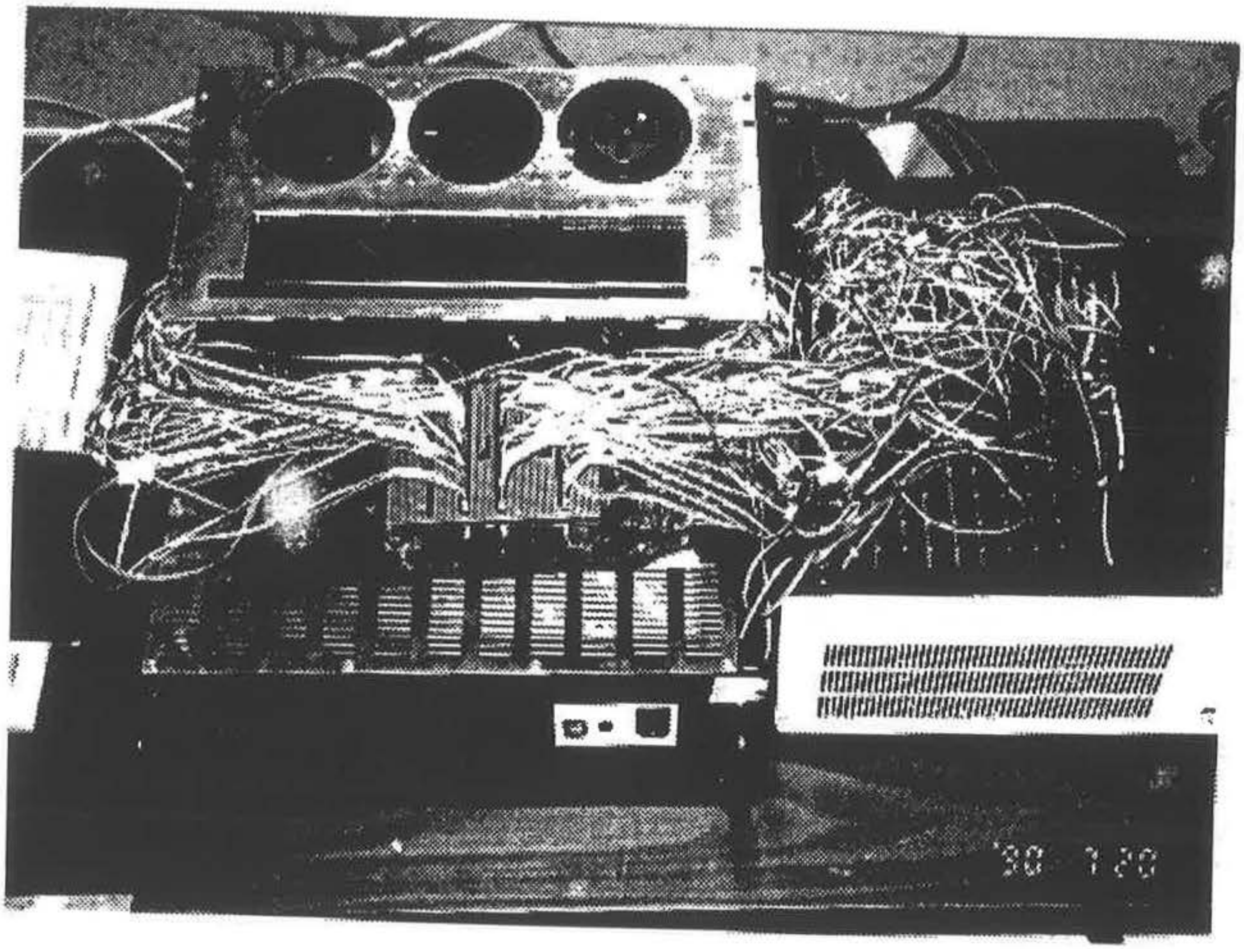
In order to deliver the global interrupt signal to all transputers in the network, we built a special hardware circuit. The circuit is basically a fan-out with one input and 74 outputs. The event channel of each transputer is connected to an output of the circuit. A data channel on the master node is connected to the input of the circuit. The global interrupt signal is generated by having the master node send to the data channel connected to the input of the circuit. Figure 4.3 shows a picture of the department's transputer-based multicomputer with the hardware extension. Seventy four transputers are physically split into two boxes, with 10 nodes in the small box and the rest of them in the big box. The global interrupt circuit is located on the top of the larger box.

4.4.3 Global Sampling and Clock Synchronization

A global interrupt is used to turn *ON* and *OFF* the monitor on all nodes dynamically. It is also used to perform global sampling and clock synchronization in the network.

A high priority controller process on the master node triggers the interrupt periodically at a pre-specified time interval. The first interrupt signal indicates the start of the parallel monitor. The controller process keeps sending until the monitor has been turned *OFF* explicitly by the user. The monitor can be restarted after it has been turned *OFF*.

On each slave node, there is a high priority backend process waiting for the interrupt signal on the event channel. Upon the arrival of the event signal, it checks a special memory location [Beers89] to determine whether the processor and each of the data channels are currently busy, and increments the counters accordingly. Since the event signals are periodic , the backend process can also update its own local clock at a predefined interval by setting the clock



'90 1 20

Figure 4.3: A Picture of the Transputer-based Multicomputer

to the expected value. The result of sampling is reported by periodically generating an event entry and writing it to the buffer pool. The overhead of sampling and clock synchronization is low since the code to be executed is exceedingly simple. It contains only a few transputer instructions and runs for less than 10 *μsec.* If resynchronization is performed every second, the overhead is less than 0.0001%. The termination of a monitoring session is detected on each slave node by not receiving the interrupt signal after a pre-defined timeout period. In the current implementation, the timeout period is set to be twice of the sampling interval.

The accuracy of clock synchronization will be affected if the monitor is not the only high priority process since the backend process will not be able to respond in a timely fashion when the interrupt signal arrives. Fortunately, in our environment, by default all user processes and Trollius server processes run in low priority. The only system processes that have to run in high priority are the kernel process and channel processes. The execution of these processes is transient. An adaptive synchronization scheme has been implemented in order to factor out the interference of these high priority processes to clock synchronization. In this scheme, the local clock is reset only if the monitor process gets control of the processor within a legitimate period of delay, say 20 microseconds; otherwise the value of the clock remains unchanged. Experimental result shows that this scheme reduces the worst case drift of the clock synchronization algorithm substantially. A limitation is that if user processes are allowed to run in high priority, the clock synchronization could be postponed indefinitely. This problem is almost impossible to avoid; however, for most applications it is common to have all user processes run at low priority. Section 5.3 reports on experimental results for the accuracy of our clock synchronization algorithm.

4.4.4 Summary

In this section, we have described the global interrupt approach and how it is used to obtain global snapshots of the system and synchronize local clocks in the transputer network. In contrast to the logical clock approach [Lamport78] has traditionally been used to order asynchronous events and obtain a consistent global state in distributed and parallel systems. The logical clock approach has also been successfully used for parallel debugging in existing systems [Fowler88][VoZe90]. However, the logical time only reflects the temporal order of events but not the physical elapsed time. Since the differences of logical timestamps are not comparable with each other, logical time cannot be used to measure the performance of message transmission. Moreover, the expense to run the logical clock algorithm is high. Therefore, a logical clock did not satisfy the requirements of our system.

Another approach to the global clock problem is the pure software clock synchronization algorithms which estimate the drifts between different clocks by passing messages around the network [Duda87][GuLa84][Shumway89]. This is a time-consuming approach since the algorithm involves exchanging a lot of messages among different nodes and the accuracy is disappointing. Chapter 5 gives a comparison between the global interrupt approach we used and the best known software synchronization algorithm on the transputer.

As compared to other techniques, the global interrupt approach has the advantage of high accuracy, low overhead and simple implementation. We have showed it can be applied to a transputer network with minimal additional hardware support.

4.5 Event Tracing

4.5.1 Event Generation

There are five types of standard events traced by the monitor: *proc_init*, *proc_exit*, *msg_send*, *recv_call*, *msg_arr*. As shown in Chapter 3 these events form a sufficient set of events which can be used to reconstruct the execution graph of the parallel program. Users can also specify their own events to be traced in the program. The probes to generate standard events are inserted into the appropriate routines in the Trollius run-time library. Appendix B discusses in detail the probe routines and the changes made to the Trollius library. In order to monitor an applications, users must recompile their programs and link to the instrumented version of the runtime library.

An additional library routine `probe()` is provided to allow user-specified events. The user is responsible for inserting the `probe()` call into the his source problem to generate the user-defined event. Our principle is to minimize monitoring overhead by tracing a minimal set of events but provide users the flexibility to monitor additional events.

Each invocation of the probe routine generates an event trace entry. It is encoded into a message and sent to the meter process on the local node using the Trollius kernel message passing mechanism. The structure of event entries is shown in Figure 4.4.

4.5.2 Buffer Management

The event trace data collected by the meter process as well as the utilization data generated by the backend process are stored locally in a buffer pool before they are sent back to the host station for display and analysis.

The buffer pool is organized as a double-buffer structure (Figure 4.5). Each buffer contains an identical number of trace entries. After one buffer is full, it is automatically switched to the other one. The buffer manager is a low priority process which periodically checks the status of buffers. Data in a full buffer are encoded into a message and sent to the host using Trollius network level message passing. The advantage of the double-buffer structure is that monitoring processes can continue writing trace entries to one buffer while the other is being flushed. Operations on the shared data structure are critical sections and are protected by disabling timeslicing during operations which access the buffer pool. If both buffers are full and new trace entries are being generated, the meter process blocks until a buffer has been emptied by the buffer manager. Since the backend sampling process cannot wait, it simply increments an overflow counter and proceeds. The overflow counter counter keeps track of the number

1	4	4	12
PROC-INIT	Timestamp	Process Id	Process Name

1	4	4	12
PROC-EXIT	Timestamp	Process Id	Not Used

1	4	4	4	4	4
MSG-SEND	Timestamp	Process Id	Event Type	Dest Node	Msg Length

1	4	4	4	4	4
RECV-CALL	Timestamp	Process Id	Event Type	Not Used	Buffer Size

1	4	4	4	4	4
MSG-ARR	Timestamp	Process Id	Event Type	Source Node	Msg Length

1	4	16
User-defined Event	Timestamp	Auxillary Information

Figure 4.4: Structure of Trace Entries

of utilization events dropped by the monitor due to overflow. At the end of each monitoring session, the value of the overflow counter on each node is reported to the host monitor.

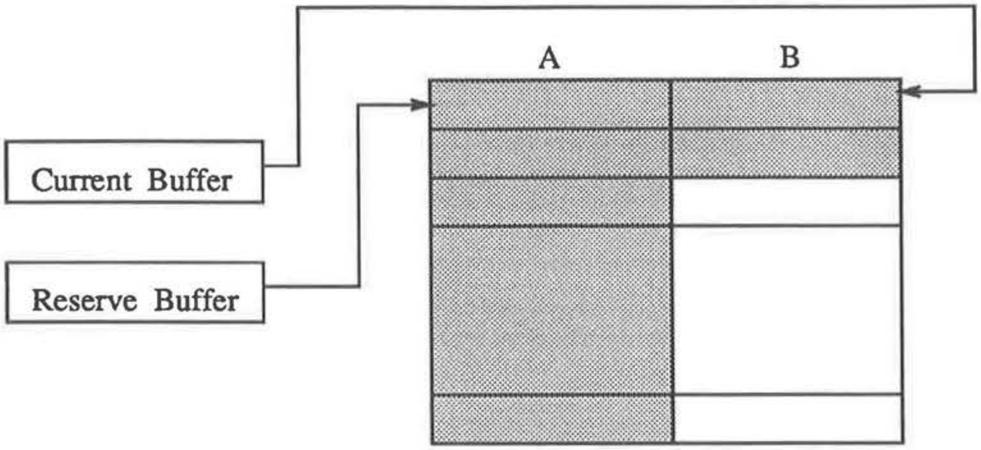


Figure 4.5: Double-Buffer Structure

4.5.3 Adaptive Reporting

The trace data at each node are sent back to the host using Trollius network level message passing mechanism. The advantage of using the same communication mechanism as the application is the simplicity in implementation. It also makes the design of the parallel monitor more portable to other systems since there no need to change the communication mechanism provided by the underlying operating system. Since the communication network is multiplexed by the monitor and the application, status messages may interfere with normal communication of the application, and affect the accuracy of the performance results measured by the monitor. Experimental results show that when the frequency of flushing buffer is high, up to 80% of the slowdown of the application is caused by monitor communication (See Section 5.2).

An adaptive reporting scheme was implemented to reduce the interference of monitoring to application communication. In this scheme, the monitor on each node keeps track of current load of the network. Monitoring data are sent only when the network is lightly loaded. The resource usage data measured by global sampling gives a very good indication of the current status of the transputer network, and this information can be used by the monitor to determine whether the trace data should be sent. Ideally, each node has complete load information of all processors and communication channels and can make the decision based on an global picture of the whole system. However, due to the inherent communication delay, the resource utilization of one node will have been already obsolete when it is propagated to the monitor on a remote node. Therefore, the information collected by the monitor does not reflect the most up-to-date consistent global view of the system. Also, it involves passing a lot of messages in the network. The approach we use is a distributed algorithm in which each processor's monitor decides whether or not to send the monitoring data based on local load information. A decision is made at each step on the path the monitoring message is routed to the host. Since the load information on the local node is always up-to-date, the monitor is able to make an accurate prediction for the next step. A predefined threshold function is used by the buffer manager on each node to determine whether the node is currently overloaded and whether to send the trace data. Let U_{cpu} denotes the CPU utilization. If link i is the one the monitor uses to send trace data to the host, let U_i denotes the utilization of this link. The threshold function f is computed by: $f = \alpha U_{cpu} + \beta U_i$ where α and β are coefficients obtained from empirical data. If $f > 0.8$, the node is considered overloaded and the sending of monitoring data is postponed. If all buffers on the local nodes are full, the buffer manager has no choice but to flush the buffers regardless of the current status of the network. The buffer size, reporting interval, and threshold function have to be selected carefully to achieve optimal performance. Chapter 5 contains an empirical study on tuning these parameters.

In the current implementation, the adaptive decision is made only at the first step when the monitoring message leaves its origin. Implementation of the complete scheme requires substantial changes to the routing mechanism of the underlying operating system and affects the portability of the monitoring system. The preliminary implementation of the adaptive reporting scheme shows up to 50% improvement over the static scheme in term of degradation in the performance of the monitored application. Experimental results indicates that the adaptive scheme improves the performance more substantially for communication intensive applications since they are more sensitive to the interference of the monitor communication.

4.5.4 Summary

Minimizing the communication overhead introduced the monitor is an important issue in monitoring multicomputer networks. Some existing systems [HaWy90] resort to a separate communication network for monitoring messages to eliminate the effect of monitoring. However, in most multicomputer networks, a separate network is not usually available for monitoring purpose and is expensive to install in the system. Some systems [Parasoft88] store and process the trace data locally until after the application computation terminates. However, in a multicomputer network where buffer space on each node is extremely limited, it is impossible to collect adequate information about the execution of any substantial application. Moreover, this approach does not permit real-time monitoring, which is desirable for many applications. The adaptive reporting scheme has proven to be an effective approach to this problem. Refinement of this scheme is expected to result in further improvement of the performance of our tool.

4.6 User Interface

The parallel monitor is designed so that it can incorporate a wide range of user interfaces. An simple command interface has been implemented for users to start and stop the parallel monitor interactively at the terminal. A programming language interface is also supported by providing two additional library routines *start_monitor()* and *stop_monitor()* so that the user may turn the monitor *ON* and *OFF* from within their programs.

Originally a simple interface was implemented to display the performance result to the user as text lines. The textual interface does not allow users to visualize the execution of their programs and was inconvenient to use. An X window-based graphical interface has been developed to display performance results to the user as easy-to-read chart and graphs. Here we give a brief description of the functionality of the graphical display.[1]

The output of the graphical display includes a main menu, the network topology, the global clock, and the execution graph. A sample view is shown in Figure 4.6. The main menu is the control panel for the parallel monitor. The window in the upper right corner in Figure 4.6 displays the topology of the transputer network. The CPU load for each node is shown in the square box representing a node in the network. By clicking on the link which connects node 0 and node 1, a small rectangle box is popped up to display the utilization of the selected transputer link. The global clock window (the one in the upper left corner) shows the current time relative to the elapsed time of the whole program. The clock value can be set, reset, start, stop or adjust speed by clicking on the corresponding buttons in the window. The window on the lower half of the screen displays the execution graph of the parallel program. Different icons are used to represent different types of events in the graph. In Figure 4.6, a filled left triangle represents a *msg_send* event. Open and filled right triangles represent *recv_call* and *msg_arr* events respectively. The communication patterns of the program can be easily visualized in the execution graph. Figure 4.6 shows a broadcasting from each node in the system. The vertical and horizontal scrolling bars allow users to conveniently browse through the execution graph or focus on only a portion of the execution graph. The display for both node utilizations and the execution graph are updated as the global clock proceeds. The user can also obtain more details of each event in the execution graph by clicking on the icon representing the event. A new window pops up with detailed description of the event being selected. For instance, if the selected event is a *msg_send*, then the sender process id, the type and length of the message, the destination node and the time the message was sent will be displayed to the user. The weighted critical path generated by the analysis tool is highlighted on the execution graph, allowing the user to examine the critical path graphically. The graphical interface is currently implemented using the InterViews [LiCaVl87] C++ graphics toolkit on top of the X window system. Our experience shows that a graphical user interface is an indispensable component of any parallel and distributed monitoring tool.

[1] The design and implementation of the graphical interface will be described in detail in Hilde Larsen's M.Sc. thesis.

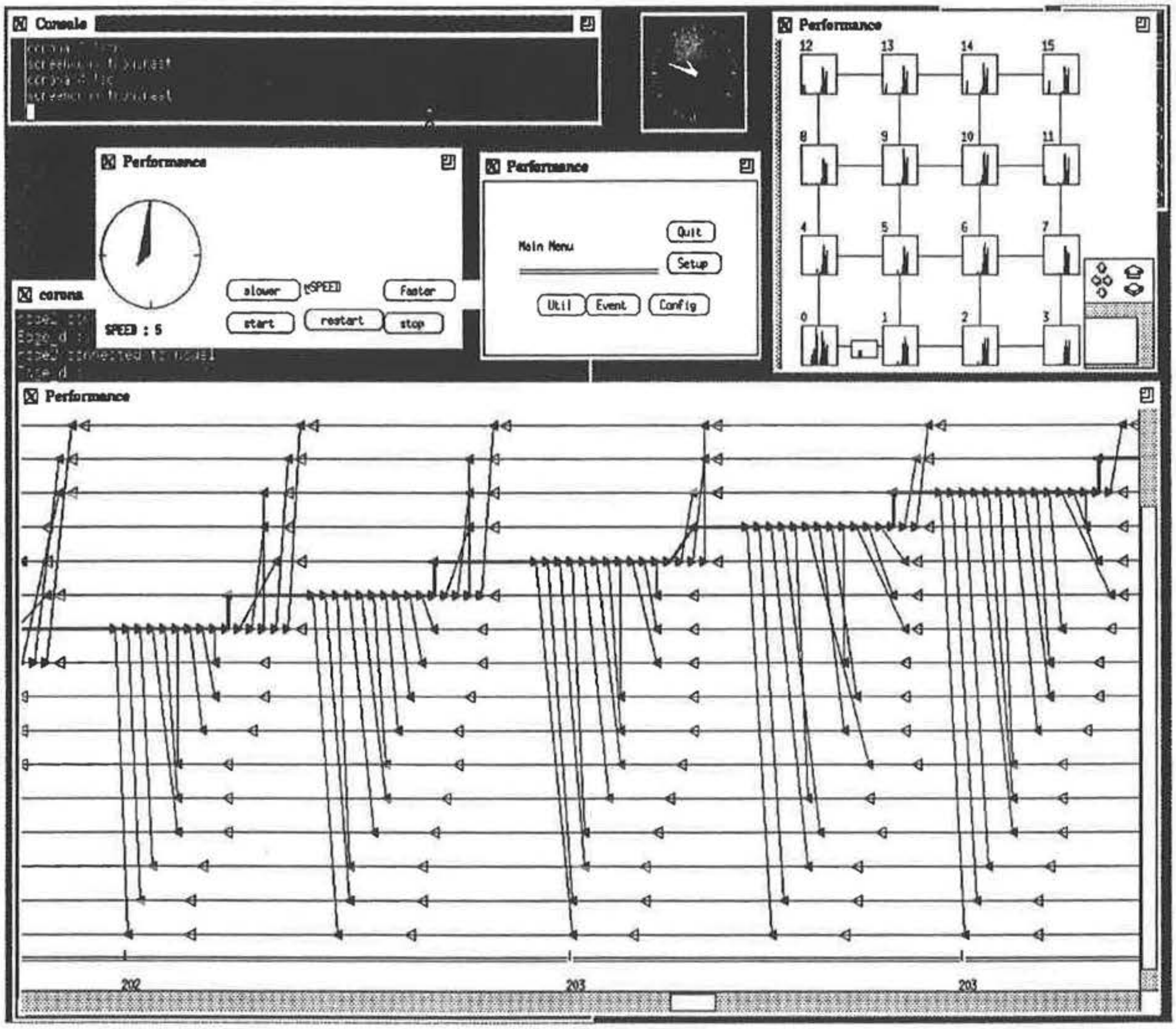


Figure 4.6: Graphical Display of Performance Result

Chapter 5

Testing and Verification

Accuracy and overhead are the two most important indicators of the success of a monitoring tool. In this chapter, we present the testing and verification results for our monitoring tool. To measure the accuracy of the monitoring data, we used an artificial application with controllable behavior and predictable performance. The parallel monitor is then used to measure these programs. The accuracy of the performance result reported by the monitor is derived by comparing it to the expected result of the program. Both artificial and real benchmarks are used to measure the overhead of the parallel monitor. Efforts have been made to isolate various sources of overhead and to tune the parameters of the monitoring program to obtain optimal performance. The accuracy and overhead of our clock synchronization technique is also discussed and compared with other software clock synchronization algorithms reported in the literature.

5.1 Validation of Monitoring Result

The performance results reported by the monitor is accurate if it correctly reflects the behavior of the application program when it runs without the monitor. In order to validate these results, the behavior of the monitored application must be known in advance. We have designed an artificial application with predictable behavior to verify the correctness of the monitoring results.

The parallel monitor reports the processor and channel utilization on each node in the transputer network. We designed two sets of programs with artificial workload to measure the accuracy of the processor and link utilization reported by the monitor. Given a predefined time interval Δt and an expected workload e, the artificial processor load program computes by incrementing a dummy counter for $l\%$ of the time during Δt and sleeps the rest of the time. Similarly, the artificial link load programs on neighbouring nodes will keep exchanging messages for $l\%$ of the time during Δt and keeps the link idle the rest of the time. To guarantee that the communication channels are busy, all artificial link load processes run in high priority and use low level transputer instructions *in*() and *out*(). The operating system level message passing primitives are not used in order to avoid unpredictable or extra context switches. Each artificial application is measured over an extended period of time: $T \gg \Delta t$. The experiment is repeated a large number of times and the average, range and standard deviation for the performance results reported by the monitor are calculated for each artificial workload. Table 5.1 shows the validation results for processor utilization. Table 5.2 shows the validation results for link utilization. In both cases, $\Delta t = 1sec, T = 10min$, and the result is computed over 10 experiments. It can be seen from the tables that both processor utilization and link utilization measured by the monitor are very accurate, with worst case deviation less than 5% and standard deviation less than 2%.

Expected Load (%)	Average Load (%)	Range (%)	Standard Deviation (%)
10	10.62	9 - 12	0.89
20	20.72	19 - 22	1.13
30	30.32	29 - 32	0.82
40	40.10	40 - 41	0.30
50	50.09	50 - 51	0.29
60	60.08	59 - 61	0.38
70	70.32	68 - 72	0.96
80	80.13	78 - 81	0.88
90	89.90	89 - 90	0.30
100	99.90	99 - 100	0.30

Table 5.1: Validation of Processor Utilization

Expected Load (%)	Average Load (%)	Range (%)	Standard Deviation (%)
10	10.41	9 - 12	0.83
20	19.46	17 - 20	0.73
30	29.00	27 - 30	0.87
40	38.27	35 - 40	1.54
50	48.80	46 - 50	1.09
60	58.60	56 - 61	1.58
70	68.82	67 - 71	1.36
80	79.14	77 - 81	1.60
90	89.71	87 - 92	1.69
100	99.31	98 - 100	0.79

Table 5.2: Validation of Communication Channel Utilization

The accuracy of the resource utilization result is affected by the interval of global sampling.

The sampling interval must be short enough for the distribution of workload to be homogeneous, but long enough to keep monitoring overhead within an acceptable range. One criteria for selecting the sampling interval is that it need not be performed more frequently than events occurs. Since the frequency of interprocess communication events in Trollius is measured in a few hundred microseconds, the sampling interval need not be less than 1 *msec*, but (probably) should not exceed 10 *msec*. Figure 5.1 shows that the monitoring overhead decreases as the sampling interval increases. The overhead shown in the chart is in fact the sum of the sampling overhead and the reporting overhead. Reporting overhead decreases as sampling interval increases because trace data are reported less frequently. However, since the reporting interval is relatively long as compared to sampling interval, its effect on the monitored program does not show linear behavior. This is why in Figure 5.1 the overhead does not decrease linearly as sampling interval grows. Based on the result in Figure 5.1, 5 *msec* seems to be the best choice since the overhead remains almost constant at 2.5% once the sampling interval is increased to 5 msec. The results given in Table 5.1 and Table 5.2 are measured with a global sampling performed every 5 *msec* on all nodes.

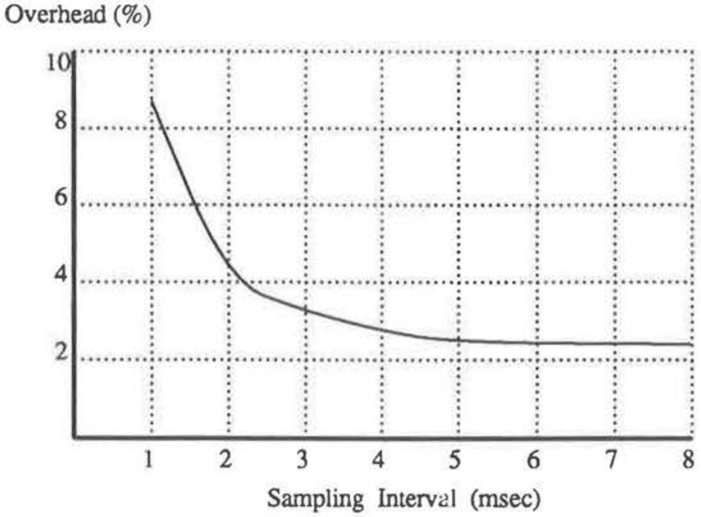


Figure 5.1: Monitoring Overhead for Different Sampling Intervals

The accuracy of performance metrics such as program execution time relies on the time measurement of the traced events, which in turn relies on the accuracy of the global clock. The accuracy of clock synchronization in the monitoring system will be discussed in Section 5.3.

5.2 Measurement of Monitoring Overhead

The overhead incurred by the parallel monitor is measured by the performance penalty (slowdown) it introduces to the monitored application. Let T be the program execution time of the parallel application when it runs without the monitor, and T_M be the program execution time when the application runs with the monitor. The overhead is computed by $(T_M - T)/T \times 100\%$.

In Section 1.1.2, we discussed the various sources of monitoring overhead. In our experiments, we isolated different sources of monitoring overhead and measured each separately. Our purpose was to identify which one contributed most to the overhead in our system. The parallel monitor is functionally decomposed into three components, labelled:

- A: global sampling and clock synchronization;
- B: event tracing;
- C: reporting.

The monitoring overhead caused by different components was measured by disabling one or more of them during different monitoring sessions. For reporting(C), we measured both the static reporting scheme (C_S) and the adaptive reporting scheme (C_A) as proposed in Chapter 4. The application used to evaluate the monitor was a parallel Cholesky's factorization algorithm implemented under Trollius on the transputers. The input of the program is a $n \times n$ matrix. During the computation, the matrix is decomposed into submatrices which are processed concurrently on different nodes. The size was held constant so that as the number of nodes increases, the granularity of the computation becomes finer. In Table 5.3, the overhead to monitor this application is shown when running on different topologies with constant input size 65 × 65. The overhead attributed to different components on each row. $A + C_S$ means the monitor only performs global sampling and static reporting. $A + B$ means it only performs sampling and event tracing without reporting the results to the host. Results for other rows can be interpreted in similar ways.

Topology	Overhead (%)					
(Mesh)	1 × 2	2 × 2	2 × 4	4 × 4	4 × 8	6 × 8
$A + C_A$	0.5	0.8	1.1	2.4	0.7	3.6
$A + B$	2.0	1.2	1.5	1.4	3.7	9.7
$A + B + C_S$	2.8	2.4	5.8	8.1	42.6	45.2
$A + B + C_A$	2.6	2.4	5.3	5.4	20.9	39.1

Table 5.3: Monitoring Overhead for Cholesky's Factorization Program

The results given in Table 5.3 show that the overhead of both sampling(A) and event tracing(B) is low, less than 4% in most cases. It also shows that the overhead of reporting(C) is reasonably low when executed on topologies with less than 16 nodes (the 4 × 4 mesh). The granularity of the 65 × 65 matrix on 16 nodes is reasonable as each node get a 4 × 4 submatrix. As the computation becomes too fine-grained on larger topologies, e.g. the 4 × 8 or 6 × 8 mesh, the overhead incurred by reporting monitoring data increases dramatically to over 40%. This is because the communication of the application is so intensive that the interference with monitoring message severely degrades the performance of the application. The result for the adaptive reporting scheme ($A + B + C_A$) indicates that substantial improvements are possible by reducing the interference of reporting to application communication.

Table 5.4 shows the overhead introduced to the Cholesky's factorization program with different size input. In this table the same 2 × 4 topology was used. The overhead is the sum of sampling, event tracing and reporting. Measurements are made for both the static reporting scheme and the adaptive reporting scheme. The number of events generated by the application for different size of input is also shown in the table. Note that overhead decreases as the input size increases. On a fixed number of nodes, the larger the size of the input matrix, the less fine-grained the parallel computation and the less overhead the monitor communication incurs. This in combination with Table 5.3 supports our claim that the major source of monitoring overhead lies in the communication bandwidth used to report monitoring data. We can also conclude from these experiments that the overhead of reporting decreases as the granularity of the parallel application grows. The result in Figure 5.3 indicates that the adaptive reporting scheme is an effective means to reduce the communication overhead of the parallel monitor. However, when the overhead is low, adaptive and static reporting scheme behave basically the same (See Figure 5.4). For parallel applications with reasonable granularity, the overhead incurred by the parallel monitor is within acceptable range (below the 15% performance penalty suggested in [Reed89]).

		Overhead (%)	
Matrix Size	Number of Events	Static Reporting	Adaptive Reporting
9 × 9	399	6.1	8.9
29 × 29	871	8.0	6.7
65 × 65	2032	5.8	5.3
144 × 144	4195	2.3	1.6
234 × 234	9388	1.1	1.1
504 × 504	14590	1.0	1.0

Table 5.4: Monitoring Overhead for Input of Various Matrix

Another source of monitoring overhead which does not affect the running time of the program is the memory space allocated to store the data collected by the monitor on each node. Memory is often is scarce resource on transputers. It is important to minimize the buffer space used by monitor so that memory can be used for scaling up the size of the problem. However, reducing the size of the buffer pool would increase the frequency of reporting, resulting in higher overhead. Therefore, a trade-off has to be made between satisfying the memory constraint and reducing communication overhead of the monitor. Figure 5.2 shows the monitoring overhead for different buffer sizes for the Cholesky's factorization program with input size 65 × 65 on a 4 × 4 mesh.

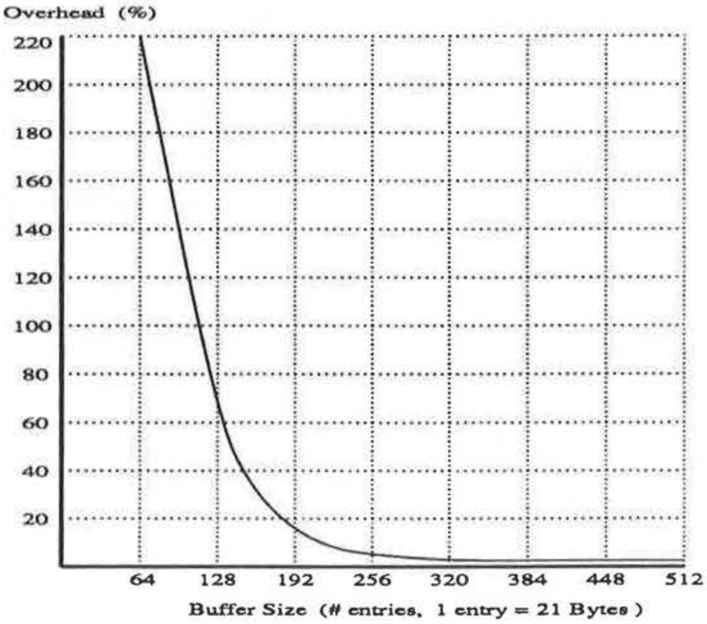


Figure 5.2: Monitoring Overhead for Different Buffer Size

The default buffer size in the current implementation is 5378 bytes which could store 256 trace entries. The user is given the flexibility to specify buffer size allocated for the parallel monitor on each node.

5.3 Clock Synchronization

A clock synchronization algorithm is *acceptable* if the drift between different clocks is small compared to the minimum interval of time between any two events. We use a global clock to order asynchronous events and measure the elapsed time of message transmission. Therefore the accuracy of the performance results we obtain heavily depends on the accuracy of the clock synchronization algorithm used in our system. This section presents our results for the global interrupt approach we use to perform clock synchronization in the transputer network. We compare this approach to other software clock synchronization algorithms.

We measured the drift between different clocks by having processes on neighbouring nodes exchange one single byte message for a predefined period of time. Assuming that when both channels are active the time to transmit a one-byte message is identical, the clock drift can be derived from the average difference of opposite direction message transmission time. The two communicating processes on the neighbouring node run in high priority and use the low level transputer assembly code *in*() and *out*() to exchange messages, in order to factor out the interference due to context switches. Table 5.5 shows the accuracy of our clock synchronization algorithm.

Resync Interval (*sec*)	0.1	0.5	1.0	2.0
Average Drift (μsec)	0.84	1.95	3.58	5.91
Maximum Drift (μsec)	6	7	8	14
Standard Deviation(μsec)	1.40	1.86	2.30	3.88

Table 5.5: Accuracy of Clock Synchronization Using Global Interrupt

Note that higher accuracy of clock synchronization can be achieved by performing a resynchronization more frequently. By performing a resynchronization every second, we achieve an accuracy of average drift less than four microseconds and maximum drift of eight microseconds. Very little overhead is incurred in our scheme since the code to be executed on each node to perform the resynchronization is extremely efficient. It contains only a few transputer instructions and runs for less than 10 μsec. If the resynchronization interval is one second, then the overhead is less than 0.0001%. Also, it takes less than 10 microseconds to send a one-byte message across a transputer link, and the message transmission time in Trollius is measured in several hundred microseconds. The accuracy of clock synchronization algorithm is more than adequate for ordering asynchronous events and measuring message elapsed time.

There have been a few clock synchronization algorithms for transputer networks reported in the literature [Shumway89][CaVi88]. We compared our scheme with the *RING-SYNC* algorithm in [CaVi88] since it reports the best accuracy among all existing algorithms.

The RING-SYNC algorithm is based on a ring-structured transputer network in which a master node periodically passes a *SYNC* message around the ring containing the local clock value and the partial delay. Upon receiving a *SYNC* message, every slave node sets the value of its clock to the sum of the clock value and partial delay in the *SYNC* message and updates the clock value in the message accordingly. When the *SYNC* message returns to the master node, it recalculates the partial delay for the next *SYNC* message. In [CaVi88], they also apply linear regression and q-degree extrapolation to estimate the drift between two resynchronizations and revise the clock value. Experimental results for the RING-SYNC algorithm have been reported in [CaVi88]. The maximum and typical clock drift are measured with and without the interference of user process, and the result is given before and after the drift correction using the q-degree extrapolation. Table 5.6 gives the summary of the best of their results when resynchronization is performed every 5 seconds.

	No Revision		1-degree Extrapolation	
	NO LOAD	W/ LOAD	NO LOAD	W/ LOAD
Maximum Drift (μsec)	100	115	12	56
Typical Drift (μsec)	100	115	8	36

Table 5.6: Accuracy of RING-SYNC algorithm

The result of the RING-SYNC algorithm for the *NO LOAD* case after drift correction using the 1-degree extrapolation seems to be almost as good as our clock synchronization using global interrupts. However, it deteriorates drastically in the presence of user processes in the system. The reason is that the RING-SYNC algorithm has to share the communication channels with the application and thus interferes with the user's communication activities. Since passing a *SYNC* message around the ring is very expensive, especially when the number of nodes in the system is large. Better accuracy cannot be achieved by performing resynchronization more frequently in RING-SYNC. As compared to the RING-SYNC algorithm, our clock synchronization algorithm using global interrupts has the following advantages:

1. *Topology independent.* Our approach makes no assumption about the interconnection of the transputer network, while the RING-SYNC algorithm only works in networks containing a ring. This limitation of the RING-SYNC algorithm implies that it cannot be directly applied to common topologies such as tree-structured networks.

2. *Application independent.* The accuracy of clock synchronization using global clock is not affected by the application since a separate network, the global interrupt circuit, is used to deliver the signal. The accuracy of RING-SYNC algorithm is seriously affected if the application is highly communicative.

3. *Lower overhead.* The overhead of the RING-SYNC is substantially higher than the global interrupt approach even if resynchronization is only performed rather infrequently. Running the q-degree extrapolation algorithm for correction consumes extra processing power on each node.

4. *Higher accuracy.* Even the accuracy of the RING-SYNC algorithm in the *ideal* case is only close to the accuracy achieved using global interrupt. The difference in the normal case with user processes in the system between the two scheme is an order of magnitude greater.

5. *Simple implementation.* The implementation of our scheme is exceedingly straightforward and the code contains only a few transputer instructions. While efficient implementation of the RING-SYNC and the q-degree extrapolation algorithm can be tricky.

The advantage of the RING-SYNC algorithm is that it is a pure software solution and does not need any extra hardware support. However, the accuracy and reliability of the global interrupt approach more than justified the minimal amount of extra hardware needed to implement it.

5.4 Summary

In this chapter, we presented the results of our experiments in measuring the accuracy and overhead of the parallel performance monitor developed on the transputer-based multicomputer.

The results indicate that both the accuracy and the overhead of our monitoring tool are within the desired range to achieve the goals we proposed in Section 1.3. By measuring the various sources of monitoring overhead we identify the communication activities of the monitor as the major source of overhead in our system. The results indicate that the adaptive reporting scheme as proposed in Chapter 4 is an effective means of reducing the interference of monitoring to application communications. A comparison is made between our clock synchronization scheme using global interrupts and a pure software clock synchronization. The results indicate that our approach is superior in accuracy, overhead, applicability and simplicity, justifying our design principle of relying on minimal hardware support to achieve performance beyond the realm of any pure software solutions.

Chapter 6

Performance Tuning: A Case Study

In this chapter, an example is used to demonstrate the use of our monitoring tool to tune a parallel application. The application we have chosen is an image reconstruction algorithm implemented on transputers.

6.1 The Parallel Image Reconstruction Algorithm

The algorithm is a parallel version of a sequential algorithm used in image processing to eliminate noise from a raw image by performing edge detection on the image. Input to the algorithm is a raw image as an $n \times m$ matrix, each element representing a pixel in the image. The algorithm is designed for a $k \times k$ 2-dimensional mesh. The input matrix is decomposed into k^2 submatrices where all processors except those in the last row of the mesh receive a square submatrix of size $\lfloor \frac{MIN(n,m)}{k} \rfloor$. All extra columns in the input matrix are sent to the last row of the mesh. Upon receiving a submatrix, each processor runs the edge detection algorithm on the subimage and exchanges the side columns of its submatrix with its nearest neighbours in order to recompute the pixels at the edges of its subimage. The computation on the subimage is iterated until convergence, i.e. until no more elements in its submatrix get updated. All processed subimages are then recombined and the parallel computation terminates.

The algorithm has been implemented under the Trollius Operating System and run on 17 transputers configured as a 4×4 mesh plus an external node which has a direct connection to the host workstation. The network topology on which the program runs is shown in the output of the graphical display of the parallel monitor (Figure 6.1). The transputer node adjacent to the host (called the *master* node) reads in the raw image from the host file system. It decomposes the input matrix into submatrices and distributes them to all transputer nodes in the mesh. Each *slave* node computes and communicates with its neighbouring nodes using Trollius network level message passing primitives. The results from all slave nodes are recombined at the master node. The reconstructed image is then written to a file in the user's file system. The program contains about 1500 lines of C code and is an integrated part of an image processing package developed in the Computer Science Department at UBC.

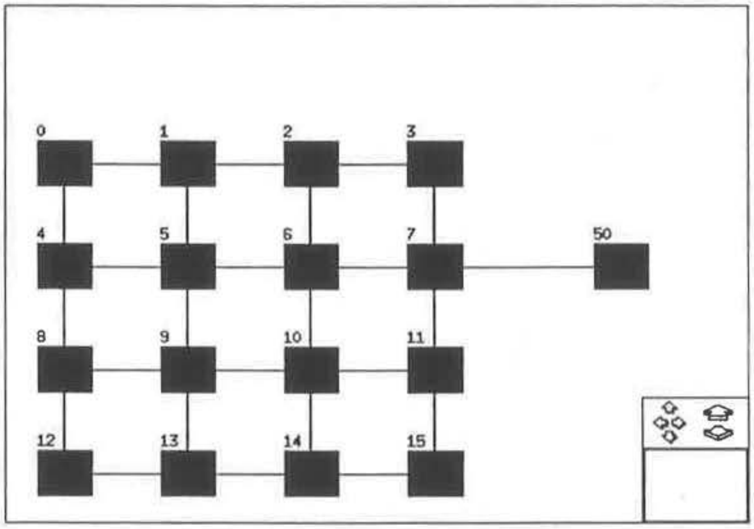


Figure 6.1: Graphical Display of Network Topology

6.2 Measurement and Analysis

The program was originally implemented and debugged on transputers without the help of the parallel monitor, and it appeared to produce desired result. The program was recompiled and linked to the instrumented version of the Trollius runtime library without modification to its source code. The input image is a 47 × 47 square matrix.

The first result we obtained from the parallel monitor turned out to be a debugging result rather than a performance result. The graphical display of the execution graph indicated that the monitor was unable to find the matching *msg_arr* events for some of the *msg_send* events on the slave nodes(Figure 6.2). This occurred near the end of the execution of the program. By clicking on the unmatched sending events in the graph, we examined the information about each of these events and discovered that some segments of the reconstructed subimage sent by the slave nodes were never received by the master node. The execution graph also indicated that all receiving events on the master node were matched. Hence, the problem was that the master node did not make enough receive calls when collecting subimages. With the help of the monitoring tool, this bug was quickly fixed. Although our tool is primarily intended as a performance monitor, it certainly can also be used to debug programs. It allows the user to gain insight into the runtime behavior of execution of the parallel program and detect problems or locations where the program is behaving strangely.

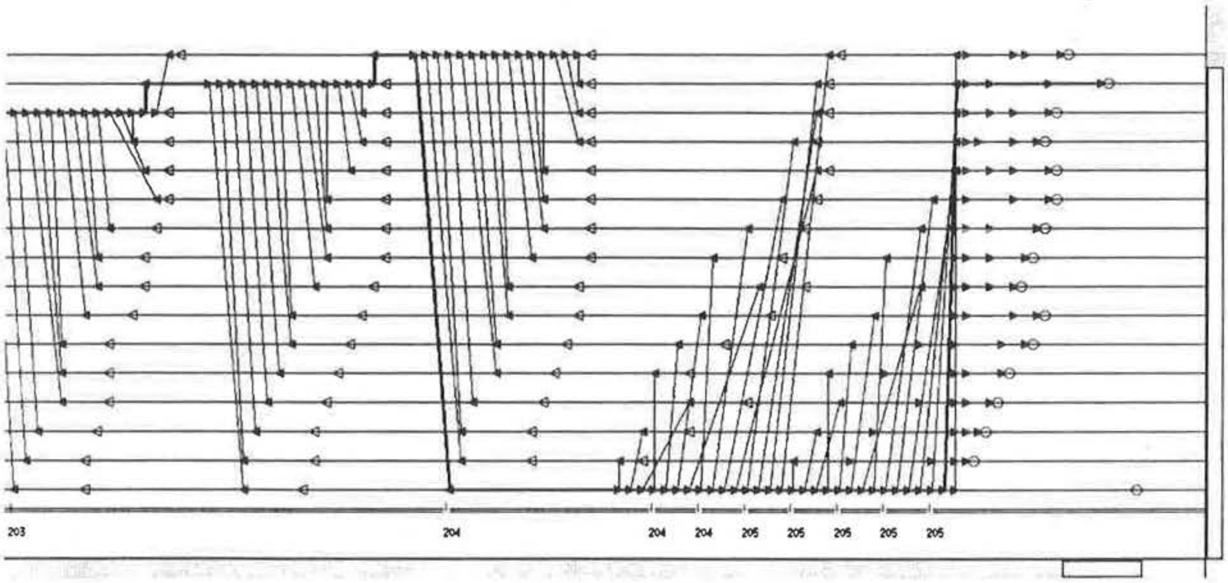


Figure 6.2: Graphical Display of Unmatched Message Events

Once having debugging the program, we ran the WCPA tool on the trace to obtain our measurement. The performance of the inital implementation was very disappointing. The speedup on 16 nodes was less than 3 and the efficiency is less than 20%. The ratio of computation vs. communication in the program was 20 : 80, which means 80% of the execution time was spent in communication. By examining the weighted critical path generated by our tool, we discovered that the communication activities to distribute and return the subimages constitutes the major portion of the critical path. In order to obtain precise measurement of the relative weight of different phases in the execution of the program, we manually inserted probes into the application to generate user-defined events, signifying the start of each phase:

```
probe(READ_IMAGE, "reading image");
probe(DISTRIBUTE_IMAGE, "distributing");
probe(COMPUTE_IMAGE, "computing");
probe(RETURN_IMAGE, "returning image");
probe(WRITE_IMAGE, "writing image");
```

These probes were placed in the main program right before the procedure calls to execute the corresponding tasks. We re-ran the program under the monitor and measured the elapsed time and relative weight of each of the phases. For instance, the elapsed time and relative weight of distributing subimages was measured by the elapsed time between the DISTRIBUTE_IMAGE event and the **COMPUTE_IMAGE** event on the weighted critical path and the percentage of the total weight between them. These measures were generated automatically by our analysis tool. The result of the analysis is shown in Table 6.1.

Phase in WCPA	Relative weight
Read Image	29%
Distribute Image	5%
Compute Image	8%
Return Image	47%
Write Image	11%

Table 6.1: Analysis Result for Image Reconstruction Algorithm

As shown in Table 6.1, the input and output of the image was weighted 40% on the critical path. This is due to the low degree of parallelism during these operations, i.e. all nodes are idle waiting while the master is reading from or writing to the host file system. Since only the master node is adjacent to the host, this I/O bottleneck is impossible to be completely removed. In the remainder of the discussion we ignore the effect of this sequential I/O bottleneck.

The computation on the subimages was only weighted 8% on the critical path. This is because all nodes are processing the subimages in parallel and a high degree of parallelism has been achieved in the system. It also indicates that the code to be execute is efficient already and further code optimization cannot improve the performance very much.

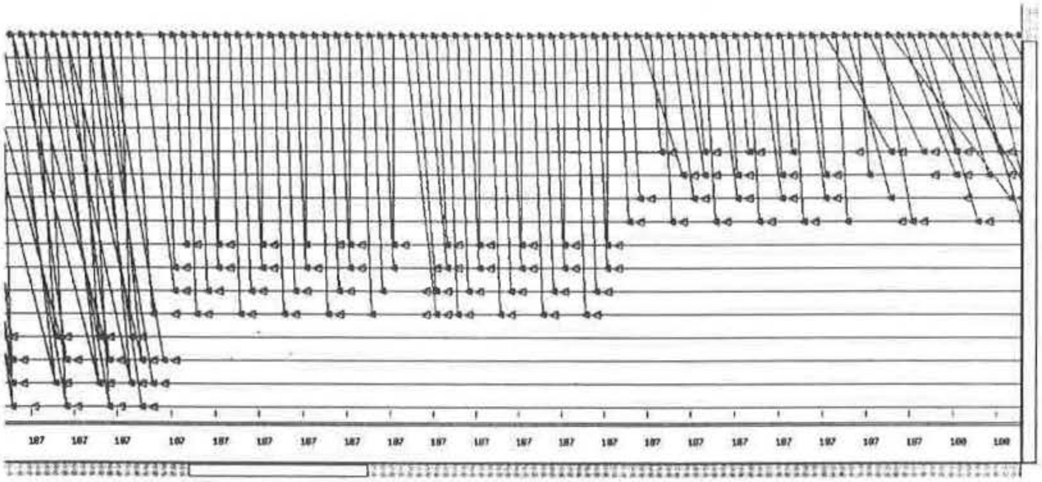


Figure 6.3: Communication Activities of Distributing Subimages

We therefore focus our attention on the distribution and gathering phases of the computation, which together were weighted 52% on the critical path. The communication pattern of the parallel program can easily be visualized in the graphical display of the execution graph. Figure 6.3 shows the communication activities in the system when subimages are being distributed from the master node to all slave nodes. We can see from the graph that the number of messages to send a submatrix appears to be excessive (A total of 11 messages are involved to send a subimage). Closer examination of each event in the graph revealed that the length of each message is only 60 bytes. To send a message using Trollius network level message passing primitives, the header attached to each message is more than 50 bytes. Thus sending each message incurred an almost 50% communication overhead. By consulting the author of the program, we found out that the subimages are distributed column by column, i.e. each column is encoded into a separate message, regardless of the size of the subimage. Since the size of our input image was relatively small and each column of the submatrix contained only 14-15 elements, there were a large number of short messages. We recommended that some of the columns be combined and encoded into longer messages. The program now sends the subimage in a single message of its size does not exceed 800 bytes. Both the number of messages and the communication overhead of each message were dramatically reduced. The program execution time of the modified version showed 55% improvement over the initial implementation. The weight of distributing and returning images on the critical path was reduced from 52% to 39%. The ratio of computation vs. communication was improved to 63:36, indicating that 63% of the time was spent in computation tasks. Speedup and efficiency have been improved by more than 100% due to the improvement of program execution time.

6.3 Summary

In this chapter, we have demonstrated how our monitoring tool is used to effectively improve the performance of a parallel application. Tuning the performance of a parallel program is a very sophisticated task due to the complicated interaction among concurrent components of the program. Effective performance tuning not only relies on the user's thorough knowledge about the program structure, but also depends on the information available to the user about the execution of the program. The information presented to the user is useful only if the user gains insight into the runtime behavior of the program and can appropriately focus on the program activities which have the most impact on the overall performance of the program. By combining a graphical display with the weighted critical path analysis package, our tool provides at a high level automatic guidance for performance tuning. In our example, the identification and resolution of such a performance problem in a parallel application has led to more than 50% improvement in program execution time.

Chapter 7

Conclusions

7.1 Synopsis

This thesis has studied the performance characteristics of parallel programs in multicomputer networks, and presented the design and implementation of a real-time performance monitor on transputers. We started with a simple performance model which is based on a graph representation of parallel programs in the multicomputer network. This performance model allows us to easily derive a variety of performance metrics for parallel programs. From this model, we also developed a new analysis method, called weighted critical path analysis (WCPA), which has proven to be helpful detecting performance bottlenecks in parallel programs. The design of a real-time performance monitor was proposed based on these ideas and then implemented on a 74-node transputer-based multicomputer. Lastly, we set up benchmarks to validate the accuracy of the monitoring results and to measure the overhead incurred by the monitor. We also demonstrated how this tool can be used to tune the performance of an actual parallel application on transputers.

We proposed in Section 1.3 a set of goals to guide the design of our performance monitoring tool. Our experience with the tool indicates that our goals have been achieved. The capability of measuring both resource utilization and tracing process events is clearly superior to other transputer performance monitoring tools. (Section 2.2). Extensibility was achieved by the modular structure and well-design interface between different components of the parallel monitor(Section 4.2). Experimental results show that both the accuracy and the overhead of the monitor are within acceptable ranges. Transparency is achieved by inserting software probes into the run-time library of the underlying operating system so that users do not have to modify their source programs to make them monitorable. We took advantage of a high level windowing environment, namely the X window system, to display performance results in a user-friendly manner. Although our monitoring tool was designed for the transputer-based multicomputer networks and implemented under the Trollius Operating System, the measurement and instrumentation techniques developed are applicable to a wide range of distributed memory parallel architectures. The performance model and the weighted critical path analysis method we proposed in Chapter 3 can be easily adapted to any message-based distributed systems, such as the LAN-based distributed environment. The use of global interrupts and the clock synchronization technique we used can be ported to most closely-coupled multicomputer networks with minimal modifications. The adaptive reporting scheme and design of the graphical interface are generally applicable to any performance monitoring tools for parallel and distributed programs.

7.2 Future Work

We conclude this thesis by suggesting possible future enhancement of our tool and speculating on future research directions.

7.2.1 Enhancement of the Monitoring Tool

The adaptive reporting scheme has not been fully implemented in the current implementation. Since trace data are sent to the host using Trollius network level message passing mechanism, the decision on whether or not to send the data can only be made at the first step when it leaves its node of origin. The monitor has no control over status messages after they are sent. The router handles both user messages and status messages in the same way. Further refinement of the adaptive reporting scheme would include modification of the routing mechanism of the operating system so that *message priorities* are supported. User messages are given higher priority and monitoring messages are given lower priority so that user messages going to the same channel as status messages are handled first. Status messages are sent only when there is no user message waiting for the same channel or when the local buffer has been filled.

Another improvement of our tool includes better integration of the monitor with the graphical interface so that operation of the parallel monitor can be controlled interactively by "clicking a button".

7.2.2 Alternatives to Nonintrusive Monitoring

To reduce the overhead caused by messages sent by the monitor, we proposed the adaptive reporting scheme(Section 4.5.3). There are other alternatives to achieve the same goal. One approach is to compensate for the overhead incurred by the monitoring when calculating performance metrics from raw trace data. For instance, to compensate for the communication overhead introduced by the monitor, the monitoring process on each node has to keep track of the number of status messages and that of user messages sent over a communication channel during a specific period of time. Using these data, it can estimate the extra queuing delay the status messages have caused and distribute the total delay to each of the user message on the same channel. The extra delay for each step on the route is then subtracted from the total elapsed time of the message, thus obtaining the corrected message transmission time. In order to be able to compensate for the monitoring overhead, we must collect enough information about the execution of the monitor itself. In essence, it is a matter of how to monitor the monitor itself. Furthermore, an appropriate queuing model has to be developed to estimate the interference the monitor has caused to the application.

A different approach that takes advantage of the global interrupt mechanism available in out system, is to stop the computation and communication activities of the application program in the whole system when performing measurement tasks and draining trace data from each node. Global interrupts can be used to stop all nodes simultaneously and restart the system after the measurement task is finished. The clock value on each node is reset to its last value when the system was stopped. This would completely factor out all the overhead of monitoring and reporting to the host. The performance results obtained should precisely reflect the behavior of the application as if it were run without the presence of the monitor and a high degree of *virtual* non-intrusiveness is achieved. One disadvantage of this scheme is that it is likely to be slow. A second problem is the difficulties in stopping the computation and communication activities of the application in a parallel system. Although we can remove all user processes temporarily from the ready queue when a global interrupt arrives, the work the system processes are doing on behalf of the application cannot be suspended halfway since some system services are needed to perform the measurement task. Moreover, process scheduling is supported by hardware on transputers; the manipulation of these process queues is tricky and error-prone. A third problem is how to deal with the user messages being transferred over a link when the system is stopped. The monitor must wait until the data transfer finishes before it can get control of the link.

Both schemes seem to be promising alternatives to achieve non-intrusive monitoring in the multicomputer networks. The possibility of implementing them on transputers will be investigated in future research.

7.2.3 Performance Steering

An interesting application of our tool is to use the information provided by the monitor to tune the performance of the application on the fly, which is known as *performance steering*. Performance steering is especially useful for programs that run for a long period of time. say several days to several weeks. In addition to displaying the performance data to the user. they can also be used as feedback to the underlying system which can control the execution of the application in order to achieve optimal performance. The dynamic load balancing technique also falls into this category. One special feature of the transputer network is that its topology can be dynamically reconfigured by simply sending instructions to the crossbar switches from the host. Since the communication pattern of the application is reflected in the execution graph generated by the monitor, it can be used to minimize the communication overhead. We may, for instance, try to directly connect nodes which communicate frequently so that messages do not have to be routed through intermediate nodes.

Bibliography

[AnLa89] T. E. Anderson and E. D. Lazowska, *Quartz: A tool for tuning parallel program performance*, Technical Report 89-09-05, Dept. of Computer Science, Univ. of Washington, Sept. 1989.

[AnJo87] F. Andre and A. Joubert, *SiGLe: An evaluation tool for distributed systems*, Proc. IEEE Intl. Conf. on Parallel Processing, 1987, pp.466-472.

[Babb87] R. G. Babb, *et al*, *Multi-level monitoring of parallel programs*, Technical Report, Dept. of Computer Science, Oregon Graduate Center, Rpt. No. CS/E 87-013, Nov. 1987.

[BaWi83] P. Bates and J. Wileden, *High-level debugging of distributed systems: the behavioral abstraction approach*, ACM SIGPLAN Notice, Vol. 18, No. 8, Aug. 1983.

[Beers89] J. Beers, Private communication, 1989.

[Beilner88] H. Beilner, *Measuring with slow clocks*, Technical Report, TR-88-003, International Computer Science Institute, Berkeley, CA, July 1988

[Bran89] W. C. Brantley, *et al*, *RP3 performance monitoring hardware*, Instrumentation for Future Parallel Computing Systems, Addison-Wesley, 1989.

[BuMi89] H. Burkhart and R. Millen, *Performance-measurement tools in a multiprocessor environment*, IEEE Trans. on Computers, Vol. 38, No. 5, May 1989.

[Burns88] G. D. Burns, *Trollius operating system definition*, Trollius Documentation Series, Ohio Supercomputer Center, Oct. 1988.

[CaTu89] W. Cai and S. Turner, *Highly transparent monitoring of real-time occam programs*, Proc. of 2nd Conf. of North American Transputer User Group, Oct. 1989, pp.41-52.

[CaWe88] P. C. Capon and A. J. West, *Monitoring Occam channels by programming transformation*, Proc. of 1988 Transputer Conference, 1988, pp. 160-169.

[CaVi88] V. Carlini and U. Villano, *A simple algorithm for clock synchronization in transputer network*, Software - Practice and Experience, Vol. 18, No. 4, Apr. 1988.

[Chan87] S. C. Chan, *Designing and implementation of an event monitor for the Unix operating system*, M.Sc. Thesis, Dept. of Computer Science, Univ. of British Columbia, April 1987.

[Couch88] A. L. Couch, *Graphical representation of program performance on hypercube message-passing mutliprocessors*, Ph.D. dessertation, Dept. of Mathematics, Tufts University, May 1988.

[EaZaLa89] D. L. Eager, J. Zahorjan and E. D. Lazowska, *Speedup versus efficiency in parallel systems*, IEEE Trans. on Computers, Vol. 38, No. 3, march 1989.

[Duda87] A. Duda, *et al*, *Estimating global time in distributed systems*, Proc. of 7th Intl. Conf. on Dist. Comp. Syst., Sept. 1987, pp.299-306.

[Emrath88] P. A. Emrath, S. Ghosh and D. A. Padua, *Event synchronization analysis for debugging parallel programs*, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, CSRD Rpt. No. 839, Dec. 1988.

[Fromm83] H. Fromm, *et al*, *Experience with performance measurement and modelling of a processor array*, IEEE Trans. on Computers, Vol. C-32, No. 1, Jan. 1983, pp. 15-31.

[Fowler88] R. Fowler, *et al*, *An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors*, Proc. of Workshop on Parallel and Distributed Debugging, May 1988, pp.163.

[Gait86] J. Gait, *A probe effect in concurrent programs*, Software - Practice and Experience, Vol. 16, No. 3, March 1986, pp.225-233.

[GaMo84] H. Garcia-Molina, *et al*, *Debugging a distributed system*, IEEE Trans. on Software Engineering, Vol. SE-10, No. 2, march 1984.

[GuLa84] R. Gusella and S. Latti, *TEMPO - A network time controller for a distributed Berkeley Unix system*, IEEE Distributed Processing Tech. Comm. Newsletter, Vol. 6, No. 2, June 1984.

[HaSh89] D. Haban and K. Shin, *Application of real-time monitoring to scheduling tasks with random execution times*, Technical Report, International Computer Science Institute, Berkeley, CA, TR-89-028, May 1989.

[HaWy89] D. Haban and D. Wybrantietz, *Monitoring and measuring parallel systems using a non-instrusive rule-based system*, Technical Report, International Computer Science Institute, Berkeley, CA, TR-89-030, May 1989.

[HaWy90] D. Haban and D. Wybrantietz, *A hybrid monitor for behaviour and performance analysis of distributed systems*, IEEE Trans. on Software Engineering, Vol. 16, No. 2, Feb. 1990.

[HeBr89] D. Helmbold and D. Bryan, *Design of run-time monitors for concurrent programs*, Technical Report, No. CSL-TR-89-395, Computer Systems Laboratory, Stanford Univ., Oct. 1989.

[HoCu87] P. A. Hough and J. E. Cuny, *Belvedere: prototype of a pattern-oriented debugger for highly parallel computation*, Proc. of 1987 Intl. Conf. on Parallel Processing, 1987, pp.735-738.

[HoLa89] D. N. M. Ho, S. W. Lau and F. C. M. Lau, *Efficient tools for transputer monitoring*, Proc. of 2nd Conf. of North American Transputer User Group, Oct. 1989, pp.27-40.

[Inmos83] Inmos Corporation, *Occam Programming Manual*, 1983.

[Inmos89] Inmos Corporation, *The Transputer Databook*, Second Edition, 1989.

[JiWaCh90] J. Jiang, A. Wagner and S. Chanson, *Tmon: A real-time performance monitor for transputer-based multicomputer*, To appear in Proc. of the 4th Conf. of North American Transputer Users Group, Oct. 1990.

[Joyce87] J. Joyce, *et al*, *Monitoring distributed systems* ACM Trans. on Computer Systems, Vol. 5, No. 2, May 1987, pp.121-150.

[KaFl90] A. Karp and H. Flatt, *Measuring parallel processor performance*, Communication of ACM, Vol. 33, No. 5, May 1990.

[KeSc87] T. Kerola and H. Schwetunm, *Monit: A performance monitoring tool for parallel and pseudo parallel programs* ACM Performance Evaluation Review, Vol. 15, No. 1, pp.163-174.

[Lamport78] L. Lamport, *Time, clocks, and the ordering of events in a distributed system*, Communication of ACM, Vol. 21, No. 7, July 1978.

[LeRo85] R. J. LeBlanc and A. D. Robbins, *Event-driven monitoring of distributed programs*, Proc. of 5th Intl. Conf. on Dist. Comp. Syst., May 1985, pp.515-522.

[LeMe87] T. J. Leblanc and J. M. Mellor-Crummey, *Debugging parallel programs with instant replay*, IEEE Trans. on Computers, Vol. C-36, No. 4, April 1987.

[LiCaVl87] M. A. Linton, P. R. Calder and J. M. Vlissides, *InterViews: A C++ Graphical Interface Toolkit*, Proc. of the USENIX C++ Workshop, Nov. 1987.

[Malony89] A. D. Malony, *et al*, *An integrated performance data collection, analysis and visualization systems*, Dept. of Computer Science, Univ. of Illinois, Rpt. No. UIUCDCS-R-89-1504, March 1989.

[MaPi88] A. D. Malony and J. R. Pickert, *An environment architecture and its use in performance data analysis*, Center for Supercomputing Research and Development, Univ. of Illinois, Rpt. No. 829, Oct. 1988.

[McDaniel77] G. McDaniel, *METRIC: A kernel instrumentation system for distributed environment*, Proc. of 6th ACM Symp. on Operating System Principles, Nov. 1977. pp.93-99.

[McHe89] C. E. McDowell and D. P. Helmbold, *Debugging concurrent programs*, ACM Computing Surveys, Vol. 21, No. 4, Dec. 1989.

[Miller84] B. P. Miller, *Performance characterization of distributed programs*, Technical Report, Computer Science Division(EECS), Univ. of California, Berkeley, TR No. UCB/CSD 84/197, Aug. 1984.

[Miller88] B. P. Miller, *DPM: A measurement system for distributed programs*, IEEE Trans. on Computers, Vol. 37, No. 2, Feb. 1988.

[Miller90] B. P. Miller, *et al*, *IPS-2: the second generation of a parallel program measurement system*, IEEE Trans. on Parallel and Distributed Systems, Vol. 1, No. 2, Apr. 1990.

[MiMa86] B. P. Miller, C. Macrander and S. Sechrest, *A distributed program monitor for Berkeley Unix*, Software - Practice and Experience, Vol. 16, No. 3, March 1986, pp.225-233.

[MiYa87] B. P. Miller, C.-Q. Yang, *IPS: An interactive and automatic performance measurement tool for parallel and distributed programs*, Proc. of 7th Intl Conf Distributed Computing Systems. Sept. 1987.

[Mohan84] J. Mohan, *Performance of parallel programs: model and analyses*, Ph.D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ., July 1984.

[Nelson89] H. Nelson, *Experience with performance monitors*, Instrumentation for Future Parallel Computing Systems, Addison-Wesley, 1989.

[OgSc85] D. Ogle and K. Schwan, *The real-time collection and analysis of dynamic information in a distributed system*, Technical Report, Computer and Information Science Research Center, Ohio State Univ., OSU-CISRC-TR-85-12, Sept. 1985.

[Parasoft88] Parasoft Corporation, *An overview of the EXPRESS system*, Technical Notes, 1988.

[Reed89] D. Reed, *Distributed Memory Working Group Summary*, Instrumentation for Future Parallel Computing Systems, Addison-Wesley, 1989

[ReFu87] D. Reed and R. Fujimoto, *Multicomputer networks: message-based parallel processing*, The MIT Press Series in Scientific Computation, 1987.

[Sart85] S. Sartzetakis, *et al*, *A real-time multiprocessor performance monitoring tool*, Proc. of IEEE Electroniccom 1985, Oct. 1985, pp.104-108.

[SeRu85] Z. Segall and L. Ruldolph, *PIE: A programming and instrumentation environment for parallel processing*, IEEE Software, Vol. 2, No. 6, Nov. 1985, pp.22-37.

[Shea89] D. G. Shea, *et al*, *Monitoring and simulation of processing strategies for large knowledge bases on the IBM Victor multiprocessor*, Proc. of 2nd Conf. of the North American Transputer User Group, Oct. 1989, pp.11-26.

[Shephard86] R. Shephard, *Extraordinary use of transputer links*, Inmos Technical Notes, Nov. 1986.

[Shumway89] M. Shumway, *Synchronizing clocks in multi-transputer networks*, Inmos Technical Notes, Aug. 1989.

[Snodgers88] R. Sondgers, *A relational approach to monitoring complex systems*, ACM Trans. on Comp. Syst., Vol. 6, No. 2, May 1988.

[SpKe88] M. Spezialetti and J. Kearns, *A general approach to recognizing event occurences in distributed computations*, Proc. of IEEE 8th Intl. Conf. on Dist. Comp. Syst., June 1988, pp.300-307.

[Sterling88] T. Sterling, *et al*, *Multiprocessor performance measurement using embedded instrumentation*, Proc. of IEEE Intl. Conf. on Parallel Processing, Vol. 1, Aug. 1988, pp.156.

[VoZe90] O. Vornberger and K. Zeppenfeld, *Graphical visualization of distributed algorithms*, Proc. of 3rd Conf. of North American Transputer Users Group, Apr. 1990, pp.223-234.

[YaMi88] C.-Q. Yang and B. P. Miller, *Critical path analysis for the execution of parallel and distributed programs*, Proc. of 8th IEEE Intl. Conf. on Distributed Comp. Syst., June 1988, pp.482-489.

[Zenith90] S. E. Zenith, *Linda coordination language; subsystem kernel architecture (on transputers)*, Research Report, Dept of Computer Science, Yale University, YALEU/DCS/RR-794, May 1990.

Appendix A

Architecture of the Transputer-based Multicomputer

The architecture of the IMS T800 transputer is shown in Figure A.1. The processor speed of all T800 transputers are pin-selected to 20 MHz. The speed of all bi-directional links are set to 20 Mbits/sec.

The architecture of the IMS C004 link switch is shown in Figure A.2. The speed of all C004 switches in the system are set to 20 Mbits/sec.

The physical connections of the transputers, crossbar switches and VME interfaces are shown in Figure A.3. The transputers are connected to the Sun 4 workstation through a IMS B011 board and a CSA Part 8 Interface Board. There are six links on the CSA Part 8 board. The four buffered links are directly connected to the transputers, and the two unbuffered links are connected to the daisy chain of the configuration links of the crossbar switches. Therefore, there are five independent data channels between the transputers and the host. There are 74 T800 transputers and 10 C004 switches in the array of transputers and crossbar switches. The first 10 transputers and first 2 switches are placed in one box, with the remaining transputers and switches in another larger box. There are 8 connections between the two boxes. The transputers in the larger box are numbered from 0 to 64, and the switches are numbered from 0 to 7. Link 0 of transputer i are directly connected to that of transputer $i+1$. Link 1, 2 and 3 of transputer i is connected Switch i, switch *succ*(i) and switch *pred*(i) respectively. All transputers are partitioned into five reset groups. Therefore, up to five users can use the transputer-based multicomputer simultaneously.

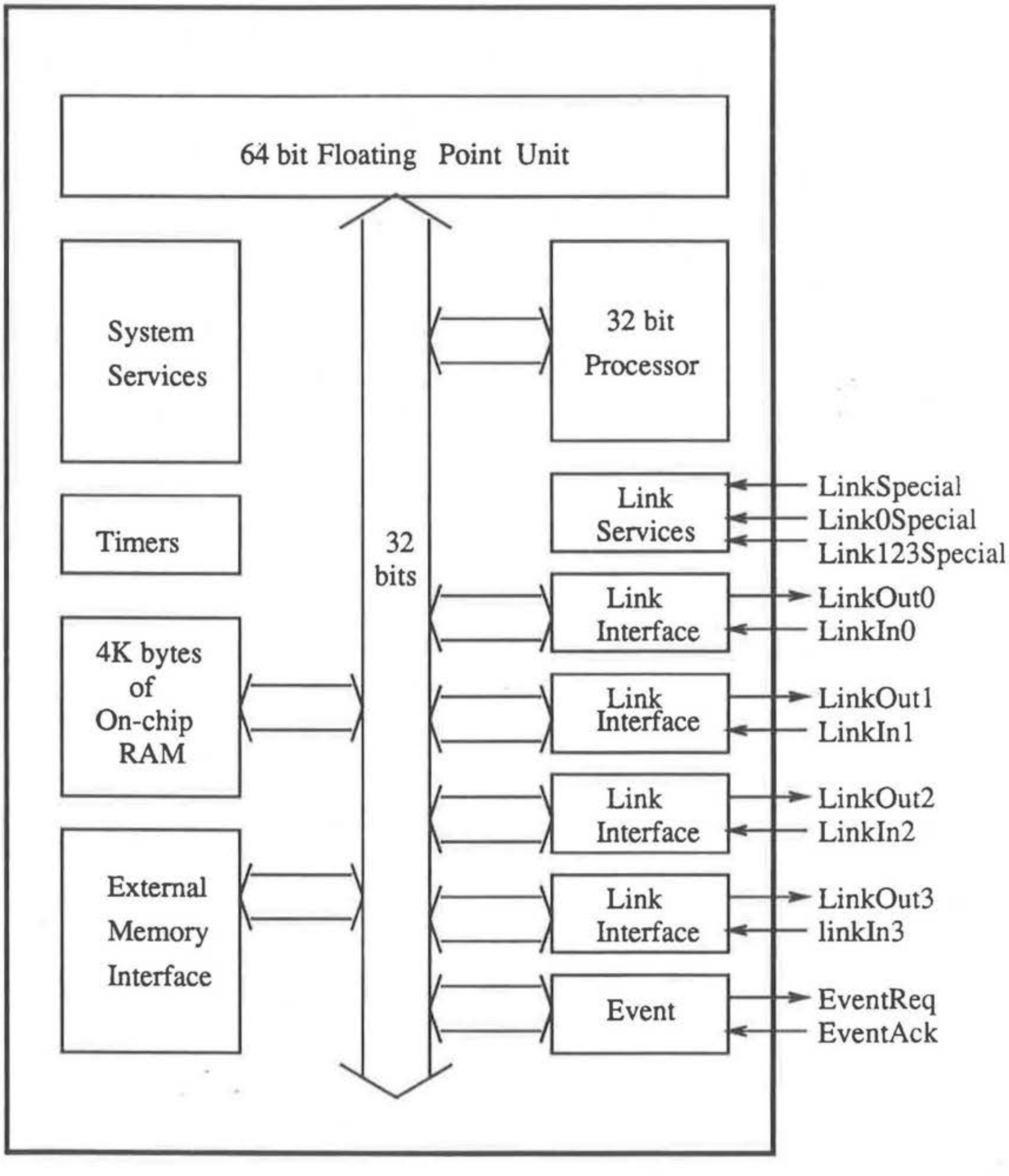


Figure A.1: Architecture of the IMS T800 Transputer

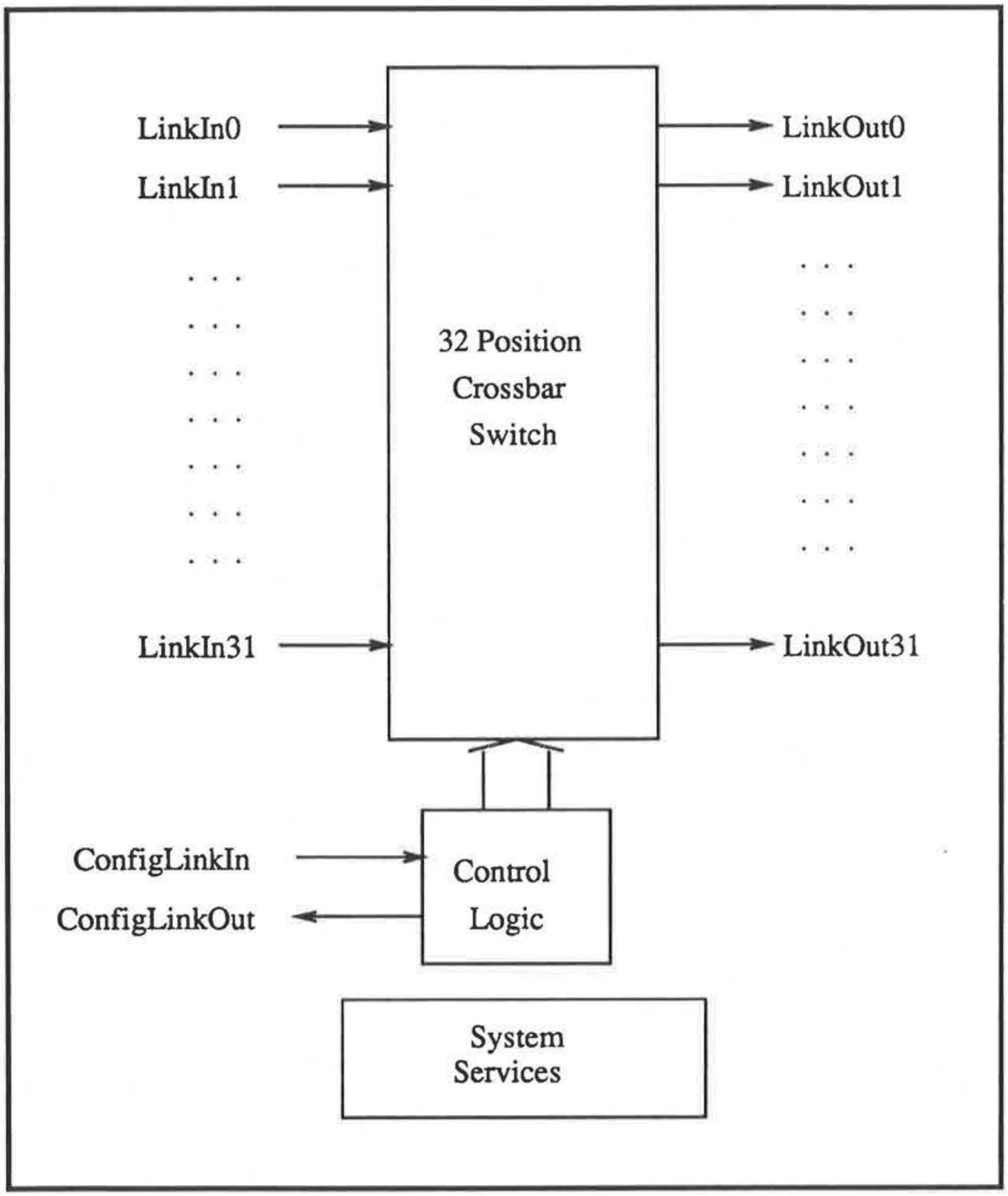


Figure A.2: Architecture of the IMS C004 Crossbar Switch

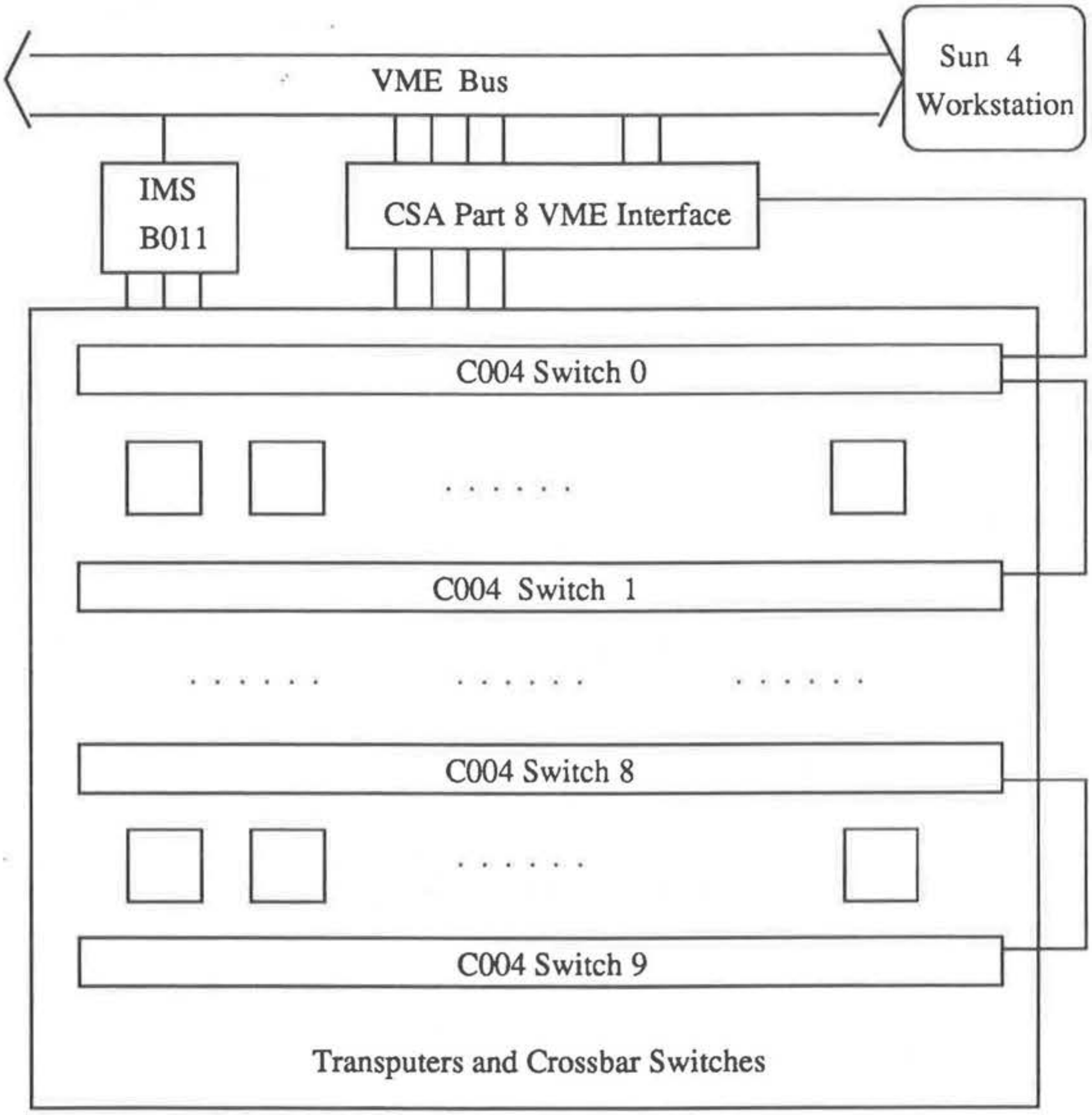


Figure A.3: Physical Connections of the Transputers and the Switches

Appendix B

Modifications to Trollius Run-time Library

The following routines are inserted into the Trollius run-time library and replace existing ones. They are used to generate the five types of standard events defined in Section 4.5 as well as user-defined events.

B.1 Definition of Monitor Parameters

```
    ......

/* event type for monitoring message */

#define MON_CMD         -100
#define MON_RES         -101
#define MON_TRACE       -102

/* monitor controlling command */

#define MON_BEG         0
#define MON_END         1

typedef struct TraceEntry
{
        char   tag;
        int    data[NUM_REGS];
} TraceEntry;
```

```
typedef TraceEntry *TracePtr;

/* all events to be monitored defined here */

#define NODE_USAGE      '\000'
#define MSG_SEND        '\001'
#define MSG_RECV        '\002'
#define RECV_CALL       '\003'
#define PROC_INIT       '\004'
#define PROC_EXIT       '\005'
#define OVERFLOW        '\006'

......
```

B.2 Probes to Generate Message Events

```
int msg_mon(nheader, trace_type)
struct nmsg* nheader;
char trace_type;
{
        struct kmsg     kheader;
        TraceEntry      trace_buf;

        trace_buf.tag = trace_type;
        trace_buf.data[1] = ltot(getpid());
        trace_buf.data[2] = ltot(nheader->nh_event);
        trace_buf.data[3] = ltot(nheader->nh_node);
        trace_buf.data[4] = ltot(nheader->nh_length);

        kheader.k_event = MON_TRACE;
        kheader.k_type = 0;
        kheader.k_flags = 0;
        kheader.k_length = sizeof(trace_buf);
        kheader.k_msg = (char *) &trace_buf;

        if (ksend(&kheader))
            return(errno);
        return(0);
}
```

```
int nsend(header)
struct nmsg* header;
{
        msg_mon(header, MSG_SEND);
        header->nh_data[0] = getnodeid();
        return(do_nsend(header, NSEND));
}

int nrecv(header)
strcut nmsg* header;
{
        int     err_code;

        msg_mon(header, RECV_CALL);
        err_code = do_nrecv(header, NRECV);
        header->nh_node = header->nh_data[0];
        msg_mon(header, MSG_RECV);
        return(err_code);
}
```

B.3 Probes to Generate Process Events

```
int mon_pinit()
{
        struct kmsg     kheader;
        TraceEntry      trace_buf;
        char*           pname;

        trace_buf.tag = PROC_INIT;
        trace_buf.data[1] = ltot(getpid());
        pname = (char *) &(trace_buf.data[2]);
        GetProcName(pname);

        kheader.k_event = MON_TRACE;
        kheader.k_type = 0;
        kheader.k_flags = 0;
        kheader.k_length = sizeof(trace_buf);
        kheader.k_msg = (char *) &trace_buf;

        if (ksend(&kheader))
```

```
            return(errno);
        return(0);
}

int mon_pexit()
{
        struct kmsg     kheader;
        TraceEntry      trace_buf;

        trace_buf.tag = PROC_EXIT;
        trace_buf.data[1] = ltot(getpid());

        kheader.k_event = MON_TRACE;
        kheader.k_type = 0;
        kheader.k_flags = 0;
        kheader.k_length = sizeof(trace_buf);
        kheader.k_msg = (char *) &trace_buf;

        if (ksend(&kheader))
            return(errno);
        return(0);
}

int kinit(priority)
int priority;
{
        int     retcd;

        ......
        retcd = kattach(priority);
        mon_pinit();
        return(retcd);
}

void kexit(status)
int status;
{
        ......
        mon_pexit();
        _kexit(status);
}
```

B.4 Probes to Generate User-defined Events

```
int probe(probe_type, aux_info)
char probe_type;
char* aux_info;
{
        struct kmsg     kheader;
        TraceEntry      trace_buf;
        char*           aux_buf;
        char            aux_len;

        trace_buf.tag = probe_type;
        aux_buf = (char *) &(trace_buf.data[1]);
        aux_len = MIN(strlen(aux_info), MAX_AUX_LEN);
        strncpy(aux_buf, aux_info, aux_len);
        aux_buf[aux_len] = '\0';

        kheader.k_event = MON_TRACE;
        kheader.k_type = 0;
        kheader.k_flags = 0;
        kheader.k_length = sizeof(trace_buf);
        kheader.k_msg = (char *) &trace_buf;

        if (ksend(&kheader))
            return(errno);
        return(0);
}
```

B.5 Monitor Controlling Routines

```
int mon_control(mon_cmd)
int mon_cmd;
{
        struct nmsg header;

        header.nh_node = MASTER;
        header.nh_event = MON_CMD;
        header.nh_type = 0;
        header.nh_flags = 0;
        header.nh_length = 0;
        header.nh_msg = NULL;
        header.nh_data[0] = mon_cmd;

        if (nsend(&header))
            return(1);
        else
            return(0);
}

int startmon()
{
        return(mon_control(MON_BEG));
}

int stopmon()
{
        return(mon_control(MON_END));
}
```