A Performance Monitoring and Analysis Environment for Distributed Memory MIMD Programs

Kayhan İmre

Ph. D.

University of Edinburgh

1993

Abstract

This thesis studies event monitoring techniques that are used for collecting, filtering and visualising event traces from parallel programs. Implementations of two experimental monitoring systems are presented. The first system is a hybrid implementation which uses extra hardware to collect event traces. The second system is a software implementation which was implemented on the Edinburgh Concurrent Supercomputer. These two systems can gather event traces from the parallel programs at a very low cost. The event abstraction mechanism is used for filtering these event traces. The generic and application-specific performance metrics are achieved by using event abstraction techniques to replace event patterns with new abstract events which are used for visualising performance related behaviour. The strengths and weaknesses of the event abstraction approach are discussed in the context of performance analysis and visualisation of message passing parallel programs.

Table of Contents

1. Performance Monitoring and Analysis Problem **1**

2. Fundamental Concepts **5**

2.1 Introduction 5

2.2 Problem Space 6

2.2.1 Parallel Architectures 6

2.2.2 Parallel Programming 9

2.3 Choice of Data to be Monitored and Analysed 10

2.4 Performance Tuning 11

2.5 Performance Observation Tools 11

2.5.1 Easy to Use Tools 12

2.5.2 Direct Support for Tuning Experiments 12

2.5.3 Application-specific Visualisation 14

2.5.4 Other Software Engineering Issues 15

2.6 Summary 16

3. Instrumentation and Monitoring **18**

3.1 Introduction 18

3.1.1 Event Sampling 19

3.1.2 Event Tracing 20

3.1.3 Passive Monitoring 20

3.2 Event Monitoring 21

3.3 A Hardware Approach — Instrumentation for the POSIE Testbed 24

3.3.1 The POSIE Testbed 24

3.3.2 POSIE Monitoring Hardware 30

3.3.3 Analysis of the POSIE Performance Monitoring System . . 35

3.4 A Software Approach — Instrumentation for the Edinburgh Concurrent Supercomputer 49

3.4.1 The ECS and CS-Tools 49

3.4.2 The ECS Performance Monitor 50

3.4.3 Analysis of ECS Performance Monitoring System 57

3.5 Comparison between POSIE and ECS Monitoring Systems 60

3.6 Summary 65

4. Performance Data Filtering **66**

4.1 Introduction 66

4.2 Event Abstraction Technique 67

4.2.1 Event Streams 68

4.2.2 Event Patterns 70

4.2.3 Front-ends for the Event Abstraction Mechanism 80

4.2.4 Event Abstraction Examples 90

4.3 Uses of Event Abstraction in Performance Tuning 99

4.3.1 Standard System Metrics 99

4.3.2 Application Specific Metrics 101

4.3.3 Correlating Event Histories 102

4.4 Summary 103

5. Performance Data Visualisation **105**

5.1 Introduction 105

5.2 Visualising Abstract Events 107

5.2.1 Performance Displays 108

5.2.2 Virtual and Real Parallelism 126

5.3 Performance Visualisation Exemplars 129

5.3.1 A Skeletal Program: Broadcast and Gather Operations . . 129

5.3.2 Matrix Multiplication Optimiser 136

5.4 Summary 142

6. Summary and Conclusions **143**

Bibliography **149**

A. Published Papers **158**

Chapter 1

Performance Monitoring and Analysis Problem

Parallel programs are difficult to implement compared with their sequential counterparts. The difficulty arises when the programmer wants to debug a program that contains competing parallel processes. Because of the loose connectivity between parallel processes, bugs in a program may not surface every time the program runs, and depending on the underlying parallel architecture, programs can change their behaviour. This property of parallel programs makes them difficult to monitor and analyse. Once the programmer implements the parallel program correctly, (s)he faces another problem: performance. It is not easy to achieve the expected speed-up straightaway. The next task to carry out is performance tuning. This task involves monitoring execution(s) of the parallel program, and analysing the results of monitoring. Depending on the results of the analysis, the program may require some modifications to make it run faster. When the programmer carries out a modification, it is also possible to create new bugs in the program. As one can imagine, the parallel program development cycle is not straightforward, and it requires extensive facilities to analyse a parallel program. Therefore, it is crucial to get information about the behaviour of the program that is under development. In this thesis, event monitoring systems are considered as

the main sources of information to analyse parallel programs. These monitoring systems collect important events of parallel program executions, and they can create an execution history for each run of a parallel program. These events can help the user to understand the timing characteristics of a program, and to associate higher level program constructs with these characteristics. Since the data obtained by monitoring is at a low level compared with the high level program constructs, special tools are needed to assist the user in extracting useful information from the low level data. These kinds of tools are crucial in performance tuning when there are large numbers of processes to analyse.

The limits on the scope of the work described in this thesis are as follows:

- Distributed memory MIMD systems are studied.
- CSP [27] type programs are considered i.e. sequential processes that communicate with each other using blocking send/receive operations.
- Event monitoring is the basis for collecting performance data.
- Either the event monitoring system provides a global timer, or local timers that are precise enough to monitor an interval of a parallel program execution are used.
- Postmortem performance analysis and visualisation tools are considered.

The research described in this thesis consists of three consecutive phases each of which corresponds to a task in examining the executions of parallel programs; monitoring, filtering and visualising. Respectively, chapters 3, 4 and 5 are dedicated to these phases.

In Chapter 2, the background information about the subject is presented, and the goals of the research are explained in the context of existing performance monitoring and visualisation approaches.

In Chapter 3, the first phase of the project, monitoring parallel systems, is presented. In this phase, two monitor implementations were carried out by the author. The first implementation was a hardware supported software monitor (i.e. hybrid monitor) for a distributed memory parallel computer testbed. Because of on-going operating system development, this system has not been used for experimenting with the real parallel programs. Only several low level measurements have been made to assess the characteristics of the testbed. However, the implementation served as a preparatory project in which the problems with monitoring and analysing parallel programs were identified. The experience which was gained during this stage was used in the implementation of a monitor for the Edinburgh Concurrent Supercomputer (a Transputer-based machine). This implementation was a software instrumentation which could collect events from parallel programs at an acceptable time penalty. It has been used to do the experiments involving realistically large numbers of processes. Later, another version of this instrumentation was implemented (by another person with collaboration with the author) to overcome memory limitations but, in most cases, this implementation caused unacceptable levels of disturbance.

Chapter 4 is dedicated to the problem of extracting information from the event traces collected by the parallel program monitors. The filtering provides a bridge between collecting event traces (i.e. monitoring) from parallel programs and presenting results to the user (i.e. visualisation). Filtering is achieved through the event abstraction mechanism which fills the gap between very low and high level information by providing abstraction levels in between them. In Chapter 4, other uses of the event abstraction mechanism, such as defining performance metrics, and application specific abstractions are presented in detail. Two prototype implementations were developed and used as research tools to conduct experiments with large event traces.

The last phase of the project, performance visualisation, is presented in Chapter 5. The approach which is developed in this stage differs from the existing visual-

isation approaches in that performance displays are not dedicated to any specific performance information. These displays are designated to visualise different aspects of the abstract events. This gives the user ability to define what to visualise through abstract events. In this approach, the "resource" concept is not restricted to obvious resources such as processors and communication links, but anything which can be represented by an abstract event is considered as a "resource". This means that user can define abstract software resources which use computational and/or communication-related resource, and use these abstract resources as a target to be optimised.

In this phase of the project, a general purpose graph plotting utility (i.e. Gnuplot) was successfully exploited to implement the performance displays.

Chapter 5 also presents results of experiments which cover all three phases of the project. In these experiments, two exemplar programs were run on ECS. The monitoring software, abstraction and visualisation tools were demonstrated on those exemplar programs.

Chapter 2

Fundamental Concepts

2.1 Introduction

Parallel computing covers several completely different architectures and programming paradigms. This diversity of architectures and programming paradigms makes it difficult to develop standard techniques for writing, debugging and performance tuning parallel programs. Most of the programming paradigms contain architecture dependent features which enable a user to develop efficient programs for a specific architecture. Up to date, neither the programming paradigms which are independent from the underlying architecture [3,4,11], nor parallelising compilers have shown significant progress towards developing fast parallel programs. However, there are several areas in which gradual progress has been made towards less architecture dependent parallel programming without sacrificing performance. For example, connectivity of processing elements is becoming less visible to a user than it was. The aim is to simplify the task of parallel programming. Parallel programming is still a difficult task in spite of advanced programming environments and very powerful parallel computers. The programmer needs advanced tools to support his or her programming effort in writing fast parallel programs. Since it is very difficult to design tools which support several parallel architectures and programming models, the performance observation tools should specialise in a particular domain of parallel computing. In this chapter, the work domain which is the subject of this thesis is presented.

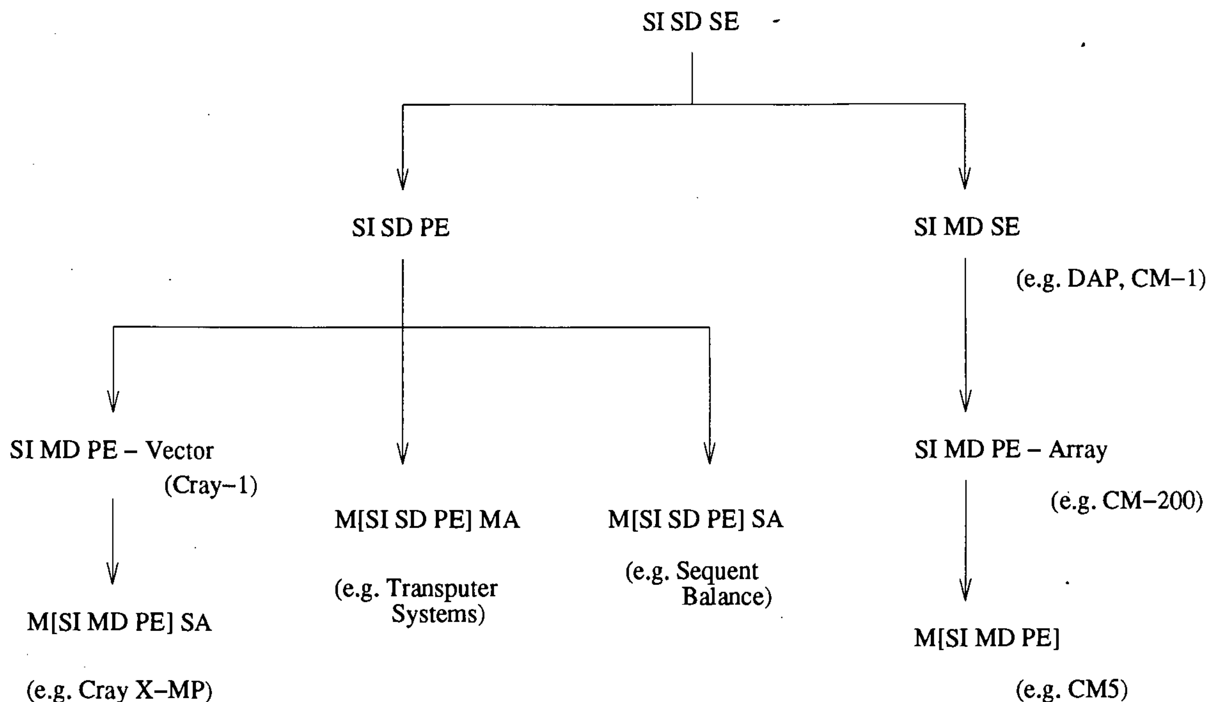


Figure 2–1: A genealogical chart of parallel architectures

2.2 Problem Space

So far in this thesis, the terms *parallel computer* and *parallel program* have been used without reference to the kind of parallel computer or parallel programming involved. In this section, assumptions about these concepts are clarified.

Unless otherwise stated, *parallel computer* refers to a distributed memory MIMD computer, and *parallel programming* refers to the message passing parallel programming paradigm.

2.2.1 Parallel Architectures

In Figure 2–1, a genealogical chart of parallel architectures is depicted [30]. This chart is a refinement of Flynn's taxonomy [16]. Parallel computers have evolved

from Single Instruction Single Data (SISD) computers to overcome the sequential processing bottle-neck. First parallelism came in the form of multiple arithmetic units. Two different approaches have been taken by different groups to provide multiple arithmetic units. The first approach provided low level parallelism, which was invisible to the programmer, by executing arithmetic operations of a sequential code in parallel (SISD PE : SISD computer with parallel execution units). Since this kind of parallelism was limited, in the other approach, parallelism was made visible to the programmer, and parallel arithmetic operations had to be expressed explicitly in the program in the form of array operations (SIMD SE : SIMD computer with serial execution units). These two approaches were not enough for highly parallel general purpose computations. By providing multiple processing elements, Multiple SISD (MSISD) and Multiple SIMD (MSIMD) computers have been built. MIMD computers (MSISD) have further divided into two different classes: Shared Memory MIMD (MSISD SA computers[1]) and Distributed Memory MIMD (MSISD MA computers [2]).

Since Distributed Memory MIMD computers (Figure 2–2) are general purpose, and can be scaled up [1,26,54], they serve to a wider user community than other parallel computers do [9].

Important Properties of Distributed Memory MIMD Parallel Computers

The speed of distributed memory MIMD parallel computers depends on the number of processing elements as well as the speed of individual processing elements and the communication network.

---

[1]SA : shared address space

[2]MA : multiple address space

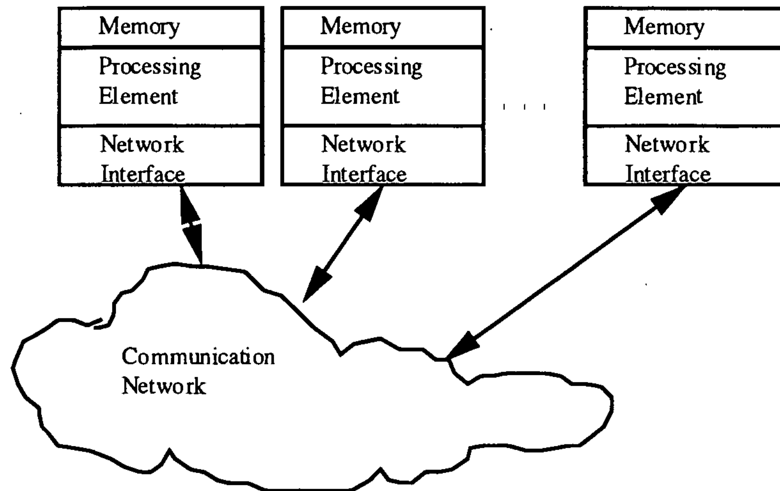


Figure 2–2: Distributed Memory MIMD Architecture

The number of processing elements which can be housed in a distributed memory MIMD computer depends on the topology of the communication network and the communication channels available on the processor boards. For example, with four channels on each processor board, it is possible to to build a binary-hypercube with up to 16 nodes, or a scalable mesh which may contain any number of processors.

The diameter of a network is the longest distance between any two nodes in the system, and the diameter is one of the parameters which can give an idea about the predicted performance of the network and therefore, the performance of the parallel computer.

Another important parameter is the intermediate channels which are used in the routes between nodes. The more nodes share the same channels, the less scalable is the system. Sharing intermediate routing channels makes the system open to channel contention. The channel contention problem can be a limitation to the scalability of a topology. For example, a mesh can be as big as desired, but bigger meshes are more sensitive to the channel contention than smaller meshes. Therefore the programs running on a mesh may not scale as well as the hardware scales.

2.2.2 Parallel Programming

Message passing is the most suitable, or at least the most popular, programming paradigm for distributed memory MIMD systems since the message passing programming paradigm is a software reflection of the architecture of a distributed memory MIMD system. Even though it is possible to implement a message passing programming environment on a shared memory MIMD system, or a shared memory programming environment on a distributed memory MIMD system, the efficiencies of these hybrid systems are lower than message passing systems on distributed memory MIMD systems.

A message passing program consists of processes and soft channels which connect the processes. When a parallel program is mapped on to a distributed memory parallel computer, processes are mapped on to processors and soft channels are mapped on to hard channels which constitute the communication network.

It is also possible to map processes on to processors dynamically, and to create and terminate processes while the program is executing. This kind of dynamic management of processes has not been very successful because there is no general purpose technique to balance the load of a parallel program without having a large overhead which wipes out the benefits of balancing the load. In this thesis, parallel programs with static mapping are investigated.

When soft channels are mapped on to hard channels, the operating system handles the delivery of the messages. Even though the message delivery system cannot be controlled directly from the program, it is possible to improve its performance indirectly by changing the process mapping. Some routers can optimise the message routing by finding alternative routes to avoid possible bottle-necks. It is sometimes very difficult to predict the performance of a dynamic router.

The number of soft channels is not restricted to the number of hard channels. It is always possible to have more soft channels than hard channels. Soft channels

are also called virtual channels, by analogy with the virtual memory concept. Multiple virtual channels can timeshare a hard channel.

2.3 Choice of Data to be Monitored and Analysed

Monitoring parallel programs is difficult because of the probe effect [18,46]. The monitored and unmonitored execution of the same program may have different performance characteristics if the amount of information to be collected is too high. The problem is to keep the monitoring information as small as possible, and at the same time, to collect enough information to be able to analyse the performance.

A number of metrics can be used to characterise the performance of a parallel program. A metric can be a single number or a set of numeric or alphanumeric data which describes the behaviour of a parallel program (profiling metrics) [29].

The most useful single value metric is parallelism. The parallelism is

$$p = T_p/T \tag{2.1}$$

where T_p is the total processor time which is spent for computation and T is the execution time of the program. The execution time is

$$T = (T_p + T_{wait})/N \tag{2.2}$$

where T_{wait} is the total waiting time and N is the number of processors. The total waiting time is the scheduler waiting time when there is no process to run.

Profiling metrics are not so simple to present. They may vary from a very simple definition of a process behaviour to the critical path [50,69] of the computation graph. All these profiling metrics require detailed traces of a parallel program. Communication related traces are especially useful in this case because one can study the balance between communication related and computational stages of a parallel program.

2.4 Performance Tuning

Performance tuning is an experimental task which involves observing the run-time behaviour of a parallel program. If the performance is unsatisfactory, the programmer investigates the possible reasons for the poor performance, then alters the program to make it faster. Every time a modification is carried out on a program, the observations must be repeated to ensure that the result expected from the last modification is achieved. This cycle continues until a satisfactory result is reached; experience shows that this is quite a difficult task, and it may take a long time.

2.5 Performance Observation Tools

In this section, we discuss performance visualisation problems in the context of performance tuning. We start with a wish list of a programmer who wants to tune his/her program, and needs help:

- easy to use tools,
- direct support for tuning experiments,
  - experiment automation,
  - experiment correlation,
- application-specific visualisation,
- other software engineering issues,
  - portability,
  - performance of the tools.

2.5.1 Easy to Use Tools

In the literature, 'easy to operate' tools are considered as 'easy to use' tools [23, 47]. This can be true to some extent. The real issue is not the initial effort to run the tool but how to interpret the results that are presented using this tool. Firstly, the user should be able to interpret the results from a large program as if (s)he were interpreting the results from a small program (i.e. the complexity of a performance view should not increase drastically when the programs get bigger). Secondly, the results must not require a large amount of interpretation (i.e. the right abstraction level which reveals the bottle-neck is desirable). Most existing tools are usually simplistic and restrictive in nature; facilities for experienced users are either completely missing or lack simplicity. For example, if the user wants to customise the tool (s)he is given no choice but to implement the required extension to the tool. The recommended choice of environment is the programming environment which is used for implementing the tool itself. Doing this requires a good knowledge of the implementation details, which is not very practical [23,37]. Thus advanced users should be provided with options to enable them to carry on more detailed analysis, or to customise the tool for keeping up with the developing requirements of the parallel program tuning task [42].

2.5.2 Direct Support for Tuning Experiments

The major failing of existing performance visualisation tools is that they are data-driven: given a large number of monitoring traces, the tools aim to allow one to visualise this information in an intelligible form [32]. Since performance visualisation tools are mainly for performance tuning purposes, we have to investigate the nature of the tuning task. This is usually repetitive and experimental, involves more than one version of the program, and uses several different configurations of the hardware platform. We believe that if the information from each iteration of the tuning task is integrated, we can achieve better results with highly parallel

systems in a relatively short time. With the current visualisation tools, it is the user's responsibility to correlate information from different monitoring traces, and therefore correlating information is restricted since it is carried out manually using another tool to present the results. In the following sections, we explain what kinds of information can be extracted from multiple monitoring traces.

Automation of Experiments

The performance tuning task consists of repetitive steps in which a programmer runs his/her program, monitors it, interprets the results, and modifies the program to make it faster. Even though the program changes after each modification, the programmer will usually be investigating similar features of it in the next tuning iteration. There are two reasons for carrying out such experiments. Firstly, the programmer wants to know whether the last modification delivered the expected results. Secondly, further tests may be required to see that the program behaves as expected when different data sets or system configurations are used. Performance visualisation tools must include some provision to allow users to define what kind of information they are seeking. If these definitions can be used for several tests, the programmer can quickly carry out many experiments.

Experiment Correlation

As well as repeating the same analysis on several versions of a program, collecting monitoring data from several runs of the same program and analysing it as one piece of monitoring information can improve understanding of a program [41]. The most obvious application of this idea is scale-up experiments. In these experiments, the programmer wants to know how a program behaves when it is mapped onto bigger systems. The program under examination is run several times; each time, the number of the processing elements is increased. Then, execution times of the runs are used to draw a graph to show how the program scales up with the

system size. None of the performance visualisation tools reported to date support this task; it must instead be done manually using a graph drawing tool. Similar, but extended graph drawing facilities can be integrated into a performance visualisation tool. For example, if program performance breaks off at some point as the number of processors is increased, or there are odd peaks on the graph, the programmer can include other performance metrics in the graph in order to find a possible explanation from this particular behaviour.

2.5.3 Application-specific Visualisation

Detailed performance studies require more specialised support than general purpose performance visualisation tools. With the help of application-specific visualisation, performance tuning experiments can be conducted quickly since the performance displays carry some semantics of the application and the results require very little interpretation. This is very important for monitoring very large programs. Using general purpose performance visualisation tools, the programmer can only capture either fine grain behavioural patterns or high level statistical information such as message counts, execution times, parallelism profile, etc. Fine grain information is not useful for tuning big programs since it is difficult to reason about overall performance just by looking at a small portion of the execution history of a program. Likewise, very high level information does not contain useful clues about behavioural patterns, it just portrays whether the program behaves well or not. Different abstraction levels should be supported by performance visualisation tools [58].

Another area which may benefit from application-specific visualisation is massively parallel computing. Even though the fundamentals are the same, the scale of massively parallel computing invalidates some of the existing performance visualisation techniques, especially those which use low level performance information [39,40].

In [32], [55] and [23], the importance of application-specific performance visualisation tools is strongly emphasised, but present performance visualisation tools are not flexible enough to allow programmers to define application-specific visualisation.

Sequential program libraries are a big part of sequential program development. In parallel programming, parallel program templates and libraries are expected to be an important part of parallel program development [12]. These templates will increase programmer productivity because the parallel programmers will be able to develop programs by writing small parts for these template programs. Application-specific visualisation can provide performance views which are specific to a template program. These performance views can be archived along with the template programs. This helps the programmer to visualise and analyse the performance of an application which is implemented by modifying a special template program.

2.5.4 Other Software Engineering Issues

Along with the issues which are specific to performance visualisation, a performance visualisation tool is expected to conform with the usual software engineering standards. Two important standards are portability and the response time of a performance visualisation tool.

The diversity of available parallel architectures and the software environments for these architectures make it difficult to design portable performance visualisation tools. There are two main reasons for this. Firstly, the software environment in which the visualisation tool is implemented changes from system to system. If standard software environments, such as X Windows, are supported, it is possible to design highly portable tools. Secondly, monitoring information is another factor restricting portability. Every monitoring environment collects its own set of primitive data for representing monitoring information. For event-based mon-

itoring, primitive event types are used for representing primitive operations such as message transmission, process blocking, etc. Since these primitive operations are semantically similar across the different architectures, it is possible to create a super-set of primitive events. Every monitoring environment can use a sub-set of this super-set. Since the meaning of each primitive event is clearly defined, whatever the format of a particular monitoring environment, conversion from a particular format to a standard one can be carried out very easily. This kind of standardisation enables portability of monitoring data.

The response time of a performance visualisation tool is another crucial factor since the amount of performance data gathered can be quite large. The trend towards massively parallel programs is one of the reasons for designing tools that work with a large amount of monitoring data. Depending on the granularity of the information or the nature of a query, tool response time can be very slow with a large amount of data. These properties should be considered very carefully when a tool is designed.

2.6 Summary

Programming distributed memory MIMD computers using the message passing programming paradigm requires well developed skills because there are so many parameters of these programming environments that it is not always possible to find the right combination of parameters to achieve optimum performance of the system. Run-time information about a parallel program can help a programmer to understand the behaviour of the program, and correct this behaviour to achieve better performance. For this performance tuning task, the parallel programmer requires special tools to extract useful information from large volumes of run-time information of a parallel program. The aim of these tools is hopefully to reduce the time spent in performance tuning, and to help programmers to take wiser actions

while tuning the performance. The information presented to a programmer can range from single value metrics to highly application specific profiling metrics which describes a particular behaviour of the program in detail.

Chapter 3

Instrumentation and Monitoring

3.1 Introduction

Collecting performance information from the execution of a parallel program is a difficult task since it is not always possible to provide non-invasive monitoring. The difficulty arises either due to technical or to economic reasons. It is not always possible to convince a computer manufacturer to include very expensive monitoring hardware in a parallel computer. Sometimes the physical structure of a parallel computer does not allow extra hardware to be added for reasons such as close distances between processing elements, heat dissipation, etc. These problems are particularly acute in a parallel computer containing massive numbers of processing elements. For these reasons the following types of instrumentation systems have emerged:

- Software monitors,
- Hardware monitors,
- Hybrid monitors.

Even though software monitors [8,45,48,49,60] are the most intrusive of all, they are widely used for monitoring parallel programs because they can be implemented very cheaply. A software monitor is implemented by embedded code which resides

in either the operating system or the monitored application. The embedded code triggers events, stores event records temporarily in the workspace of the monitored program, and sends them to a central analyser.

Hardware monitors [10,57,62,65] are the least intrusive. A passive hardware unit is connected to the internal bus of processor board or to a communication bus/channel, and monitors the program control and/or the data flow. Sometimes these monitors can slow-down bus/channel cycles to capture data but the main problem with this type of instrumentation is that it requires very sophisticated pattern recognition hardware. To implement such pattern recognition hardware is very expensive, and also, the flexibility and usability of pure hardware instrumentation are restricting factors.

Hybrid monitoring [7,21,22,28,43,52,56,68,70] is a compromise between software and hardware instrumentation. The triggering of monitoring data is carried out by inserted code, and the hardware takes care of the rest. In this chapter, implementations of a hybrid and a software monitor are presented.

Monitoring can be carried out in several different ways, each of which can be accommodated in software, hardware or hybrid instrumentation systems. These monitoring approaches are event sampling, event tracing and passive monitoring [51,64].

3.1.1 Event Sampling

Event sampling is a monitoring technique which periodically checks the status of processing elements and collects statistical information about the activities of each processing element. This can be done by a periodic interrupt which is asserted by the system timer or an additional timer associated with the monitoring device.

Since sampling does not capture complete information about every occurrence of a particular event during the execution of a parallel program, it is impossible

to use this kind of monitoring information for detailed behavioural analysis of a parallel program. Instead, event sampling provides statistical information about the program's activities.

3.1.2 Event Tracing

Event tracing is a monitoring technique which triggers the event collection mechanism whenever the flow of control passes instrumentation code which is embedded in the application or the operating system. In contrast to event sampling, the event tracing technique can capture the precise order of events which occur within a process (sequential code). Triggering of an event causes an event record to be written to the monitoring store or to special monitoring hardware. Event records contain some information about the event type and the time when it occurred. It is also possible to use a passive event triggering mechanism which does not record any event information but triggers hardware which records the status of the processor.

3.1.3 Passive Monitoring

Passive monitoring systems do not require any software triggering, but instead monitor address, data and control busses to find a condition which is defined as a triggering pattern. The pattern can vary from an access of a specific memory location to a series of accesses/instructions which signal a particular event in a parallel program. Defining patterns for a hardware monitor can be problematic because in some cases a pattern can match with unwanted cases, and there may be no way of separating the desired cases from the undesired cases.

Hardware monitors are usually used for monitoring limited properties of parallel computers such as communication traffic and memory access frequencies. It

is difficult to relate very low level information which is collected by a hardware monitor to a parallel program's internal activities.

3.2 Event Monitoring

Event monitoring provides detailed information about parallel program execution. An event history of a parallel program is a valuable source of information that can support a wide range of performance displays ranging from the statistical to fine granularity time-line displays (Gantt Charts). Since fine granularity information is expensive to gather, the monitoring system can significantly disturb the program that is monitored. The problem is to keep the disturbance at an acceptable level, at which the probe effect [18,44,46] does not occur.

Every event in a program history is represented in a primitive event trace that is captured and recorded by the monitoring instrumentation. These event traces contain the instances of a small number of event types that are representatives of various state changes in a computation. Examples of event types that are used in this context are shown in Figure 3–1.

The instances of these primitive events can represent the detailed execution history of a parallel program. Different instances of the same event are differentiated by their time-stamps. Given any instance of an event, the program stage in which the event is signaled can be identified by tracing the events in the history. Since the time-stamps are generated by hardware and there is a global clock signal, total ordering of the events in an execution history can be achieved [35]. These primitive events are the basic building blocks in an execution history, and each of them is an instant in time (i.e. a primitive event occupies zero time). To present only these primitive events alone does not give useful information to the user, however, since it is difficult to observe such an event in a large interval especially if there are many events in that particular interval. Instead of presenting individual primitive

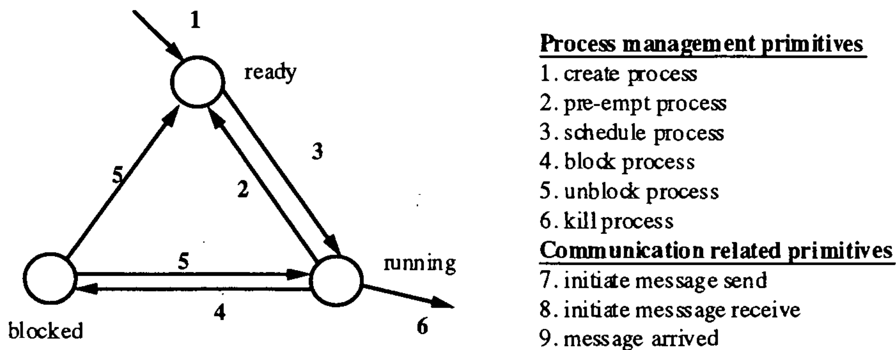


Figure 3–1: Scheduling graph and primitive event types

events, presenting compound events that replace groups of primitive events can be more meaningful to the user. For example, a "schedule process" event immediately followed by a "pre-empt process" event can be replaced by a "running" event that represents an interval during which a particular process is running on a particular processor. Although this simple replacement can dramatically improve the presentation of the history, higher levels of replacements are needed to obtain better history views that relate closely to the steps of execution of a program. If this is done, any information associated with the view also can be easily related to the program. For example, a unique combination of communication patterns can be used as an anchor in a history, and this then becomes a reference point for other events that may be created. Identifying some important stages in a program history is vital because the performance of the program can be analysed by reference to them.

The primitive events can represent the computation graph of a parallel program. This graph represents the dependencies between various parts of the computation. By working out these dependencies, one can extract valuable information from this graph. For example, the critical path of a computation can help the user to find those parts of a program that slow down the computation [47]. The computation graph is an abstraction of primitive events that correspond to the vertices of

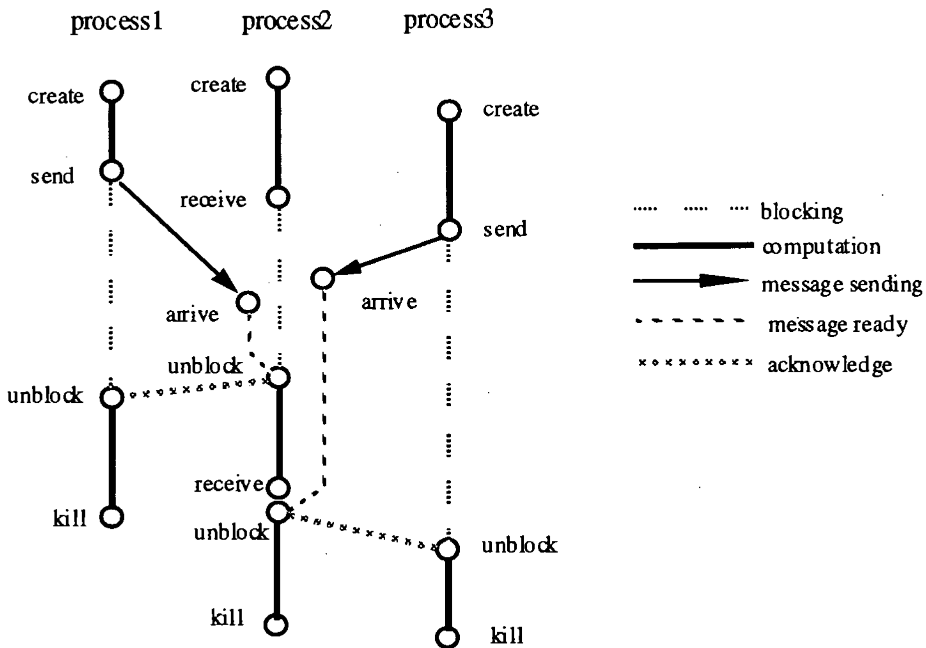


Figure 3–2: A computation graph

this graph. There is an arc between every pair of events that are adjacent within the same process boundary (Figure 3–2).

There is also an arc between two events that are the sending and receiving ends of a message transmission. Since a primitive event is instantaneous, there is no cost attached to it. But there is a cost for an arc between two events. The cost is the absolute difference between the time-stamps of those events that enclose the arc. The events on both ends of an arc can also be used for identifying the type of its cost (See Figure 3–2 for the cost types). For example, a "send message" and "unblock process" pair makes an arc that represents the cost of blocking a process.

3.3 A Hardware Approach — Instrumentation for the POSIE Testbed

3.3.1 The POSIE Testbed

The aim of the POSIE Project is to investigate distributed memory parallel systems. As part of this project a special testbed has been designed. This testbed will be used for monitoring actual distributed memory programs. All the decisions taken since the beginning of this project are aimed towards a hardware supported monitoring environment. The reasons for designing such a system are

- to monitor distributed programs without disturbing them,
- to monitor them in the lowest level of detail (which is not possible with software instrumentation),
- to support real-time analysis of monitored data to achieve run-time decision mechanisms for load balancing.

The POSIE testbed consists of a distributed memory parallel system for running the programs, monitoring hardware for collecting performance data from the system, and a monitor board for the real-time analysis of performance data. The processing part of the testbed consists of processor boards which communicate through a bus structured local area network called Centrenet [31]. Because of the bus structure, there is a direct connection between any processor pair. The aim is not to investigate system design issues such as topology of processors or message routing techniques, which is why this simple connection structure has been chosen to make message passing easy. Besides making message passing easy, there is no need for other hardware topologies because we are investigating software which is

insulated from the actual message routing through the network. It is also possible to emulate different topologies by adding a software layer between the bus driving software and higher software layers.

While a distributed program is running on the processors a network of instrumentation interfaces monitors the whole system. These interfaces are connected to another bus which is separate from Centrenet, and used only for monitoring purposes. The reason for separating the monitor bus from Centrenet is so as not to disturb the computation and the message traffic. In addition to this, the monitoring bus is not an ordinary bus, but it has an additional dedicated function which is time-stamping. Time-stamping is a very important aspect in terms of ordering events which are happening independently on different processors. Of course, time-stamping is also used for calculating statistics such as CPU utilisation, completion time, frequencies of different events etc. Timing is supported by a global clock signal on the monitoring bus.

Performance measurement is done by the monitor using the performance data collected by the monitoring hardware. These data are in the form of individual records representing the state changes, called "events". For example, sending or receiving a message, waiting for a message, or scheduling a process to a processor can be an event. Events are completely software dependent, and they can be created where they are needed. The creation of an event is done by software instrumentation which causes the event record to be passed to the monitoring hardware. The software instrumentation is a piece of code which writes some event information to the monitoring hardware as if it were a memory location. Because simple instructions can be used for creating event records, and the monitoring hardware handles the event records, the software instrumentation overhead is minimum. This approach is a hybrid between pure hardware and pure software instrumentation.

The disadvantage of a pure hardware approach is the inflexibility and the high cost. Because in a pure hardware approach, some kind of pattern-matching unit is used for capturing events, very complex hardware mechanisms are needed. It is also very expensive to design a hardware monitor which is capable of capturing variable patterns. On the other hand this approach does not cause any undesirable intrusion. Software instrumentation is the most intrusive approach but it gives a lot of flexibility to implement events. In this approach, some instrumentation code is inserted into the application program code to create events. While the program is running, the instrumentation code is executed, and monitor software events occur. In order to time-stamp these events, the instrumentation code has to read a hardware or software clock value from somewhere, and add it to each event record. The event records which are created by the instrumentation are stored on the processor board and forwarded to an analyzer via the same network as that used for message communication. As a result of the extra load on the network, performance of the application decreases. The hybrid approach used in POSIE inherits the flexibility of the software approach and the non-intrusivity of the hardware approach. The performance monitoring testbed is shown in Figure 3–3.

Processor boards are connected to Centrenet so that they can communicate with each other, and they are also connected to monitoring interfaces through local busses. Each processor board is monitored by one of these slave monitoring interfaces (SMI). SMIs are connected to the master monitoring interface which controls them. SMIs are responsible for capturing events from processor boards and keeping them until they are read by the master monitoring interface (MMI). The master monitoring interface maintains the global clock signal which is used by the slave interfaces to increment hardware clocks. The master monitoring hardware is connected to the monitor board through another local bus. The monitor board is an ordinary processor board but it is only used for monitoring purposes. Because the analysis programs run on the monitor board, the processing power of the other processors is not used for any task other than application programs. The

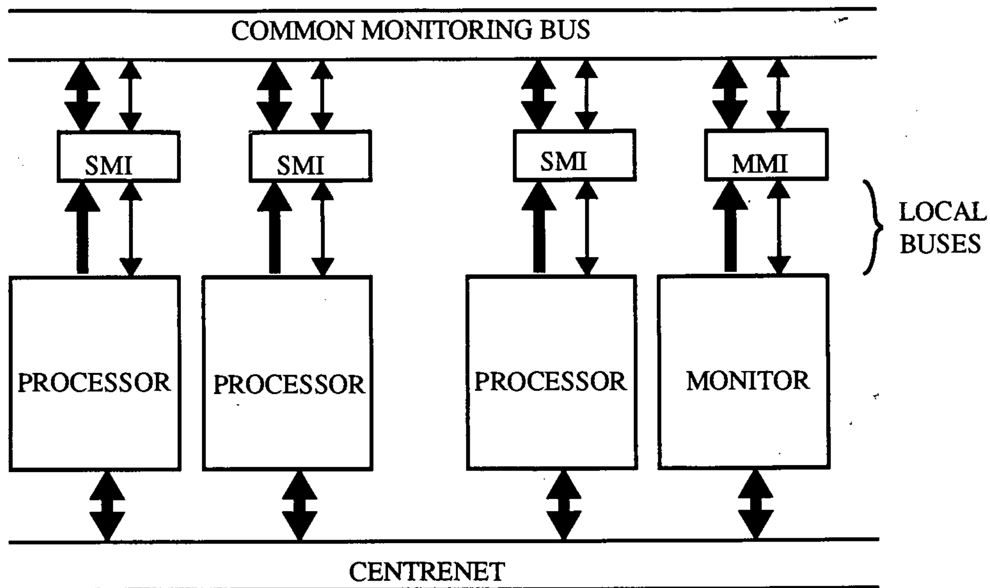


Figure 3–3: POSIE Performance Monitoring Testbed

monitor board is also connected to the local area network, but it does not monitor the network. This can be done by a separate board which monitors messages transmitted via the network. This board only monitors interactions between processes on different processors without causing any change to the operation. The Centrenet connection of the monitor board is only used for loading analysis programs to the board or sending messages to the operating system to take actions such as process migration.

Processor Boards

The processor board consists of four main parts: the processing section, the dynamic memory, the network controller and the local bus drivers for connection to the SMI (see Figure 3–4). The processing section contains an 8 MHz Motorola 68010 processor and a Motorola 68451 memory management unit. Each board carries 2 Mbytes of dynamic RAM, controlled by a National Semiconductor 8422

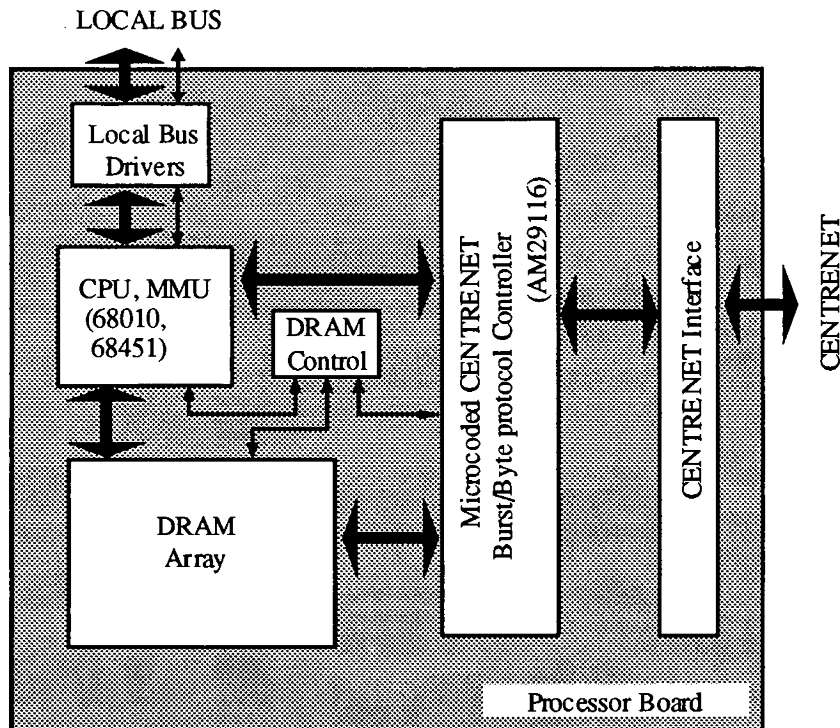


Figure 3–4: The POSIE Processor Board

device, chosen because it supports dual port access to RAM (see below). The average instruction rate is 1.2 MIPS. The local bus drivers are memory mapped devices, and a single write instruction issued by the processor causes a 16-bit event data item to be written to the slave monitoring interface. The overhead of the monitoring instrumentation may be one or two write instructions per event, depending how much event data is needed.

Centrenet is a tree-structured network consisting of nodes (Starpoints) interconnected by optical fibre links. Each Starpoint provides ports for up to 16 devices to connect to the network. The entire POSIE testbed is built into a single Starpoint rack, with each processor board and the monitoring processor on a separate card, connected to a separate Centrenet port. This approach also has the advantage that the built-in Starpoint power supply provides power for the Testbed.

Communication between POSIE processor boards is via the Centrenet Starpoint bus. It involves the sending of messages in the standard Centrenet Burst

Protocol, which supports high bandwidth transport of data blocks up to 32Kbytes in length, with acknowledgement. Communication with the outside world is also based on this burst protocol, while communication between processor boards and terminals use the Centrenet Terminal Protocol, a lower bandwidth protocol.

The Starpoint bus is able to support communication between processors at 10 Mbytes/s actual data rate, after subtracting the overhead of addressing and control information. This rate can be achieved simultaneously between up to 8 pairs of processors on the Starpoint bus. In order to support such a high rate of message transfer without excessively loading the POSIE 68010 processors themselves, the Centrenet Burst Protocol Interface[1] (CBPI) was developed. This interface takes care of all the complexity of the Burst Protocol, including retransmission errors, and runs fast enough to use the full 10 Mbytes^{-1} Starpoint port bandwidth. To minimise the load on the processor, the CBPI operates autonomously, communicating with the processor only through shared memory, and via a single interrupt line in each direction. The entire dynamic RAM space of the processor is accessible to the CBPI DMA controller, and RAM access is arbitrated by the NS8422 dual port dynamic RAM controller, providing transparent access from both the CBPI and main processor. Transmission of a burst is effected by the processor simply writing a short transmit control block, which contains control information such as the destination address and a pointer to the buffer (or list of buffers) to be sent, and asserting the interrupt signal to the CBPI. The CBPI interrupts the processor when transmission is complete, with error status returned in a word in the transmit control block. The CBPI is capable of transmitting an entire list of messages, to different destinations if desired, without processor intervention. On receipt of an incoming burst, the CBPI places the incoming data into buffers from a free buffer list previously set up by the processor, and when the entire burst

[1]The Centrenet Burst Protocol Interface was designed by Tim Hopkins, Dept. of Computer Sci., Univ. of Edinburgh, 1988.

has arrived, sets up a descriptor containing error status, source address and other useful information, and interrupts the processor. The CBPI is able to continue to receive bursts in this way, even if the interrupt is not serviced immediately.

Further functions of the CBPI include maintenance of counts of certain network errors, again in shared memory locations initialised by the processor, and provision of a low bandwidth Single Packet Channel for supporting the terminal protocol - this appears to the host as a single 16-bit wide I/O port mapped into host memory space. The CBPI is implemented using microcoded hardware, with microcode stored in 1K words of 48-bit wide RAM, downloaded by the 68010 on reset, allowing easy modification to the microcode. The microcode is sequenced by an AMD 2910 device. The CBPI data path is built around an AMD 29116 part, which contains 32 16-bit registers and a flexible ALU optimised for I/O control operations, while for reasons of performance and ease of microcoding, some low-level operations are implemented directly in Programmable Array Logic devices, for example the timeout counter and the controller for DMA into shared memory. 16 word by 16-bit wide FIFOs connect the CBPI data bus to the shared memory data bus, and smooth out any changes in data rate between the network and the DMA channel to memory.

3.3.2 POSIE Monitoring Hardware

The Slave Monitoring Interface

These interfaces are the most critical part of the monitoring instrumentation because they actually monitor the processor boards and capture events. An event is notified to the slave monitoring interface by the processor board writing event data to the interface. At this point, the interface writes the data into the first level FIFO, along with the time-stamp. If the FIFOs are full, it can either slow down the processor or discard the new event data. This option can be set depending on

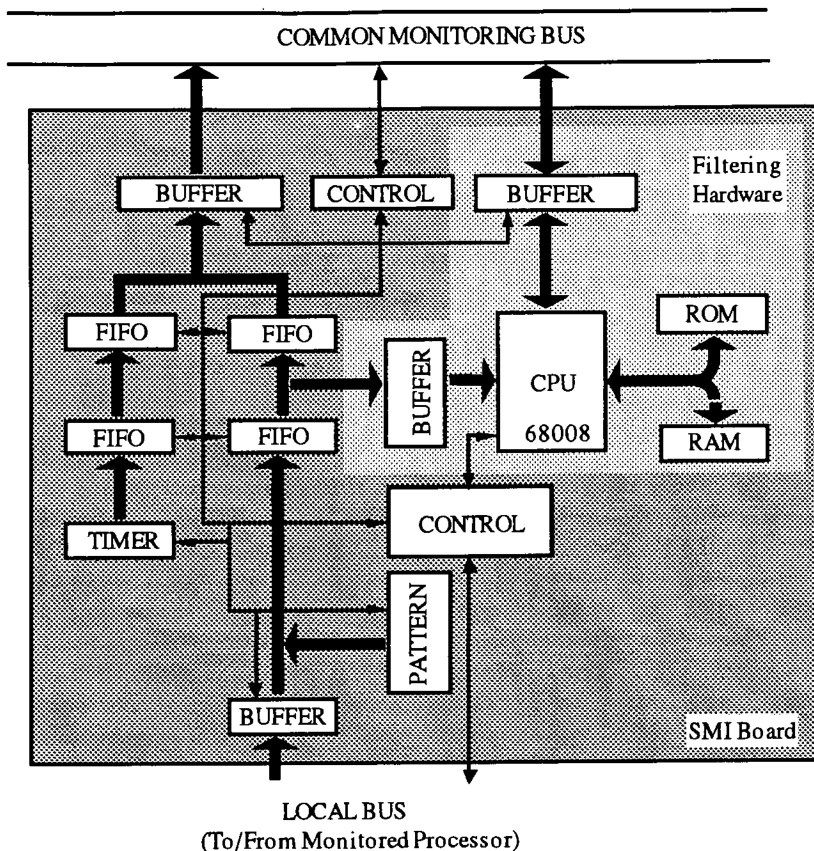


Figure 3–5: The Slave Monitoring Interface

the application or the nature of the experiment. The time-stamp is 16-bit binary value which represents fine tuned time within a time-slice of the global clock. In order to extend the time range, a synchronisation event is put into the FIFOs by enabling a special pattern for this event type (see Figure 3–5).

These synchronisation events are interpreted by the monitor as separators between time-slices, and are used for incrementing software clocks which are handled separately inside the monitor for each processor board. The timers on the slave monitoring interfaces are maintained in synchronisation by a global clock signal which is generated by the master monitoring interface. Any loss of synchronisation events can be detected and proper action can be taken by software running on the monitor.

The filtering hardware consists of a processor, static buffer space, a bootstrap

ROM and some buffers to the busses. This filtering processor can be programmed by the monitor, and it is able to perform other tasks such as counting events, and calculating some basic statistics related to monitoring. The filtering hardware includes a MC68008 microprocessor, an 8-bit version of the 68000 family. This processor was selected because its 8-bit wide data path reduces the chip count on the monitoring interfaces. At the same time, it is compatible with the 68000 instruction set, which is important for software development. It is connected to the data path between the first and second level FIFOs, and to the monitoring interface controller. When filtering is required, the 68008 sets a flag bit to the controller, and starts to monitor the data path. Having set the flag, the controller waits for another flag before transferring data from the first level FIFO to the second level. If the controller receives the filter flag, it removes data from the first level FIFO without writing to the second level. If the filter flag is negated, the controller transfers data from the first level to the second level FIFO, and deletes it from the first level. The filtering hardware does not change anything on the data bus but filters data by informing the controller. This operation slows down data gathering on the slave monitoring interface, but the monitoring bus load and the event monitor's load decrease as a result of filtering. If the first flag is negated by the 68008, the controller does not look at the other two signals. In this case, the filtering hardware is transparent to the the monitoring process so that it does not affect any operation.

The control logic of the slave monitoring interfaces is designed to be fast enough to capture events without slowing down the processors. If the master monitoring interface is not fetching event records from the slave monitoring interfaces fast enough, however and the event discarding option is not set, the FIFOs can overflow. This situation arises when the monitor board is too slow in reading the event records from the master monitoring interface. Only in this case the processor boards slow down. The solution to this problem is either to set the

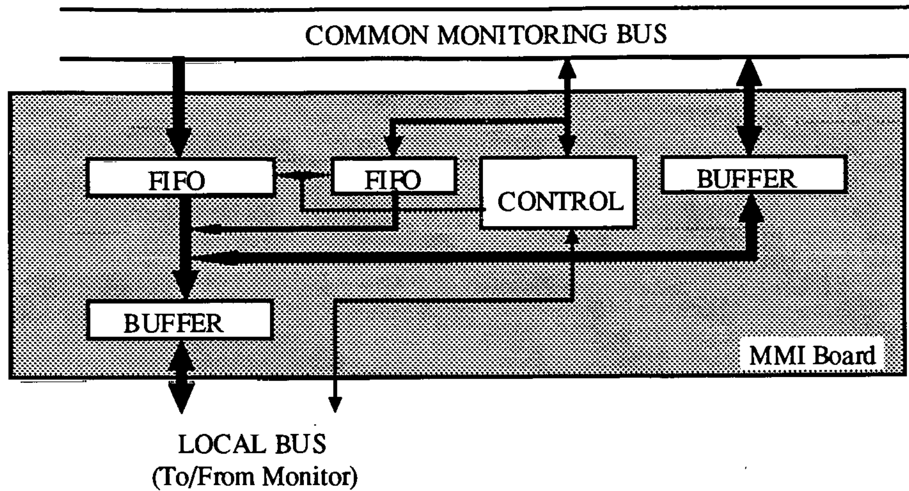


Figure 3–6: The Master Monitoring Interface

event discarding flag, or to increase the degree of filtering on the slave monitoring interfaces.

The Master Monitoring Interface

The master monitoring hardware (MMI) controls the operation of the common monitoring bus (Figure 3–6). One of the main reasons for having a hardware monitor is not to disturb the computation; another is not to disturb communication. Thus the MMI and SMIs take care of the communication of monitoring data, and the event monitor (Figure 3–3) does the computation.

Data is fetched from a SMI within two bus cycles. In the first bus cycle, the MMI sets the SMI address. In the second bus cycle the SMI decodes the address, and puts data on the bus along with a status flag. If this flag is set the data on the bus is valid, otherwise there is no data on the bus. Another signal is used for flagging that the SMI has decoded the address. The absence of this flag is interpreted as there being no SMI at that address. Bus operation is synchronous and in the second bus cycle the MMI copies the contents of the bus to its FIFOs if the data is valid. Along with data, the SMI address is also written to the

FIFO which is at the same level as the data FIFOs. The SMI address is used for identifying the processor board from which the data originated.

The MMI also maintains the global clock signal to the SMIs. Every SMI uses identical clock hardware and a single source of clock signal prevents relative timer drifts. In addition to the data gathering function of the MMI, it is also used as an interface for downloading simple software to the filtering processors on the SMIs. It can also read data directly from the filtering processors in order to access the results which are produced by them. This facility enables the filtering processors to do more complex filtering such as calculating processor utilisation, counting events etc.

The Event Monitor Board

The MMI is also connected to a processor board in the same way as the SMIs, except that the board to which the MMI is connected acts as the event monitor board. This flexibility allows a standard processor board to be used as an event monitor without the need to design a separate system. The MMI and SMIs are simple interfaces, and they can be modified to change or improve some of the functions. The event monitor board itself has no dedicated hardware function for monitoring, but it is used for running real-time monitoring and analysis programs.

In the case of running load balancing algorithms, the event monitor uses Centrenet to send messages to the kernels running on the processor boards. These messages contain some feedback data to improve the parameters of the system. The Centrenet connection is also used for downloading analysis programs before the monitoring experiment is started.

3.3.3 Analysis of the POSIE Performance Monitoring System

In this section, the following characteristics of the POSIE performance monitoring system are discussed:

- the cost of preparing and triggering event records,
- the data capacity of the POSIE monitoring hardware

The Cost of Preparing and Triggering Event Records

The POSIE monitoring system relies on event information notified to the monitoring hardware by software instrumentation embedded either in the operating system or in the application which is being monitored. Depending on the event type, the contents of an event record can vary. The event record usually includes the processor number, process identification, event type, time-stamp and channel identification (if the event is communication related). Some of this information can be gathered by the monitoring system automatically without any cost to the monitored computation but the rest of the information must be passed to the monitoring system by the instrumentation.

The POSIE monitoring hardware can identify processor numbers and give time-stamps to events but the event type, process identification and channel identification must be gathered by the software instrumentation. The overhead of preparing an event record can be higher than the overhead of recording since the latter involves a single write instruction to the monitoring hardware.

There are several ways of preparing event records, each of which involves a different monitoring overhead:

Naive event records: The simplest approach to preparing an event record is to include all of the information which is required by the event. This technique is

```
message_send(chan_id,message_buffer) /* Operating System Code */
{
   ... get process identification

record_event(5,process_id,chan_id)    /* 5 : SEND event */

   ... packetise the message
   ... send packets to the network
   ... block the current process

record_event(6,process_id,chan_id)    /* 6 : BLOCK event */

   ... schedule another process
   ...
}
```

Figure 3–7: An example of "naive event records" usage

used for general purpose instrumentation which is embedded inside the operating system or the system libraries. When the monitored application calls a function from the operating system, the task related information can be gathered from the system variables, put together to form an event record, and the resulting event record written to the monitoring interface. Since the data bus width of the POSIE monitoring system is 16 bits, event records which are longer than this require multiple write instructions. In Figure 3–7, a code segment from the operating system instrumented using this naive approach is depicted. In this example, event type and the channel and process identifications are written[2] to the monitoring interface.

Context sensitive event records: If the events are too frequent, and the records are too long, the overhead of monitoring can easily reach an unacceptable level. Event pattern combinations should thus be studied carefully to remove the

[2]The "record_event" function writes its parameters to the SMI

```
process1()  /* User Code */
{

record_event(1,toNeighbourProcess,``process1'')
 /* 1: ASSOCIATE_S event (associate a channel to a sender process) */

   ...
   message_send(toNeighbourProcess, ``Here is a message'')
   ...
}

process2()  /* User Code */
{

record_event(2,fromNeighbourProcess,``process2'')
 /* 2: ASSOCIATE_R event (associate a channel to a receiver process) */

   ...

   message_receive(fromNeighbourProcess, message_buffer)
   ...
}

message_send(chan_id,message_buffer) /* Operating System Code */
{


record_event(5,chan_id)    /* 5 : SEND event */

   ... packetise the message
   ... send packets to the network
   ... block the current process

record_event(6,chan_id)    /* 6 : BLOCK event */

   ... schedule another process
   ...
}
```

Figure 3–8: "Context-sensitive event records" usage—first example

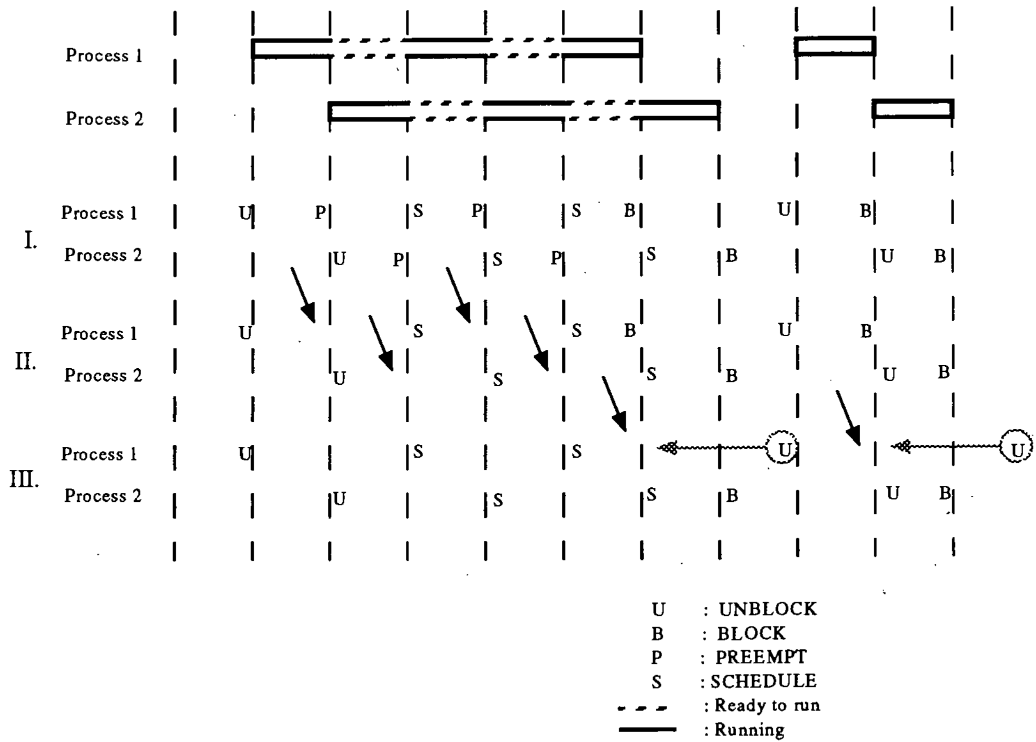


Figure 3–9: "Context-sensitive event records" usage—second example

redundant information from the event records. This will further reduce the monitoring overhead by curbing the number of writes to the monitoring interfaces, and by stealing less from the application's time. For example in Figure 3–8, the process identification field has been removed from all event records except for event numbers 1 and 2 which are triggered once at the beginning of the application processes. These events associate channel identifications to processes. When the event traces are interpreted, the missing process information of the "SEND" and the "RECEIVE" events can be retrieved from the "ASSOCIATE" events (ASSOCIATE_S and ASSOCIATE_R).

The previous example demonstrated how a common piece of information can be removed from event records. The next example shows how to stop some events from being triggered. The missing information about these events can be recovered by examining some specific event patterns. In Figure 3–9, the space-time diagram of two processes which run on the same processor is depicted. The rectangles

represent the interval in which a process is either running (solid lines) or ready to run (dotted lines). In the figure, three different forms of the same event trace are also given to show the effects of filtering. The first event trace (I) was collected using the naive approach (total 16 events). In the second trace (II), "PREEMPT" events (P) were not collected because we know that they are triggered T_c time before either "SCHEDULE" or "UNBLOCK" event. Here, T_c represents the context switching time. Unless we are interested in measuring the OS performance, it is possible to filter "PREEMPT" events and to use an average T_c value to restore "PREEMPT" events. In the example, the event trace was reduced from 16 events to 12 events. It is still possible to reduce the size of this event trace by filtering some of the "BLOCK" events. If another process is scheduled immediately after (nearly T_c time after) a process blocks (i.e. ready queue is not empty when the process is blocked), a single "SCHEDULE" event can be sufficient enough to represent this situation. It is easy to decide whether the missing event is "UNBLOCK" or "PREEMPT" by looking at the next event which is triggered from the blocked/preempted process. If the next event is "UNBLOCK" (circled "U" events in Figure 3–9), the missing event is "BLOCK" event, otherwise it is "PREEMPT" event. In Figure 3–9, the last event trace (III) shows the final result, and has the lowest event count: 10 events.

There is one question to be answered about this technique: Is the instrumentation which decides whether an event is to be triggered or not, more expensive than simple instrumentation which triggers all events straightaway? It is possible to find a suitable place in the operating system to plant instrumentation code for each event type so that the existing decision paths of the operating system are exploited. This requires no extra instrumentation code to make a decision on filtering.

Enumerated event records: Enumeration is a technique which is used for reducing the size of individual event records. This can be achieved in several different

```
process()  /* User Code */
{
   ...

record_event(1001)   /* 1001 : receive from ``fromNorth'' */

   message_receive(fromNorth,message)

record_event(1002)   /* 1002 : receive complete */
record_event(1003)   /* 1003 : send to ``toSouth'' */

   message_send(toSouth,message)

record_event(1004)   /* 1004 : send complete 1 */
record_event(1005)   /* 1005 : send to ``toEast'' */

   message_send(toEast,message)

record_event(1006)   /* 1006 : send complete 2 */

   ...
}
```

Figure 3–10: An example of "enumerated event record" usage

ways. The sizes of individual record fields can be kept small by enumerating the data stored in these fields. For example, the system may assign 32-bit identification numbers to channels, and only up to a few hundred channels are used in most of the applications. By assigning a unique number to each new channel, it is possible to reduce the size of channel identification field, for example, from 32 bits to 10 bits assuming that the channel count is less than 2^{10}.

It is also possible to use the enumeration technique for reducing the number of event record fields. If the contents of some record field combinations are known at compile-time, these fields can be replaced by a single field which contains a number representing each possible combination. In Figure 3–10, for example, the multiple field event record has been reduced to single field event record.

The Data Capacity of the POSIE Monitoring Hardware

In this section, the data load characteristics of the POSIE monitoring hardware are analysed.

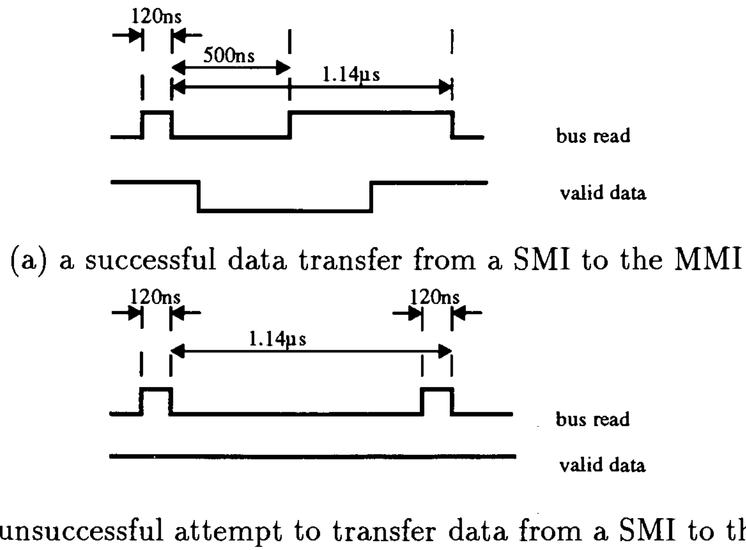
The SMIs of the monitoring interfaces have been designed to cope with sudden bursts of event generation. There are FIFO buffers on both the SMIs and the MMI. These buffers can store events temporarily while the frequency of event generation is higher than a threshold value. It is normal that the frequency of event generation by the software instrumentation depends on the granularity of the application which is being monitored.

It is important to know how the monitoring hardware behaves during burst of events, and how long it survives before the FIFOs are full. In the case of the FIFOs being full, there are two actions that can be taken: to slow down the application, or to discard the events. Neither case is desirable since the monitoring data are either incorrect or incomplete due to missing events.

In Figure 3–11 and Figure 3–12, the timing characteristics of the SMI are summarised. The first figure shows two different monitoring bus cycles: the SMI's FIFO is not empty (Figure 3–11-a), and a non-responding SMI[3] (Figure 3–11-b). In the second figure, both input and output characteristics of the SMI are given. As can be seen from the figure, the monitoring bus cycle ("bus read" and "valid data") is roughly three times faster than the event generation speed (the line marked "event" shows consecutive writes to SMI).

In Figure 3–13, the timing model of the POSIE monitoring hardware is given. This model is not restricted by the timing characteristics of the POSIE monitoring

[3]There are two possible cases where a SMI does not respond to the MMI's data transfer request: either the SMI does not exist in that particular slot, or its FIFOs are empty.



(a) a successful data transfer from a SMI to the MMI

(b) an unsuccessful attempt to transfer data from a SMI to the MMI

Figure 3–11: Two possible monitoring bus read cycles (The MMI asserts the "bus read" signal)

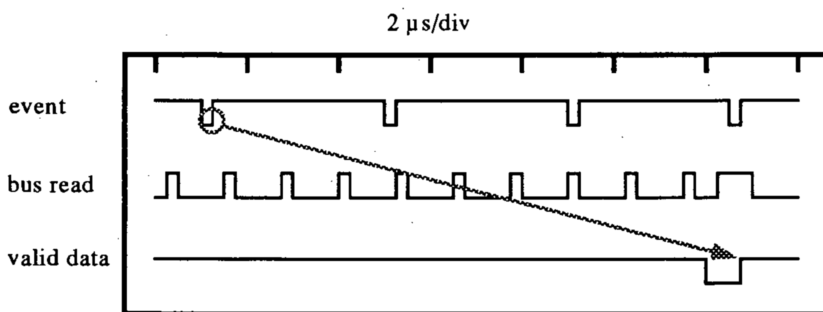


Figure 3–12: A typical example of recording an event and transferring it to the MMI

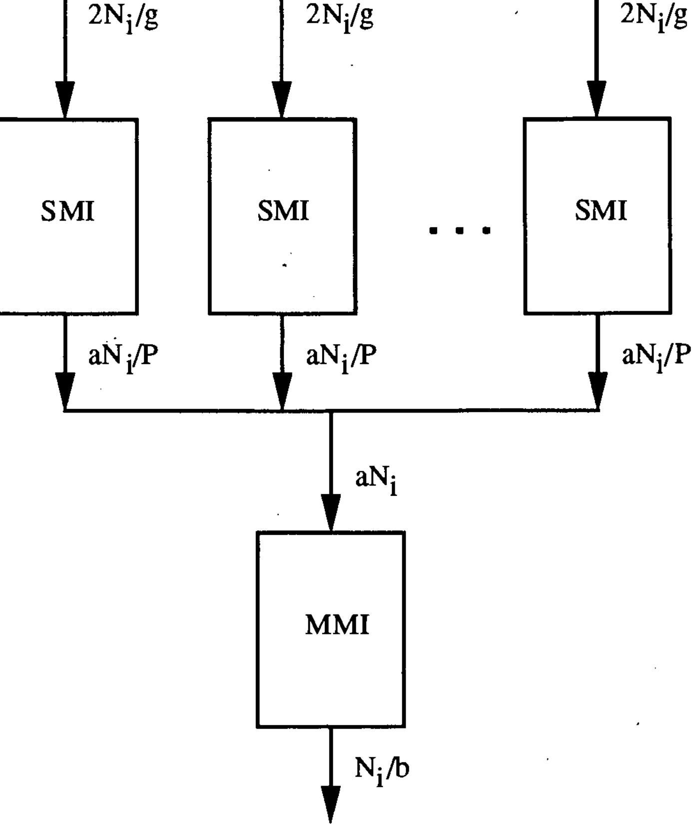


Figure 3–13: The timing model of the POSIE monitoring hardware

hardware. The same model can be applied to different monitoring setups of the same kind in order to analyse the requirements before the monitoring hardware is built. It is possible to change the timing characteristics of this monitoring hardware depending on the speed of the processors, the number of processors in the system, and the expected granularity of the software to be monitored. Parameters such as the depth of FIFOs and the speed of the monitoring bus can be altered to change the timing characteristics of the monitoring hardware.

Before presenting examples of how the POSIE monitoring hardware behaves under different loads, the definition of the parameters, and the timing characteristics of the POSIE monitoring hardware are presented as follows:

N_i : The number of instructions that can be executed per second by a processor or the monitor processor. It is assumed that the monitor board is identical to the monitored processor boards.

g : The granularity of triggering events from a monitored board. Since it is possible that event write instructions can be executed at a different rate to other instructions, granularity is represented as $g = g_i + e$ where g_i is number of instructions between two event writes, and e is the length of an event write instruction. For example, in the POSIE testbed, an instruction is executed 2 times ($e = 2$) faster than an event write instruction.

a : The ratio of the instruction cycle of a monitored processor to the data transfer cycle of the monitoring bus.

P : The number of monitored processors.

b : The number of instructions in the loop used for reading the data from the MMI.

d_{smi} : The depths of the SMIs' FIFOs. (Notice that 2-way interleaved FIFOs are used. For the POSIE implementation, d_{smi} is 2×1024.)

d_{mmi} : The depth of the MMI's FIFO.

The SMIs' and MMI's FIFOs can be filled in at different speeds depending on each others' status. There are four different cases:

The first case is that the MMI's FIFO is not full. In this case, the speed of filling the SMIs' FIFOs (Equation 3.1) is the difference between the speed of generating

events and the speed of transferring data on the monitoring bus. Equation 3.2 gives the time to fill the SMIs' FIFOs completely.

$$N_{smi} = \frac{2N_i}{g} - \frac{aN_i}{P} \tag{3.1}$$

$$t_{smi} = \frac{d_{smi}gP}{(2P - ag)N_i} \tag{3.2}$$

In the case that the MMI's FIFO is full, the speed of transferring data on the monitoring bus is bounded by the speed of emptying events from the MMI's FIFO. The speed of filling SMIs' FIFOs and the time to fill them can be calculated using Equation 3.3 and Equation 3.4, respectively.

$$N'_{smi} = \frac{2N_i}{g} - \frac{N_i}{bP} \tag{3.3}$$

$$t'_{smi} = \frac{d_{smi}bPg}{(2bP - g)N_i} \tag{3.4}$$

The speed of filling MMI's FIFO depends on the availability of data in SMIs' FIFOs. If they always have data in them, the monitoring bus is fully utilised (i.e. in every bus cycle, data are transferred between FIFOs). In this case, the speed of filling the MMI's FIFO (Equation 3.5) is the difference between the maximum bus speed and the speed of emptying data from the MMI's FIFO. Equation 3.6 gives the time to fill the MMI's FIFO completely.

$$N_{mmi} = aN_i - \frac{N_i}{b} \tag{3.5}$$

$$t_{mmi} = \frac{d_{mmi}b}{(ab - 1)N_i} \tag{3.6}$$

If the SMIs' FIFOs sometimes contain data, the monitoring bus is under-utilised. The speed of generating events and the speed of transferring them to the MMI becomes the same. The speed of filling MMI's FIFO and the time to fill them can be calculated using Equation 3.7 and Equation 3.8, respectively.

$$N'_{mmi} = \frac{2N_iP}{g} - \frac{N_i}{b} \tag{3.7}$$

$$t'_{mmi} = \frac{d_{mmi}bg}{(2bP - g)N_i} \tag{3.8}$$

The behaviour of the monitoring hardware as a whole can be classified as in the following cases:

Case 1: The SMIs' FIFOs are filled more slowly ($2N_i/g \leq aN_i/P$) than they are emptied, and the MMI's FIFO is filled faster ($2N_iP/g > N_i/b$) than it is emptied by the monitor processor.

$$\frac{2P}{a} \leq g < 2bP \tag{3.9}$$

$$t_{total} = t'_{mmi} + t'_{smi} \tag{3.10}$$

Case 2: The SMIs' FIFOs are filled faster ($2P/a > g$) than they are emptied. There are two subcases:

Case 2.1: The SMIs' FIFOs become full first ($t_{smi} < t_{mmi}$).

$$g < \frac{2bP}{\frac{d_{smi}}{d_{mmi}}P(ab - 1) + ab} \tag{3.11}$$

$$t_{total} = t_{smi} \tag{3.12}$$

Case 2.2: The MMI's FIFO gets full first ($t_{smi} \geq t_{mmi}$).

$$g \geq \frac{2bP}{\frac{d_{smi}}{d_{mmi}}P(ab - 1) + ab} \tag{3.13}$$

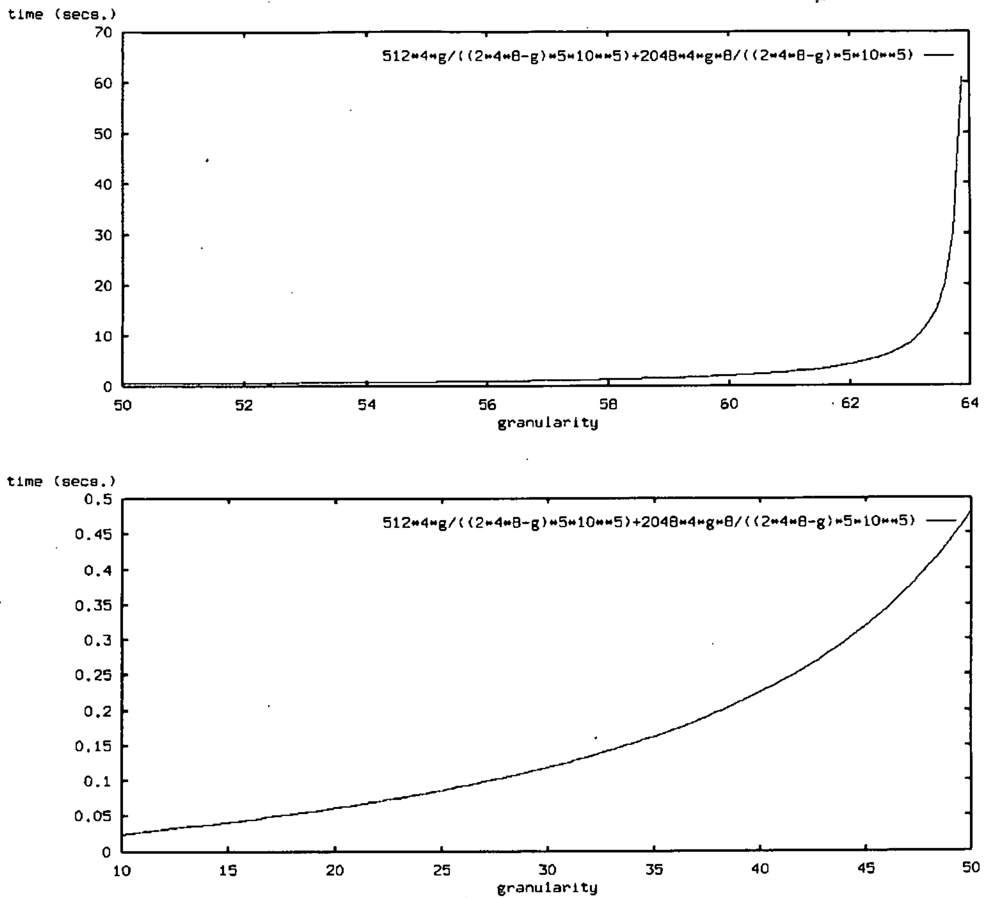


Figure 3–14: Timing characteristics of the POSIE monitoring hardware

$$t_{total} = t_{mmi} + t'_{smi}(1 - \frac{t_{mmi}}{t_{smi}}) \tag{3.14}$$

There is another case which has not been mentioned. The SMIs' FIFOs and the MMI's FIFO are filled more slowly than they are emptied ($2N_i/g \leq aN_i/P$ and $2N_iP/g \leq N_i/b$). This is the normal behaviour when there is no burst of events.

Figure 3–14 shows how the POSIE monitoring hardware behaves under a burst of events. These graphs are produced using Case 1 with the following parameters:

$P = 8$, $a = 1.6$, $b = 4$, $d_{smi} = 2048$, $d_{mmi} = 512$, $10 \leq g < 64$.

In this configuration, as long as g is larger than 64 (i.e. the average number of instructions between two event writes to the monitoring hardware is 62 or more) there is no danger of overloading the monitoring hardware. In the case of bursts ($g < 64$), the graphs in Figure 3–14 represent the maximum tolerance time (given a g value) to a burst of events before the FIFOs get full. These graphs show that, for example, the monitoring hardware can tolerate $g = 40$ up to approximately $0.2sec.$ (or any $0.2sec.$ interval should not have a granularity which is smaller than 40 instructions). At this granularity level, 38 instructions (average) can be executed (see the definition of g) between two event writes.

To evaluate the usability of the POSIE monitoring hardware, and to assign its overhead figures, the following OS figures have been used: context switching from one process to another takes 260 machine instruction times, and a blocking intra-processor communication time costs 435 machine instruction times to the sender process. It is expected to trigger only one event during context switching, and 4 events during communication:

$<$ *initiate send and block the process* $>$,

$<$ *initiate receive and block the process* $>$,

$<$ *unblock the receiver process* $>$,

$<$ *unblock the sender process* $>$.

Let us assume that a process is continuously sending messages to another process (on the same processor board) without doing any other computation, and the receiver process is also receiving these messages without doing any other computation. In this extreme case, the average distance between two event writes is approximately 109 instructions. With this instrumentation, it is not possible to overflow the monitor's FIFOs, and the worst monitoring overhead to an application program is 1.8%.

It is possible for user-defined events to overflow the monitoring hardware FIFOs if the granularity of the event triggering does not comply with the limitations of

the hardware (see graphs in Figure 3–14). At the limit ($g = 64$), the monitoring overhead can reach up to 3.1%.

3.4 A Software Approach — Instrumentation for the Edinburgh Concurrent Supercomputer

Not all systems are suitable for hardware instrumentation for several reasons. Firstly, financial reasons prevent one from adding hardware instrumentation on top of the existing parallel machine. Secondly, there may be no room for any additional hardware since most commercial parallel computers follow the leading edge of the technology, and fit as much hardware as possible into a small space. In this case, there is no option but to implement a software monitoring system. In this section, an implementation of a software monitoring system for the Edinburgh Concurrent Supercomputer [63], (ECS), a MEIKO Computing Surface, is introduced. This instrumentation is used in parallel program performance tuning experiments. Even though this implementation is intrusive, it provides valuable information for performance tuning.

3.4.1 The ECS and CS-Tools

CS-Tools [15] is a programming environment which enables users to write parallel programs using the C or FORTRAN programming languages. A parallel program consists of independent sequential programs that communicate with each other through **CS-Tools** library functions. These library functions provide both blocking and non-blocking communication facilities. Each sequential program is considered as a process, and processes address each other using transports. Each transport acts like a channel to which more than one process at a time can send messages, and there is only one receiver process. The receiver process makes

known itself to the world by writing a name and an associated address of a transport to a global look-up table. The sender process finds out about the address of a destination transport by querying the name of the transport at the look-up table. Processes do not need to know about the location of the other processes since CS-Tools hides the message routing details from the users. This facility in the environment makes programming easier at the expense of a small performance degradation, since user cannot directly address physical channels and talk to neighbouring processors efficiently. Nevertheless, the user has an opportunity to improve the performance of the message passing mechanism by defining a topology. The user defines a processor topology, and maps the process topology onto this processor topology so that processes which are likely to cause channel contention are close enough, and the soft channels which are likely to be heavily loaded do not share the same hard channels. The closer the processes, the better the chances of efficient communication. By doing this, the user can ensure that there is a very small chance of channel or processor contention. These issues are addressed in more detail in the next section.

3.4.2 The ECS Performance Monitor

The ECS Monitor exists in two versions, both of which have advantages in different situations. While the first version is less intrusive, it has some restrictions relative to the second version. The instrumentation to be used depends on the situation. If there is enough memory for both the application and the monitoring system, the first version is always preferable since it affects the application less than the second version does. In the case of insufficient memory, the second version is the single option.

ECS Performance Monitor Version 1

This version of the monitoring system consists of small monitoring codes which are linked to the user processes at compile-time, and a central monitor which is a stand-alone process dedicated to monitoring. Each monitoring code is responsible for collecting monitoring data from the process to which the monitor is linked. The monitoring information is held locally until all processes of the application terminate themselves. Then the monitoring information from the individual processes is collected together to produce an execution history of the parallel program. Since collecting event traces from processing elements happens after the program completes its execution, the instrumentation overhead does not include communication costs. The cost of instrumentation is a constant value for each communication activity. Therefore the total cost of the instrumentation is a function of communication frequency.

The monitored activities are process creation/termination and message passing operations. The instrumentation replaces CS-Tools functions with instrumented versions. These new functions can record every send/receive operation by creating special events for each occasion. Two events are created for each message passing operation: one marks the beginning and the other completion. For example, it is possible to identify how long a send operation takes by looking at the time-stamps of the "send" and "unblock" events recorded for the communication.

Since this instrumentation only monitors application programs and not the operating system, OS performance cannot be measured in detail. Nevertheless, by looking at the execution times of applications one can gain an insight into the OS performance. For example, it is not possible to monitor scheduler activities with the existing instrumentation. Let us consider two processes running on the same processor. With the current instrumentation, it is possible to observe that these two processes are executing at the same time. Since scheduling activities are not monitored, we cannot identify when a context switch between a pair of

EX BIBL. UNIV. EDINBURG

processes has happened. Nevertheless, the context switching overhead can be calculated using the existing information. For example, the durations of two different computational activities (e.g. a code segment from a process) are measured separately in a quiet environment (i.e. there is no other process running on the same processor). Then these computational activities are run in parallel. The difference between the execution time when the activities run in parallel and the total execution time of the activities when they are run separately gives the time spent for the context switching.

It is possible to install an embedded code inside the operating system to monitor the scheduling activities. Since the scheduling frequency can be much higher than the frequencies of the application activities (e.g. sending a message, blocking for a synchronisation, etc.), monitoring the OS related activities is expected to be more expensive than monitoring an application program. Besides, the OS is not normally accessible to software instrumentation in comparison with applications that can be instrumented easily.

ECS Performance Monitor Version 2

The second version of the ECS performance monitor[4] uses the same underlying philosophy as version 1. Both have the same kind of instrumentation that triggers the monitoring system when an event is to be recorded. The difference is in the way monitoring information is collected. In the second version, the monitoring information is collected on special monitor processors while the application program is still running. In order to decrease the monitoring overhead, an efficient implementation is required. This requires a proper understanding the underly-

---

[4]This version was designed by the author, and implemented by William Hern [25] as a part of EPCC Summer Scholarship Programme.

ing communication mechanism. The following characteristics of communication networks were considered when the second version was being implemented:

- The distance between monitored and monitoring processors is important.
- The important portion of a communication cost is housekeeping and routing costs, not the actual transfer cost.
- Many to one connections are susceptible to channel and probably processor contention.

To satisfy the first item, processors in an application are divided into small groups so that one monitor processor can be allocated close to all the processors in a group. This grouping also helps with the contention problem. Since every monitor processor is responsible for a small number of application processors, the chances of channel contention around the monitor processor are low. Of course contention depends on the frequency of the monitor messages that carry monitoring information, and to reduce the frequency of monitor messages, small messages are combined together to make a smaller number of large messages. This also reduces the amount of information to be transferred since there is no need to repeat common data items. Combining small monitor messages and sending them as a single large message is also faster than sending them separately, since the routing overhead is nearly the same regardless of the size of the message.

When a large number of processors are to be monitored, it is crucial to organise the monitoring testbed so as to decrease the monitoring overhead. If the user does not define a processor topology for his/her application, defining a monitoring testbed is straightforward. This is carried out by grouping processors, and assigning a monitor processor to each group. Applications that require specific hardware topologies need more attention than the ones without topology requisite. Since the topology can impose some restrictions on how the monitoring testbed is organised,

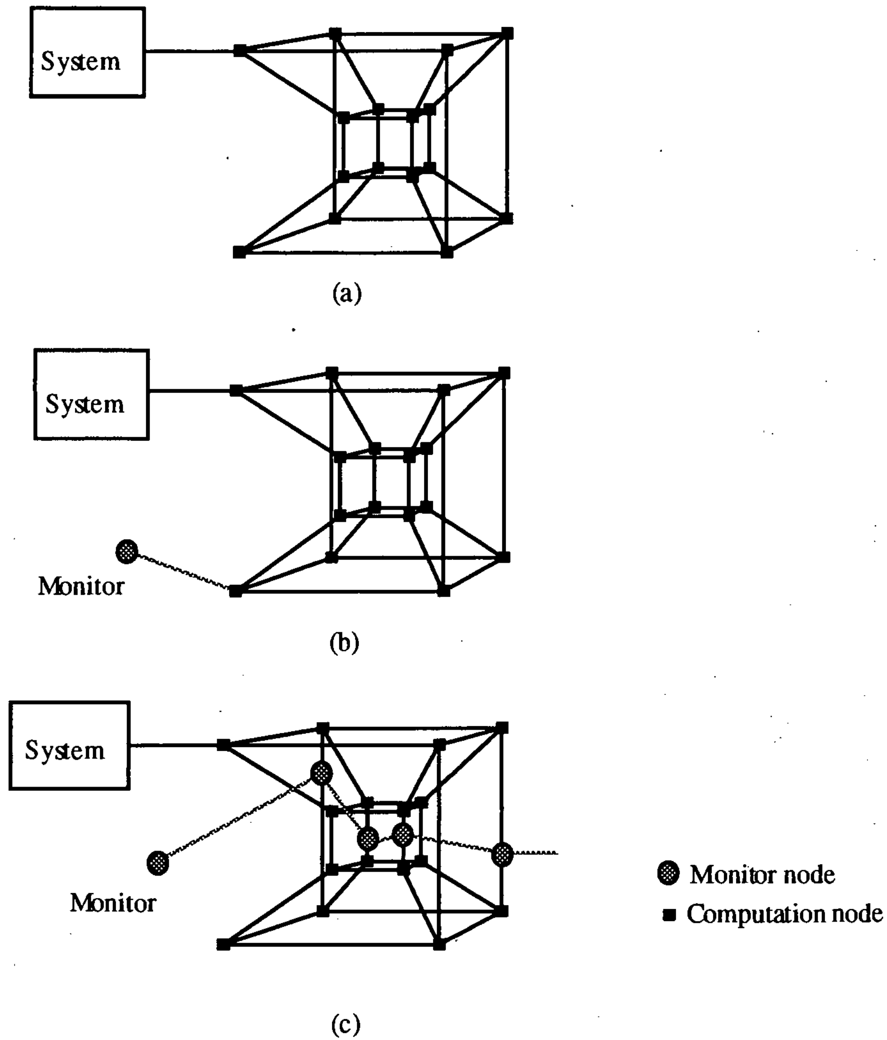


Figure 3–15: The unmodified and two modified 16-node hypercube topologies for ECS

there are not many options for defining a testbed other than cutting some connections and adding some extra PEs for monitoring. Of course this can affect the message routing strategy since there are now more options for routing a message, or some routes are longer. For some popular topologies, it is possible to provide predefined testbeds.

In Figure 3–15, an implementation of a 16-node hypercube topology and two different monitoring testbeds for this particular topology are given. The first figure (Figure 3–15-a) is the original 16-node hypercube topology which is used in ECS. In this topology, all four links of each node are used for constructing the hypercube except for one of the links of the node zero (top, leftmost corner). This node is special because it provides a bridge between the other nodes in the topology and the rest of the system (i.e. the user and the file system). Since the system connection is provided by a normal communication link, and there are only four links available on each node, one of the links of an original hypercube topology cannot be provided in this particular implementation.

In Figure 3–15-b, a monitoring testbed (testbed 1) for the hypercube is given. A monitor node is added to the topology using an existing spare link. All of the nodes send their monitoring traces to this monitor node. Although it is easy to implement this testbed, there are problems with it. Firstly, some of the nodes are 5 or 4 hops away from the monitor node. The more hops are used for sending monitoring data, the more disturbance to the monitored application is expected. The communication network is disturbed due to extra traffic. The computations on the node are also disturbed due to extra computation which is carried out to route the monitoring messages. Secondly, the monitor node is connected to the other nodes through a single link. Since the monitor messages from all nodes will use this single link, it is likely to suffer from the channel contention. As a consequence of this, a large number of monitor messages will be buffered on the intermediate nodes between the monitored nodes and the monitor node. This will delay the delivery of application messages.

In Figure 3–15-c, another monitoring testbed (testbed 2) for a 16-node hypercube is given. This testbed uses four more monitor nodes than the first testbed. Each of these additional nodes is responsible for monitoring a hypercube dimension which contains four computation nodes. Each monitor node is added to the existing hypercube topology by cutting a link between two computation nodes, and connecting these two computation nodes directly to the monitor node.

There are some advantages of the second testbed over the first testbed. Firstly, in the second testbed, there are larger number of links which connect the monitor to the rest of the nodes. This will reduce the contention which is caused by monitor messages. Secondly, the distances between the monitored nodes (computation nodes) and the monitor nodes are not more than 2 hops. This will reduce the monitor related message traffic on the network since each monitor message travels through a small number of links. Last but not least, using multiple monitor nodes will increase the allowable size of monitoring data since each monitor node which is added to the system will provide additional memory space for storing monitoring data.

There are also disadvantages of adding multiple monitor nodes to a hardware topology where there is no spare link for connecting the computation nodes to the monitor nodes. Firstly, inserting a monitor node will add an extra hop to some of the routes in the net. This will introduce a time penalty to a message transmission which uses this route. Secondly, extra links of the monitor nodes will provide more alternatives to route messages. This can reduce the transmission time of some messages, or hide a contention which can happen in the original topology which does not have extra links and monitor nodes. It is also possible that, in some cases, extra monitoring hardware can improve the performance of the monitored application.

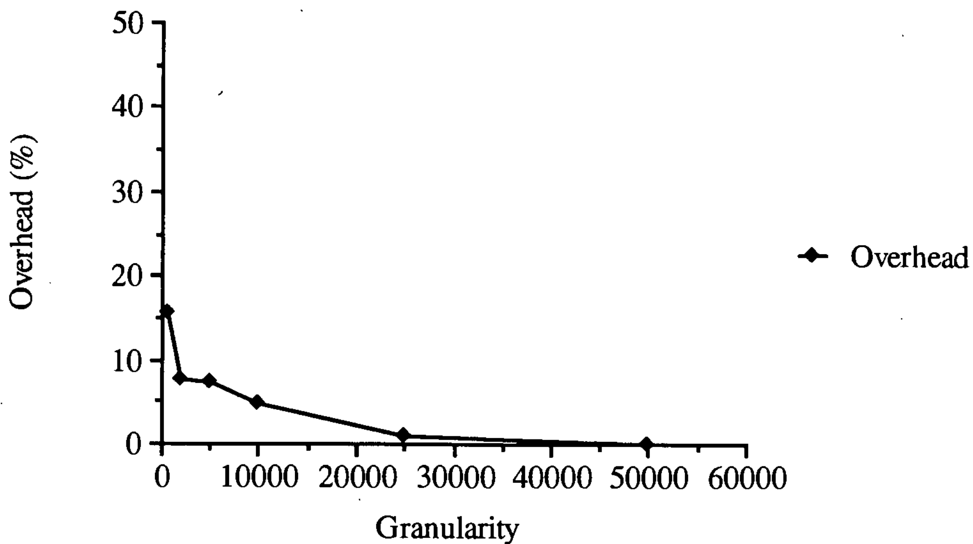


Figure 3–16: Overheads of the ECS Monitor (version 1) at different granularity levels

3.4.3 Analysis of ECS Performance Monitoring System

There are several factors that increase the overhead of monitoring. The first factor is the granularity of the application. Since the ECS monitor collects information about the communication activities of the application, the granularity of the application will change the number of communications that take place. The higher the number of communications, the greater the overhead incurred. In Figure 3–16, ECS monitor version 1 is used for monitoring an application (see Section 5.3.1) in which the overhead decreases as the granularity increases. This application program is a good test program because its granularity can be changed to flood the system with messages or to carry out a large number of local computations. Since ECS monitor version 1 does not use any communication channels while the application is running, the overhead of the monitor is small. The maximum overhead for the test program is 16% where the execution is dominated by communications (i.e. maximum event triggering is expected in communication intensive cases of an application).

ECS monitor version 2 has been tested under several different conditions and for three different granularity levels. In the experiments, the testbeds which were explained in Section 3.4.2 (see Figure 3–15 (b) and (c)) were used for monitoring a specific application which relied on hypercube topology for better performance although the program could run on other topologies. For example, one set of experiments was carried out without defining a specific topology. In the ECS, if the user does not define a topology, the system defines it, and the application is mapped on to this system-defined topology. The reason for using a system-defined topology was to test whether the system could provide a better message routing strategy for a system-defined topology. It is possible that the message router is optimised for certain topologies, and the user-defined topologies cannot be utilised efficiently by the router. For example, if the router does not know that the destination node for a message send is a neighbour node, the message can be routed to its destination through a longer route. By observing the router's behaviour it is possible to choose efficient topologies for this particular version of the router, or alternatively, it is possible to implement efficient routers for some specific topologies. Then, these topologies are kept in a library, and can be used by the system unless the user provides an application specific topology.

In Figure 3–17, the results of the experiments are depicted. Each of them shows the overheads of monitoring the application at a different granularity level: fine (a), medium (b) and coarse (c) grains. The buffer size is the number of events that has to be gathered before they are sent to the nearest monitor node. In these three figures, buffer sizes between approximately 50 to 100 result in the lowest monitoring overhead.

It can also be seen from these graphs that the second testbed is usually better than the first one. In the second testbed, there are four monitor processes (on dedicated monitor processors). These monitor processes are located close to the processors which they monitor. The minimum distance between a monitor and a monitored process is either 1 or 2 hops, but the message router may choose an

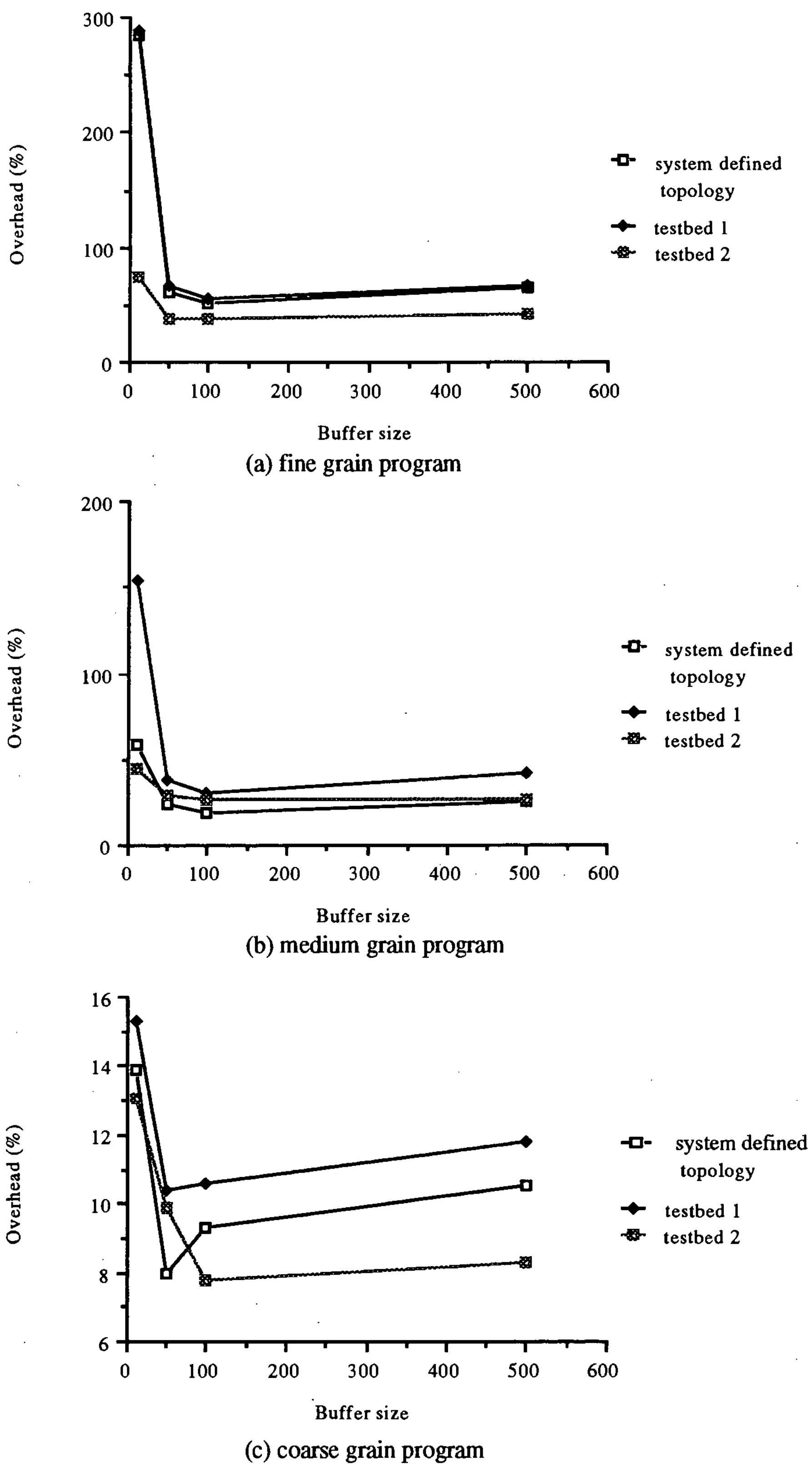


Figure 3–17: Overheads of the ECS Monitor (version 2) under different conditions

alternative route which is longer than 2 hops. This monitoring configuration relies on the router to use the shortest possible route. The more efficiently the monitoring information is transferred to a local monitor, the less monitoring overhead is incurred.

In the first testbed, the minimum distance between a monitor and a monitored process ranges from 1 to 5 hops. It is again possible that the router may choose routes which are longer than the minimum. In this respect, the second testbed is certainly better than the first testbed.

It is usually possible to reduce the overhead of ECS performance monitor version 2 by allocating monitor processors which are close to the monitored processors. Unlike version 2, version 1 has a fixed configuration, and the user cannot improve it. On the other hand, version 1 has a much lower overhead than version 2.

3.5 Comparison between POSIE and ECS Monitoring Systems

In the previous sections, two different performance monitoring systems were presented. In this section, we discuss these implementations in terms of intrusiveness, scalability, cost, capability, and portability. Table 3–1 summarizes the comparison of the two implementation..

Intrusiveness

Intrusiveness of the monitoring systems is the most important problem to be considered. In both of these implementations, various decisions were taken in order to minimise this intrusiveness. In the POSIE monitoring system, the event monitoring system is supported by hardware. In the ECS monitoring system, extra processing elements are used so that realistic monitoring can be carried out.

Table 3–1: Comparison between POSIE and ECS Monitoring Systems

	ECS Monitor (S/W)	POSIE Monitor (H/W)
		Application Related Events
Event Types	SEND MESSAGE	SEND MESSAGE
	RECEIVE MESSAGE	RECEIVE MESSAGE
	UNBLOCK PROCESS	UNBLOCK PROCESS
	CREATE PROCESS	CREATE PROCESS
	TERMINATE PROCESS	TERMINATE PROCESS
		OS Related Events
		PREEMPT PROCESS
		SCHEDULE PROCESS
		MESSAGE ARRIVAL
		BLOCK PROCESS
	Software Instrumentation	*Software Instrumentation*
Event Triggering	Event identification	Event identification
	Process identification	Process identification
	Channel identification	Channel identification
	Processor identification	*Hardware Instrumentation*
	Time-stamping	Processor identification
		Time-stamping
Event Buffering	Processes' workspaces	Dedicated hardware buffers for each processor
Event Collection	Existing communication links	Dedicated monitoring bus
Cons and Pros	Intrusive (normally 5%-15%)	Expensive to implement
	No real-time support	Unscalable
	No global timing	Global timing
	Portable	Non-intrusive (nearly, < 1%)
	Scalable	Real-time support
	Sometimes the only alternative	

The first difference is that the ECS Monitors cannot catch some events that are OS related and not visible at the application level. OS related events are shown together with application related events in Table 3–1. To monitor these events OS instrumentation is needed. Since the frequencies of these (especially "preempt" and "schedule") events are much higher than application related events, it is highly intrusive to monitor them using a software monitor.

In the ECS monitor, monitoring an event consists of triggering, time-stamping, buffering and transferring. These stages are all carried out by using software and all incur a time penalty. The POSIE monitor, on the other hand, has no cost for all these activities except for triggering, which is a small piece of inserted code inside the OS. In a software monitor, time stamping is one of the expensive stages of monitoring an event. Since it involves accessing a shared resource and reading a hardware clock, its overhead is usually high. Transferring monitoring information is the most expensive stage of all. This stage involves loading existing communication network with extra messages. Several implementation decisions can be taken to reduce this load. Firstly, buffering event records and sending them as big chunks of data decreases the overhead because sending a message incurs a constant overhead regardless the size of the data to be sent. This overhead includes the extra bandwidth, which is consumed by the fixed size headers and trailers of a message, and the extra computation power, which is spent routing messages through the nodes.

Using some of the characteristics of the events, it is also possible to compress some of the information. For example, each event has 32-bit time-stamp. Since the time interval between two consecutive events can usually be measured using a 16-bit timer, these relative time values can be sent to the central monitor instead of 32-bit absolute time-stamps. If the time difference between two consecutive events is greater than a 16-bit value, a dummy event is created to cover the time gap that is beyond the 16-bit time-stamp. This technique is also used in the POSIE monitor

to improve the performance of the monitoring bus. In both implementation, the event trace size can be reduced by up to 33%.

Scalability

Since distributed memory MIMD systems are getting larger (i.e. more PEs), their monitors have to scale up with them. There are some constraints that restrict scalability of a monitoring system. One of them is the timer that is used for time-stamping. Since it is sometimes physically not possible to provide a global timer signal to large number of PEs, a monitoring system which relies on a global timer may become unscalable. If the partial ordering of the events is sufficient for the purpose, a global timer can be ignored totally. In the POSIE monitor, a global timer signal is available for the PEs, which are in the same physical cabinet. It is not difficult to support a global timer for several cabinets but its scalability is restricted. Since every cabinet of the POSIE machine has its own monitor, however, the data collection infrastructure scales up with the machine. The ECS Monitor is highly scalable as long as there are enough processors to support monitoring. Grouping PEs and assigning a monitor to each group is the key factor for the system's scalability. By grouping the PEs it possible to exploit the locality of the monitoring information to reduce the network traffic.

Cost

Obviously, implementing a hardware supported monitoring system is much more expensive than implementing a software monitoring system in terms of time spent on implementation and other resources. At first sight, the ECS monitor can also be considered as an expensive implementation. For example, sometimes half or quarter of the PEs are used for monitoring purposes. There is no real investment cost of the ECS Monitor, however, since ordinary PEs are used for monitoring and they can be returned to the PE pool when there is no need for them. Having

invested in a dedicated hardware monitor, it is impossible to allocate its resources for any other purpose, and it can considerably increase the cost of the parallel machine.

Portability

Since the ECS monitor is dedicated to a CS-Tools environment, and does not require any modification to the OS, it can be ported to any system that runs CS-Tools. On the other hand, the POSIE monitor is dedicated to the POSIE machine; it cannot be ported to another system. Even though the POSIE monitor is not portable, it is possible to build portable monitoring hardware. This can be done by building a stand-alone system which can collect data through its probes. A probe is a simple hardware interface between a processor board and the monitoring system. On the processor side of the probe, there is a customised interface which recognises only specific write cycles to the monitor, and on the monitoring system side, there is a standard interface to transfer monitor data to the monitoring system. The probes can be kept very simple by implementing only the processor board specific hardware on the probes, and moving other common functions to the monitoring system. For example, the monitoring system will time-stamp and buffer the data which is recognised by the probe. Physical connections between a processor board and a probe which is specific to this particular board can be implemented in several different ways, and it is an engineering problem. The whole system will look like a logic analyser with probing hardware attached to the end of each probe cable.

3.6 Summary

In this chapter, two different performance monitoring test-beds are presented. One of the test-beds is hardware supported software instrumentation which uses dedicated hardware to capture and collect the monitoring traces which are triggered by software instrumentation embedded into the operating system. The second testbed is also software instrumentation but there is no dedicated hardware for monitoring purposes; system resources are shared with the monitored application. In some monitoring configurations, extra computational resources (processors) and communication links are used for improving the instrumentation.

Since the hardware supported monitoring test-bed (POSIE performance monitor) has an instrumented code within the operating system, it is possible to monitor some activities, such as process scheduling, which cannot be monitored using instrumentation such as the ECS performance monitor.

While the lowest monitoring overhead is achieved by using hardware support, it is also possible to use pure software approaches and to achieve acceptable levels of monitoring overhead.

Due to ongoing operating system development, the POSIE testbed was not used for further experiments to monitor real parallel programs. Its functionality and performance characteristics were measured using diagnostics programs. All experiments with the real parallel programs were carried out on the ECS.

Chapter 4

Performance Data Filtering

4.1 Introduction

In performance analysis of parallel programs, reducing the size of the trace information which is collected from an execution of a parallel program is an important problem. Yet little work has been done to attack this important problem. Most of the performance analysis (post-execution) or visualisation tools do not allow users to select information of their own preference. Instead, predefined filters are used for extracting information about well-known performance metrics. This is a restricting factor for a user wishing to explore application specific characteristics.

In the literature, there are two main approaches to reducing the trace information and extracting useful information from parallel program traces. These two approaches mainly concentrate on debugging parallel programs. The first is called the relational approach [61] since relational database type queries are applied to event traces files in which each event corresponds to a relational database record (a tuple).

The second is based on behavioural abstraction [5,6,38]. In this approach, the behaviour of each process of a parallel program is defined as an event pattern template. These templates are used for recognising the patterns in event traces and creating abstract events from these patterns. Then, the abstract events are

used to compare the actual behaviour with the expected behaviour. [2], [13], [32], [34] and [36] are motivated by the behavioural abstraction approach.

The approach presented in this chapter uses the event abstraction technique which is similar to the behavioural abstraction approach. Performance metrics of a parallel program are represented as abstract events. Since the user defines abstract events specific to any case, (s)he is not restricted to using standard metrics. The user can define application or domain specific metrics which directly reflect a particular performance characteristic of a parallel program in terms of internal structures of the program.

4.2 Event Abstraction Technique

Event histories of parallel programs are valuable information sources for performance analysis, but the problem is to extract useful information from massive amounts of low level event traces. Here, the programmer needs a mechanism to filter out unwanted details of the history and expose the performance related information. Filtering can be achieved by replacing some of the lower level event patterns with higher level events. The process of replacing patterns can be repeated until a satisfactory result is achieved. The replacements can range from very simple pattern replacements to very complex hierarchical replacements (Abstract events). Every new replacement event inherits some information from the events that are replaced, and adds new information on top of them. Since the number of events decreases and their meanings become closer to the structure of the parallel program under development, it becomes easier for the programmer to interpret an event history. For example, some primitive events can be replaced by new events that represent communication and computation intervals. Then, these new events are replaced by other new events that represent small stages of each process. By using these small stages, parallel or non-parallel stages in the program

can be identified. These new events can be created by finding the intersections of the "stage" events. The resulting view of the program history can reveal problems with the program. If this view is found to be inadequate for tuning, the programmer can go further and find the stages which do have significant effects on performance. For example, this can be achieved by finding the critical path of the history, and intersecting critical path events with the events that represent parallel and non-parallel stages of the program. The resulting events inherit the meanings of the both constituent events: critical non-parallel stages or critical parallel stages.

The event abstraction mechanism can be used for improving program execution visualisation, searching patterns in the execution histories, or measuring user defined metrics.

4.2.1 Event Streams

An event trace file is a collection of event records which represent the run-time behaviour of a parallel program. Each event record corresponds to an event which happened during the execution of the parallel program. Figure 4–1 depicts a fixed size event record format. An event trace file can contain thousands of these records. Even though the event records in a trace file look like a single stream of events , the logical structure of an event trace is not a single stream. Each process in a parallel program has its own individual event stream which is similar to an event stream of a sequential program. Since a message passing program consists of more than one sequential process, organising an event trace as a collection of small event streams is quite natural.

Each event stream in an event trace represents the behavioural pattern of the process to which it belongs. By tracing the communication events in a stream, one can gain an understanding on the interactions of a particular process. In Figure 4–2, the directed graph of a parallel program and its event trace is exemplified. In

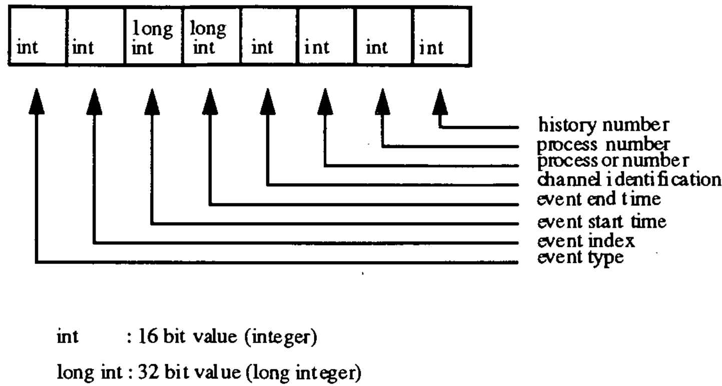
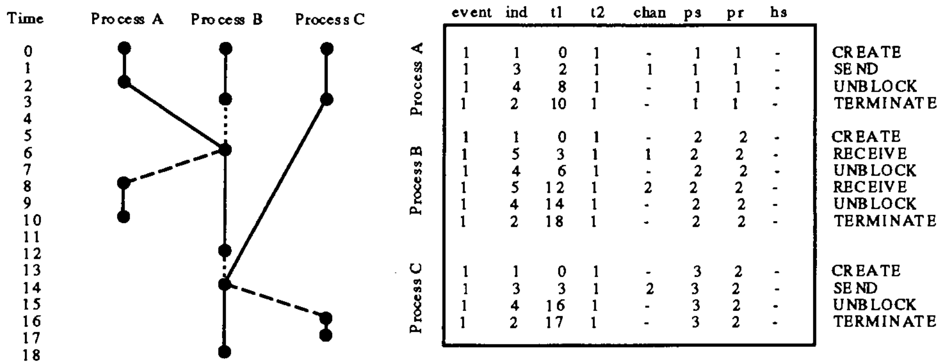


Figure 4–1: The event record format



	event	ind	t1	t2	chan	ps	pr	hs	
Process A	1	1	0	1	-	1	1	-	CREATE
	1	3	2	1	1	1	1	-	SEND
	1	4	8	1	-	1	1	-	UNBLOCK
	1	2	10	1	-	1	1	-	TERMINATE
Process B	1	1	0	1	-	2	2	-	CREATE
	1	5	3	1	1	2	2	-	RECEIVE
	1	4	6	1	-	2	2	-	UNBLOCK
	1	5	12	1	2	2	2	-	RECEIVE
	1	4	14	1	-	2	2	-	UNBLOCK
	1	2	18	1	-	2	2	-	TERMINATE
Process C	1	1	0	1	-	3	2	-	CREATE
	1	3	3	1	2	3	2	-	SEND
	1	4	16	1	-	3	2	-	UNBLOCK
	1	2	17	1	-	3	2	-	TERMINATE

Figure 4–2: An example Program Activity Graph (PAG) and event trace

this simple example, there are three processes, two of which send messages to the third one. It is possible to express the run-time behaviour of the processes as a finite state machine using the communication events as tokens. For example, the process B in Figure 4–2 can be expressed as the string of events "abcbcd" where 'a' is *<create process event>*, 'b' is *<initiate message receive event>*, 'c' is *<unblock process event>*, and 'd' is *<terminate process event>*.

While it is fairly easy to understand individual processes of a parallel program by expressing individual event streams as finite state machines, the overall behaviour of the parallel program can be very difficult to express using the same technique. Even though the time-stamps of events can provide a total ordering for a particular execution of a parallel program, it is likely that a completely

different ordering would be observed in another execution of the same program. This is because the dependencies between the processes are loosely coupled. In other words, we cannot express a parallel program as a totally ordered string of events. Thus the notation for expressing the behavioural patterns of a parallel program should contain facilities to express partially ordered interactions of a parallel program. These partially ordered interactions between the processes of a parallel program are the main reason for keeping separate event streams, and the interpretation of interprocess relations should be open to the user's preferences. Either the user considers time-stamps of the events to provide total ordering, or (s)he interprets some of the events in a partially ordered context and ignores the existing time-stamps. By separating the event streams of individual processes, we provide flexibility to the users. In the next section, the communication pattern is used to exemplify how this event stream structure simplifies the user's task of analysing the program.

4.2.2 Event Patterns

An event trace of a parallel program consists of primitive events which represent atomic interactions of the execution of the program. These events cannot be used directly to understand the run time behaviour of a parallel program unless the parallel program is small. In a small program, the user can recognise event patterns which are manifestation of high level interactions between processes. What is meant by "high level interactions" depends on what the user is investigating. The user may be looking for a series of interactions between specific processes, or may be trying to extract application specific metrics from the event trace. This can be carried out manually by colouring the event patterns on a Gantt chart. The user tries to understand the program by tracing these coloured event patterns which are familiar to him/her since (s)he recognises and assigns meanings to them. The coloured Gantt chart presents concise information. Since interesting patterns are

highlighted it is possible to zoom out and to view the event trace from a wider angle.

Massively parallel programs can be very difficult to understand in detail even if its processes are replications of a few simple process types. The manual colouring mechanism is also not easy to use because there are many patterns to colour. The same colouring task can be carried out on massively parallel programs provided that a special tool supports the colouring task in finding the patterns in event traces. Since there are not many pattern types, but many instances of several pattern types to be coloured, the colouring task can be carried out by allowing the user to define the patterns, and letting the computer find and colour them.

A proper pattern definition mechanism is essential for carrying out the computer aided colouring task. The colouring mechanism should be as easy to use as manual colouring, and should also allow the computer to interpret definitions and generate results efficiently. In the rest of this section, a pattern definition mechanism is discussed. In Section 4.2.3, user aspects of this mechanism are discussed in the context of implementing a tool based on colouring event patterns (i.e. the event abstraction mechanism).

Temporal Relations

In the performance measurement context, time is the main resource which is to be optimised. As well as using absolute time (e.g. January 19), we can also use relative time (e.g. yesterday) to indicate the time of an event. Both absolute and relative time are equally used to measure performance characteristics of a parallel program. Relative time can be more descriptive about the behaviour of a parallel program since it is used to express temporal relations between different parts of a parallel program. For example, it may not make too much sense to say that "a computation phase takes 30 ms" but it may make greater sense to say "a computation phase takes 30 ms and approximately 20% of this phase overlaps

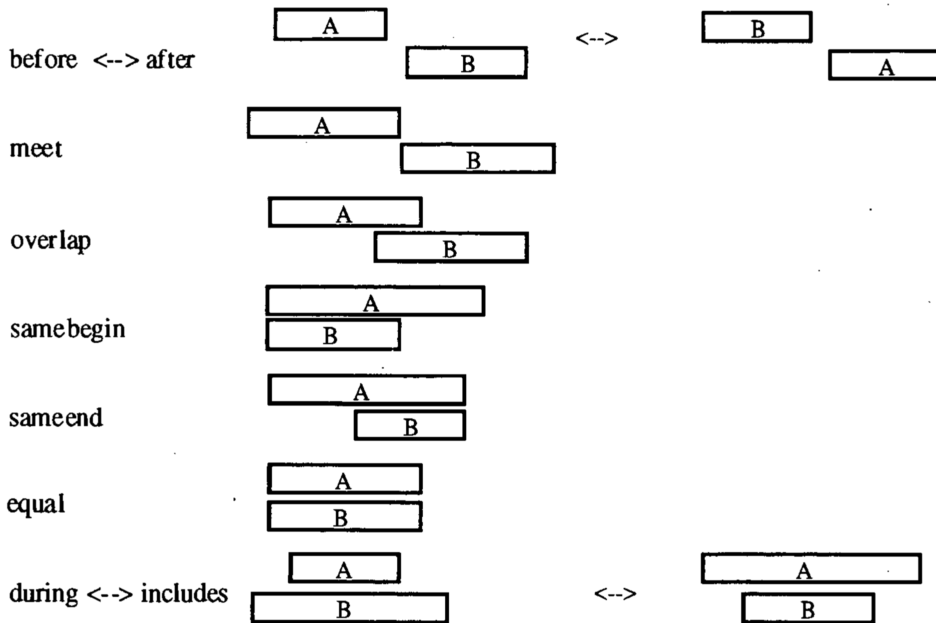


Figure 4–3: Temporal relations between two events

with the communication phase". Relative time, in this example, is used to express parallelism between two different phases of a parallel program.

In Figure 4–3, temporal predicates used in a temporal database query language [33] are depicted. Given two events, these predicates can express any temporal relation between these two events. In the implementation of the event abstraction language which is presented in this thesis, a similar mechanism is used to express temporal relations.

Total Ordering vs. Partial Ordering

Physical time can be misleading unless it is interpreted correctly. Events in an event trace are ordered using physical time which can help the user to determine whether these events occurred sequentially or parallel. In this case, we rely on totally ordered event traces to analyse an execution of a parallel program. In fact, most of the events in an event trace are not totally ordered even though physical time-stamps are attached to the event record. The physical time-stamp

only represents the temporal relations for an instance of an event pattern. The same pattern can include events which are ordered differently in the next execution of the same code, or more generally, in the execution of a similar piece of code which is capable of generating the same type of event pattern. For instance, the communication pattern is a very simple example which demonstrates partially ordered relations between events in the same pattern.

In Figure 4–4, nine different orderings of a communication pattern are depicted. A communication sequence can be identified by four events, two of which belong to a sender process ("SEND" and "UNBLOCK"), and two of which belong to the receiver process ("RECEIVE" and "UNBLOCK"). These nine different orderings of the events in the communication pattern can be observed in an event trace and each instance is a special case of the communication pattern whose events are ordered by using physical clock. By studying the physical clock, for example, we can tell if the receiver blocked too long until it received the message, or if the sender was too early to send the message, etc. When we want to express what a communication pattern is, it is not very efficient to define every possible combination of the communication pattern. A general definition can be given using a logical clock which represents "happened before" relations.

In Figure 4–5, partial ordering of a communication pattern is depicted. In this figure, "SEND" and "RECEIVE" events are represented in the same time interval. This means that these two events can occur in any order in the interval. We cannot define any "happened before" relation between these two events, they are concurrent events. The same things can be said for the "UNBLOCK" events which are defined as concurrent events. We can define the "happened before" relations between the events of the communication pattern as shown in Figure 4–6.

It is important to include facilities to express partially ordered event patterns in a pattern based language for event abstraction. This provision can simplify

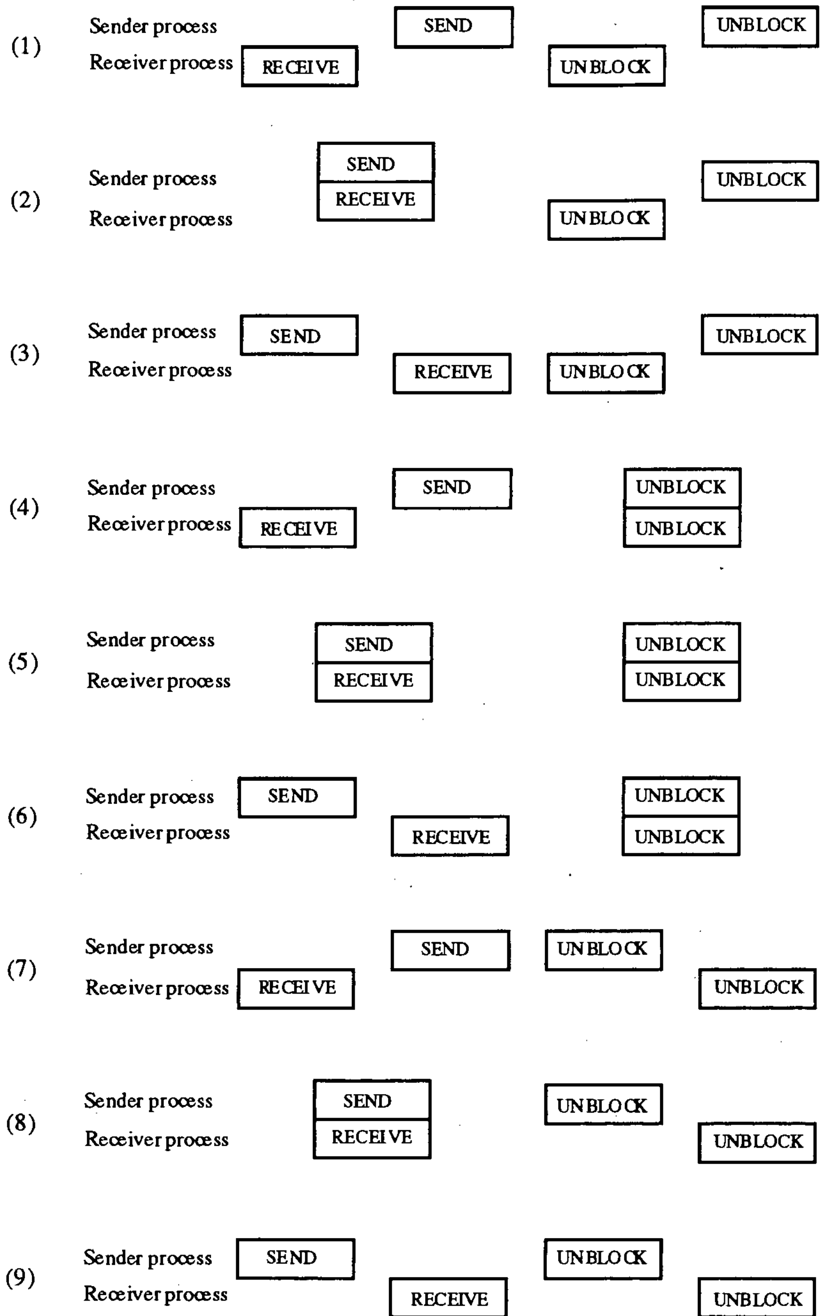


Figure 4–4: Possible orderings of a communication pattern

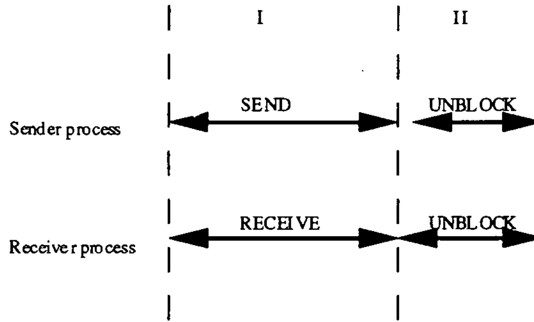


Figure 4–5: Partial ordering of a communication pattern

SEND event → UNBLOCK event of sender process,
SEND event → UNBLOCK event of receiver process,
RECV event → UNBLOCK event of sender process,
RECV event → UNBLOCK event of receiver process

Figure 4–6: The "happened before" relations between the events of the communication pattern

pattern definitions in most of the situations, such as the communication pattern which is exemplified here. In this example, the temporal relations of the events which constitute the communication pattern, can be expressed in one definition (Figure 4–5) using partial ordering information instead of nine definitions (Figure 4–4) using total ordering (i.e. ordering using physical clock information).

Filtering, Aggregating and Inheriting

Event abstraction is achieved by filtering and aggregating event traces. Every time an aggregation occurs, the event trace contains a smaller number of events. Aggregate events replace patterns of events, and inherit some information from these events. Thus the granularity of aggregate events is usually bigger than the events they replace, though there are some cases where aggregate events can be smaller than the patterns which they replace. As long as the number of events to be visualised is reduced, small granularity aggregate events do not present any

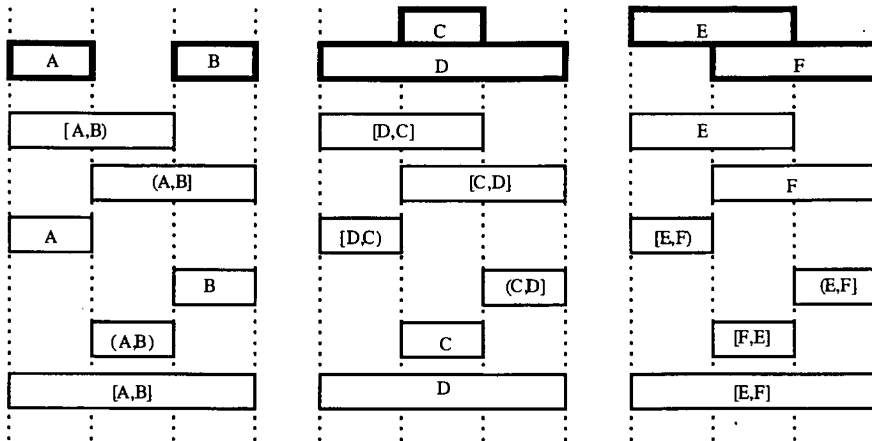


Figure 4–7: Temporal event creation operations

problem with visualisation. These small granularity aggregate events are intended to mark important turning points of an event history rather than long periods of the history.

An aggregate event can be defined using the time references of the events in a pattern which is going to be replaced by the aggregate event. In Figure 4–7, possible aggregate event combinations which can be created from two-event patterns are depicted. In this figure, there are three separate event pairs: one non-overlapping, one in which one event includes the other, and one overlapping events. Two events divide the time interval into three sub-intervals. There are six different possibilities for creating new aggregate events out of these combinations. The notation (inside the rectangles in Figure 4–7) is adopted from the one which is used for expressing intervals. For example, [A,B) reads as "the interval starting from event A ending at event B, including event A but excluding event B. One of the most interesting combinations in this figure is [F,E] (or C). This combination represents the intersection between two overlapping events, which represents the parallelism between the two. For example, let us say the events are computation phases from two different processes. The aggregate event which is achieved using the intersecting region represents the parallelism between these computation

phases. Similarly, we can use [E,F) or (E,F] to represent the sequential parts of the two computation phases.

The format of an aggregate event record is exactly the same as the format of a primitive event record. When an aggregate event is created the first information to be inherited is the time-stamps as explained previously. Other information is provided from the event pattern which the aggregate event replaces. Since it is possible that the pattern contains events which originate from different processors or processes, the user selects the information to be inherited. For example, in a communication pattern, there are two processes which contribute the pattern: the sender and the receiver processes. The user can choose the process to which the new aggregate is attached. Let us consider that the new aggregate event represents the blocking time of a receiver process where the message is not available. In this case, it is natural to attach the new process to the receiver process. The processor information is also inherited from the receiver process for obvious reasons. There is one more information field in the event record to be filled in: the channel identification. In the case of a communication pattern, channel information can be inherited from the either of the processes since the channel identifications of the sender and the receiver are identical.

After the pattern is found and the new aggregate event is created, the events which form the event pattern can be erased from the event trace. Since it is not always desirable to erase all events of the pattern, the user decides which events are to be deleted. Once an event is deleted from the event trace it eliminates the possibility of matching the same pattern and creating the same aggregate event more than once.

Erasing events from an event trace is the simplest form of filtering. Once an event is erased, it cannot be recovered unless the event trace file is read again, and the proper abstractions are repeated to get another copy of the same event. The user needs some facilities for inspecting the abstractions backwards to gain deeper

understanding about how an abstraction was achieved, and what events are used to get there. This kind of inspection can be facilitated by storing the filtered events hierarchically. This can be done either by the system automatically or by the user manually. Storing filtered events automatically is not always possible since a large number of events can be created and filtered repeatedly. This causes some storage management problems such as deciding how to erase some of the filtered events to overcome memory shortage. Another way of storing filtered events is to provide different filtering operators which can delete events from the event trace and store them in hierarchical fashion. In following paragraphs, some proposed event filtering operators are discussed.

The event filtering operators which are proposed in this thesis are depicted in Figure 4–8. The aim of these filtering operators is to help the user to keep filtered events in hierarchical tree structures which represent the history of filtering until the current abstraction is achieved. These trees can be browsed by the user through special tools to investigate the details of an abstraction. For example, let us consider an abstraction of a "program phase". The user can spot the longest phase among the many phases, and investigate the detailed interactions using the filtered events. The filtered events in this case may represent "communication and computation sub-phases" which constitute the "program phase". The events which represent the "communication and computation sub-phases" may have more filtered events which represent the details of the sub-phases.

In Figure 4–8, filtering operators are explained using an example. In this example, the events above the horizontal line represent the events in the event trace, and the events below the line represent the filtered events which are no longer visible in the trace. The operators are "HIDE", "REPLACE", "RESTORE" and "DELETE".

The "HIDE" operator is the most conservative filtering operator of all. This operation basically removes the events from the event trace and connects them to

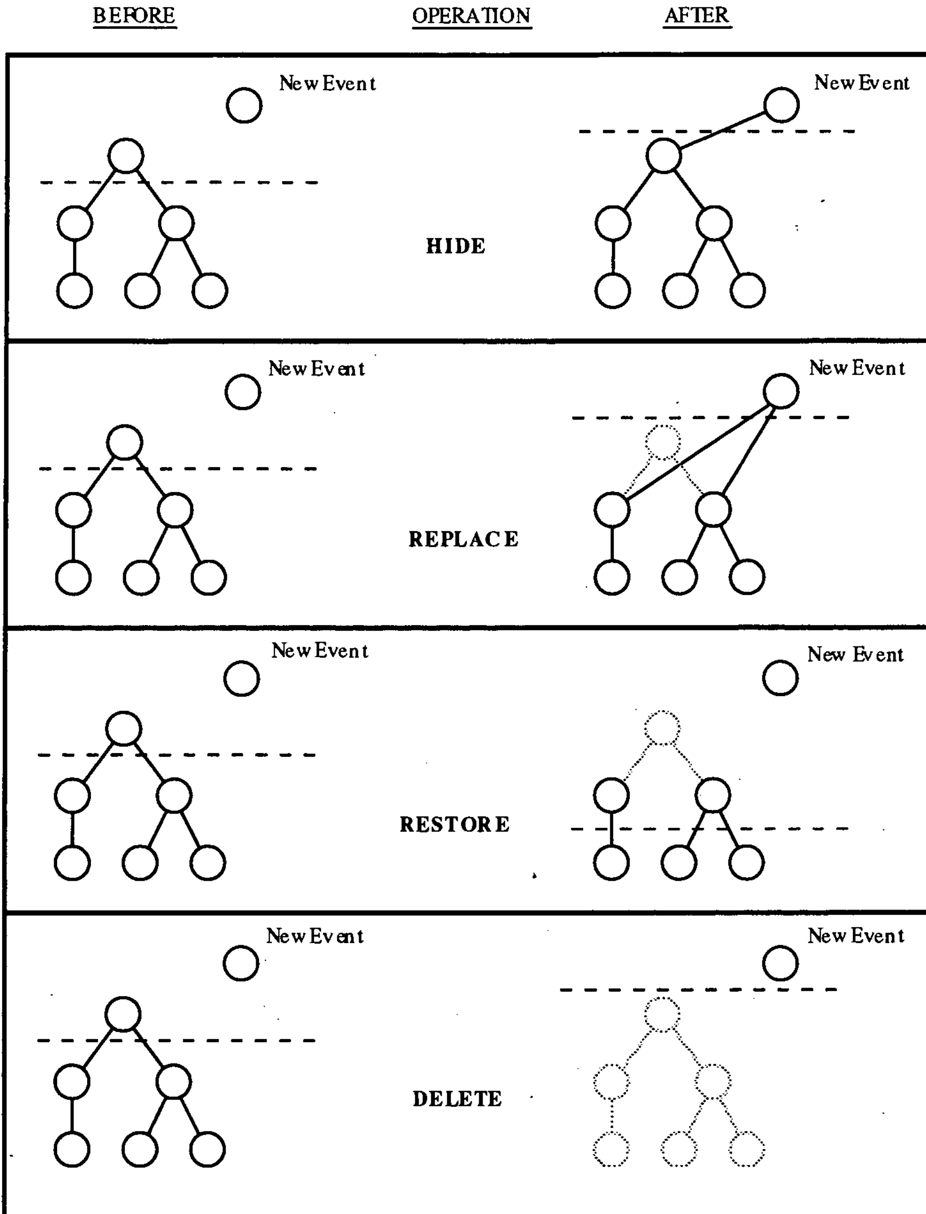


Figure 4–8: Event filtering operations

the newly created event. Even though this is not very economical operation in terms of storage space, it can be very useful when the user wants to see the details of an abstraction. Sometimes an abstraction mechanism requires repetitive event creation and filtering, and the user does not want to keep interim abstractions. In this case, "REPLACE" operator can be used for filtering interim events from the event trace. The events which are filtered by the interim events are inherited by the new event. Since this abstraction mechanism is intended for interactive usage, it should permit the user to recover details of an abstraction when necessary. To be complete in this context, a reverse filtering operator is also provided. This is the "RESTORE" operator which deletes an event and restores the filtered events to the event trace. This operator is not intended as an "undo" operator which recovers the previous state of the abstraction but as a complementary operation which can provide the low level details when they are required. The last filtering operator ("DELETE") is the most destructive filtering operation of all. It deletes events and their child events (filtered events) when the user does not require them any more.

4.2.3 Front-ends for the Event Abstraction Mechanism

A tool for the event abstraction mechanism consists of three main parts: a front-end for entering the event pattern definitions, an abstraction engine which carries out the pattern matching, filtering and aggregating, and an output unit which formats and transfers the data to the visualisation tool. In this section, front-end options for the event abstraction mechanism are discussed. Since the goal of the event abstraction mechanism is to make detailed performance analysis data available to a user, and speed-up the performance tuning phase of the program development cycle, the tool should provide fast and efficient facilities to the user in analysing event traces. Two front-ends have been implemented for entering event pattern definitions. One of them is a graphical user interface which allows the

user to define and modify event patterns very quickly, and the other is a textual interface which is an extension of the C programming language.

The Graphical Front-end

There are always advantages in using a graphical representation for data which has more than one dimension. Event traces of parallel programs which have the dimensions of time and space fall into this category. The space dimension reflects the multiplicity of either data or instructions in MIMD programs, while the time dimension reflects the changes in data or in the instruction sequences over time. Thus the graphical front-end for the event abstraction mechanism uses the same dimensions for defining event patterns.

In Figure 4–9, the communication pattern of a message passing parallel program is defined using the graphical front-end. In this figure, there are three pattern definitions, each of which represents the total ordering of a possible communication sequence. Detailed information concerning the orderings of the communication pattern can be found in Section 4.2.2. In the graphical representation of a pattern definition, each event is represented as a rectangle containing the event name. Events can then be grouped together in large rectangles to define the process or the processor to which they belong. For example, in the communication pattern definitions, each definition contains four event rectangles, one new event rectangle, and two process rectangles. The horizontal positions of event rectangles represent temporal relations, as previously defined in Figure 4–3.

The representation of the temporal relations also accommodates a notation to facilitate event patterns which contain partial ordering. For example, the communication pattern has some events which cannot be ordered using a logical clock. Figure 4–10 depicts this pattern as an example of partially ordered event patterns. The "RECEIVE" event and the "UNBLOCK" event of the sender process are encapsulated by a rectangle. This rectangle defines two levels of tem-

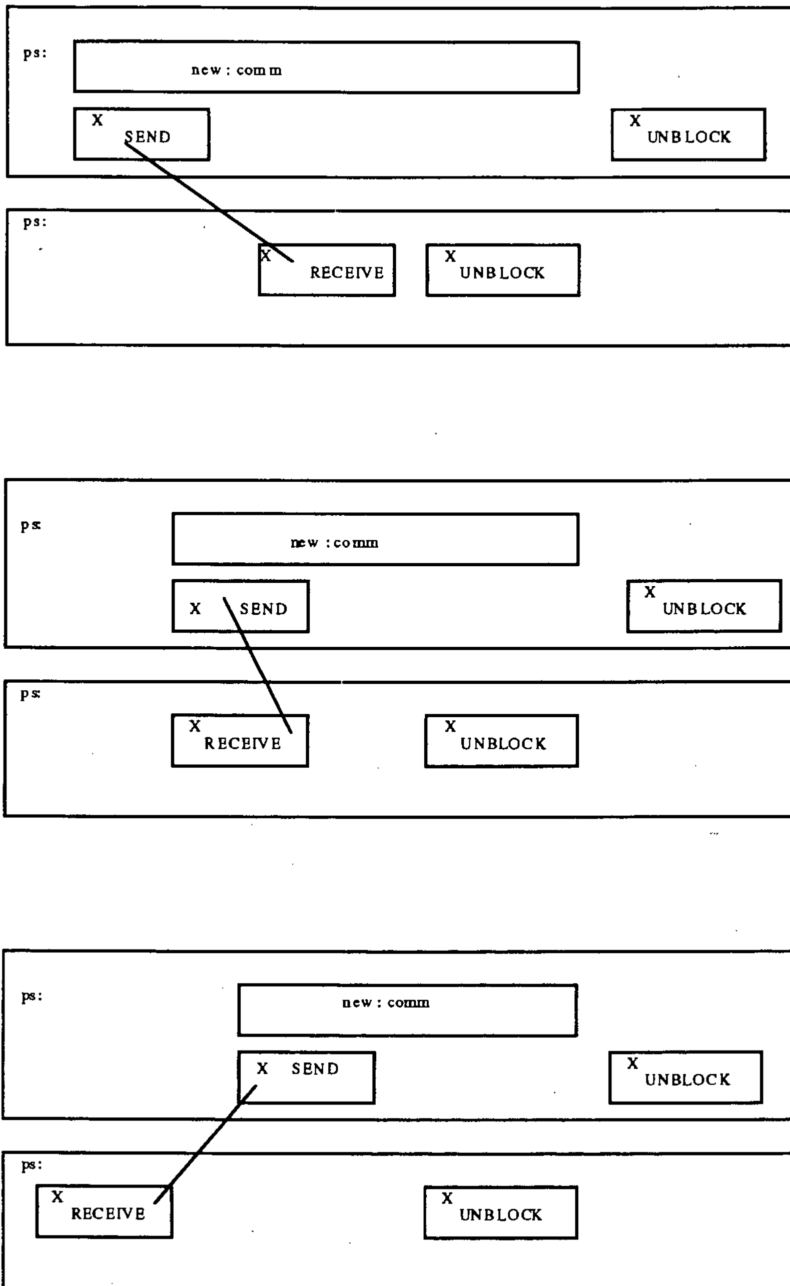


Figure 4–9: Possible orderings of a communication pattern using graphical interface for defining event patterns

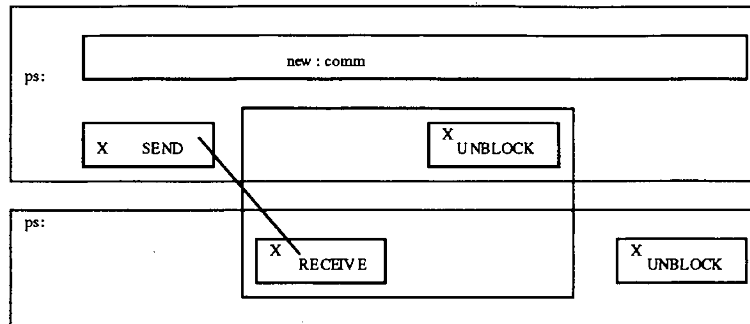


Figure 4–10: A communication pattern using graphical interface for defining event patterns

poral regions (inside and outside the rectangle) each of which has an event pair: $(RECEIVE, UNBLOCK_{sender})$ and $(SEND, UNBLOCK_{receiver})$, respectively. The temporal relations $SEND \rightarrow RECEIVE$ and $UNBLOCK_{sender} \rightarrow UNBLOCK_{receiver}$ are excluded because corresponding event pairs are not at the same level. However, the temporal relations $RECEIVE \rightarrow UNBLOCK_{sender}$ and $SEND \rightarrow UNBLOCK_{receiver}$ are preserved because corresponding event pairs are at the same level. The other temporal relations $RECEIVE \rightarrow UNBLOCK_{receiver}$ and $SEND \rightarrow UNBLOCK_{sender}$ are also preserved because the corresponding event pairs belong to the same event stream (i.e. they belong to the same process) in which events are completely ordered.

The line which connects "SEND" and "RECEIVE" events in Figure 4–10 defines that these two events should have the same channel identification. When the pattern is found in an event trace, the abstraction engine carries out filtering operations. The 'X' sign in an event rectangle tells the abstraction engine to filter the event. In the example, four of the events in the pattern are filtered. Finally, a new aggregate event(s) is created by the abstraction engine. In the example, there is one aggregate event definition whose event type is "comm". This new aggregate event inherits its first time-stamp from the "SEND" event (of the process which is defined at the top part of the pattern), and the second time-stamp from the "UNBLOCK" event (of the process which is defined at the bottom part of the pattern).

The graphical syntax which is used for defining event patterns is also suitable for an interactive front-end. For example, the user can highlight events from an existing Gantt chart display by choosing them with a pointing device such as a mouse or light pen. Then, by drawing lines (channel identification) and rectangles (process, processor boxes) in this view, the user can define the patterns very quickly.

Textual Front-end

It is not always desirable to have an interactive graphical front-end. In some situations, the user may require more powerful facilities than the graphical front-end can provide, and may want to skip some of the steps, such as selecting events, modifying pictures, etc. Thus for the event abstraction mechanism, a textual front-end has been defined, as an extension of the C Programming Language, containing a library of abstractions. This has to be implemented once, but efficiently, since they are used frequently. The textual front-ends also do not demand as much system resources as a graphical front-end does (e.g. graphical terminal, window manager).

Event patterns are defined in a declarative fashion which provides a similar kind of interface to the graphical front-end. Each event in a pattern is declared with the conditions which have to be held when the pattern is searched in an event trace.

Each line of an event pattern definition is a C function call which operates on the event trace. There are four functions : `eventfind()`, `nextevent()`, `create_event()` and `filter_event()`. These functions are placed by the C Preprocessor in an appropriate place in the program which searches the event patterns in the event trace. The parameters of each function call set the conditions which the event has to satisfy.

```
#define PATTERN1_EVENT1 eventfind(&e1,"PRIMITIVE",SEND,0,4000,\
      0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,name("distributor"))
```

Figure 4–11: A line from an event pattern definition

A line from an example event pattern definition is given in Figure 4–11. This line reads "find a SEND event of process 'distributor' where the event is before clock tick 4000".

The first parameter of the "eventfind" function is a pointer to the data structure in which the details of the event is filled in when there is a successful match. The second and the third parameters are respectively the abstraction level and the event type. The fourth and fifth parameters are the temporal conditions which represent the minimum and the maximum start time of the event. Similarly, the next two parameters are the temporal conditions for the finish time of the event. The last three parameters are used for identifying channel, history, processor and process information.

The "nextevent" function has the same parameters as the "eventfind" function except for the first parameter. This is an additional parameter which points to the previous event. The "nextevent" function searches for the next event at the designated abstraction level without being affected by the events at other abstraction levels.

Each event definition in an event pattern is defined as PATTERNn_EVENTm where n is the pattern number and m is the event number within the pattern.

```
#define PATTERN1_EVENT1 eventfind(&e1,"PRIMITIVE",SEND,0,INFINITY,\
      0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN1_EVENT2 nextevent(&e1,&e2,"PRIMITIVE",UNBLOCK,\
      0,INFINITY,0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN1_EVENT3 eventfind(&e3,"PRIMITIVE",RECEIVE,\
      0,e2.t1,0,INFINITY,e1.chan,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN1_EVENT4 nextevent(&e3,&e4,"PRIMITIVE",UNBLOCK,\
      e1.t2,INFINITY,0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)

#define PATTERN1_CREATE1   create_event(&new_event,"comm",0,\
      e1.t1,e4.t2,e1.chan,e1.hs,e1.pr,e1.ps)

#define PATTERN1_FILTER1 filter_event(&e1)
#define PATTERN1_FILTER2 filter_event(&e2)
#define PATTERN1_FILTER3 filter_event(&e3)
#define PATTERN1_FILTER4 filter_event(&e4)

eventtype e1,e2,e3,e4,new_event;
compose()
{
while(pattern1());
pr_parallelism(name("comm"));
}
```

Figure 4–12: Textual definition of a communication pattern

```
find consecutive SEND and UNBLOCK events from process x
find consecutive RECEIVE and UNBLOCK events from process y
      where
           the channel identifications of
           SEND and RECEIVE events are equal
             and
           UNBLOCK (process y) event is after SEND event
             and
           RECEIVE event is before UNBLOCK (process x)
create an event named ''comm'' (of process x)
    which starts at the same time with SEND event of process x
            and
          ends at the same time with UNBLOCK event of process y
filter SEND, RECEIVE and both UNBLOCK events
```

Figure 4–13: Structured English definition of a communication pattern

The pattern definition of a communication event is given in Figure 4–12. This event abstraction can be expressed in structured English as shown in Figure 4–13. In this example, all of the event patterns which match with this definition are found and replaced with new events named "comm" which represents the transfer time of a message from the sender process to the receiver process.

Reducing user errors: Graphical vs. Textual Definitions

Since one of the purposes of performance analysis and visualisation tools is to speed-up the performance tuning task, it is important to provide efficient tools which help users to do experiments quickly. One of the factors which causes slow-down is user errors, and in this context, the graphical front-end has some advantages over the textual interface.

Table 4–1: Graphical and Textual Definitions in the context of user errors

	Clerical	*Syntactic*	*Semantic*	*Correction Handling*
QL Parameters	*Probability of Error*			*Effort*
QL Types	*Query Formulation Errors*			
Graphic or Pictorial	Low-Medium	Medium	Low-Medium	Medium
Linear Keyword	High	Medium-High	Medium-High	High
Record at a time	High	High	High	High
Mathematical	High	High	Medium-High	High

Table 4–1 is an extract from a taxonomy of database query languages [67]. It is possible to apply these results to the event abstraction case considering that both finding event patterns in an event history and querying a database are similar tasks. In the table, four different types of query languages (QL) are compared in the context of the error types *clerical*, *syntactic* and *semantic*. The textual QL types in Table 4–1 (all types except the first one) are similar in terms of handling a query, but different in expressing it. The *linear keyword* uses English-like phrases with a definite syntax which consists of words from a specific reserved list. The *record-at-a-time* formalism is an extension of an existing programming language, and the syntax of the queries are adapted from the host language. The *mathematical* formalism uses short and succinct expression of powerful operations. The graphical and textual front-ends for the event abstraction mechanism correspond to the *graphic or pictorial* and *record-at-a-time* types, respectively. In Table 4–1, the probabilities of user errors involving *graphic or pictorial* types are the smallest. Also, correction handling effort for the *graphic or pictorial* types is less than the others.

The *clerical* errors are usually typos which are often related to text editing.

This kind of errors are less likely in the graphical front-end since there are much fewer textual objects in graphical definitions than textual definitions.

The *Syntactic* errors can be detected easily in both graphical and textual front-ends. However, graphical definitions are more advantageous than textual definitions since valid definitions of graphical objects can be forced by using a special purpose graphic editor. For example, this kind of editor can handle composite objects (e.g. grouping event rectangle, event name and filtering operator), restrict illegal positioning of graphical objects (e.g. overlapping event rectangles), and provide special purpose editing capabilities (e.g. the line which connects two event rectangles is updated automatically when one of the rectangles is moved). In text editors, syntax errors cannot be detected until a meaningful part of the definition is entered (e.g. a separator). These kinds of syntax errors cannot be prevented by text editors but a warning message can be produced (e.g. beep sound, flashing cursor, etc.).

The *semantic* errors are the most important error type because some of the erroneous conditions are syntactically valid definitions which are not conceptually correct or not suitable for a specific situation. As a result of this kind of error, the user can get a wrong reply to his/her query, and may then lead the user to an incorrect decision [53]. Recovering from these kinds of errors can be time-consuming because they may go undetected for some time. The graphical definitions are easier to understand because they resemble the Gantt chart display which is familiar to many users. This similarity should reduce *semantic* errors to some extent. It is even possible to provide extra capabilities to the graphical front-end to reduce semantic errors. For example, the task of defining event patterns can be made more interactive by finding the partially defined event patterns (i.e. incremental search) in an example event trace, and superimposing these instances of events onto their definitions. This feed-back mechanism will give the user opportunity to browse a relevant part of a real event history before the definition is complete.

Some *semantic* errors are impossible to make when the graphical front-end is used. Cyclic temporal references are an example of such errors. A simple illustration of this type of error can be given for a pair of events a and b. The temporal relations $a \rightarrow b$ and $b \rightarrow a$ represent a cyclic temporal relation between a and b. Since this situation is impossible for event histories, it should be avoided. The graphical front-end does not allow the user to define cyclic temporal references because it is impossible to arrange the rectangles in such a way. In a textual pattern definition, the cyclic temporal relations cannot be seen easily when there are many events (e.g. more than three) in the pattern, and this type of error can be made easily. For example, when the user wants update a temporal condition which is attached to an event, (s)he has to check other event definitions which give reference to this particular event, and if necessary, they have to be updated explicitly by the user. If the user fails to update any of the references, a cyclic temporal relation can be defined unintentionally. Since temporal conditions are defined by the positions of the event rectangles in a graphical event pattern definition, any change to one of the temporal conditions (i.e. adding/deleting/moving/resizing an event rectangle) will also update other temporal conditions automatically.

Not all errors can be prevented by the user interfaces, but graphical user interfaces can reduce the possibility of making such errors.

4.2.4 Event Abstraction Examples

An Application Specific Metric: Broadcast and Gather Phases

In order to show how the visualisation approach can be applied to analysing parallel programs, some experiments were conducted on a test case program to visualise its performance characteristics. The test program is not a complete application, but a template program which performs global operations on a distributed data set. A complete application could be implemented using this template program.

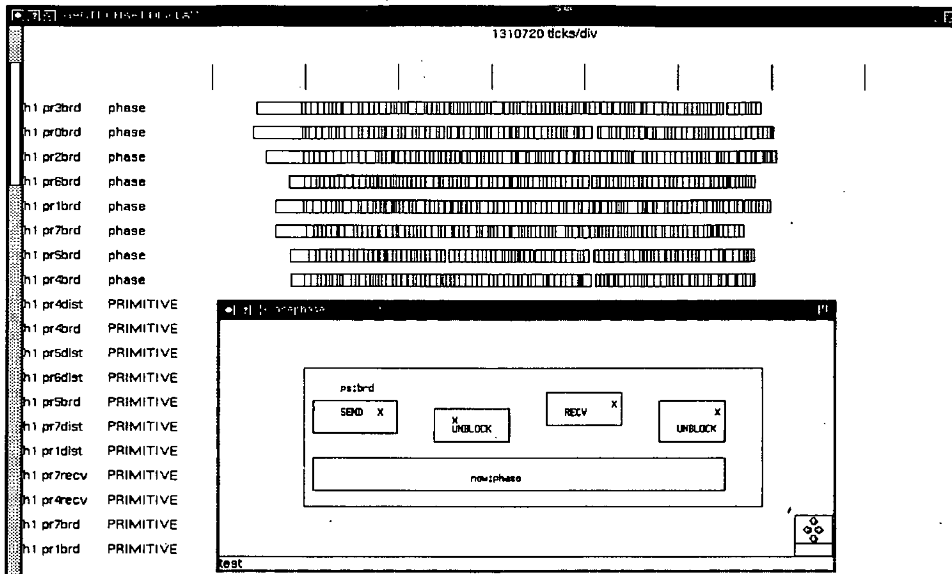


Figure 4–14: Broadcast and gather phases of the template program.

In the experiments, the performance characteristics of this template program were explored (see section 5.3.1 for the details of the program).

One important characteristic is the speed with which broadcast and data gather operations occurred; these operations are investigated closely. The initial Gantt chart display of the monitoring trace is depicted in Figure 4–14. In this case, primitive event traces are replaced by an abstract event which is called a 'phase'. The definition of this abstraction is also depicted in the same figure. In the definition, four primitive events from the 'broadcaster' process are replaced by an event called 'phase'. These four primitive events represent two consecutive communication operations: the broadcaster sends a request message and receives the answer after some time. The new event represents the elapsed time for a broadcast and gather phase. By visualising these 'phase' events, we can identify if any of the phases takes longer than the others.

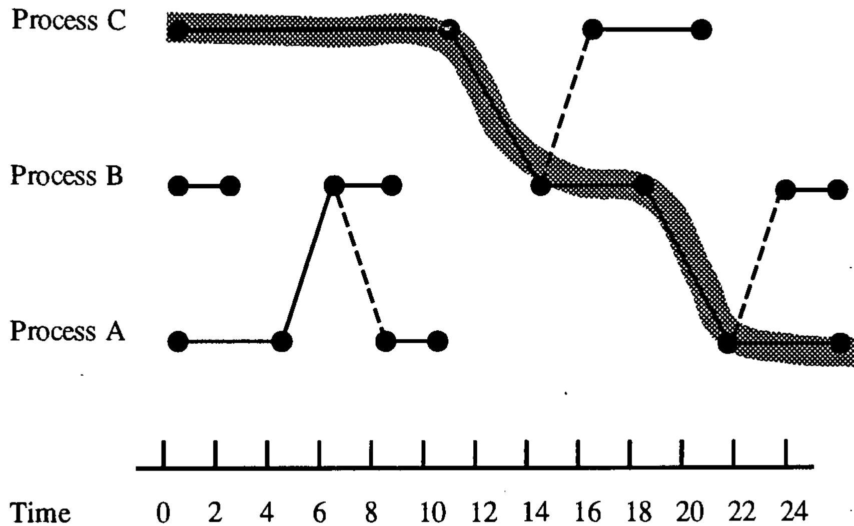


Figure 4–15: The critical path of a simple PAG

A Standard System Metric: The Critical Path

The critical path [69] is a profiling type of performance metric that not only gives an idea about the performance characteristics of a parallel program but also helps the user to locate the problematic sections of the program. The critical path of a parallel program is the longest CPU time weighted path through the Program Activity Graph (PAG), which can be constructed using the primitive events of a parallel program trace. In Figure 4–15, the critical path of a simple PAG is depicted. The PAG segments which are on the critical path can be identified by abstract "Critical Path" events which overlap with these segments. Since the critical path events individually mark the segments of the event trace, they can be mixed with any abstract event to locate any particular behaviour which can seriously change the overall performance. For example, the user can achieve abstract events which answer questions such as "Is the broadcast phase on the critical path?" or "Which process is on the critical path most?".

The critical path can be found by using the event abstraction mechanism. Event patterns are declared which check communication points of the program starting from the beginning. In Figure 4–17, an initial abstraction to find the

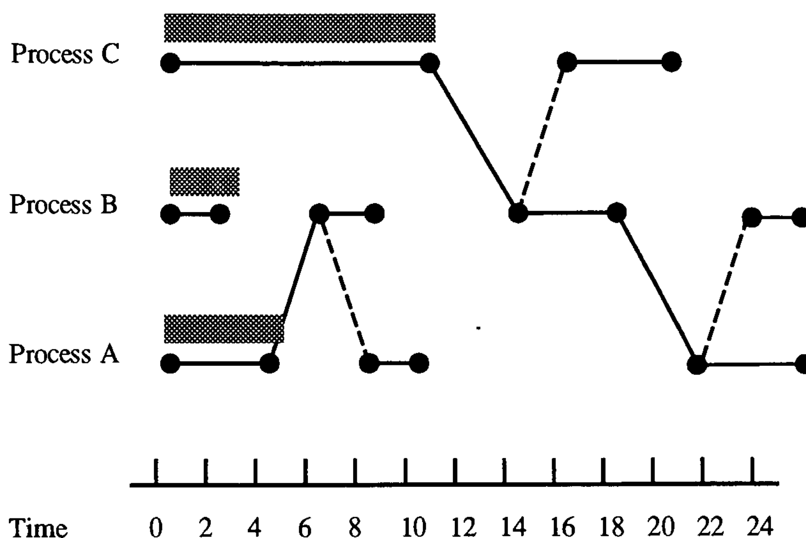


Figure 4–16: Initial costs for CP

```
#define PATTERN1_EVENT1 eventfind(&e1,"PRIMITIVE",CREATE,\
                   0,INFINITY,0,INFINITY,\
                   NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN1_EVENT2 nextevent(&e1,&e2,"PRIMITIVE",SEND,\
                   0,INFINITY,0,INFINITY,\
                   NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)

#define PATTERN1_CREATE1 create_event(&new_event,"CPM",CR,\
                   e1.t1,e1.t2,NULL_CHAN,e1.hs,e1.pr,e1.ps)
#define PATTERN1_CREATE2 create_event(&new_event,"PRIMITIVE",COST,\
                   0,e2.t2-e1.t1,e2.chan_id,e1.hs,e1.pr,e1.ps)

#define PATTERN1_FILTER1 filter_event(&e2)
#define PATTERN1_FILTER2 filter_event(&e1)
```

Figure 4–17: CP pattern number 1: Initial costs

critical path is given. In this abstraction, the computation in each process from the creation of the process to the first send operation is marked as an initial cost which is represented by the "COST" event, i.e. the initial cost represents the computation time which is spent before the first communication. The initial costs of the example PAG is given in Figure 4–16. The "COST" event will be used later to represent the cost of reaching an arbitrary communication point.

The pattern definition in Figure 4–17 only finds the cost event for processes in which the first communication event is message send. There is a separate event pattern definition which finds the cost events for processes in which the first communication event is message receive. This second pattern can be defined by replacing the "SEND" event in "PATTERN1_EVENT2" with "RECEIVE" event.

In order to find the critical path, it is necessary to visit every communication point in chronological order to calculate the cost of reaching the ends of these points. Since a communication takes places between two processes, each communication point corresponds to a DAG node from which two inwards and two outwards edges emerge—one inward and one outward edge belong to the first process and the other edges belong to the second process. The third pattern definition of the critical path definition is used for comparing the computational costs of two incoming edges (Figure 4–19). This abstraction chooses the most expensive incoming edge and marks it as the last edge of the critical path until this particular communication point. An example is depicted in Figure 4–18.

The next two patterns (pattern number 4 in Figure 4–21 and pattern number 5 which is explained below) are used for calculating the computation cost of reaching the next communication point from the current communication point of the same process, and for adding this cost to a total sum. An example is shown in Figure 4–20. There are two pattern definitions: one for the sender processes and one for the receiver processes. In Figure 4–21, one of these two pattern definitions is

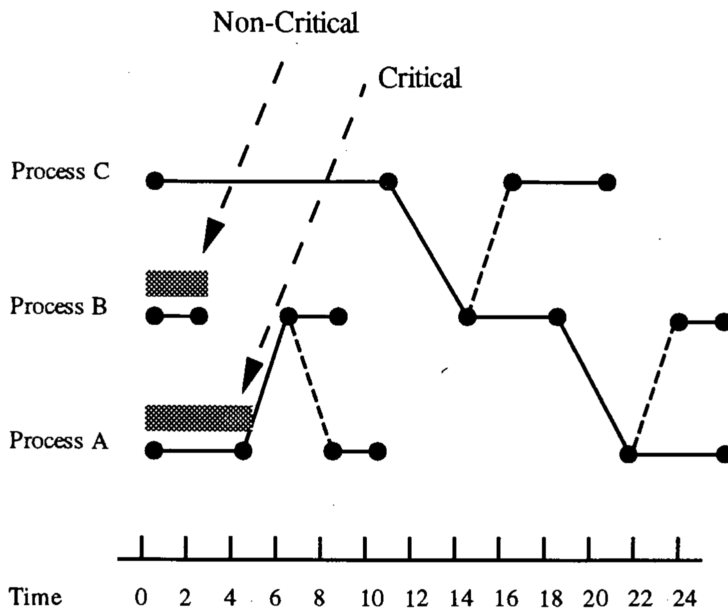


Figure 4–18: Choosing the path which is CP candidate

given. The other pattern definition for the receiver processes can be produced by replacing the "SEND" event of PATTERN4_EVENT2 (Figure 4–21).

The patterns which have been defined up to now implement the procedure for visiting the communication points (DAG nodes of an event history), and for marking the critical path from the beginning. When two separate local critical paths join at a communication point, the costliest has to be chosen and the other one has to be marked as non-critical. All the critical path marks along this rejected local critical path have then to be erased. This procedure is implemented by declaring a pattern definition which erases (filters) the critical path marks (events) from this local critical path. An example of this situation is shown in Figure 4–22, with its event pattern definition being given in Figure 4–23. This filtering definition erases all critical path events from the particular local critical path starting from the end and going back towards its starting point.

These six pattern definitions are used for creating critical path events in the order shown in Figure 4–24. Since each communication point is marked with critical path events as either critical or non-critical, any abstraction can be carried out on

```
#define PATTERN3_EVENT1 eventfind(&e1,"PRIMITIVE",COST,0,0,\
                    0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN3_EVENT2 nextevent(&e1,&e2,"PRIMITIVE",UNBLOCK,\
                    0,INFINITY,0,INFINITY,\
                    NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN3_EVENT3 eventfind(&e3,"PRIMITIVE",COST,\
                    0,0,0,e1.t2,e1.chan_id,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN3_EVENT4 nextevent(&e3,&e4,"PRIMITIVE",UNBLOCK,\
                    0,INFINITY,0,INFINITY,\
                    NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)

#define PATTERN3_CREATE1 create_event(&new_event,"CPM",CR,\
                    e2.t1,e2.t2,e1.chan_id,e1.hs,e1.pr,e1.ps)
#define PATTERN3_CREATE2 create_event(&new_event,"CPM",NC,\
                    e2.t1,e2.t2,e1.chan_id,e3.hs,e3.pr,e3.ps)
#define PATTERN3_CREATE3 create_event(&new_event,"PRIMITIVE",\
                    PROCESSED,e1.t1,e1.t2,\
                    NULL_CHAN,e1.hs,e1.pr,e1.ps)
#define PATTERN3_CREATE4 create_event(&new_event,"PRIMITIVE",\
                    PROCESSED,e1.t1,e1.t2,\
                    NULL_CHAN,e3.hs,e3.pr,e3.ps)

#define PATTERN3_FILTER1 filter_event(&e3)
#define PATTERN3_FILTER2 filter_event(&e1)
```

Figure 4–19: CP pattern number 3: Choosing a CP candidate

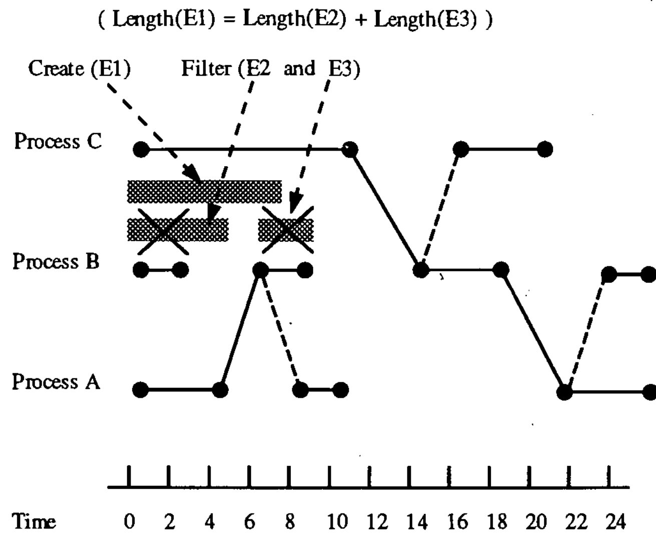


Figure 4–20: Calculating the cost of a node

```
#define PATTERN4_EVENT1 eventfind(&e1,"PRIMITIVE",PROCESSED,0,0,\
                   0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN4_EVENT2 nextevent(&e1,&e2,"PRIMITIVE",UNBLOCK,\
                   0,INFINITY,0,INFINITY,\
                   NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN4_EVENT3 nextevent(&e2,&e3,"PRIMITIVE",SEND,\
                   0,INFINITY,0,INFINITY,\
                   NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)


#define PATTERN4_CREATE1 create_event(&new_event,"PRIMITIVE",COST,\
                   e1.t1,e1.t2+(e3.t2-e2.t1),\
                   e3.chan_id,e1.hs,e1.pr,e1.ps)


#define PATTERN4_FILTER1 filter_event(&e3)
#define PATTERN4_FILTER2 filter_event(&e2)
#define PATTERN4_FILTER3 filter_event(&e1)
```

Figure 4–21: CP pattern number 4: Calculating the cost of a node

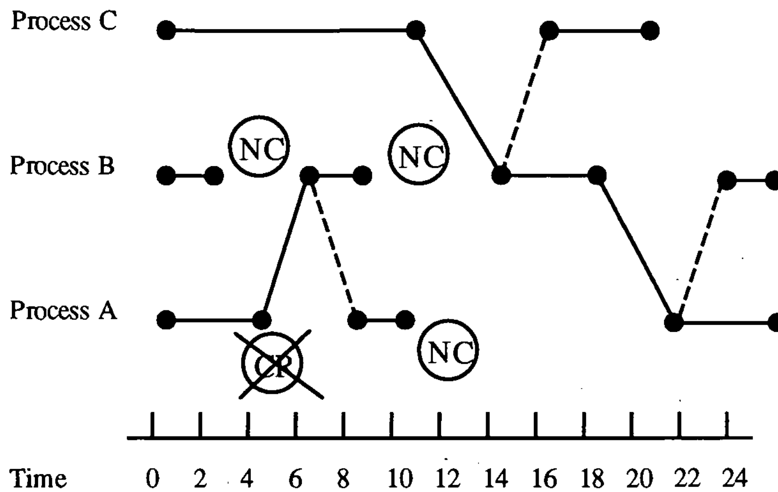


Figure 4–22: Dropping a candidate from CP

```
#define PATTERN6_EVENT1 eventfind(&e1,"CPM",CR,0,INFINITY,\
                   0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN6_EVENT2 nextevent(&e1,&e2,"CPM",NC,\
                   0,INFINITY,0,INFINITY,\
                   NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN6_EVENT3 eventfind(&e3,"CPM",NC,\
                   e1.t1,e1.t1,e1.t2,e1.t2,\
                   e1.chan_id,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN6_EVENT4 nextevent(&e3,&e4,"CPM",NC,\
                   0,INFINITY,0,INFINITY,\
                   NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)


#define PATTERN6_CREATE1   create_event(&new_event,"CPM",NC,\
                   e1.t1,e1.t2,e1.chan_id,e1.hs,e1.pr,e1.ps)


#define PATTERN6_FILTER1 filter_event(&e1)
```

Figure 4–23: CP pattern number 6: Dropping a candidate from CP

```
    ...
while(pattern1()) ;
while(pattern2()) ;
while(pattern3()||pattern4()||pattern5()) ;
while(pattern6());
    ...
```

Figure 4–24: The definition of CP pattern matching order

top of the critical path abstraction in order to determine the connection between a particular program behaviour and the critical path. For example, the user can create another abstraction by finding the intersections between a particular computation phase and the critical path of the program. If this particular phase of the computation is mostly on the critical path, it is possible that any improvement on this particular computation can also improve the overall performance.

4.3 Uses of Event Abstraction in Performance Tuning

4.3.1 Standard System Metrics

The performance characteristics of a parallel program can be extracted from its event history. This is done by defining a set of standard events that represent the various performance metrics. Since these metrics can be calculated without knowledge of the algorithm that is being analysed, the event definitions for the performance metrics can be considered as general purpose definitions. Therefore those metrics on which performance analysis frequently depends can be defined once and kept in a library. The other solution to this problem is to implement

them within the tools as a fixed feature. But the advantage of using event libraries is that they can be modified easily to support various kinds of event histories. For example, if there is an event history that contains different types of events from ones previously encountered, then new library definitions can be created to support the new event type.

Besides performance metrics, the user can customise a library that contains standard event definitions to enhance it with frequently used event types. Since there is no standard way of analysing an event history of a program, an event library offers the user an opportunity to create an environment that reflects his/her own choices and personal experiences of analysing programs. The idea of creating a library for standard events also makes the performance analysis tool very flexible when a new analysis style is found and a user wants to add it to the performance analysis environment. For example, the critical path method [69] has been implemented and is part of the library. This method is useful when a user wants to identify the process interactions that dominate the computation. The definitions for finding the critical path may be applied to any segment of the event history. In this way, different program stages can be analysed separately. The user controls this process by defining the intervals. These intervals, represented by the composite events that are used as conditions within the definitions, are the manifestations of a program stage in which the user is interested. When any occurrence of this event type is created and added to the history, the event definitions for finding the critical path are activated and the interval event is replaced by new events that represent the critical path. The other library definitions work in the same way as the critical path definitions. The user's responsibility is to define the interval events that are inputs to the library definitions. Since a knowledge of the programming model is used for creating the interval events, the results of the library function can be directly related to the program stages.

4.3.2 Application Specific Metrics

General purpose performance displays are useful to some extent. They can display information, which the programmer can browse, for example, showing how fast the program is running or how much communication occurs. The programmer can speculate from these displays about possible reasons for the bottle-necks or regions with poor performance. However, except for the time coordinates on the charts, there is no way of relating performance data to the program stages or the effects of a microscopic behaviour on the overall performance. But, a performance view that depicts performance metrics within the context of program stages, or accommodates user defined queries that address application specific performance problems, can be more useful than the traditional performance displays, because the programmer can analyse the history in more detail and relate the results to the program more easily.

An event history of a parallel program contains enough information to reconstruct what happened during the execution of the program. It is possible to represent the reconstructed program history in terms of milestone events which are manifestations of internal activities of the parallel program. The user then investigates these activities to find performance bottle-necks and tune the performance. The investigation of these program activities can be carried out by referring to the milestone events which represent the program's behaviour model. The activities that have an effect outside a process can be defined as externally observable properties of the process. For message passing systems, communication activities are the only ones that can be considered within this category. The communication patterns of a process contain useful information to identify particular processing stages that are separated by communication activities. The programmer can exploit this information to reconstruct a program's behavioural model by assigning an event to each stage or to a super-stage that is a combination of other stages. In this way, an event history of a parallel program can be translated into a hierarch-

ical history. The aim of this is to achieve a brief history of events that is closer to the programming model than the history of primitive events. Unlike debugging, performance analysis may not require the complete definition of a behavioural model. In the performance analysis case, the programmer can find milestones in the program by identifying unique patterns of communication in its history. These milestones can be referenced to analyse various timing characteristics of a parallel program.

Since the event traces contain information about the stages of a program and temporal properties of these stages, intermediate results of a behavioural model reconstruction can be presented to the programmer to inform him about the timing characteristics of the program. Though these intermediate results are useful to the programmer, they are also the basis for further analysis techniques. Besides, the summarised form of the history can still be very large to browse, and may not reveal the information that the programmer requires for performance tuning. In the following sections, different uses of these intermediate results are given.

4.3.3 Correlating Event Histories

An event history can be used for analysing performance of a particular execution of a parallel program. Since the performance of a program can be different under different conditions, analysing only one execution of the program may not be suitable for performance tuning purposes. In the following circumstances correlating different histories can be useful:

- Investigating data dependent behaviour of a parallel program,
- Experimenting with scalability of a parallel program implementation,
- Comparing different mapping strategies,
- Testing non-deterministic parts of a parallel program.

Since the user can control the way in which history correlation is carried out, it is possible to focus on particular parts of a parallel program, and filter out the unnecessary information that relates to other parts (or different detail levels) of the program. For example, the user may want to know about an interaction between two processes, and (s)he wants to investigate the effects of mapping decisions on the contention level of the channel between a particular process pair. Here, the user acts by defining event patterns that represent the interactions between these two processes, and by creating an abstract event for this particular interaction. Then the user can run the program using different mapping strategies, and collect the execution histories. By applying the same abstraction rules to these histories, the user can select the events representing the interactions between the pair of processes. These selected events can reveal the timing characteristics of the interactions which are investigated. For example, finding the longest/shortest communication instances, or visualising any repetitive pattern of these communication instances may help to the user to gain insight into how channels are loaded when different mapping strategies are used.

4.4 Summary

Programming distributed memory MIMD computers that contain a large number of processing elements and achieving highly parallel programs is not easy. The parallel program development cycle requires support for performance tuning. Since large parallel programs are difficult to monitor and analyse, the user needs comprehensive tools to understand the behaviour of parallel programs. Event monitoring is a technique for collecting data from a parallel program execution to understand the detailed behaviour of the program. The event abstraction mechanism proposed here provides an infrastructure to simplify execution histories. Since the user can select event patterns from the history, and construct application specific

performance displays, this mechanism delivers information that is directly related to the tuning task.

Definitions for event abstractions can be stored as library functions. Then, these definitions can be reused for similar classes of problems. The ability to create and modify libraries has some advantages. Firstly, the user can customise the tool for particular problems. Secondly, new techniques for analysing parallel programs can be incorporated into the tool. Finally, the tool can be adapted to the changing demands of the field.

There is one drawback of the event abstraction approach. The user has to learn how to use the visual query language to define queries. The syntax of the language is not difficult to learn, however, since it resembles the Gantt chart display which is widely used for performance visualisation purposes. The tool is also accessible to users who do not know the language. They can use standard libraries to analyse their programs, and get the basic performance visualisation.

In summary, the proposed event abstraction mechanism provides detailed, application specific performance tuning support.

Chapter 5

Performance Data Visualisation

5.1 Introduction

The importance of scientific visualisation has been accepted for a long time because the representation of large amounts of numerical information in a suitable abstract graphical form helps a user to understand the numerical information without him/her going through every detail of that information. Performance visualisation is no different from other forms of scientific visualisation. Given a large amount of event trace information, the goal is to abstract this information in graphical form from which the user can gain insight into the performance of a parallel program. Graphical forms may not contain detailed information but they can easily exhibit data tendencies otherwise difficult to observe in textual form.

In currently available tools, visualisation is achieved through performance displays which visualise a restricted number of aspects of performance data (eg. ParaGraph [23,24], PIE [39], Multiple Views [37], HyperView [42], IPS-2 [47,50], Moviola [17]). Since the semantics of the event patterns which are to be visualised are predefined, every performance display can present only a small number of generic event patterns, and not application specific ones. This is a restriction which is imposed on how the semantics of event patterns can be interpreted, making standard visualisation tools inflexible for customisation and/or many application specific queries.

The approach which is presented in this thesis differs from the existing visualisation tools in separating the domain specific knowledge from the performance displays, and giving the control of defining domain specific knowledge to the user (i.e. there is no predefined semantics attached to certain event patterns). The user tries to get a visual answer to his or her query about one aspect of the program's performance. The query facility is provided by the event abstraction mechanism. When an abstract event is created, the user assigns a meaning (i.e. the user defines the semantics of an event pattern in his or her mind) to the new abstract event which is a manifestation of what is happening inside the program. There is a fixed number of visual display formats which can present abstract events in different graphical forms. Depending on the meaning of the abstract event, the same type of visual display can visualise different aspects of a parallel program. Since we can easily identify manifestations of a parallel program activity as event patterns and visualise this specific information, our visualisation approach can support performance displays ranging from basic standard ones to highly application specific displays.

Prototyping of visual displays has been achieved by using a general purpose graph plotting utility (i.e. GnuPlot [20]). The capabilities of these kinds of utilities show that they can be used for implementing performance visualisation tools. There are several advantages to be gained from using general purpose graph plotting utilities:

- They simplify the implementation
- They usually have extensive graphic capabilities compared with performance visualisers
- Since they are extensively used by the parallel computing community, users are familiar with these utilities and it is very easy for them to handle perform-

ance data for other uses such as documenting performance characteristics of a program, preparing presentations, etc.

5.2 Visualising Abstract Events

The primitive events of a parallel program contain the most detailed information about the behaviour and the performance of the program. Visualising these events alone would reveal most of the bottle-necks of a parallel program. Early parallel program visualisation tools reflected this situation by implementing graphical displays which visualised individual details of the primitive events in the form of event names, icons, etc. This approach can be satisfactory for visualising small parallel programs which either have a small number of processes or do not run for a long time.

The relevant primitive events or the primitive event patterns which manifest a specific behaviour of the program have to be highlighted so that they can become visible among the other events when there are large number of processes to be visualised. The distribution, quantity and durations of these highlighted patterns can create a better view of program behaviour. The user might be interested in visualising the distribution, durations, summaries or temporal relations of these abstract event occurrences. This is achieved through performance displays which can visualise the same event abstraction in different graphical forms, each of which presents a different level of information from a different angle of view (Figure 5–1[1]).

---

[1]This system runs on the Sun workstation. By the time the experiments were carried out, the event traces had been transferred from ECS to the workstation using 'ftp' utility (file transfer program). It is now possible to use the ECS Sparc front-end, and there is no need to transfer large event traces between two different systems.

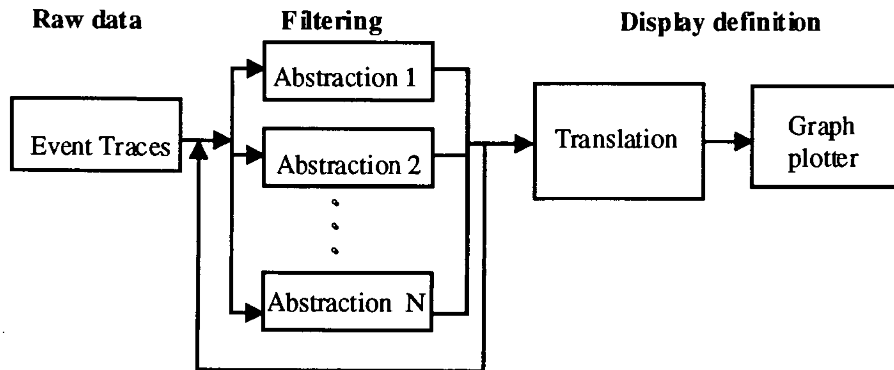


Figure 5–1: The conceptual structure of the performance visualisation environment

5.2.1 Performance Displays

Performance displays are the crucial part of performance visualisation. In this thesis, the basic graphical display types are chosen to visualise the performance. When these display types and the large number of abstract event definitions are combined together, very useful performance displays can be achieved. In addition to the standard performance displays which are provided as a library, the user can also define what to visualise, which makes this approach extensible without implementing new performance displays.

In table 5–1, a summary of the performance displays which are presented in this thesis is given.

The following graphical display types are widely used as performance displays by performance visualisation tools:

- Gantt charts
- 2-d graphs (line or scatter)
- Matrix displays

Table 5–1: Performance visualisation displays

Display Type	Display Dimensions	Cons & Pros
Gantt Chart	Space-Time	+easy to interpret +detailed information -restricted to small intervals -not suitable for visualising large number of processes
3-D display	Space-Time-Metric	+suitable for visualising large programs +many forms of display available -not suitable for detailed behaviour visualisation
Parallelism display	Time-Metric	+provides summary type of information which is not obvious in Gantt chart and 3-D displays
Profile display	Metric-Metric	+summarises resource usage +suitable for any size program -no direct use of locating performance problems

- Contour plots
- 3-d plots
- Bar charts
- Kiviat graphs

Each of these displays can visualise various forms of performance data from a different point of view. Performance visualisation tools usually accommodate several of these graphical forms and their derivatives because no single display type, on its own, is enough to visualise every aspect of parallel program performance data.

Gantt Chart Displays

The Gantt chart is one of the oldest graphical forms. It was used by H.L. Gantt [19] as a performance display in industrial management. This display type can visualise detailed performance information in two dimensions: space (processes

or processors in the computer performance visualisation case) and time. The activities which occur in a parallel program can be represented on the chart in several ways: rectangles, icons, lines or textual information, but there is one thing in common. The positions and the lengths of the representations on the chart are proportional to their real life timescales. For performance visualisation the activities to be displayed are the events which occur during an execution of a parallel program.

The distribution of events in a history can present a problem when using a Gantt chart. Here, the problem is that there are groups of events, and between the groups, there are considerably long time gaps. This can be explained using an analogy. Let us consider that we are preparing an industrial map of a country and we want to investigate the interactions between different sectors of industry, marketing and environmental issues in order to increase the efficiency of the industry and reduce the environmental damage.

The first thing to consider is to draw a map to see the distribution of the factories, the raw material sources, the cities, the ports, the roads and the geographic objects such as rivers, lakes, mountain ranges, etc. A map containing all these features would help to understand and work out the complex relations between these objects. The important part of drawing this map is to represent all these objects in such a way that the relations between them reveal the problems and give clues as to how to solve them.

The simplest approach to drawing the industrial/environmental map is to represent the factories with icons (eg. a different icon for each different factory type, and with the size of the icon indicating the relative size of each factory). This map would correspond to a Gantt chart used to display primitive events in a parallel program. The problem with this kind of presentation is that there will be so many factories around the industrial cities that the icons which represent the factories will overlap with each other in these areas making the map unreadable. For each

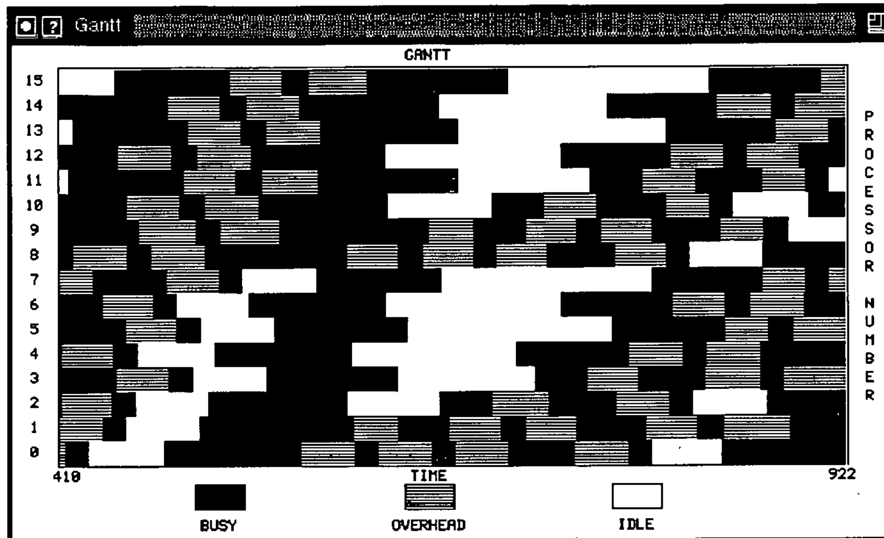


Figure 5–2: Gantt chart display of "ParaGraph" Visualisation Tool

industrial area we will need a special supplementary map which shows the area in a smaller scale. To analyse the map, the analyst should go through these small scale maps. The same problem exists in the performance visualisation: around the synchronisation points, the events will concentrate and reduce the readability of the performance display to nil.

An approach to increase the readability of a map without reducing the scale is to bring the information to the level of the current scale by finding a new abstract representation for it. For example, we can use bars (similar to a bar chart) to represent the capacity of each branch of the industry. This highly abstract representation will give aggregate information without reducing the readability of the map but enhancing the visualisation of information by narrowing it to a specific aspect. The same approach is also used in performance visualisation. Some specific primitive event patterns are represented in abstract events that can be visualised in several graphical forms to emphasise a different aspect such as frequency, duration, temporal properties.

Predefined event abstractions such as idle/busy/overhead periods (see Figure 5–2) can provide generic visualisation that can be applied to any parallel

program. It is also possible to use event patterns to create an abstraction which is meaningful to a specific application. For example, in the industry map analogy, we can modify the industry map which shows the capacity of each branch of the industry to show for example the volume of poisonous chemicals which are poured into the North Sea. In this map, each industrial branch in the surrounding regions is represented by a bar which shows the amount of the contribution. Since this map shows only the contributing industrial estates, the other industrial estates (in the same region) which do not contribute the pollution will be excluded. This selective representation of the map shows more information to someone who is interested in the pollution level of the North Sea, and the map is applicable to the countries around the North Sea. This latest version of the map corresponds to the application specific visualisation. This is achieved by recognising some event sequences from the event streams of the processes of the parallel program, and creating abstract events for these sequences.

Let us consider the GMAT [59] example in Figure 5–3. In this example, every event is represented by an icon whose vertical projection corresponds to its owner process and its horizontal projection to its time-stamp. GMAT's solution to the problem of fitting larger intervals into a Gantt chart is to compress the regions in which no event occurs, and to represent this compressed region with a bar type meter which represents the length of the compressed region. With the GMAT approach, larger intervals can be fitted into a Gantt chart given that there are gaps between event groups. This can be true for programs with small number of processes but for parallel programs with large numbers of processes the distribution of events is likely to be even.

An alternative solution to the problem of fitting larger program intervals has been adopted by several performance visualisation tools which are intended for visualising parallel programs with large numbers of processes. This solution is to replace icons which represent transient points in the history with pattern-filled or coloured rectangles which represent intervals of program activities. A small

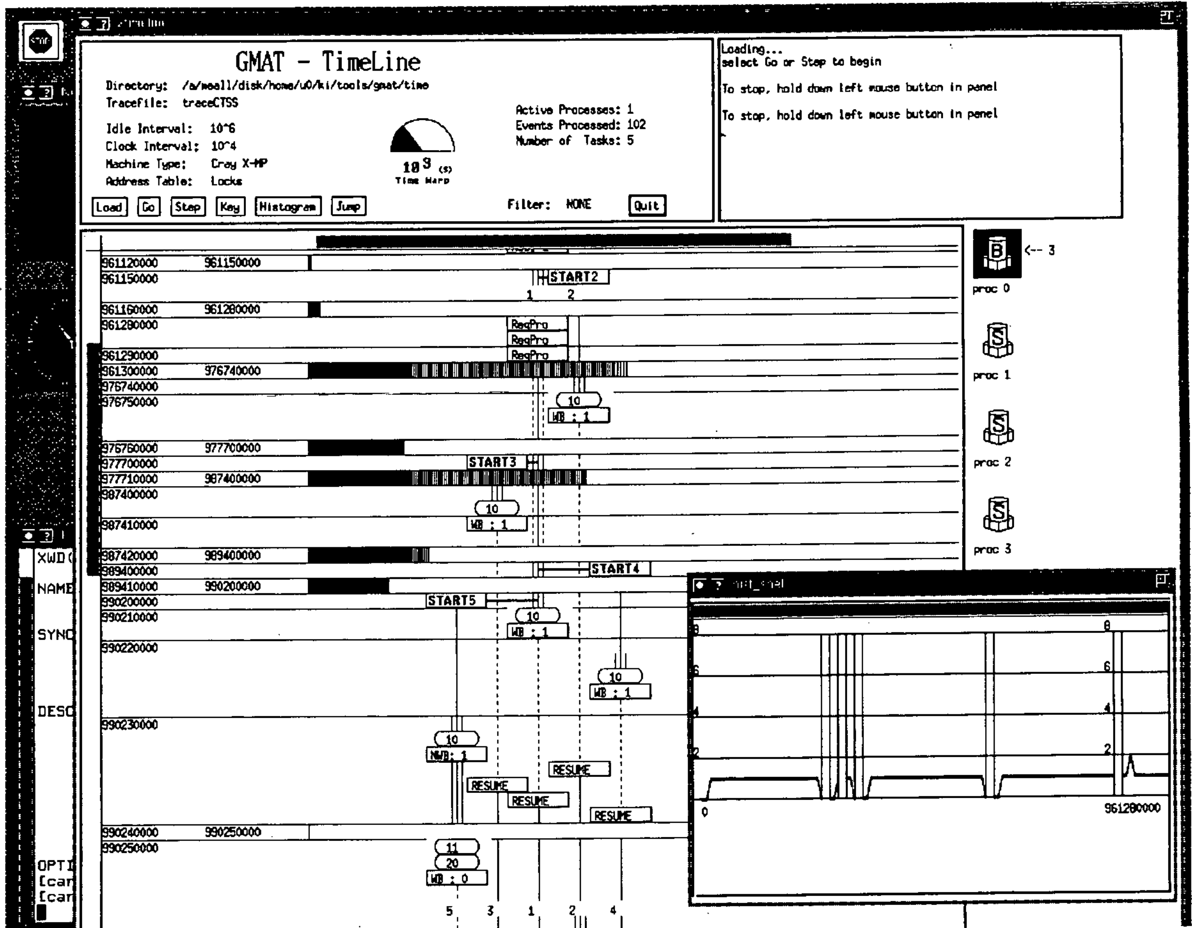


Figure 5–3: Gantt chart display of "GMAT" Visualisation Tool

number of program activity types is used for constructing what happened during the execution of the program. In Figure 5–2 a Gantt chart display is given from the ParaGraph [23,24] parallel program visualisation tool. In this display, there are three different types of interval types: "busy", "overhead" and "idle". It is easy to spot if there is a long busy/overhead/idle interval but it might be difficult to trace instances of these three interval types unless the region which is investigated is close to the beginning of the history.

Our solution to visualising large intervals on a Gantt chart is similar to the second approach, in terms of classifying intervals, but unlike the second approach there are user defined interval types which correspond to the user defined events. Since the user can combine as many events as he or she wants, the interval size that can be displayed on a Gantt chart can vary from one abstraction to another.

In Figure 5–4, an example abstraction of a parallel program[2] is visualised on a Gantt chart. In this example, the durations of the abstract events are usually much longer than the gaps between consecutive events (e.g. the gap between "e2" and "e3" in Figure 5–4). This property of the example makes it difficult to visualise both abstract events and the gaps between them. When the user zooms out, it is difficult to see intervals between two events. When the user zooms in to see it, the abstract events will not fit into the display. The user is forced to zoom in and out continuously to inspect the details of the display. The Gantt chart is not very suitable for visualising micro and macro events at the same time. The next display type (i.e. 3-D displays) solves this problem by defining a separate dimension for the durations of events. This is further explained in the following section.

---

[2]This is the skeletal program which is explained in Section 5.3.1. The program ran on 8 Transputers, and ECS Monitor V1.0 was used to monitor the execution.

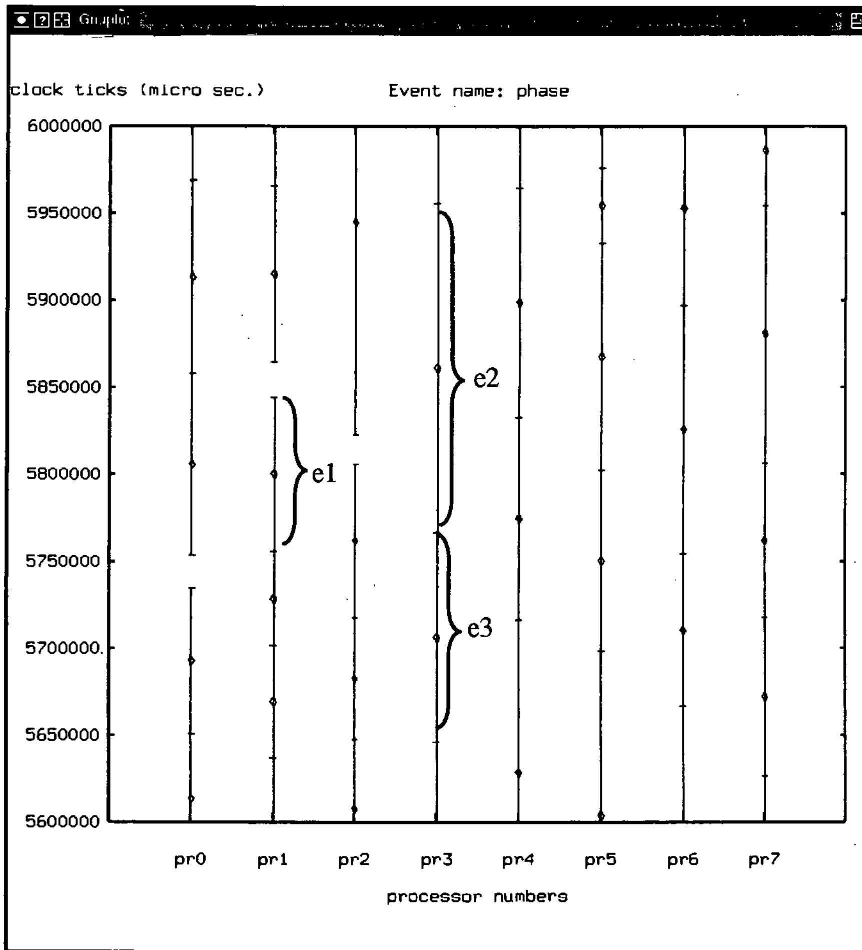


Figure 5–4: A prototype Gantt chart display using GnuPlot

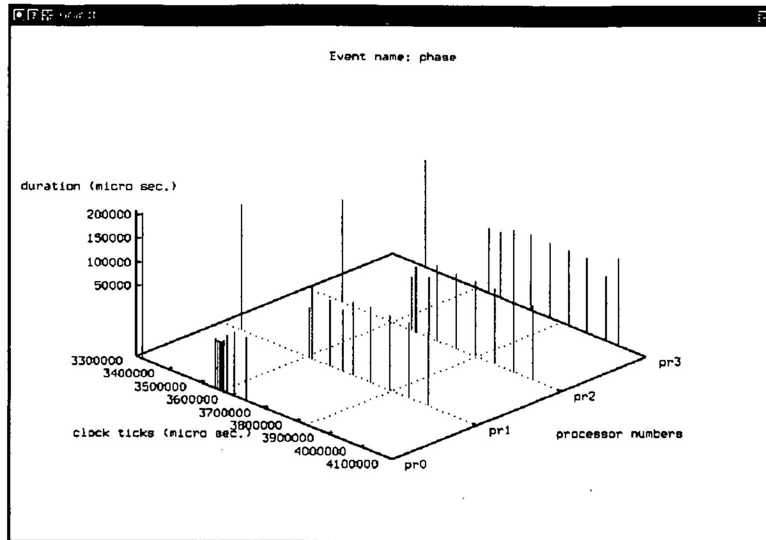


Figure 5–5: Visualisation of the stages of a program

3-D Displays

The 3-D displays discussed in this thesis are an enhanced form of the Gantt chart. By adding another dimension to the Gantt chart, 3-D displays can be achieved. The third dimension represents the duration of the abstract events which are visualised. This dimension is called the metric dimension because it provides a common basis to compare the durations of abstract events. This display type can be seen as a Gantt chart in which events are visualised as perpendicular columns.

In Figure 5–5, a small data set is visualised as a 3-D graph. Since one end of each event is aligned with the X-Y plane, the other end serves as a reference point to compare its duration with other events durations. A peak in this kind of graph indicates an exceptionally long activity which may contribute to a slowing down of the program under test.

Because the 3-D displays have separate dimensions for the durations (z-axis) and the time-stamps (x-axis) of the events, the scales of each dimension can be adjusted independent from each other, the user can visualise micro and macro events at the same angle of view.

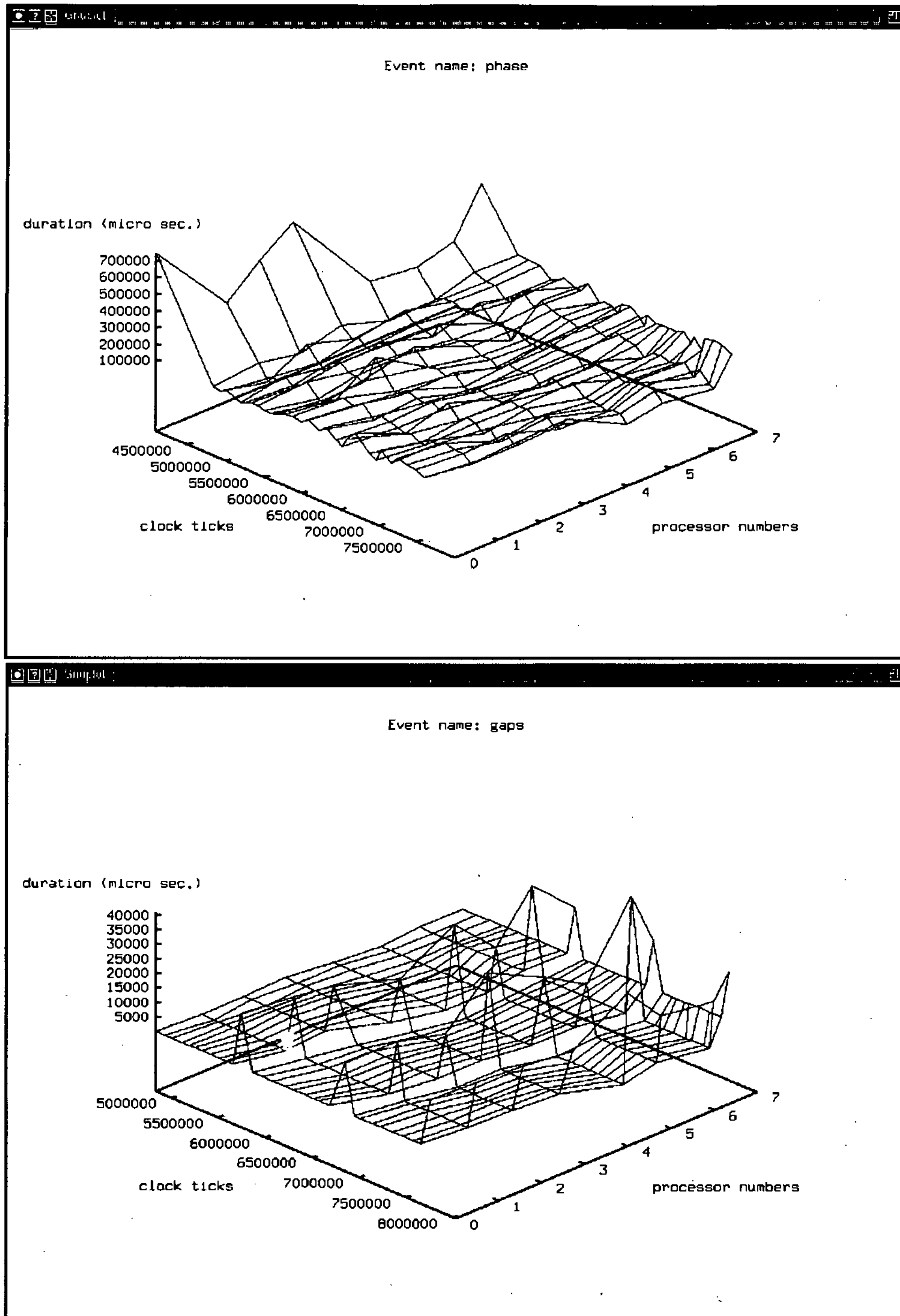


Figure 5–6: Visualisation of macro and micro events

In Figure 5–6, an example of the visualisation of micro and macro events is given. These two displays have been generated using the same event trace file and the same abstraction level which were used for generating the Gantt chart in Figure 5–4. The top window in Figure 5–6 is used for visualising the abstract events. These abstract events (application specific abstraction) represent computational phases of the program. Another abstraction is used for representing the gaps between these computational phases. This is visualised in the bottom window of Figure 5–6. Note that both displays are at the same angle of view (i.e. both cover 3,000,000 units = 3 secs. of the program history), but the scales of z-axis are different; the z-axis of the graph at the bottom is 20 times larger than the z-axis of the graph at the top.

In the bottom display of Figure 5–6, the gaps between computational phases are visualised. The periodical peaks in the graph correspond to the input/output related activities. It is possible to read the durations of individual I/O activities without changing the scale of the graph (eg., the first I/O activity of processor 0 takes approximately 0.015 secs.). On a Gantt chart, the same readings would not be taken without zooming in to the places where the I/O activities occur.

So far in the examples, two different styles of 3-D displays are used: impulses (Figure 5–5) and surface (Figure 5–6). There are several different styles of 3-D display type. Depending on the nature of the data that is visualised, the user can switch between these different styles of the same display type. Each can be useful in different situations.

The following styles of 3-D graphs are available and can be used for visualisation:

- Surface
- Line
- Scattered (dots in GnuPlot)

- Bars (impulses in GnuPlot)
- Surface contours
- Base contours

A 3-D surface graph can be useful when large number of processors are to be visualised (eg. Figure 5–13). Since small variations between neighbouring data points can make the surface jagged and general trends in the data therefore difficult to see , it is possible to improve this type of display by clipping the hidden surfaces from the graph. It is also possible to apply surface smoothing techniques to the surface graphs. This will hide small variations of data and emphasise the important peaks and dips on the graph. By doing this, some details are lost but it enables visualisation of large data sets, especially from massively parallel systems.

The line and bar styles can be useful when the user wants to emphasise the variation between data points which originate from the same processor or the same process. Even though these types of variations in data, which usually reflect the behaviour of individual processes, are the main concern, the line and bar styles are still useful for presenting data across different processes.

The 3-D scattered display type is suitable for visualising the distribution of abstract events. This type of display can be used for monitoring activities throughout very large event histories, since the dots which are used for representing data points do not crowd the display. In this type of display, the duration of events cannot be seen very easily since the points are floating in the 3-D space of the graph. The whole point of using the scattered format is to see groupings of data points. This grouping can be a single line of a process, and can manifest a behaviour related to a single process. These groupings can also be in an area or in a volume.

3-D Contour graphs are similar to surface graphs. They show less detail of the data points but the overall picture of the graph can be more visible since small variations between neighbouring points are filtered. By changing the number

of contours which are drawn, the user can control the level of detail that is to be visualised. This technique can be considered as a visual filtering technique since it serves as a device to reduce the amount of unimportant and/or unwanted information.

It is also possible to visualise more than one abstraction on the same graph (i.e. superimposing two graphs of different abstraction levels). Such graphs may become overcrowded, however, so that the display cannot be interpreted easily. In these cases, visual filtering techniques such as using contours only, or surface smoothing are useful. It is also possible to use different styles of the 3-D graph in a single view.

In a manner similar to Gantt chart displays, 3-D displays can also suffer from overlapping, or nearly overlapping data. Data points which are too close to each other may appear as a single data point. Since it is not always easy to zoom in and check if there are any overlapping data points, an alternate 3-D display format is used to unfold all overlapping data points. The dimensions of the 3-D display are translated from "space, time, metric" to "space, sequence number, metric". Every event in an event stream which belong to a particular process is given a sequence number starting from the beginning of the stream. Then, these events are visualised using sequence numbers instead of their time-stamps. Since the distance between any two consecutive events is the same, there is no risk of displaying more than one data point as one. In this display, the order between events which originate from different processes is not preserved but the order of events of the same process is untouched.

Another potential use for 3-D displays whose y-axis is made up from sequence numbers is history comparison. Assuming that the same event abstraction can be achieved for two different event histories, then visual comparison of two histories can be achieved.

Parallelism Displays

A parallelism display is a two dimensional graph type which shows the distribution of parallel activities over time. Unlike the parallelism displays of other performance visualisation tools, the parallelism display presented here is not restricted to the parallelism of raw computation. Traditionally parallelism displays show the degree of parallel intervals where processors compute.

The standard parallelism display provides a good metric which explains how the computational resources are used. It does not necessarily mean that keeping processors busy is an indication of a fast parallel program. However, we can conclude from low parallelism that the program is not performing well and therefore, it cannot be fast enough to gain an advantage over its sequential implementation. A parallel program may contain both duplicate computations and housekeeping activities such as coding, decoding and interpreting messages, which do not exist in the equivalent sequential implementation. This extra code can be identified as unproductive computation, and the rest as useful.

Apart from useful/unproductive computation argument, conventional parallelism displays present only very low levels of information for performance tuning. They just show how much computational resource is used without providing any information about how these resources are used.

The parallelism displays presented here are suitable for visualising any kind of resource usage. The definition of the "resource" is not restricted to system resources such as processors and channels. Software resources which use system resources are also considered as abstract resources to be exploited efficiently in order to achieve fast parallel programs.

Software resources vary from a process to a small code segment within a process code. They can be represented as computational units such as program phases, a step of a loop, turnaround time of a server process, broadcasting time, etc. The list is extensive since every application has its own specific software resources. Visual-

ising abstract software resources may reveal valuable information for performance tuning.

In Figure 5–7 four different parallelism displays of a test program (see section 5.3.1 for detail of the example program) are shown. All four of the displays visualise the same interval of the same program but each display visualises a different aspect of the program. The top display is the standard parallelism display which shows how computational resources are used. Since this program is a template, it does not use many computational resources but implements global data gathering operations using communication related resources.

The second display in Figure 5–7 shows how many processes block on sending messages. Similarly, the third display shows how many processes block on receiving messages. Even though these two displays do not directly show how much communication resource is used, they show the effects of communication on the execution of processes. These views are more important than operating system related views such as message routing, buffering and de/packetising information because the user usually cannot do anything directly to alter system related parameters in order to optimise an application program.

The last display in Figure 5–7 visualises how many "broadcast and gather" phases are carried out in parallel at a given time. All these displays are created using simple abstractions of primitive events. The first display shows the intervals which are marked by event pairs. These pairs are: ["CREATE", "SEND"], ["CREATE", "RECEIVE"], ["UNBLOCK", "SEND"],["UNBLOCK", "RECEIVE"] and ["UNBLOCK", "TERMINATE"].

The event pair for the second display is ["SEND","UNBLOCK"], and the event pair for the third display is ["RECEIVE","UNBLOCK"]. These simple abstractions are generic abstractions which can be applied to any program without any knowledge of the application, and meaningful performance displays can be achieved. The fourth display differs from the previous three in this respect. The

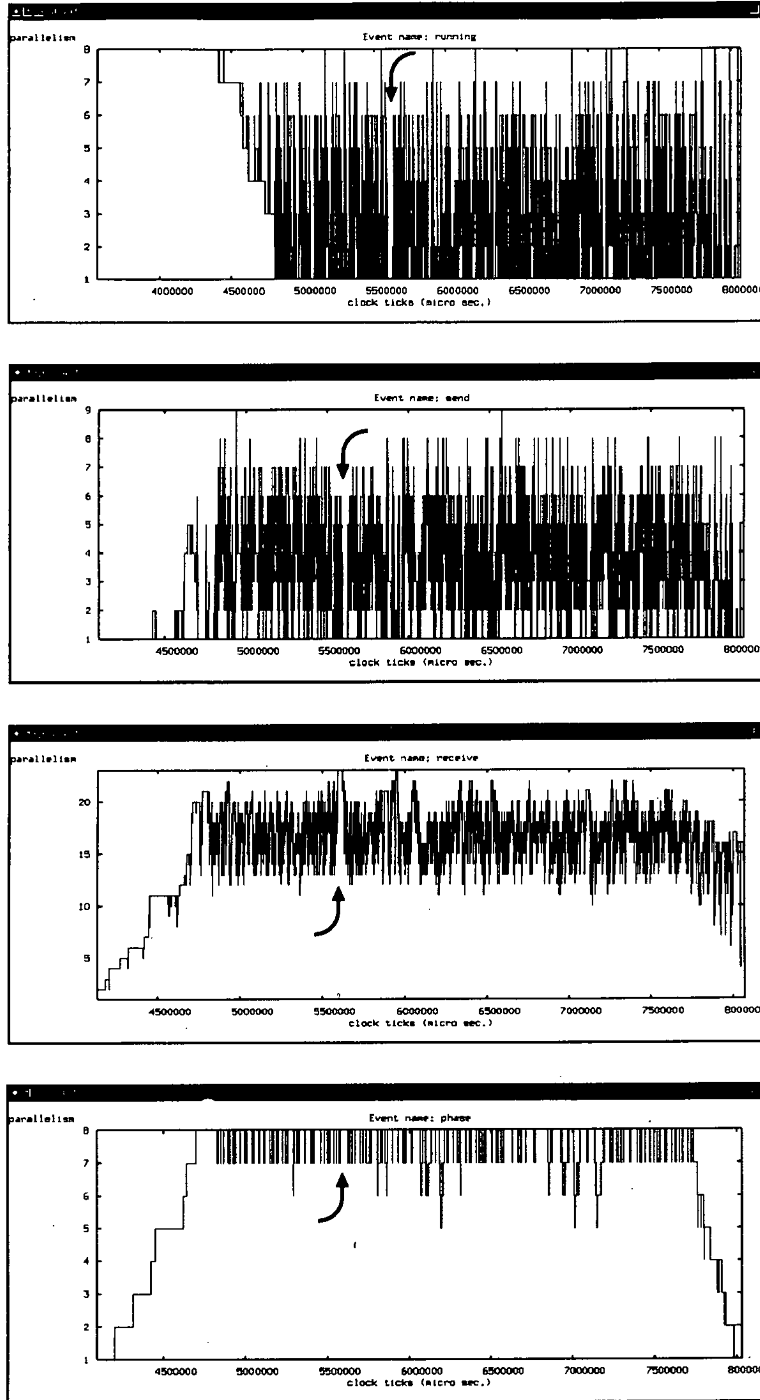


Figure 5–7: Parallelism views of a program

abstraction for this last display only produces meaningful results for the program described in Section 5.3.1. The abstraction for this display is explained in Section 4.2.4.

The information presented in the displays in Figure 5–7 can be more meaningful when the interpretation of individual displays are related to each other. For example, a specific region of the event history is identified by an arrow on each display. We can interpret this region as poorly parallel (top display) where most of processes wait for incoming messages (second and third displays), and the parallelism level of the monitored behaviour (bottom display) has not been affected by these circumstances. It can be concluded that the processes which are waiting for messages are participating in the task which is visualised in the bottom display.

Even though parallelism displays visualise summarised information, using multiple parallelism displays, each of which represents a different abstraction level, it is possible to gain detailed understanding about a behavioural pattern without zooming in to the region and without visualising the low level information.

Utilisation Displays

The utilisation display is a summarised version of the parallelism display. This display informs the user about the percentage of the execution time which is spent for each parallelism level.

This display gives information about how a parallel program performs, and does not give any information about where and/or when the performance is poor or good. Since this display summarises information which can be also visualised by the other performance displays, it looses details such as the ability to visualise critical path, parallel slackness etc.

In Figure 5–8, two utilisation display examples are given. Both of the displays visualise the "broadcast and gather" example (see Section 5.3.1). The first display

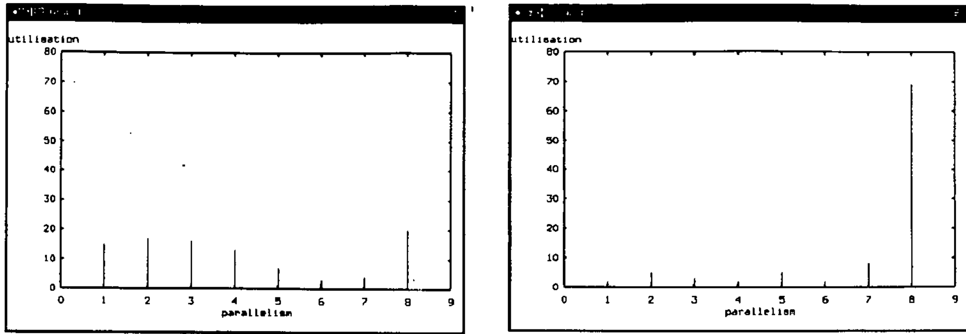


Figure 5–8: Profile view of "running" and "phase" events

shows how computational resources (processors) are used in the program. For an 8 node topology, the program used computational resources very poorly. It spends its time equally (nearly) using 1,2,3,4 and 8 processors. By looking at the parallelism display (see the first display Figure 5–7), we can also say that 8 processors are mainly used during the initialisation stage of the program.

In Figure 5–8, the second display visualises the "Broadcast and Gather" phases of the same example program. In each phase, a request message is broadcast to each node, and replies are gathered. The resources which are used by each phase are a network of processes which implement the phases. These processes consume a complex combination of computational and communication resources. When two or more requests are made at the same time, the request will use the same process network and therefore there will be competition between the requests. Here, the utilisation display shows the distribution of the multiple requests. For example, from the display we can tell that for 70% of the execution time eight requests are processed in parallel. This is an example of how utilisation displays can be used for presenting a summarised form of a highly abstract information about a particular behaviour of a parallel program.

5.2.2 Virtual and Real Parallelism

Since it is possible to map more than one process on to a processor, two or more processes may be running on the same processor. The scheduler timeslices these processes without functionally affecting them. This may only affect the execution times of the processes, and eventually, the total execution time of the application. An indication is required for identifying the decrease in performance of a parallel application due to excessive processor sharing.

The execution time of a code segment of a process may vary depending on the number of processes sharing a processor. Since execution time is the prime concern for the user, (s)he wants to know how many processes are competing (i.e. the number of processes which are in the "ready' and the "running"' states) for a processor at a given time. The ready-to-run processes would be running at the same time if they were mapped onto separate processors. This kind of parallelism is called virtual parallelism. Virtual parallelism is equal to or higher than real parallelism. For a parallel program execution, if the difference between the level of virtual parallelism and the level of real parallelism is significant, this can be an indication of a performance bottle-neck as a result of a wrong mapping strategy.

Two different versions of parallelism display are provided: Virtual and real parallelism displays. In the real parallelism display, the abstract events (from the same processor) which overlap are handled as a single event which is union of the overlapping events. In the virtual parallelism view, every event is handled separately. As a result of this, for example, two intersecting events from the same processes contribute 1 unit of parallelism for the non-overlapping and 2 units for the overlapping interval. An example is presented in Figure 5–9. The abstract events which are used in this example are shown as a Gantt chart. At the bottom of Figure 5–9, real and virtual parallelism displays of this example are depicted.

The reason for mapping more than one process on to a processor is to take advantage of parallel slackness [66]. When a process blocks for a communication

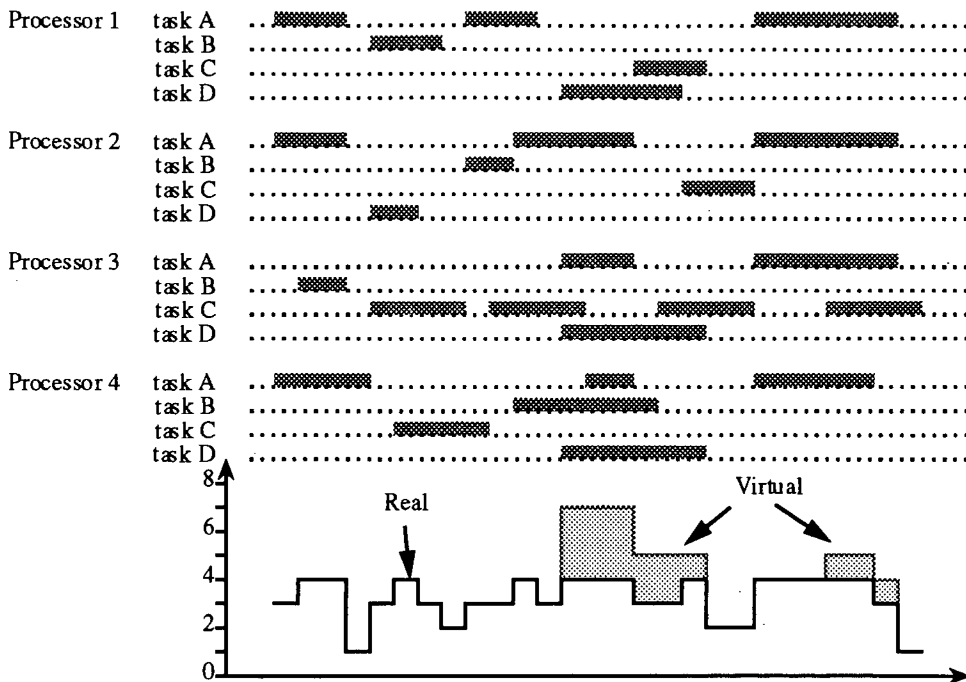


Figure 5–9: Virtual and real parallelism views of an example

related operation, the processor can be assigned to another process. In the optimum case, only one of the processes is ready to run, and the other processes are waiting for a communication. When the running process blocks for a communication, another process completes its communication and starts running. In this extreme case, no processor is idle nor are any two processes ready to run at the same time. Since the executions of processes never overlap in this extreme case, the real and the virtual parallelism views of the program will be identical.

The degree of parallel slackness can be determined from the differences between the real and virtual parallelism displays. The closer the similarity between the virtual parallelism view and the real parallelism view, the better the exploitation of parallel slackness.

The comparison between the real and the virtual parallelism views can be made if the abstraction of the event history represents the intervals during which

a process is running or ready to run, but there is a potential for achieving the same results for more abstract levels of a history.

Let us consider an application which uses task farming. A farm process is a stand-alone server process which is dedicated to calculate a specific function. As well as having several different farm processes in a task farming program, there can be also several copies of the same farm process.

Since the aim is to achieve the best possible turn-around time from these farm processes and the number of processors is restricted, each farm processor can contain several different types of farm process. Assuming that each type of request is made at a different time, mapping several different farm processes onto a single processor does not affect the turn-around times of these processes. However this assumption is not always true. There will be multiple requests to these farm processes. In this case, two or more requests are handled in parallel on the same processor. This will affect the turn-around time of the farm processes since they time-share the same processor.

Using virtual and real parallelism views, multiple requests to the farm processes can be visualised easily. In order to visualise this behaviour, a simple abstraction is required to identify intervals in which a farm process handles a request. This is the interval which starts with the arrival of the request message, and ends when the result is sent back to the process which made the request. These intervals are represented by the instances of an abstract event. The differences between the real and virtual parallelism views of this abstraction will visualise contention (if there is any) of an abstract resource which is the farm processes.

5.3 Performance Visualisation Exemplars

In this section, visualisation examples from two different programs are presented. The interpretations of both general purpose and application specific performance displays are made in the context of these example programs.

The first example is a template program which implements global operations on a distributed data set. This is a good example of programs which perform global operations on distributed data sets.

The second example is an optimisation program which finds an optimum multiplication order of the matrices of a given expression. This program is a good example of barrier synchronisation which is a frequently used technique to synchronise processes between stages of a parallel program.

5.3.1 A Skeletal Program: Broadcast and Gather Operations

The example program presented in this section is a skeletal program [14] which consists of a network of processes that implement the "broadcast and gathering" operations. The broadcasting node sends the request messages sequentially to all its nearest neighbours which will forward the message to their neighbours, and so on. The topology of the network is a binary n-cube. By exploiting properties of binary n-cubes, it is ensured that every node receives the same message only once. Once the request message reaches a leaf node, the leaf node handles the request, and returns an answer to the node that forwarded the request. A forwarding node waits for answers from the nodes to which it forwarded the original requests. When the forwarding node gets replies from its all recipients, it combines its answer and all other answers into one message, and returns this combined answer to the

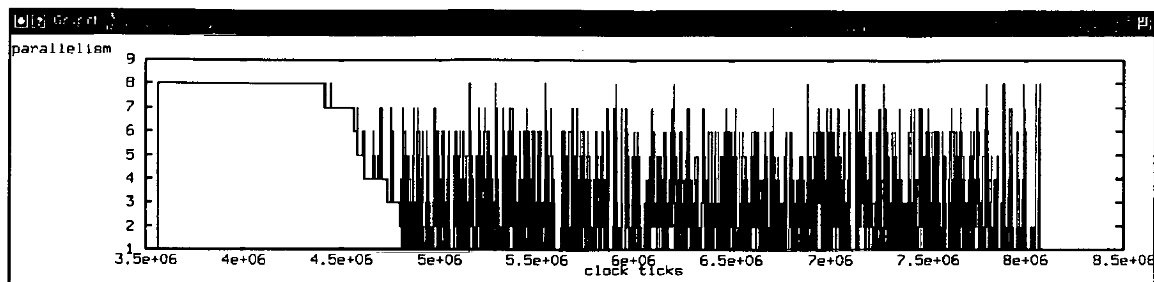


Figure 5–10: A parallelism view of 8 node "Broadcast and Gather" program

node that forwarded the original request. This is called a "broadcast and gather" operation.

In the implementation, three different processes are used. The first one is the "broadcaster" process which acts like a client. The second process is the "receiver" which handles the incoming request/reply messages to a node (processor), and redirects them to the "distributor" process which is responsible for distributing request/reply messages to the nearest neighbours or to the local client ("broadcaster") if the local client is the requester of this particular operation. Each "broadcaster" can make only one request at a time but "broadcasters" from different nodes can make requests in parallel. The infrastructure for handling requests (i.e. the network of "receiver" and "distributor" processes) can handle system-wide multiple requests without blocking any of the nodes.

The mapping of processes on to the processors is fixed; each processor has one "receiver" and "distributor" pair. The "broadcaster" process is also mapped to the processors from where global operations are going to be required.

In the experiments, 8, 16, 32 and 64 node implementations are monitored and visualised.

Figure 5–10 shows a parallelism display of computational resources. As can be seen from this display, the level of parallelism varies dramatically throughout the program. It is not possible to say how the program behaves but this display gives a clear indication of poor computational resource usage.

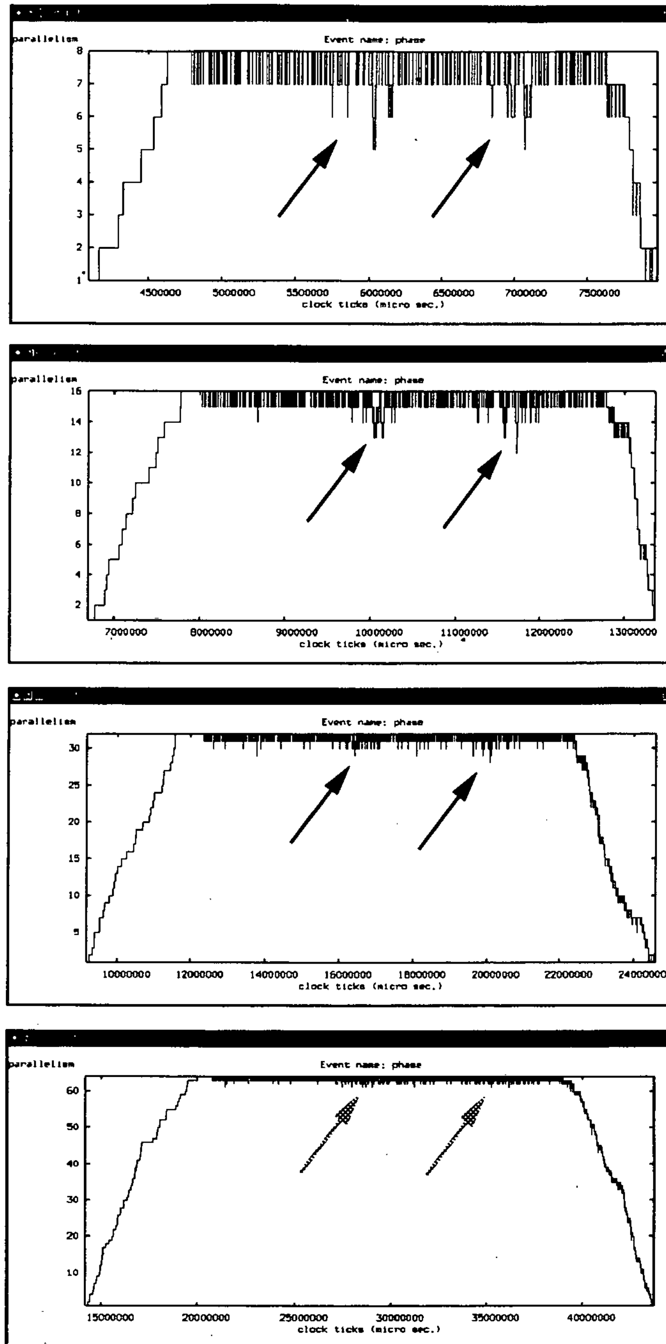


Figure 5–11: Parallelism views of "phase"s of 8, 16, 32 and 64 node "Broadcast and Gather" program

In Figure 5–11, four different parallelism views of the example program are given. Each of the displays visualises how the program behaves in terms of the "broadcast and gather" operations. The main target in visualising this program is to see how these operations are performed, how long they take, and what the overall performance is.

The abstractions which are used for visualising the "broadcast and gather" operations are explained in section 4.2.4. The abstraction represents an interval in which a "broadcast and gather" operation takes place. The length of the abstract event is relative to the duration of the operation to which it corresponds.

In the example program, each node carries out 30 operations, and then, it terminates. Each node also prints the results after every 10 operations. The effects of these print operations can be easily recognised in the first two displays in Figure 5–11. After one third and two thirds of the total execution time (see arrows in Figure 5–11), the level of parallelism decreases temporarily while the print operations are carried out. The same effects are less visible on the last two displays in Figure 5–11 since the number of the processes is high, and the individual processes tend to drift. As a result, the processes reach the printing points at different times, and the fall in the parallelism level is spread over a large interval.

Figure 5–12 and Figure 5–13 are the visualisation of the abstraction level whose parallelism displays are shown in Figure 5–11. In these 3-D displays, we can see how long the individual operations last, and easily compare them with each other. As can be seen from the displays, the first operation of each node takes much longer than the rest of the operations of the same process. This is due to the fact that not all processes start at the same time when they are downloaded to the processors. When all processes are up and running, the system settles down, and the "broadcast and gather" operations are performed at their normal speeds.

In Table 5–2, some readings from the performance displays are given. Four

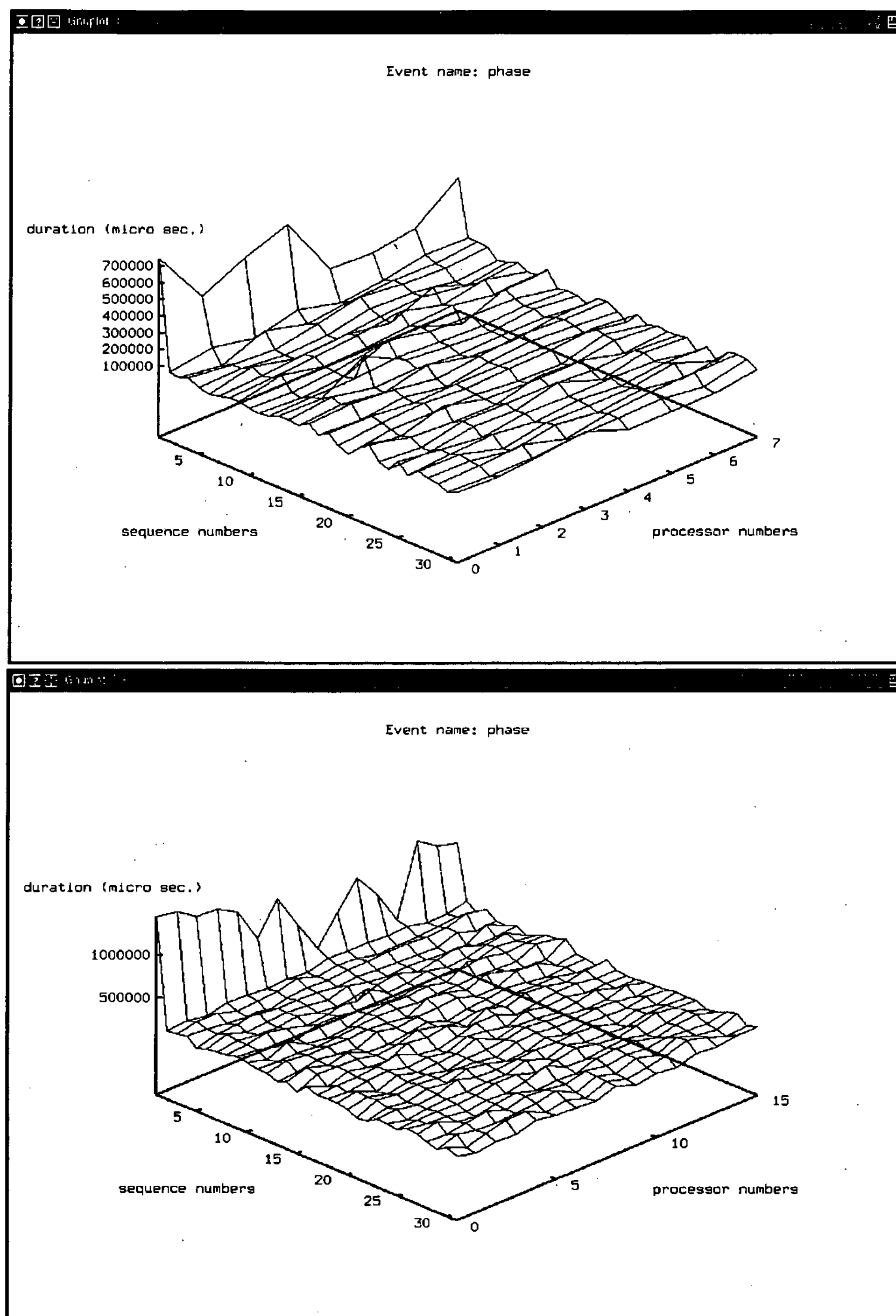


Figure 5–12: Visualisation of the stages (8 and 16 nodes)

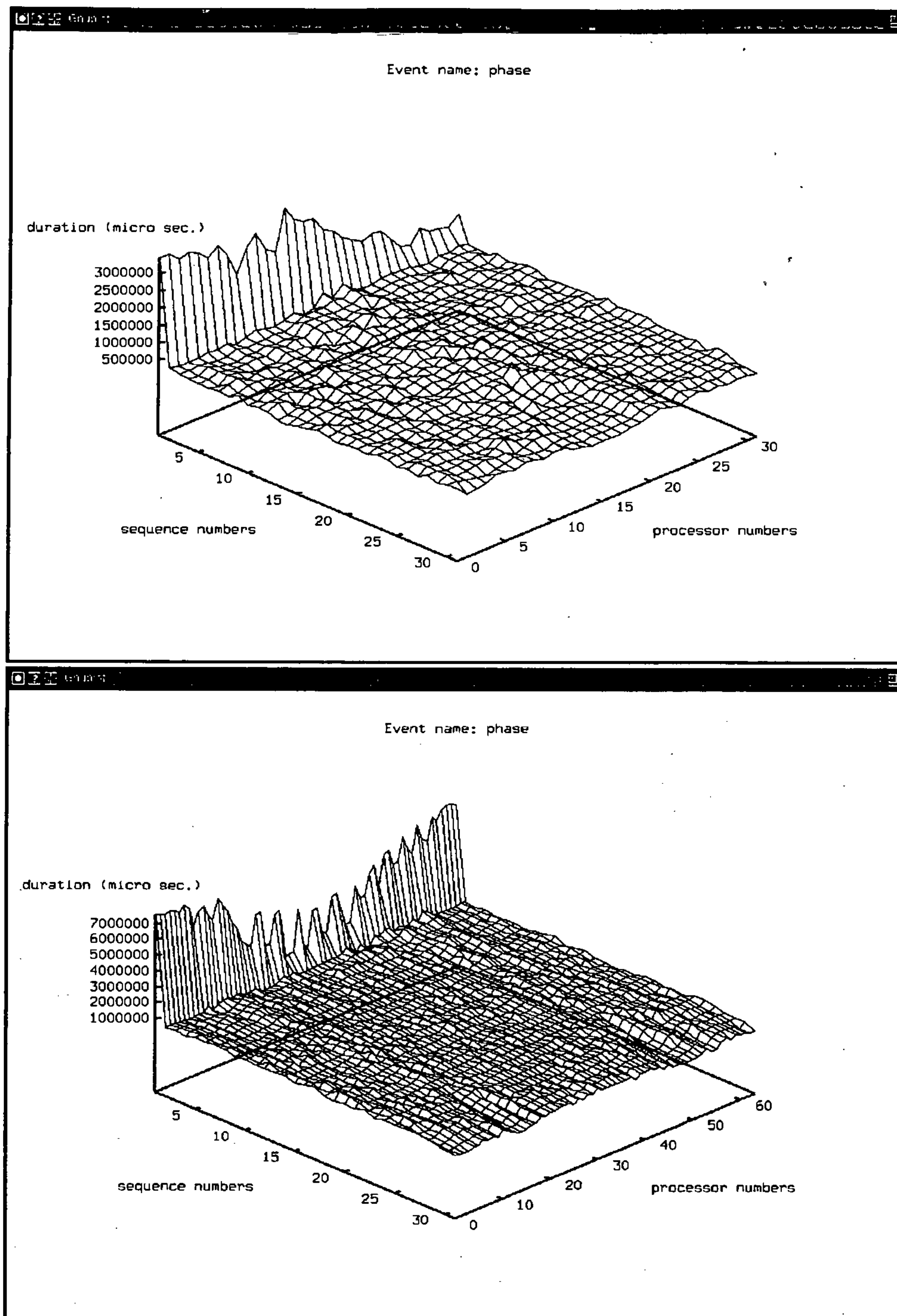


Figure 5–13: Visualisation of the stages (32 and 64 nodes)

Table 5–2: Performance measurement results

Number of nodes n	Duration of first broadcast T_f (secs)	Duration of all broadcasts T_t (secs)	$T_T - T_F$ $T_r = T_t - T_f$	Total number of node accesses $N_a = n^2 * 29$	MFlops/sec $N_a * 1000/T_r$
8	0.4	4	3.6	1856	0.52
16	0.8	7	6.2	7424	1.20
32	1.6	14	12.4	29696	2.39
64	3.5	29	25.5	118784	4.66

different executions of the program are summarised in each row of the table. The second column of the table represents the average duration of the first "broadcast and gather" operations. The numbers are approximate readings from the 3-D displays (Figure 5–12 and Figure 5–13). The readings in the third column are from Figure 5–11.

In order to calculate how much calculation can be carried out by using only these global operations, we need some information from the source code of the program. Each node requests 30 operations. Each operation causes accesses to every node to collect the results. In response to the request, every node carries out a computation which is equivalent to 1K floating point arithmetic operations.

The last column of Table 5–2 shows the speed of the skeleton program in terms of arithmetic operations which are carried out as a part of the "broadcast and gather" operations. In the calculations, the first broadcast operation from each node has been excluded because they were exceptionally long (see Figure 5–12 and Figure 5–13).

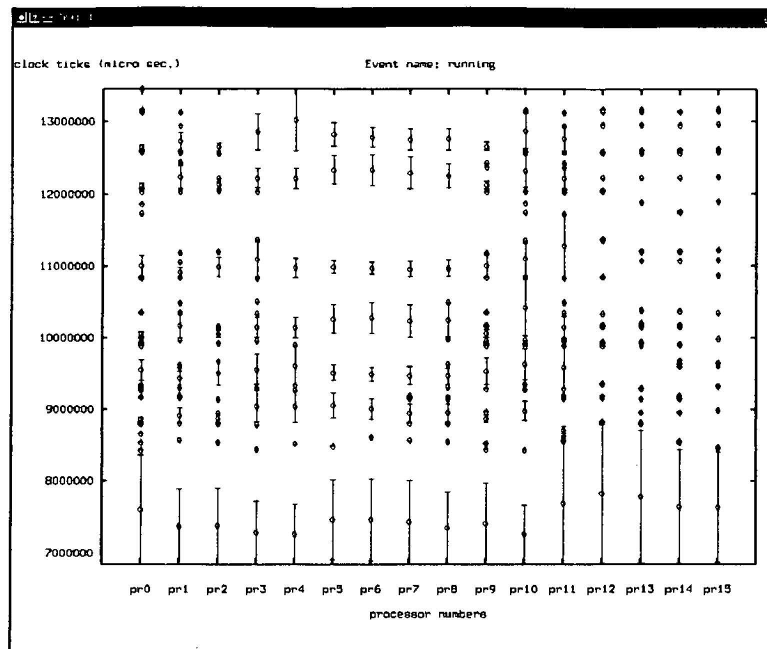


Figure 5–14: Gantt chart view of the computational intervals

5.3.2 Matrix Multiplication Optimiser

In this visualisation example, the programmer[3] and the person who visualises the performance (i.e. the writer of this thesis) are different. Since it is difficult to use application specific visualisation on an unfamiliar program, all displays which are used for visualising this program are general purpose (or generic), and their definitions (i.e. abstractions) are kept in a library for public use.

In the program, there are three different process types: one "master" process, ten "lines" processes, and five "worker" processes which are mapped on to sixteen processors. The "master" process implements the barrier synchronisation between the stages of the program. The "lines" processes act like storage processes which hold data, and the "worker" processes receive data from these storage processes.

---

[3]This exemplar program was implemented by Anna Hondroudakis, Dept. of Computer Sci., Univ. of Edinburgh, 1991.

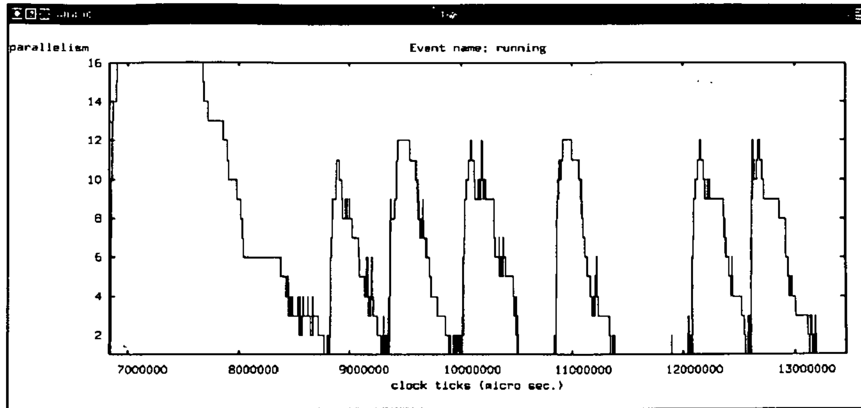


Figure 5–15: Parallelism view of the computational intervals

In Figure 5–14, a Gantt chart display of the program[4] is presented. The abstraction which is used in this display represents the intervals where processors carry out computation. The "master" process is mapped on to processor 0, the "lines" processes are mapped on to processors 1-10, and the "worker" processes are mapped on to processors 11-15.

The first noticeable anomaly in the display is the idle intervals where only a small number of processors are computing. This can be due to the barrier synchronisation which blocks the processes between the stages of the program. These intervals can be seen on a parallelism display which visualises the same interval and the same abstraction level of the program (Figure 5–15). We can easily see the intervals where parallelism falls dramatically. In fact, there are seven stages of the program, and the low parallelism intervals of the parallelism display correspond to the barrier synchronisations between the stages.

The second anomaly which can be seen in Figure 5–14 is that the first worker (processor number 11) is doing the most computation whereas the other workers (processor numbers 12-15) are doing very little computation.

---

[4]The program ran on 16-Transputer ECS domain, and ECS Monitor V1.0 was used to monitor the execution.

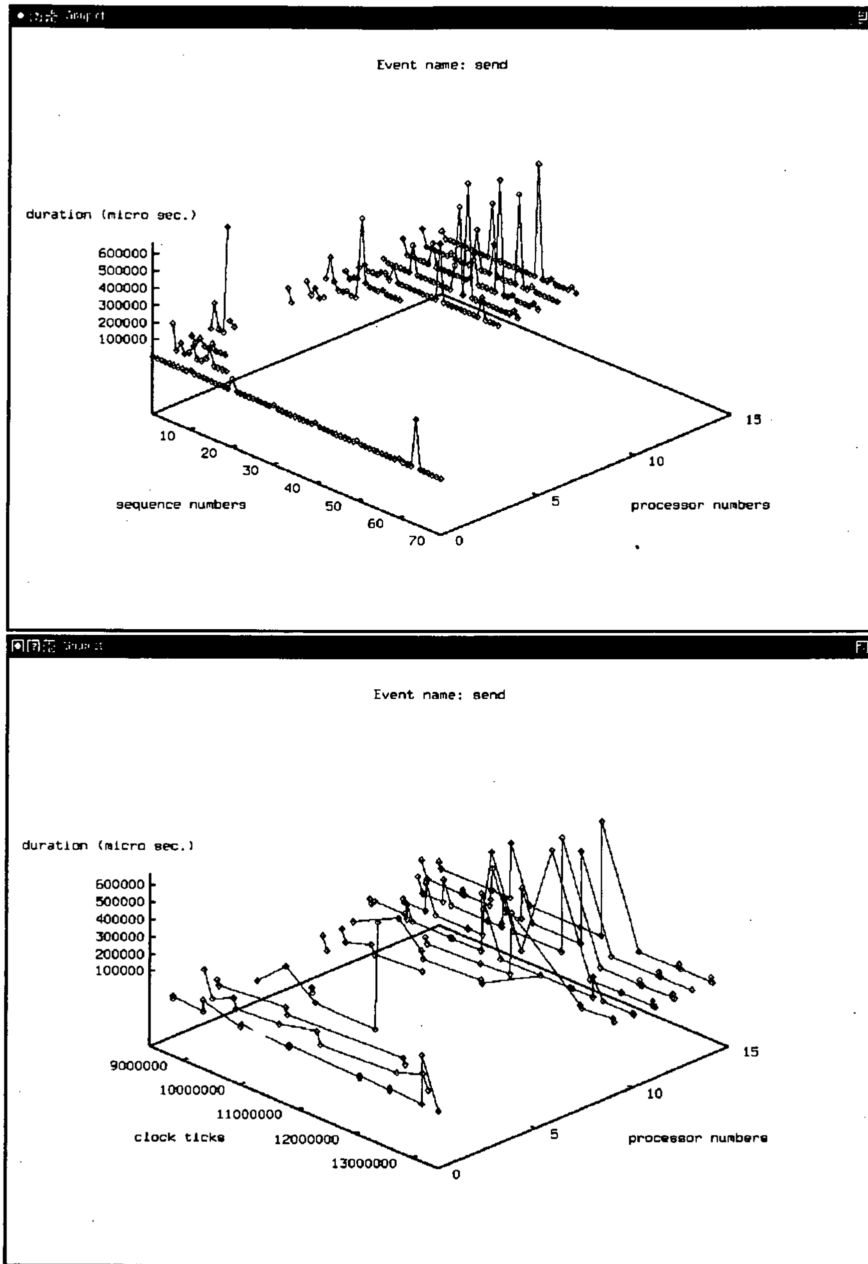


Figure 5–16: 3-D views of blocking intervals due to message sends

In Figure 5–16, message transmissions from the processes are visualised. Some of the message sends from the "workers" are exceptionally long. These exceptionally long message sends are for the barrier synchronisation. In the first display, sequence numbers are used for unfolding the overlapping data points. The second display uses the real time-stamps to visualise the message sends. These two displays can also be customised to visualise communications between specific processes, processors, group of processes, etc. For example, we can ask for "communications with the master process", "communications between the master and the worker processes", or "communications between the workers and the lines processes".

Similar to the message sends, it is also possible to visualise the receiver ends of the message transmissions. These new displays (Figure 5–17) will present different information from those which visualise the sending ends. As can be seen from these displays, worker processes are usually spending their time waiting for the messages to arrive. This shows that there is not enough work to keep these workers busy.

In Figure 5–18, the critical path information is visualised. This is the same event trace whose computational intervals are visualised in Figure 5–14. These two displays can be correlated to find out what is slowing the whole computation down. In Figure 5–14, the process (the first worker process) which is running on processor 11 looks as if it is doing heavy computation compared with the other processes (especially other workers). We can also confirm from Figure 5–18 that the process on processor 11 is one of the important contributors to the critical path of the computation. A conclusion which can be drawn from these displays is that the loads of the "worker" processes are not well-balanced.

In this example, the standard performance displays have been used for visualising a program. It is also possible to use very restricted application specific displays. This can be done, for example, using process names, and narrowing down the information to a specific type of communication. This may help the

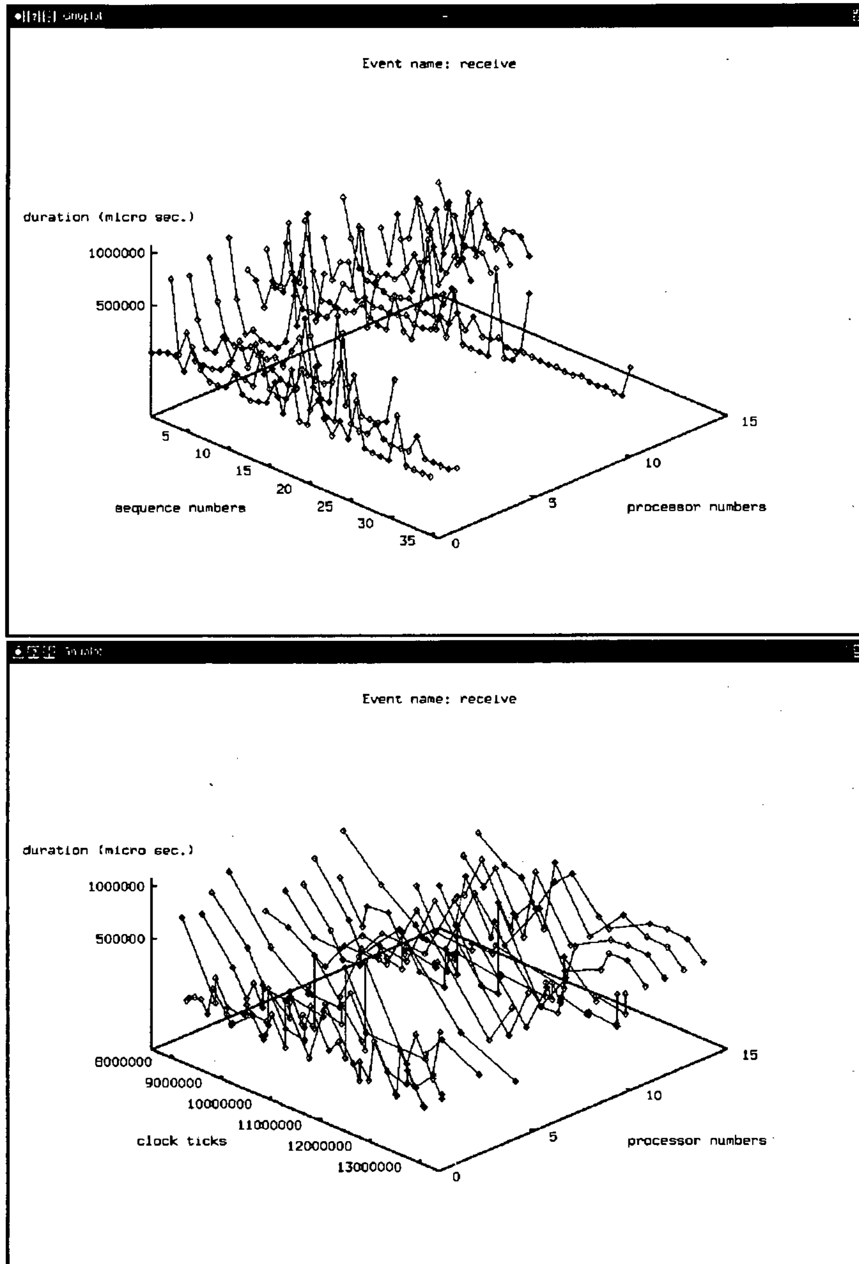


Figure 5–17: 3-D views of blocking intervals due to message receives

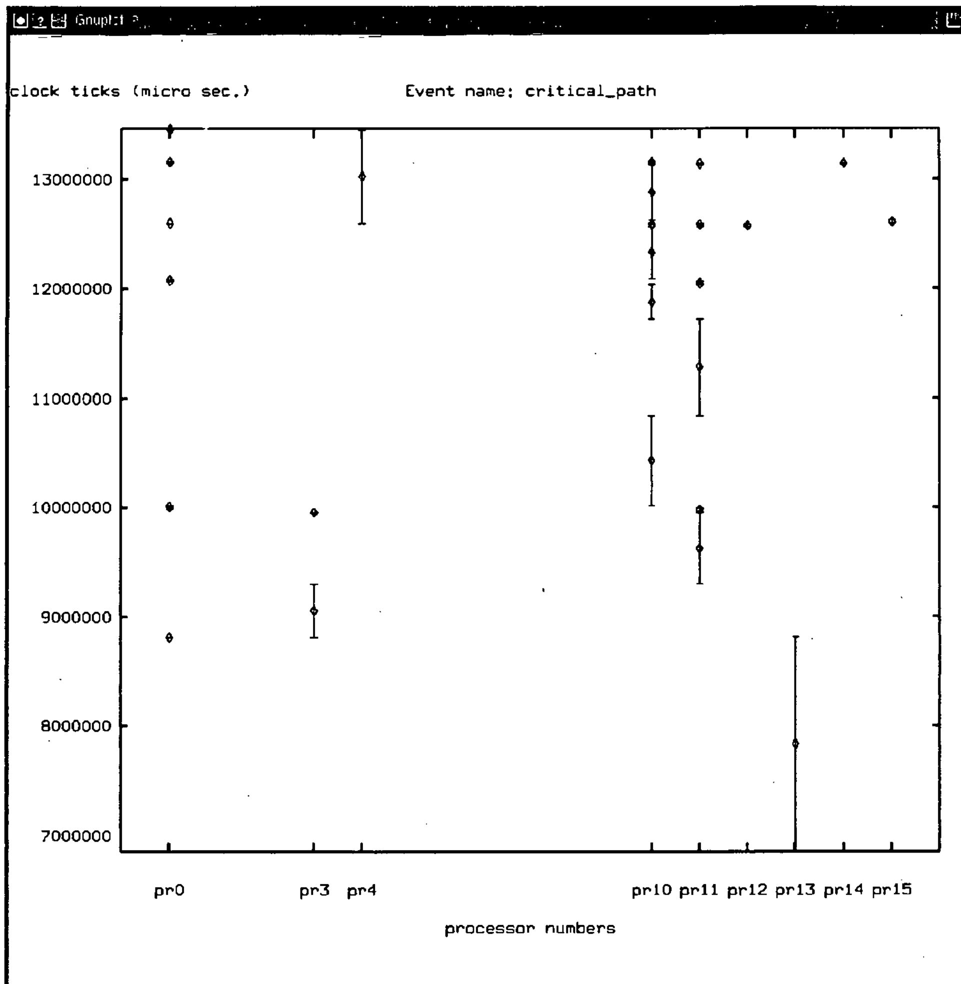


Figure 5–18: Gantt chart view of the critical path

user to gain insight into the interactions which are happening between the processes. Application specific visualisation can only be carried out efficiently by the programmer.

5.4 Summary

In this chapter, a novel performance visualisation technique is presented. The parallel program traces are abstracted, and filtered using a language which uses pattern matching to select the related information from a large volume of program traces. This selected information is presented in graphical forms to locate and measure the duration of a program activity, or to see the distribution of a particular program activity.

Using application specific abstraction, more meaningful visual displays than conventional performance displays can be achieved, if the user has some knowledge about the program which is visualised.

The event abstraction technique is also used for defining performance metrics, and for visualising them. This technique can support performance displays ranging from very simple parallelism displays to abstract views of the critical path of an application.

In order to visualise the event abstractions, a standard graph plotting utility (i.e. GnuPlot) has been used as a prototyping tool. There are not many display types in this graph plotting utility, but wide variety of event abstractions makes the tool powerful because it is possible to produce large number of different views for the same program trace. The different views of the same event trace can reveal different characteristics of the program, therefore the user has a better chance to investigate different aspects of a parallel program.

Chapter 6

Summary and Conclusions

The main theme of this thesis is to develop techniques which help the programmer to analyse and tune the performance of existing distributed memory MIMD parallel programs. For this purpose, event monitoring techniques have been investigated. All three stages of event monitoring; collecting, filtering and visualising event traces have been covered.

Two monitoring systems have been implemented to investigate the intrusiveness of the event collection techniques, and to carry out experiments on real parallel programs.

The first monitoring system investigated was a hybrid monitoring system, implemented on the POSIE testbed. In this monitoring system, very low intrusion levels have been achieved. Each event costs a few machine instructions including both the preparation and the recording of the event.

The implementation of monitoring hardware has been achieved by designing small probes each of which monitors a processor board. A central monitor board controls these slave probes. This implementation can be extended to bigger systems by distributing the central monitor, and by allowing each monitor to control a subset of the slave monitoring probes.

Also in this implementation, a clock compression technique has been investigated. This technique takes advantage of frequent events. Instead of sending full time-stamps to the monitor, only the lower half of the time-stamps are sent, and when there is an overflow to the higher half, the monitor is notified by an extra overflow event. In this specific implementation, this compression technique can reduce the bus traffic between probes (SMIs) and the monitor (MMI) by up to 33%.

It was also found that filtering events while they are being collected by the probes (i.e. using hardware in the probes to filter events) was useful for load balancing purposes, but there is not much use of this for post-mortem purposes. However, this filtering hardware can be modified to support the abstraction mechanism. Instead of collecting several low level primitive events, recognising some frequent event sequences, and replacing them with a single event can reduce the amount of information to be collected without losing important events. It is not necessary to provide the full event abstraction mechanism but just a fast and simple event sequence detector. This is functionally a subset of the event abstraction mechanism, and operates on locally available event sequences.

Efficient monitoring of parallel programs using software instrumentation can also be achieved. Two different versions of software instrumentation have been implemented, and their intrusion levels have been measured using a test program whose granularity could be changed. It was found that using communication channels for monitoring purposes can increase the intrusion level dramatically. Since memory technology is improving towards providing large capacity and low cost devices, in future it will be possible store enough monitoring information locally on each processor board.

The most expensive part of the software instrumentation was found to be the clock handling. Since a system function call is issued to access the system clock, and the operating system is also sharing the same clock with the monitor, reading

the clock is expensive. By adding a dedicated clock to the processor boards, the clock reading overhead could be reduced. It is also possible to implement a compression technique in hardware in order to reduce the size of the time-stamps. The techniques which are used in both monitoring systems (i.e. POSIE and ECS Monitors) can be adopted. This will reduce the sizes of the traces to be stored locally.

Filtering of events from the traces of parallel programs has been achieved through abstraction of events. In this approach, event patterns are replaced with more abstract events to reduce the size of the trace information, and to achieve coarse granularity event information that allows the programmer to view a program trace from a wider angle of view.

The event abstraction technique was also proved to be a powerful query technique which can used for defining performance metrics ranging from very simple ones to complex profiling type performance metrics, such as the critical path of a computation history.

Application specific and/or domain specific abstractions can also be achieved through the event abstraction technique. It is possible, for example, to provide template abstractions for a template (e.g. skeletal programming) program. These template abstractions can then be used for analysing a particular program which is implemented by adding extra code to the template program.

Since the definitions of the event abstractions can be kept in a library, it is possible to extend the existing capabilities of the tool to support new analysis types, or different variations of the message passing programming paradigm. For example in CM-5 [30], it is possible to implement a mixture of MIMD and SIMD modes. It is easy to support such a new paradigm by recognising new primitive events, and by redefining the abstractions to interpret new event patterns.

In the early stages of the design, the style of the event pattern definitions was carefully chosen to be suitable for a graphical interface. This decision led to the

implementation of a graphical interface which simplified the task of defining the event patterns. Because of the spatial characteristics of the graphical objects, the graphical interface also provided a natural protection against some of the erroneous conditions which can be defined using the textual interface.

A novel visualisation approach has been developed. In this approach, a small number of general purpose display types has been used to visualise different characteristics of abstract events. It was shown that defining the semantics of the performance information through abstract events could lead to a wide range of performance displays which ranged from standard ones to application specific performance displays. Application specific performance visualisation provided new opportunities that could lead to a better performance visualisation of big parallel programs.

Firstly, the application specific visualisation provided a convenient way of selecting information to be visualised. This made it possible to visualise some of the detailed behavioural patterns using summary type displays. This is a positive step towards visualising massively parallel programs.

Secondly, it was showed that the micro activities of a parallel program could be related to its macro activities by magnifying micro activities. This helps the user to investigate high level details of a parallel program first, and then, if necessary, to investigate low level details by magnifying them.

Thirdly, the combination of the application specific visualisation and the parallelism displays (i.e. real and virtual parallelism displays) provided a new approach to spot contention related performance bottle-necks. Unlike other performance visualisation techniques, in this approach, the term contention is not restricted to channel contention. Contention can occur on any resource, ranging from channels (hard-resources) to processes or parts of processes (soft-resources). The differences between real and virtual parallelism views of an abstraction can reveal the contention for the resources which are represented by the abstraction.

One of the important results which has been achieved in this work is in providing a better support for the performance tuning task. This task involves experimenting with the different versions/configurations of a parallel program, to find possible performance bottle-necks and to achieve better performance. The detailed performance results of these executions has to be correlated with one another. The approach presented in this thesis provides two ways of achieving this goal. Using the abstraction mechanism, it is possible to handle multiple event histories. This will result in abstractions which will reflect the differences or similarities between different executions of the program. Again using the abstraction mechanism, it is possible to focus into a specific behaviour of these executions, and by visualising each of them separately, we have the opportunity to visually compare two or more different executions.

Further progress towards visualising large scale computations (i.e. massively parallel MIMD programs) has been achieved by taking advantage of the abstraction mechanism. Summary type performance displays (e.g. parallelism and utilisation displays) can be specialised to visualise any particular behaviour(s) (which is/are represented by the instances of an abstract event) of a massively parallel program. This enables the user to summarise a behavioural pattern which gives more detailed information than the existing performance displays of the same kind. This can be considered as a conservative approach which does not invent new performance displays for massively parallel programs but utilises the classic displays to visualise different types of performance information.

In this thesis, only a small domain of performance monitoring and analysis of parallel computers has been investigated. There are many areas which require further investigation.

One area which can be investigated is the performance analysis and visualisation libraries. This investigation can aim at several targets. Firstly, different programming paradigms can be supported, and libraries created which contain ab-

stractions to interpret event patterns of each programming paradigm. Secondly, domain specific event abstraction libraries can be built for classes of algorithms or skeletal algorithms [14]. Thirdly, the metrics which are exemplified in this thesis are not exhaustive, they only constitute a small subset of all metrics. The existing performance metrics, especially profiling metrics, can be classified, and they can be defined as event abstractions in a special performance metrics library.

Another area to be investigated is the visualisation side of the event abstractions. The display types which are used in this thesis are not exhaustive. It is also necessary to investigate display types which are specific to some cases, such as visualising the critical path, and dynamic displays which can animate the event abstractions.

In this thesis, two prototype user interfaces for the event abstraction mechanism were presented. Further study is needed to explore the parameters of the potential user interfaces which will best exploit the benefits of the event abstraction mechanism. It is also necessary to study the event abstraction technique in the context of the performance tuning task in order to evaluate how this technique affects the quality of the tuning task.

Last but not least, a parallel implementation of the event abstraction technique requires special attention. Since a parallel computer with a large number of processing elements can produce very large event traces, transferring these traces to a sequential machine, and processing them there can create bottle-necks. It is possible to keep event traces locally in the processing elements, and then the same parallel machine which is used for running the monitored application can be used for processing the event traces. This will require a parallel search algorithm which finds event patterns in a distributed data set. This approach will reduce the delay between running the monitored application and getting its performance results.

Bibliography

[1] G.M. Amdahl, "Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities," AFIPS Conference Proceedings, April 1967, pp. 483-485.

[2] F. Baiardi, N. De Francesco, and G. Vaglini, "Development of a Debugger for a Concurrent Language," IEEE Trans. on Software Engineering, Vol. SE-12, No. 4, April 1986, pp. 547-553.

[3] H.E. Bal, "A Comparative Study of Five Parallel Programming Languages," Technical Report, No. IR-250, Faculteit der Wiskunde en Informatica, Vrije Universiteit, Amsterdam, June 1991.

[4] H.E. Bal, J.G. Steiner, A.S. Tanenbaum, "Programming Languages for Distributed Computing Systems," ACM Computing Surveys, Vol. 21, No. 3, September 1989, pp. 261-322.

[5] P. Bates, "High-level debugging of distributed systems: The behavioural abstraction approach," J. System Software, No. 3, Elsevier Science Publishing Co., 1984, pp. 255-264.

[6] P. Bates, "Debugging Heterogeneous Systems Using Event-Based Models of Behavior," ACM Sigplan Notices, Workshop on Parallel and Distributed Debugging, No. 24(1), January 1989, pp. 11-22.

[7] T. Bemmerl, and A. Bode, "An Integrated Environment for Programming Distributed Memory Multiprocessors," Lecture Notes in Computer Science, No. 487, Proceedings of Distributed Memory Computing, EDMCC2, April 1991, pp. 130-142.

[8] T.Bemmerl, O. Hansen, and T.Ludwig, "PATOP for performance of Parallel Programs," Parallel Programs," Proceedings CONPAR 1990 - VAPP IV, Sep. 1990, Springer-Verlag, pp. 840-851.

[9] G. Bell, "The Future of High Performance Computers in Science and Engineering," Communications of the ACM, Vol. 32, No. 9, September 1989, pp. 1091-1101.

[10] H. Burkhart, R. Millen, "Performance-Measurement Tools in a Multiprocessor Environment," IEEE Trans. on Computers, Vol. 38, No. 5, May 1989, pp. 725-737.

[11] N. Carriero, D. Gelernter, "How to Write Parallel Programs: A Guide to the Perplexed," ACM Computing Surveys, Vol. 21, No. 3, September 1989, pp. 323-357.

[12] K.M. Chandy, and C. Kesselman, "Parallel Programming in 2001," IEEE Software, November 1991, pp. 11-20.

[13] W.S. Cheng, V.E. Wallentine, "DEBL: A Knowledge-Based Language for Specifying and Debugging Distributed Programs," Communications of the ACM, Vol. 32, No. 9, September 1989, pp. 1079-1084.

[14] M.I. Cole, "Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation," PhD Thesis, University of Edinburgh, Department of Computer Science, CST-56-88, October 1988.

[15] "CS-Tools — A Technical Overview," Technical Report, Meiko Limited, 1990.

[16] M.J. Flynn, "Some Computer Organizations and Their Effectiveness," IEEE Trans. on Computers, Vol. C-21, No. 9, September 1972, pp. 948-960.

[17] R.J. Fowler, T.J. LeBlanc, and J.M. Mellor-Crummey, "An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors," —.

[18] J. Gait, "A debugger for concurrent programs," Software Practice and Experience, Vol. 15, No. 6, pp. 539-554.

[19] H.L. Gantt, "Organizing for Work," 1919.

[20] GnuPlot User's Manual, Version 3.0.

[21] D. Haban and G. Shin, "Application of Real-Time Monitoring to Scheduling with Random Execution Times," IEEE Trans. on Software Engineering, Vol. 16, No. 12, December 1990, pp. 1374-1389.

[22] D. Haban and D. Wybranietz, "Hardware Supported Monitoring in Distributed Computer Systems," Technical Report, Universitat Kaiserslautern Fachbereich Informatik, Feb 1986.

[23] M.T. Heath, and J.A. Etheridge, "Visualizing the Performance of Parallel Programs," IEEE Software, September 1991, pp. 29-39.

[24] M.T. Heath, and J.A. Etheridge, "ParaGraph: A Tool for Visualizing Performance of Parallel Programs," Users Guide, Oak Ridge National Laboratory, Oak Ridge, TN, November 1991.

[25] W. Hern, "An Event Monitor for the Computing Surface," EPCC Summer Scholarship Report, Edinburgh Parallel Computing Centre, September 1991.

[26] M.D. Hill, "What is Scalability?" ACM Computer Architecture News, Vol. 18, No. 4, December 1990, pp. 18-21.

[27] C.A.R. Hoare, "Communicating Sequential Processes," Comm. of the ACM, Vol. 21, No. 8, August 1978, pp. 666-677.

[28] R. Hofman, R. Klar, N. Luttenberger, B.Mohr, and G. Werner, "An Approach to Monitoring and Modeling of Multiprocessor and Multicomputer Systems," in Proceedings of Performance of Distributed and Parallel Systems, T. Hasegawa, H. Takagi, and Y. Takahashi (Eds.), Elsevier Science Publishers B.V. (North-Holland), IFIP, 1989.

[29] J.K. Hollinsworth, and B.P. Miller, "Parallel Program Performance Metrics: A Comparison and Validation," in Proceedings of Supercomputing '92, Minneapolis, 1992.

[30] R.N. Ibbett, "Parallel Computer Architectures: Where Next?" in Proceedings of Transputers '92, "Advanced Research and Industrial Applications," M. Becker, L. Litzler, and M. Tréhel (Eds.), IOS Press, 1992.

[31] R.N. Ibbett, D.A. Edwards, T.P. Hopkins, C.K. Cadogan, and D.A. Train, "CENTRENET—A High-performance Local Area Network," Computer Journal, Vol. 28, No. 3, 1985, pp. 231-242.

[32] J. Joyce et al., "Monitoring Distributed Systems," ACM Trans. on Computer Systems, Vol. 5, No. 2, May 1987, pp.121-150.

[33] J. Kim, H. Yoo, and Y. Lee, "Design and Implementation of a Temporal Query Language with Abstract Time," Information Systems, Vol. 15, No. 3, 1990, pp. 349-357.

[34] R. Klar, A. Quick, and F. Sötz "Tools for a Model-driven Instrumentation for Monitoring," —.

[35] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," CACM, Vol. 21, No. 7, July 1978, pp. 558-565.

[36] B. Lazzerini, F. Lopriore, "Abstraction Mechanisms for Event Control in Program Debugging," IEEE Trans. on Software Engineering, Vol. 15, No. 7, July 1989, pp. 890-901.

[37] T.J. LeBlanc, J.M. Mellor-Crummey, R.J. Fowler, "Analysing Parallel Program Executions Using Multiple Views," Journal of Parallel and Distributed Computing, No. 9, 1990, pp. 203-217.

[38] T.J. LeBlanc, B.P. Miller, "Workshop Summary," ACM Sigplan Notices, Workshop on Parallel and Distributed Debugging, No. 24(1), January 1989, pp. ix-xxi.

[39] T. Lehr, Z. Segall, D.F. Vrsalovic, E. Caplan, A.L. Chung, and C.E. Fineman, "Visualizing Performance Debugging," IEEE Computer, October 1989, pp. 38-51.

[40] A.D. Malony, "Performance Observability," PhD Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, October 1990.

[41] A.D. Mallony, D.H. Hammerslag, and D.J. Jablonowski, "Traceview: A Trace Visualization Tool," IEEE Software, September 1991, pp. 19-28.

[42] A.D. Malony, D.A. Reed, "Visualizing Parallel Computer System Performance," Technical Report, Department of Computer Science Centre for Supercomputing Research and Development, University of Illinois, September 1988.

[43] A.D. Malony, D.A. Reed, J.W. Arendt, R.A. Aydt, D. Grabas and B.K. Totty "An Integrated Performance Data Collection, Analysis, and Visualization System," Technical Report, Department of Computer Science Centre

for Supercomputing Research and Development, University of Illinois, March 1989.

[44] Marinescu et al., "Models for Monitoring and Debugging Tools for Parallel and Distributed Software," Journal of Parallel and Distributed Computing, No. 9, 1990, pp. 171-184.

[45] G. McDaniel, "METRIC: A Kernel Instrumentation System for Distributed Environments," Proc. of 6th ACM Symposium on Operating Systems Principles, Nov. 1977, pp. 93-99.

[46] C.E. McDowell, D.P. Helmbold, "Debugging Concurrent Programs," ACM Computing Surveys, Vol. 21, No. 4, December 1989, pp. 593-622.

[47] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S-S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," IEEE Trans. on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990, pp. 206-217.

[48] B.P. Miller, "DPM: A Measurement System for Distributed Programs," IEEE Trans. on Computers, Vol. 37, No. 2, Feb. 1988, pp.243-248.

[49] B.P. Miller, C. Macrander, S. Sechrest, "A Distributed Programs Monitor for Berkeley UNIX," Software-Practice and Experience, Vol.16(2), February 1986, pp. 183-200.

[50] B.P. Miller, C-Q. Yang, "IPS:An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed Programs," Proceedings of the 7th International Conference on Distributed Computing Systems, Berlin, West Germany, Sep. 1987, pp. 482-489.

[51] A. Mink, R. Carpenter, G. Nacht, and J. Roberts, "Multiprocessor Performance Measurement Instrumentation," IEEE Computer, September 1990, pp. 63-75.

[52] B. Mohr, "Performance Evaluation of Parallel Programs in Parallel and Distributed Systems," Proceedings of CONPAR 1990 - VAPP IV, Sep. 1990, Springer-Verlag, pp. 176-187.

[53] D.A. Norman, "Design Rules Based on Analyses of Human Error," Communications of the ACM, Vol. 26, No. 4, April 1983, pp. 254-258.

[54] D. Nussbaum, and A. Agarwal, "Scalability of Parallel Machines," Communications of the ACM, Vol. 34, No. 3, March 1991, pp. 57-61.

[55] C.M. Pancake, "Software Support for Parallel Computing: Where Are We Headed?" Communications of the ACM, Vol. 34, No. 11, November 1991, pp. 53-64.

[56] D.A. Reed and D.C. Rudolph "Experiences with Hypercube Operating System Instrumentation," Technical Report, Department of Computer Science Centre for Supercomputing Research and Development, University of Illinois, April 1990.

[57] M.H. Reilly, "A Performance Monitor for Parallel Programs," Academic Press Limited, London, 1990, ISBN 0-12-586330-6.

[58] G-C. Roman, K.C. Cox, "A Declarative Approach to Visualizing Concurrent Computations," IEEE Computer, October 1989, pp. 25-36.

[59] M. Seager, S. Campbell, S. Sikora, R. Strout, and M. Zosel, "Graphical Multiprocessing Analysis Tool (GMAT)," Technical Report, Lawrence Livermore National Laboratory, Computing and Mathematics Research Division, January 1988.

[60] Z. Segall, A. Singh, R.T. Snodgrass, A.K. Jones, and D.P. Siewiorek, "An Integrated Instrumentation Environment for Multiprocessors," IEEE Transaction on Computers, Vol. C-32, No. 1, January 1983, pp. 4-13.

[61] R. Snodgrass, "A Relational Approach to Monitoring Complex Systems," ACM Trans. on Computer Systems, Vol. 6, No. 2, May 1988, pp. 157-196.

[62] T.L. Sterling, A.J. Musciano, D.J. Becker, "Multiprocessor Performance Measurement Using Embedded Instrumentation," Proc. of the 1988 Int. Conf. on Parallel Processing, Vol. 1, August 1988, pp. 156-165.

[63] N. Stroud, "Introduction to the Edinburgh Concurrent Supercomputer," Edinburgh Parallel Computing Centre, EPCC-UG-1991-7, March 1991.

[64] C.B. Stunkel, B. Janssens, and W.K. Fuchs, "Address Tracing for Parallel Machines," IEEE Computer, January 1991, pp. 31-38.

[65] J.J.P. Tsai, K-Y. Fang, and H-Y. Chen, "A Noninvasive Architecture to Monitor Real-Time Distributed Systems," IEEE Computer, March 1990, pp. 11-23.

[66] L.G. Valiant, "General Purpose Parallel Architectures," Handbook of Theoretical Computer Science, J. van Leeuwen (Ed.), Elsevier Science Publishers B.V. (North-Holland), 1990, pp. 944-971.

[67] Y. Vassiliou and M. Jarke, "Query Languages—A Taxonomy," Human Factors and Interactive Computer Systems, Proceedings of the NYU Symposium on User Interfaces, Y. Vassiliou (Ed.), New York, USA, May 1982, pp. 47-82.

[68] D. Wybranietz, "Measurement Techniques for Distributed Systems During Operation," Lecture Notes, ACM Sigmetrics, Proc. 1989 Int. Conf. on Meas-

urement and Modeling of Computer Systems, Berkeley, California, USA, May 1989.

[69] C-Q. Yang, B.P. Miller, "Critical Path Analysis for the Execution of Parallel Distributed Programs," IEEE 8th Int. Conf. on Distributed Computing Systems," San Jose, California, June 1988, pp. 366-373.

[70] M. Zitterbart, "Monitoring and Debugging Transputer-Networks with NETMON-II," Proceedings of CONPAR 1990 - VAPP IV, Sep. 1990, Springer-Verlag, pp. 200-209.

Appendix A

Published Papers

The following papers were published in

Proceedings of the International Conference on Parallel Computing '91, London, 3-6 September 1991, D.J. Evans, G.R. Joubert and H. Liddell (Eds.),

and in

Proceedings of the Workshop on Performance Measurement and Visualization, Moravany, Czechoslovakia, 23-24 October 1992, G. Haring and G. Kotsis (Eds.).

Elsevier Science Publishers B.V. accept that the author(s) reserve(s) the following:

1. All proprietary rights such as patent rights.
2. The right to use all or part of this article in future works of their own, press releases, reviews or text books.
3. The right of reproduction and limited distribution for own personal use or for company use for individual clients, provided that source and copyright are indicated and copies per se are not offered for sale.

Parallel Computing '91
D.J. Evans, G.R. Joubert and H. Liddell (Editors)
© 1992 Elsevier Science Publishers B.V. All rights reserved.

A Visual Performance Analysis Environment

Kayhan İmre

Department of Computer Science and
Edinburgh Parallel Computing Centre,
University of Edinburgh, Room: 3416, Edinburgh EH9 3JZ, Scotland

Abstract

Performance tuning is an important task in parallel program development cycle. Fine performance tuning requires detailed information about how program behaves when it is executed. Event monitoring can provide detailed information about a parallel program execution. In this approach, important events of a parallel program are recorded to build an execution history of the program. Then, these events are used for analysing the program by correlating event patterns to the program itself. The important problem with event histories is their sizes. Depending on the problem size and the duration of the monitoring, they can grow very fast. It is difficult for a person who carries out tuning task to trace event patterns in a large amount of data. There is a need for step-by-step refinement techniques to extract useful information from event histories. In this paper, an event abstraction language, and some techniques to analyse event histories by using this language are presented.

1. Introduction

Parallel programs are difficult to implement compared with their sequential counterparts. The difficulty arises when the programmer wants to debug program that contains competing parallel process. Because of loose connections between parallel processes, bugs in the program may not surface every time the program runs, and depending on the underlying parallel architecture program can change behaviour. This property of parallel programs makes them difficult to monitor and analyse. Once the programmer implements the parallel program correctly, (s)he faces another problem: performance. It is not easy to achieve the expected speed-up straightaway. The next task to carry out is performance tuning.

This task involves in monitoring execution(s) of the parallel program, and analysing the results of monitoring. Depending on the results of the analysis, program may require some modifications to make it run faster. When the programmer carries out a modification, it is also possible to create new bugs in the program. As one can imagine, parallel program development cycle is not straightforward, and it requires extensive facilities to analyse a parallel program. Therefore, it is crucial to get information about the behaviour of the program that is under development. In this paper, event monitoring systems are considered as main sources of information to analyse parallel programs. These monitoring systems collect important events of parallel program executions, and they can create an execution history for each run of a parallel program. These events can help the user to understand the timing characteristics of a program, and to associate higher level program constructs with these characteristics. Since the monitoring data are low level compared with the high level program constructs, special tools are needed to assist the user in extracting useful information from the low level data. These kinds of tools are crucial in performance tuning when there are hundred of processes to analyse.

In this paper, a visual performance analysing language and its use in performance tuning of parallel programs are presented. The language is based on pictures that define event patterns and actions to be taken when these patterns are found in an event history. Every picture (pattern) contains events whose locations define temporal relations among them, and it also contains representations for identifying communication channels, histories, processors and processes. An event recogniser searches these patterns in an execution history, and upon finding a satisfying instance, the recogniser can add new events to the history or it can filter some events from the history. The following tasks can be carried out by using this language to analyse performance of a parallel program.

Program behaviour model construction. The user can exploit event traces to identify different phases of a parallel program. Since the phases are application specific information, and automatic phase detection tools are not precise, the user defines the phases in order to extract the program behavioural model.

Standard queries. The standard performance metrics, some well-known query types such as finding the critical path in a computation graph were implemented by using this language. The user can add these kinds of queries to a query database, or modify the existing ones.

Building application specific views. The program behaviour model and standard queries are combined together to achieve application specific performance views. Since the performance metrics are presented in

conjunction with the behavioural model, the programmer can directly use this information to tune the performance of the program.

Correlating different histories. The capability of addressing more than one execution history simultaneously enables user to correlate the differences and the similarities between the histories that belong to the same program but different executions under different constraints. This is important for analysing data dependent properties of a parallel program, for investigating the effects of the different mapping strategies, or for evaluating system related properties.

The results of a query are presented on a Gantt chart display. The pictures that define a query also resembles a Gantt chart display. In this respect, it is easy to define queries and to get the results in the same form of representation.

2. A Visual Performance Analysis Tool

2.1. Background

Event monitoring provides detailed level information about parallel program executions. An event history of a parallel program is a valuable source of information that can support wide range of performance displays ranging from the statistical ones to fine granularity time-line (Gantt Charts) displays. Since fine granularity level information is expensive to gather, monitoring system can disturb the program that is monitored. The problem is to keep the disturbance at an acceptable level, which probe effect does not occur. In our experiments, two different monitoring systems are used for monitoring programs. The first one is a hybrid system that has software instrumentation with hardware support [Im91]. The disturbance level of this system is very low since some functions of the monitoring instrumentation are carried out by a special hardware monitor which collects and timestamps the events. These events are triggered by a very small software instrumentation. The second monitoring system is a software system that uses the resources on which the monitored application runs. This monitoring system has higher disturbance level than the first one does. Both monitoring systems write to a standard event trace file that can be analysed by the software on a Sun Workstation. Figure 1 illustrates both monitoring systems and user interface system on a workstation.

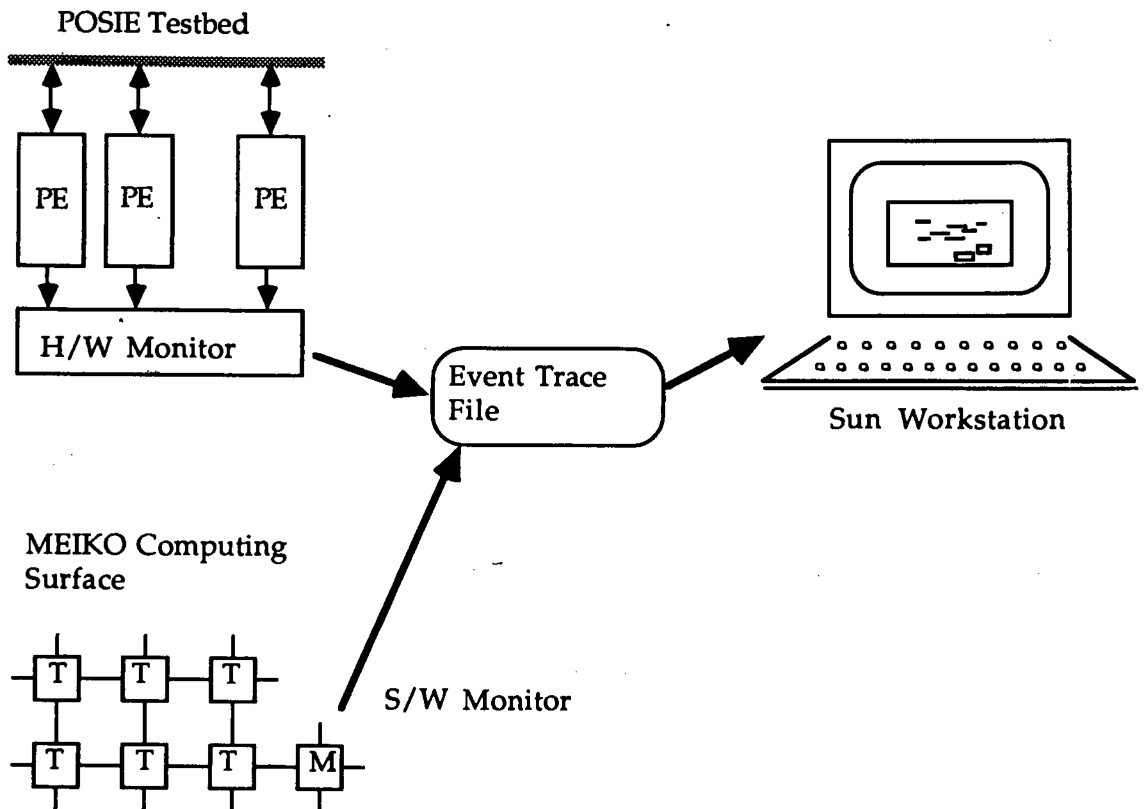


Figure 1. Experimental parallel program monitoring systems.

Every event in a program history is represented by a primitive event trace that is captured and recorded by the monitoring instrumentation. These event traces are the instances of small number of event types that are representatives of various state changes in a computation. The event types that are used in this context are shown in Figure 2.

The instances of these primitive events can represent detailed execution history of a parallel program. Different instances of the same event are differentiated by their time-stamps. Given any instance of an event, the program stage in which the event is signaled can be identified by tracing the events in the history. Since the time-stamps are generated by hardware and there is a global clock signal, events in an execution history can be ordered totally [La78]. These primitive events are the basic building blocks in an execution history, and each of them is an instant in time (i.e., a primitive event occupies zero time). To present only these primitive events alone does not give useful information to

the user since it is difficult to present such an event in a large interval especially if there are many events in that particular interval. Instead of presenting individual primitive events, presenting compound events that replace groups of primitive events can be more meaningful to the user. For example, a "schedule process" event and immediately following that, a "pre-empt process" event can be replaced by a "running" event that represents an interval of when a particular process is running on a particular processor. Although this simple replacement can dramatically improve the presentation of the history, higher levels of replacements are needed to obtain better history views that are close to the program's stages. If this is done, any information associated with the view also can be related to the program easily. For example, a unique combination of communication patterns can be used as an anchor in a history, then it is a starting point for the other events that are going to be created on top of it. Identifying some important stages in a program history is important because we can analyse the performance of the program by referencing to them.

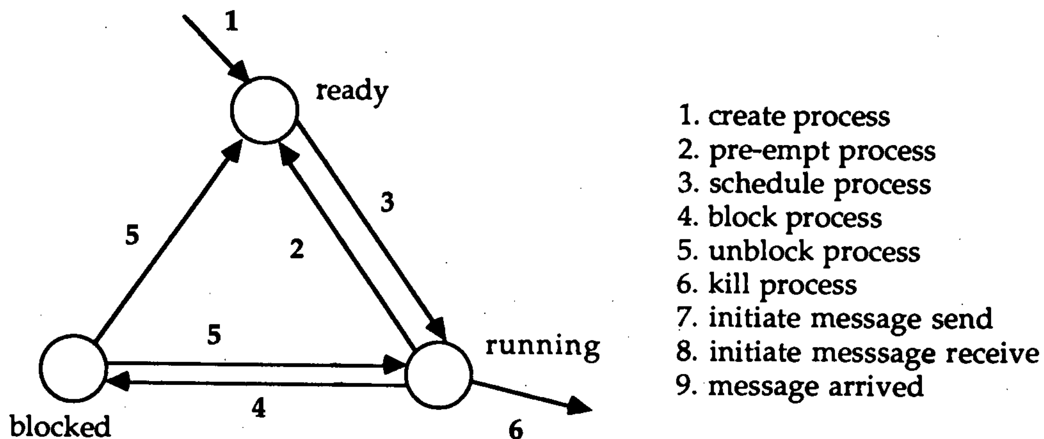


Figure 2. Scheduling graph and primitive event types.

The primitive events can represent the computation graph of a parallel program. This graph represents the dependencies between various parts of the computation. By working out these dependencies, one can extract valuable information from this graph. For example, critical path of a computation can help the user in finding out about the program parts that slow down the computation [MiCl90]. The computation graph is an abstraction of primitive events that correspond to the vertices of this graph. There is an arc between every event pair that are adjacent within the same process boundary (Figure 3).

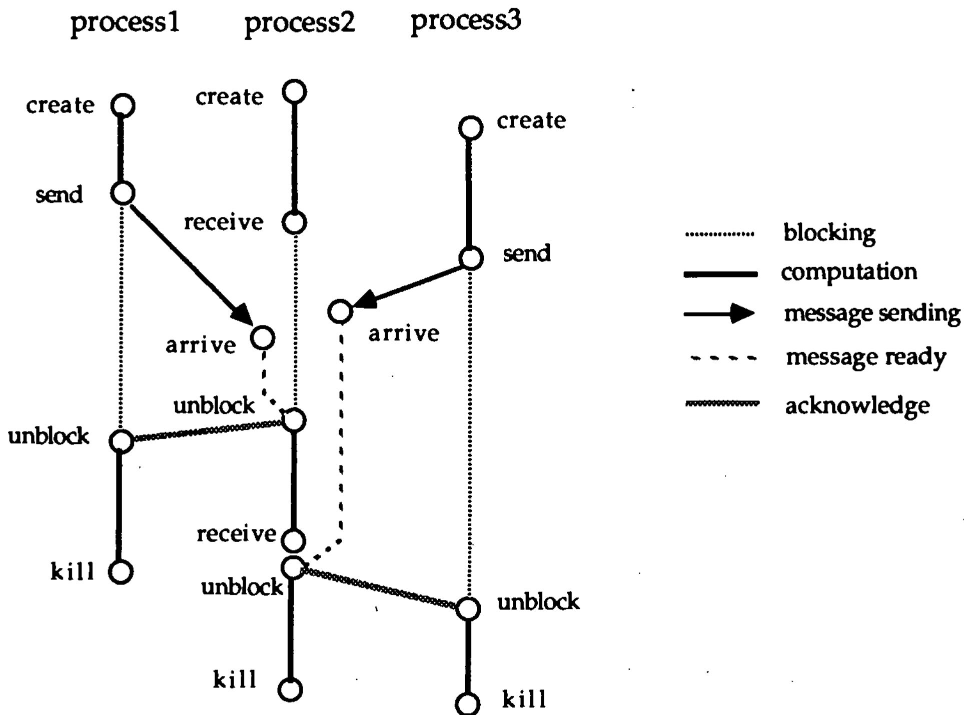


Figure 3. A computation graph.

There is also an arc between two events that are sending and receiving ends of a message transmission. Since a primitive event is instantaneous there is no cost attached to it. But, there is a cost for an arc between two events. The cost is the absolute difference between the time-stamps of those events that enclose the arc. The events on the both ends of an arc also can be used for identifying the type of its cost. For example, "send message" and "unblock process" pair makes an arc that represents the cost of blocking a process.

2.2. The language

The visual language, which is used for defining queries, provides a mechanism to create new events hierarchically. This is carried out by defining each new event type out of possible combinations of existing events. When an instance of this combination is found in the history, new event is created and added to the history. These combinations are defined by using a graphical notation. The user can compose a picture for every event definition. In a picture, an event is represented by a rectangle and its horizontal location that is relative to the other rectangles defines temporal relationships between the event itself

and the others. These definitions are compiled by the visual language compiler, and they are translated into an intermediate code. This intermediate code is input to the event recogniser that searches a history to find the instances of user defined events. When all events, which are defined in the picture, are found in the history, the time stamps of these events are tested against the temporal relationships to validate or invalidate this instance. Once the instance is validated, operations for creating new events, and filtering some of the existing ones are carried out. The event filtering mechanism can be used for two different purposes. First, it is the mechanism for deleting unwanted details from the history. Secondly, it can be used for controlling pattern matching mechanism since it enables user to change the patterns in the history dynamically. For example, let's assume that there are two different definitions that define intersecting set of patterns. Here, one can filter events from the history to prevent the other from recognising the same pattern. In the language, there are four kinds of filtering operations. The first two can delete events from the history and keep them in a tree structure whose root is the new event that is created while with the filtering is carried out. The third one deletes an event from the history, and if there are events that are connected to these events, it returns them to the history back. The last one deletes an event along with any event that is connected to it.

2.3. An Example

To show how we can use the language to analyse an execution history of a parallel program, we carried out an experiment on a very small program. The program has three processes that are arranged as a pipeline (Figure 4). Since this example is intended to show how things work, and how we can visualize the results, it is kept small. Therefore, the bottle-neck of the program, which is "process B," can be seen from the Gantt chart display easily without any further analysis (Figure 5). In this view of the program, the "run" events represent the intervals that processes compute. This event type is an abstraction of some primitive level event patterns such as "unblock" event followed by "send" event, "unblock" event followed by "receive" event. The definitions of these kinds of patterns guide the system in creating more readable views of the program history since the user can select information that (s)he wants to know.

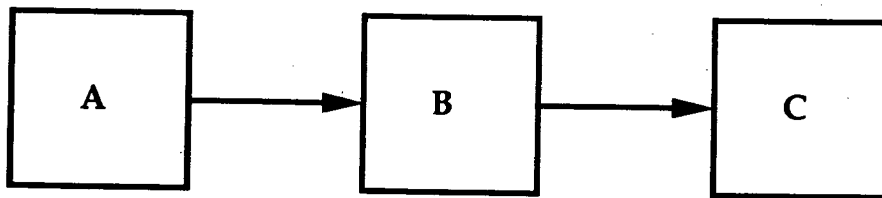


Figure 4. Process graph of the example program.

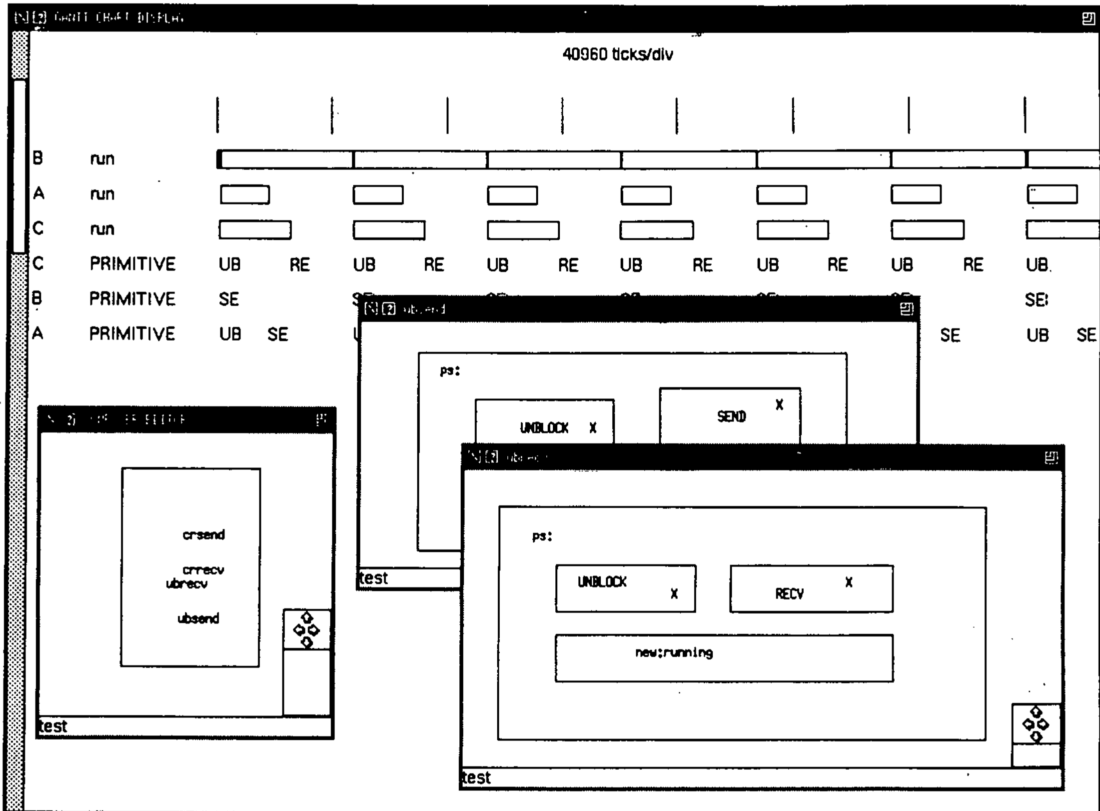


Figure 5. The first display of the example program's history (small segment of the history.)

When the execution history scales up (i.e., computation contains large number of processes, and larger interval of the execution is monitored) it is difficult to recognize the bottle-necks because the details of the execution history can make the manifestation of the bottle-necks less obvious. Here, we can use event abstraction mechanism as a query system. Since we know that the slowest element dominates the computation time and slows down the execution, we designed a general purpose query to find out which process is the slowest and how much time it spends in computing while the others are idle. This extra time

can be shared between other processes. For instance, some computation from one element of the pipeline can be transferred to the next element. In this way, parallelism can increase and the program runs faster.

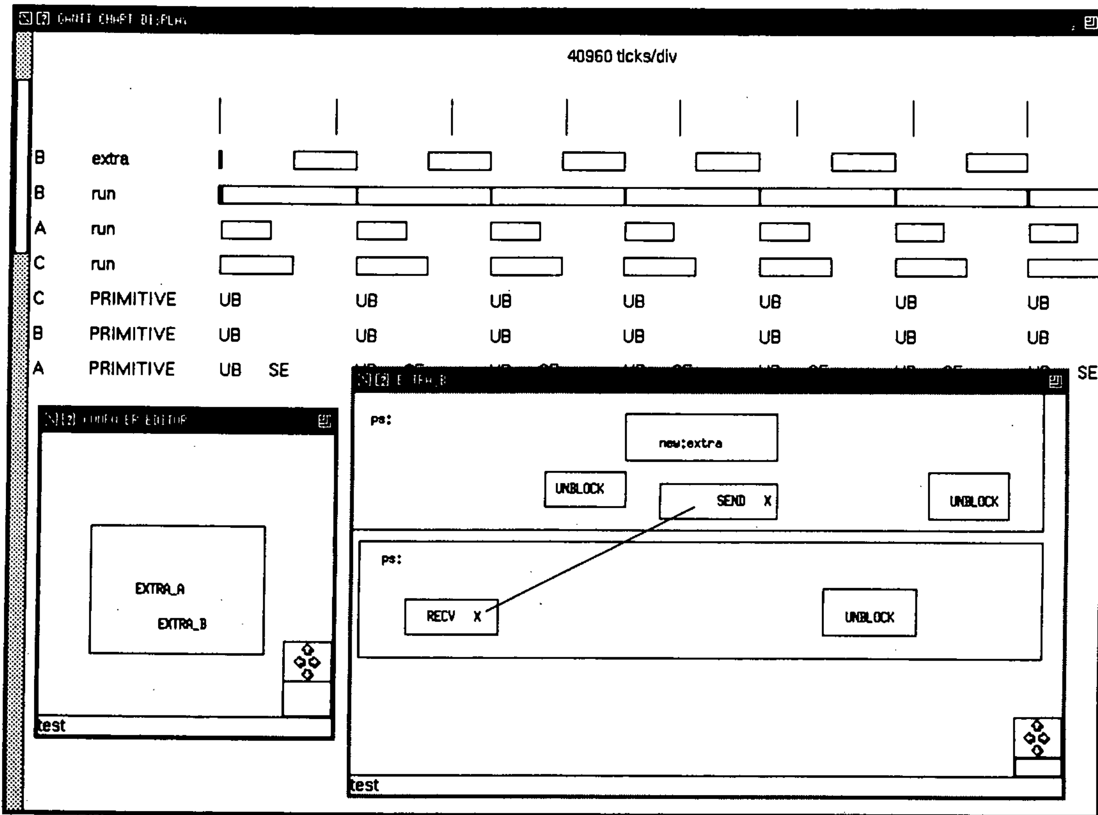


Figure 6. The results of a query.

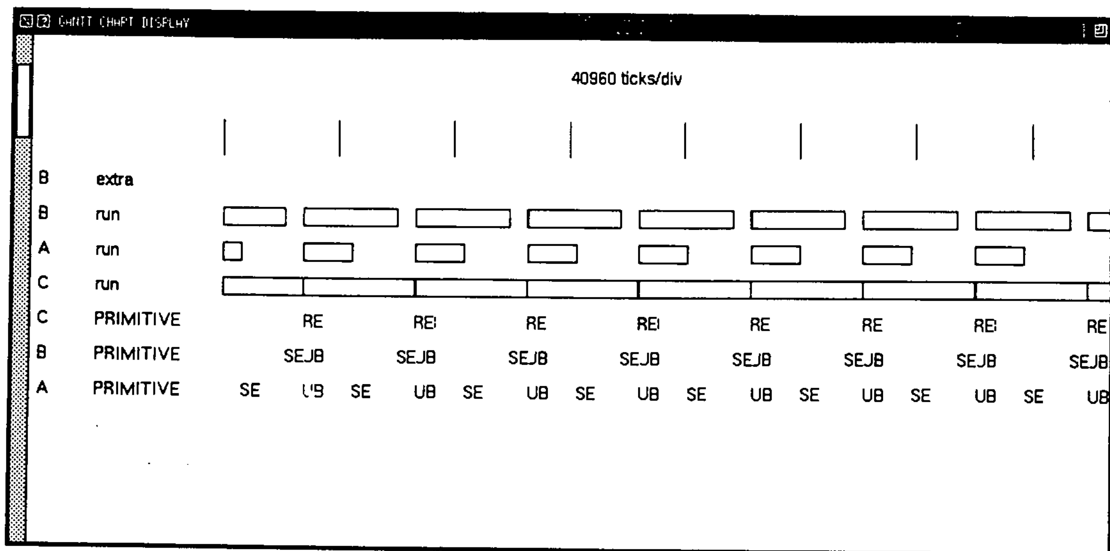


Figure 7. Tuned version of the example and the results of the query.

In Figure 6, we give some event patterns that represent communication between process pairs. These pattern definitions express the situation that receiver attempts to receive message and it blocks since the sender has not sent the message. This query identifies the processes that are slow to produce results. When the system finds this pattern in the history, it replaces the "send" and "receive" events with a new event that is called "extra." This new event represents the interval where the sender computes while the receiver is waiting for it. If we can balance this extra interval between two processes (i.e., moving half of the extra computation to the receiver end) we will improve the performance. The tuned version of the program is given in Figure 7. Though the length of each pipeline phase (i.e., the time between data enters to the pipeline and the result exits from the other end) has not changed, the total execution time of the program decreases. Since we improved the overlapping between the phases of the pipeline, it is possible to compute more phases in a given interval of time. Even though the last modification of the program improved the performance, there are still opportunities for more improvement. By repeating similar tweaking task, we can approach to the ideal case that every element of the pipeline has equal length.

In this example, we used event abstraction mechanism to find the slow processes in a parallel program that contains pipelined processes. In our example, there was only one process that fits into this category. If the pipeline is longer

than this, it is possible to find more than one processes that slow down the program. The query, which is used here, can find all slow processes, and creates "extra" event for each instance of the communications between process pairs. It is up to the user to decide the tuning strategy. Since each slow process is pointed by "extra" event, and these events are displayed on a separate row for each process, the user easily works out how much of the computation has to be transferred to the next process in the pipeline. In our example, except the first one (It is the beginning of the computation,) "extra" events are approximately 60% of the "run" events that belong to "process B." Since the aim of this tuning task is to balance this extra time between the sender and the receiver, we have to move 30% of computation from "process B" to "process C." This query not only helps the user in finding the bottle-neck of the program it also gives extra information that is directly related to the performance tuning task.

We presented how performance tuning task is carried out using the proposed event abstraction mechanism. The benefits of this mechanism are twofold: first, it provides an assistance in detecting problems. Secondly, it can be used for extracting information that is specific to an application. For example, we showed how to find out about the interval of computation that can be balanced between two processes in a pipeline.

3. Performance Measurement and Tuning Techniques

3.1. Program behaviour model reconstruction

An event history of a parallel program contains enough information to reconstruct the program's behavioural model. The activities that have an effect outside a process can be defined as externally observable properties of the process. For the message passing systems, communication activities are the only ones that can be considered within this category. The communication patterns of a process contain useful information to identify particular processing stages that are separated by communication activities. The programmer can exploit this information to reconstruct program's behavioural model by assigning an event to each stage or to a super-stage that is a combination of other stages. In this way, an event history of a parallel program can be translated into a hierarchical history. The aim of this is to achieve a brief history of events that is closer to the programming model than the history of primitive events is. Unlike debugging, performance analysis may not require the complete definition of a behavioural model. In the performance analysis case, the programmer can find milestones of the program by identifying unique patterns of communication in the history. These patterns can be referenced as important activities of the history, and then more definitions can be carried out on top of them.

Since the event traces contain information about the stages of a program and temporal properties of these stages, intermediate results of behavioural model reconstruction can be presented to the programmer to inform him about the timing characteristics of the program. Though these intermediate results are useful to the programmer, they are the basis for the further analysis techniques. Besides, the summarized form of the history still can be very big to be browsed, or they do not reveal the information that the programmer requires for performance tuning. In the following sections, different uses of these intermediate results are given.

3.2. Standard queries

Performance characteristics of a parallel program can be extracted from its event history. This is carried out by defining a set of standard events that represent the various performance metrics. Since these metrics can be calculated without the knowledge of algorithm that is analysed, the event definitions for performance metrics can be considered as general purpose definitions. Therefore these metrics on which performance analysis frequently depends can be defined once and kept in a library. The other solution to this problem is to implement them within the tools as a fixed feature. But the advantage of using event libraries is that they can be modified easily to support various kinds of event histories. For example, if there is an event history that contains different types of events than the previously available ones, some library definitions can be modified to support the new type of history, and the modified definitions are added to the library as a new feature.

Besides performance metrics, the user can customise a library that contains standard event definitions to enhance it with frequently used event types. Since there is no standard way of analysing an event history of a program, an event library is a good opportunity for the user to create an environment that reflects his choices and personal experiences of analysing programs. The idea of creating a library for standard events also becomes handy when a new analysis style is found and wanted to be added to the performance analysis environment. For example the critical path method [YaMi88] is implemented and added to the library. This method is useful when the user wants to identify the process interactions that dominates the computation. The definitions for finding the critical path may be applied to any segment of the event history. In this way, different program stages can be analysed separately. The user controls this process by defining the intervals. These intervals are represented by the composite events that are used as conditions within the definitions. When any occurrence of this event type is created and added to the history, the event definitions for finding the critical path are activated and the interval event is replaced by new events that represent the critical path. The other library definitions work the same way with the critical path definitions do. The user's responsibility is to

define the interval events that are inputs to the library definitions. The interval events are defined as part of program behaviour model reconstruction. Since the knowledge on programming model is used for creating the interval events, the results of the library function can be directly related to the program stages.

3.3. Application specific performance views

General purpose performance displays are useful to some extend. They can display information, which the programmer can browse, for example, how fast the program running or how much communication is carried out. The programmer can speculate these displays about possible reasons for the bottle-necks or the regions with poor performance. Except the time coordinates on the charts, there is no way of relating performance data to the program stages or the effects of a microscopic behaviour on the overall performance. But, a performance view that depicts performance metrics within the context of program stages, or accommodates user defined queries that address application specific performance problems can be more useful than the traditional performance displays, because the programmer can analyse the history in more detail and relate the results to the program more easily.

Without a proper tool programmer does not have many options other than tweaking the program, running it and observing the results. Since every experiment involves in program modification, compiling, run-time monitoring and interpretation of monitoring results, it is time consuming to repeat these experiments many times. In our case, by spending more time on analysing a program history, the programmer can gain much better understanding, make more educated decisions and therefore the number of experiments can be reduced.

By using the visual performance analysis environment, definitions for program behaviour model reconstruction and performance metrics can be combined to analyse performance characteristics of the various stages of a parallel program. Program behaviour model is used as a guide-line to identify history segments that correspond to the various stages of a parallel program. Once program stages are identified, their associated performance metrics can be calculated easily. A program stage is an interval in which a program part of interest is executed, and it is defined by the user as a stage by referencing two activities that encapsulate the interval.

3.4. Correlating different event histories

An event history can be used for analysing performance of a particular execution of a parallel program. Since the performance of a program can be different under different conditions, analysing only one execution of the program may not be suitable for performance tuning purposes. In the following circumstances correlating different histories can be useful:

- Investigating data dependent behaviour of a parallel program,
- Experimenting scalability of a parallel program implementation,
- Comparing different mapping strategies,
- Testing non-deterministic parts of a parallel program.

Since the user can control the way which history correlation is carried out, it is possible to focus on particular parts of the program, and filter out the unnecessary information that relates the other parts(or different detail levels) of a parallel program. For example, the user may want to know about an interaction between two processes, and (s)he wants to investigate the effects of mapping decisions on contention level of the channel between these two processes. Here, the user acts by defining event patterns that represent the interaction between these two particular processes, and by creating an abstract event for each instance of the interaction. Then user can run the program using different mapping strategies, and monitor the executions. By applying user definitions to the histories, which are collected from the executions of the program, the user can create an abstraction level of that particular channel communication. A set of build-in definitions can handle multiple histories and correlate the histories using this particular abstraction level. The correlation can be carried out in several ways, such as finding the longest communication instances in the histories or finding instances that cause other communication instances to be delayed. The advantage of this approach is that user can focus any detail of a parallel program and the system libraries take care of the correlation task. The user can also define new correlation strategies from scratch or by modifying existing system libraries. Since the same tool is used for carrying out both user definitions and system library definitions, modification of the existing definitions is not difficult.

4. Conclusion

Programming distributed memory MIMD computers that contain a large number of processing elements and achieving highly parallel programs are not easy. Parallel program development cycle requires support for performance tuning. Since large parallel programs are difficult to monitor and analyse, the user needs comprehensive tools to understand the behaviour of parallel programs. Event monitoring is a technique for collecting data from a parallel program execution to understand detailed behaviour of the program. When the detail level of monitoring couples with the size of a parallel program, the result is a very large execution history. The proposed event abstraction mechanism provides an infrastructure to simplify execution histories. Since user can select event patterns from the history, and construct application specific performance displays, this mechanism deliver information that is directly related to tuning task.

Definitions for event abstractions can be stored as library functions. Then, this definitions can be reused for similar class of problems. The capability of creating and modifying libraries has some advantages. Firstly, the user can customize the tool for particular problems. Secondly, new techniques for analysing parallel programs can be included into the tool. Finally, the tool can be adapted to the changing demands of the field.

There is one drawback of the system. The user has to learn how to use the visual query language to define queries. The syntax of the language is not difficult to learn since it resembles Gantt chart display which is widely used for performance visualisation purposes. The tool is also accessible for the users who do not know the language. They can use the standard libraries to analyse their programs, and get the basic performance visualisation.

In summary, the proposed event abstraction mechanism provides detailed, application specific performance tuning support.

5. References

[Im91] K. Imre, "An Implementation of a Hardware Monitoring System and a Performance Analysis Environment," Proc. of Workshop on Load Balancing and Performance Monitoring for MIMD Computers, Southampton, 12th April 1991.

[La78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," CACM, Vol. 21, No. 7, July 1978, pp. 558-565.

[MiCl90] B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S-S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," IEEE Trans. on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990, pp. 206-217.

[YaMi88] C-Q. Yang, B.P. Miller, "Critical Path Analysis for the Execution of Parallel Distributed Programs," IEEE 8th Int. Conf. on Distributed Computing Systems, San Jose, California, June 1988, pp. 366-373.

Experiences with Monitoring and Visualising the Performance of Parallel Programs

Kayhan İmre

Department of Computer Science and
Edinburgh Parallel Computing Centre,
University of Edinburgh,
Edinburgh EH9 3JZ, Scotland

Abstract

Tuning the performance of parallel programs is a crucial but difficult task which requires a good understanding of the program to be tuned. The aim of performance monitoring and visualisation tools is to give this good understanding to a programmer. Any parallel computation that has large number of processes makes most of the visualisation techniques obsolete since the volume of performance data to be displayed is much higher than the volume of information that a human observer can comprehend. To extract useful information from large volumes of performance data and to visualise these data in a proper form, new techniques are required. In this paper, we present a technique to extract useful information from large volumes of performance data, and to visualise this information in several different graphical forms. Since the user can assign semantic knowledge to the information to be visualised, it is easy and very flexible to create abstract views which can be interpreted in the context of application-specific or case-sensitive performance visualisation.

1. Introduction

Designing and implementing an efficient parallel program is not easy. Even though it is possible to apply optimisation techniques to the sequential processes of a message passing parallel program individually, the entire program requires much more effort to obtain the best achievable performance. One should carefully consider communication network and processing elements to make use of the available processing capacity. Overloading resources or causing contention will slow down the entire program. While designing a program, to think about complicated system details makes parallel program development a difficult task. Since it is very difficult to implement efficient parallel programs straightaway from the design, some extra work is to be carried out later to make the implementation faster (i.e. performance tuning).

In this paper, we concentrate on providing post execution tools to help the programmer understand and tune a parallel program efficiently. Here, the meaning of the word 'efficient' is twofold: tuning should be done as quickly as possible, and the resulting performance of the parallel program should be the best achievable. Neither objective is easy

to achieve because there is no standard way of analysing a parallel program–each user must develop special skills to tune the performance of particular parallel programs. While novices need to explore their programs' behaviour, experienced users will have their own style of solving problems. Bearing these requirements in mind, the tools for performance tuning should accommodate some facilities to allow user to explore the problem, and to customise the tools depending on their skills.

In our approach, we provide a tool which can be adapted to the changing needs of the programmer. We can also provide libraries which implement standard and domain-specific performance displays.

The scope of the work is limited to the performance of message passing systems even though the tools can be customised to support other programming paradigms. In this work, we consider event tracers as a prime source of information since they can provide both detailed behavioural (qualitative) and statistical (quantitative) data.

2. Instrumentation and Monitoring

An event history of a parallel program is a valuable source of information that can support a wide range of performance displays ranging from the statistical displays to fine granularity time-line (Gantt Charts) displays. Since fine granularity information is expensive to gather, the monitoring system can significantly disturb the program that is monitored. The problem is to keep the disturbance at an acceptable level at which the probe effect does not occur.

Every event in a program history is kept in a primitive event trace that is recorded by the monitoring instrumentation. These event traces contain the instances of a small number of event types that are representatives of various state changes in a computation. The event types that are used in this context are shown in Figure 1.

These primitive events can used for constructing the detailed execution history of a parallel program. Different instances of the same event are differentiated by their order in the trace. Given any instance of an event, the program stage in which the event is signalled can be identified by tracing the events in the history. To present only these primitive events alone does not give useful information to the user , however, since it is difficult to observe such an event in a large interval, especially if there are many events in that particular interval. Instead of presenting individual primitive events, presenting abstract events that replace groups of primitive events can be more meaningful to the user. For example, a "schedule process" event immediately followed by a "pre-empt process" event can be replaced by a "running" event that represents an interval during which a particular process is running on a particular processor. Although this simple replacement can dramatically improve the presentation of the history, higher levels of replacements are needed to obtain better history views that relate closely to the steps of execution of a program. If this is done, any information associated with the view also can be easily related to the program. For example, a unique combination of communication patterns can be used as an anchor in a history, and this then becomes a reference point for other events that may be created.

The computation graph of a parallel program represents the dependencies between various parts of the computation. By working out these dependencies, one can extract

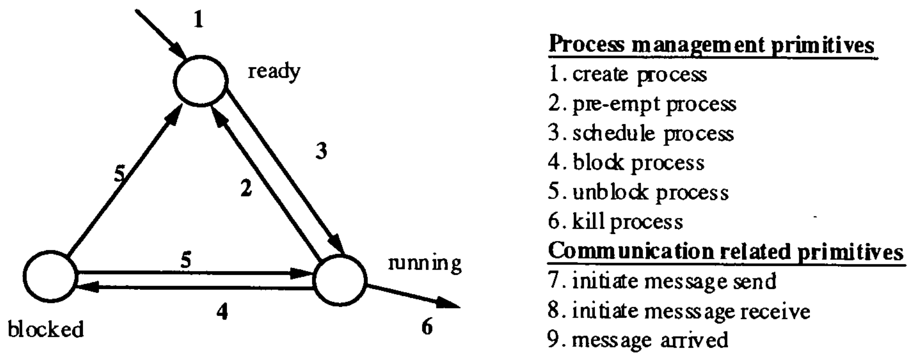
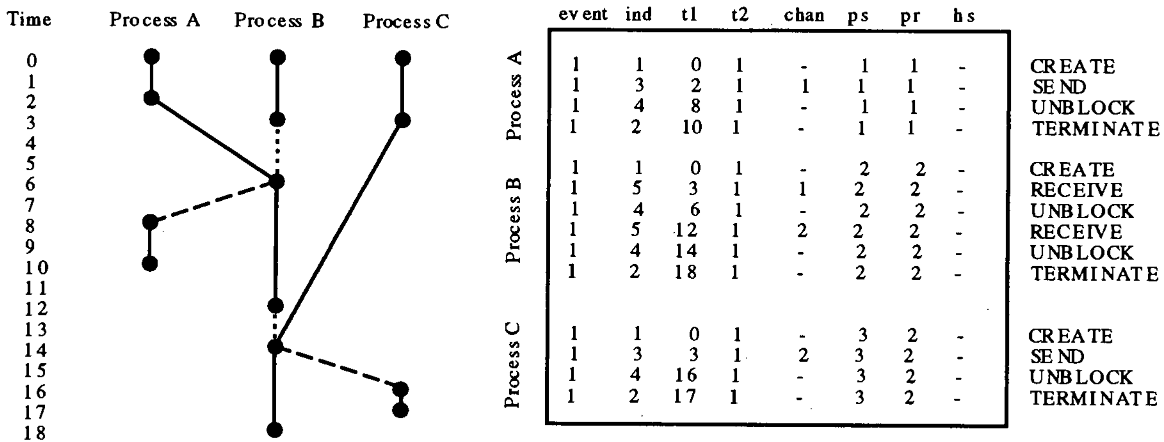


Figure 1. Scheduling graph and primitive event types



	event	ind	t1	t2	chan	ps	pr	hs	
Process A	1	1	0	1	-	1	1	-	CREATE
	1	3	2	1	1	1	1	-	SEND
	1	4	8	1	-	1	1	-	UNBLOCK
	1	2	10	1	-	1	1	-	TERMINATE
Process B	1	1	0	1	-	2	2	-	CREATE
	1	5	3	1	1	2	2	-	RECEIVE
	1	4	6	1	-	2	2	-	UNBLOCK
	1	5	12	1	2	2	2	-	RECEIVE
	1	4	14	1	-	2	2	-	UNBLOCK
	1	2	18	1	-	2	2	-	TERMINATE
Process C	1	1	0	1	-	3	2	-	CREATE
	1	3	3	1	2	3	2	-	SEND
	1	4	16	1	-	3	2	-	UNBLOCK
	1	2	17	1	-	3	2	-	TERMINATE

Figure 2. A trace and its computation graph

valuable information from this graph. For example, the critical path of a computation can help the user to find those parts of a program that slow down the computation [5]. The computation graph is an abstraction of primitive events that correspond to the vertices of this graph. There is an arc between every pair of events that are adjacent within the same process boundary (Figure 2).

There is also an arc between two events that are the sending and receiving ends of a message transmission. Since a primitive event is instantaneous, there is no cost attached to it. But there is a cost for an arc between two events. The cost is the absolute difference between the time-stamps of those events that enclose the arc. The events on both ends of an arc can also be used for identifying the type of its cost. For example, a "send message" and "unblock process" pair makes an arc that represents the cost of blocking a process.

2.1. An instrumentation for POSIE Testbed

The Posie testbed consists of a distributed memory parallel system for running the programs, monitoring hardware for collecting performance data from the system, and a monitor board for the real-time analysis of performance data. The processing part of the testbed consists of processor boards which communicate through a bus structured local area network called Centrenet. Because of the bus structure, there is a direct connection between any processor pair. The aim is not to investigate system design issues such as topology of processors or message routing techniques, which is why this simple connection structure has been chosen to make message passing easy. Besides making message routing easy, there is no need for other hardware topologies because we are investigating software which is insulated from the actual message routing through the network. It is also possible to emulate different topologies by adding a software layer between the bus driving software and higher software layers.

While a distributed program is running on the processors a network of instrumentation interfaces monitors the whole system. These interfaces are connected to another bus which is separate from Centrenet, and used only for monitoring purposes. The reason for separating the monitor bus from Centrenet is so as not to disturb the computation and the message traffic. In addition to this, the monitoring bus is not an ordinary bus, but it has an additional dedicated function which is time-stamping. Time-stamping is a very important aspect in terms of ordering events which are happening independently on different processors. Of course, time-stamping is also used for calculating statistics such as CPU utilization, completion time, frequencies of different events etc. Timing is supported by a global clock signal on the monitoring bus.

Performance measurement is done by the monitor using the performance data collected by the monitoring hardware. These data are in the form of individual records representing the state changes, called "events". For example, sending or receiving a message, waiting for a message, or scheduling a process to a processor can be an event. Events are completely software dependent, and they can be created where they are needed. The creation of an event is done by software instrumentation which causes the event record to be passed to the monitoring hardware. The software instrumentation is a piece of code which writes some event information to the monitoring hardware as if it were a memory location. Because simple instructions can be used for creating event records, and the monitoring hardware handles the event records, the software instrumentation overhead is minimum. This approach is a hybrid between pure hardware and pure software instrumentation.

The disadvantage of a pure hardware approach is the inflexibility and the high cost. Because in a pure hardware approach, some kind of pattern-matching unit is used for capturing events, very complex hardware mechanisms are needed. It is also very expensive to design a hardware monitor which is capable of capturing variable patterns. On the other hand this approach does not cause any undesirable intrusion. Software instrumentation is the most intrusive approach but it gives a lot of flexibility to implement events. In this approach, some instrumentation code is inserted into the application program code to create events. While the program is running, the instrumentation code is executed, and monitor software events occur. In order to time-stamp this event, the instrumentation code has to read a hardware or software clock value from somewhere, and add it to the event record. The event records which are created by the instrumentation are stored on

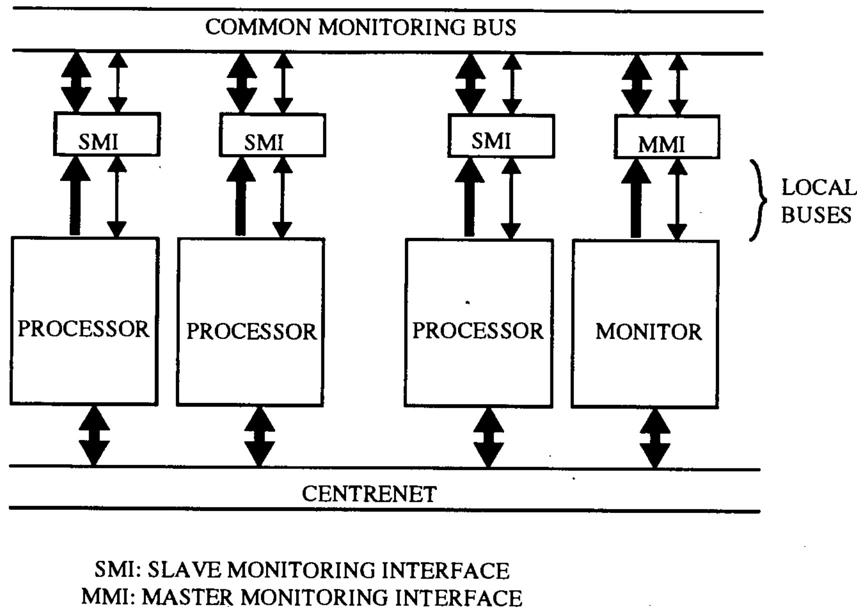


Figure 3. Posie Performance Monitoring Testbed

the processor board and forwarded to an analyzer via the same network as it is used for message communication. As a result of the extra load on the network, performance of the application decreases. The hybrid approach used in POSIE inherits the flexibility of software approach and the non-intrusivity of the hardware approach. The performance monitoring testbed is shown in Figure 3.

Processor boards are connected to Centrenet so that they can communicate with each other, and they are also connected to monitoring interfaces through local busses. Each processor board is monitored by one of these slave monitoring interfaces (SMI). SMIs are connected to the master monitoring interface which controls them. SMIs are responsible for capturing events from processor boards and keeping them until they are read by the master monitoring interface (MMI). The master monitoring interface maintains the global clock signal which is used by its slave interfaces to increment hardware clocks. The master monitoring hardware is connected to the monitor board through another local bus. The monitor board is an ordinary processor board but it is only used for monitoring purposes. Because the analysis programs run on the monitor board, the processing power of the other processors is not used for any task other than application programs. The monitor board is also connected to the local area network, but it does not monitor the network. This can be done by a separate board which monitors messages transmitted via the network. This board only monitors interactions between processes on different processors without causing any change to the operation. The Centrenet connection of the monitor board is only used for loading analysis programs to the board or sending messages to the operating system to take actions such as process migration.

2.2. An instrumentation for the Edinburgh Concurrent Supercomputer

Not all systems are suitable for hardware instrumentation for several reasons. Firstly, financial reasons prevent one from adding hardware instrumentation on top of the existing parallel machine. Secondly, there may be no room for any additional hardware since most commercial parallel computers follow the leading edge of the technology, and sequeze as much hardware as possible into a small space. In this case, there is no option but to implement a software monitoring system. In this section, an implementation of a software monitoring system for the Edinburgh Concurrent Supercomputer, (ECS), a MEIKO Computing Surface, is introduced. This instrumentation is used in parallel program performance tuning experiments. Even though this implementation is intrusive, it provides valuable information for performance tuning.

The ECS monitor consists of small monitoring codes which are linked to the user processes at compile-time, and a central monitor which is a stand-alone process dedicated to monitoring. Each monitoring code is responsible for collecting monitoring data from the process to which the monitor is linked. The monitoring information is held locally until all processes of the application terminate themselves. Then the monitoring information from the individual processes is collected together to produce an execution history of the parallel program. Since collecting event traces from processing elements happens after the program completes its execution, the instrumentation overhead does not include communication costs. The cost of instrumentation is a constant value for each communication activity. Therefore the total cost of instrumentation is a function of communication frequency.

The monitored activities are process creation/termination and message send/receive operations. The instrumentation replaces CS-Tools functions with instrumented versions. These new functions can record every send/receive operation by creating special events for each occasion. Two events are created for each message passing operation: one marks the beginning and the other completion. For example, it is possible to identify how long a send operation takes by looking at the timestamps of the "send" and "unblock" events recorded for the communication.

Since this instrumentation only monitors application programs and not the operating system, OS performance cannot be measured in detail. Nevertheless, by looking at the execution times of applications one can gain an insight into the OS performance. OS activities are also more frequent than application activities, since each application activity may invoke several OS activities. Therefore, monitoring the OS can be more expensive than monitoring an application program. Besides, the OS is not normally accessible to software instrumentation in comparison with applications that can be instrumented easily.

3. Performance Data Filtering

Event histories of parallel programs are valuable information sources for performance analysis but the problem is to extract useful information from massive amounts of low level event traces. The user needs a special mechanism to filter out unwanted details of the history and expose the related information. Filtering can be achieved by replacing some of the lower level event patterns with higher level events. The process of replacing patterns can be repeated until a satisfactory result is achieved. The replacements

can range from very simple pattern replacements to hierarchical replacements (Abstract events) [1]. Every new replacement event inherits some information from the events which are replaced, and has a new meaning. Since the number of events decreases and their meanings become closer to the structure of the parallel program which is being analysed, it becomes easier for the programmer to interpret an event history. For example, some primitive events can be replaced by new events that represent communication and computation intervals. Then, these new events can also be replaced by another new event that represents, for example, stages of each process. This resulting view of the program history can be used for investigating the parallelism of these stages. If this view is found to be inadequate for tuning, the programmer can go further and find the stages which do have significant effects on performance. For example, this can be achieved by investigating the correlation between the critical path of the history and the events that represent parallel and non-parallel stages of the program.

The event abstraction mechanism can be used for improving program execution visualisation, searching patterns in the execution histories, or defining metrics.

3.1. The front-end for the event abstraction mechanism

The tool for the event abstraction mechanism consists of three main parts: a front-end for entering the event pattern definitions, an abstraction engine which carries out the pattern matching, filtering and aggregating, and an output unit which formats and transfers the data to a graph plotter. In this section, front-end options for the event abstraction mechanism are discussed. Since the goal of the event abstraction mechanism is to make detailed performance analysis data available to a user, and to speed-up the performance tuning task, the tool should provide fast and efficient facilities to the user in analysing event traces. Two front-ends have been implemented for entering event pattern definitions. One of them is graphical user interface which enables user to define and modify event patterns very quickly, and the other one is a textual interface which is an extension of the C programming language.

3.1.1. Graphical front-end

There are always advantages of using graphical representation for multi-dimensional data. An event trace of a parallel program is one of them. Event traces of distributed memory MIMD programs have two dimensions: space and time. The space dimension comes from the 'Multiple' part of 'MIMD'. The space dimension reflects the multiplicity of either the data or the instructions of MIMD programs. The time dimension reflects the changes in data or in the instruction sequences over time. The graphical front-end of the tool for event abstraction mechanism uses these dimensions for defining event patterns.

In figure 4, the communication pattern of message passing type parallel program is defined using graphical front-end. In this figure, there are three pattern definitions each of which represents the total ordering of a possible communication sequence. In the graphical representation of a pattern definition, each event represented as a rectangle and an event name in it. Then events can be grouped together in a rectangle to define the process or the processor they belong to. For example, in the communication pattern definitions, each definition contains four event rectangles, one new event rectangle, and two process rectangles. The horizontal positions of event rectangles represent the temporal relations.

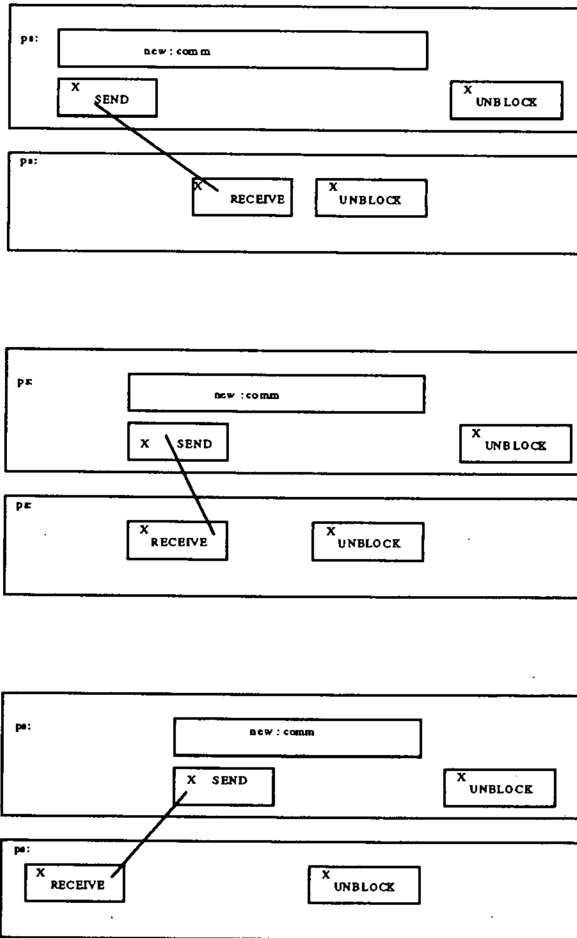


Figure 4. Possible orderings of a communication pattern using graphical interface for defining event patterns

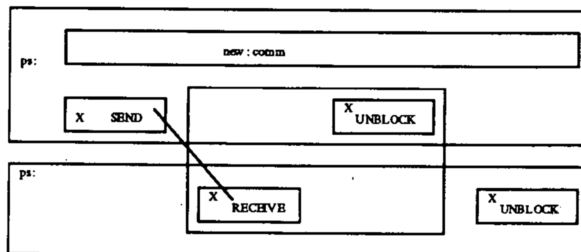


Figure 5. A communication pattern using graphical interface for defining event patterns

The representation of the temporal relations also accommodates a notation to facilitate event patterns which contain partial ordering. For example, the communication pattern has two events which cannot be ordered using logical clock: "SEND" and "RECEIVE". In figure 5, the notation for presenting partially ordered event patterns is given: The "RECEIVE" and the sender process' "UNBLOCK" events are encapsulated by a rectangle. This isolates any temporal relations of the events which are inside the rectangle from the rest of the events which are outside the rectangle. There is one exception: The order between events which belong to the same event stream is preserved. For example, in the communication pattern definition, the "RECEIVE" event is immediately followed by the "UNBLOCK" event even though they are separated by the rectangle which isolates the temporal relations. All of the "happened before" [3] relations are expressed in the definition in figure 5.

The line which connects "SEND" and "RECEIVE" events in figure 5 defines that these two events should have the same channel identification. When the pattern is found in an event trace, the abstraction engine carries out filtering operations. The 'X' sign in an event rectangle tells the abstraction engine to filter the event. In the example, four of the events in the pattern are filtered. Finally, a new aggregate event(s) is created by the abstraction engine. In the example, there is one aggregate event definition (i.e. "comm"). This new aggregate event inherits its first time-stamp from the "SEND" event (of the process which is defined at the top part of the pattern), and the second time-stamp from the "UNBLOCK" event (of the process which is defined at the bottom part of the pattern).

The graphical syntax which is used for defining event patterns is also suitable for interactive front-end. For example, the user can highlight events from an existing gantt chart display by choosing them with a pointing device such as mouse. Then, by drawing lines (channel identification) and rectangles (process, processor boxes) in this view, the user can define the patterns very quickly.

3.1.2. Textual front-end

It is not always desirable to have an interactive graphical front-end. In some situations, the user may require more powerful facilities than graphical front-end can provide, and want to skip some of the steps, such as selecting events, modifying pictures, etc. The library abstractions fall into this category. They have to be implemented once and efficiently since they are used frequently. The textual front-ends also do not demand as much system resources as a graphical front-end does (e.g. graphical terminal, window manager).

The textual front-end was defined as an extension of C Programming Language. Using very simple facilities of this programming language, the textual front-end has been implemented easily.

Event patterns are defined in a declarative fashion. Each event in a pattern is declared with the conditions which has to be held when the pattern is searched in an event trace.

Each line of the event pattern definitions is a C function call which operates on the event trace. There are four functions : `eventfind()`, `nextevent()`, `create_event()` and `filter_event()`. These functions are placed by the C Preprocessor in an appropriate place in the program which searches the event pattern in the event trace. The parameters

```
#define PATTERN1_EVENT1 eventfind(&e1,"PRIMITIVE",SEND,0,4000,\
      0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,name("distributer"))
```

Figure 6. A line from an event pattern definition

of each function call sets the conditions which the event has to satisfy.

In figure 6, a line from an example event pattern definition is given. This line reads `''find a SEND event of the process 'distributer' where the event is before clock tick 4000''`.

The pattern definition of the communication event is given in figure 7. This event abstraction can be expressed in structured English as shown in figure 8. Every event pattern which matches with this definition is replaced with a new event called "comm". This new event represents the transfer time of a message from the sender process to the receiver process.

4. A Novel Approach to Performance Visualisation

In this section, a novel approach to performance visualisation is presented. This approach provides a flexible performance visualisation environment by separating the semantic knowledge of the analysis [6], [4] from the visualisation tool. The semantic knowledge of monitoring information can be defined or modified by the user through facilities that are provided in the visualisation tool. A general purpose plotting facility can be used for displaying performance data in different graphical formats. Interpretation of each performance view depends of semantic knowledge which is assigned to the performance data. We have developed a prototype tool to test this approach.

The first step in visualising trace information is to use a Gantt chart view. This initial view is the finest grain information about a parallel program, and is usually not very useful for tuning purposes since it contains large amount of unwanted information. It is also difficult to visualize a long interval of an execution, or a large program (i.e. a program with many processes).

The next step in visualizing trace information is to enhance the granularity by creating new events with longer durations so that one can zoom out and browse larger intervals of the traces. Since the user defines the new events, the semantic knowledge of these events is familiar to him or her. The user can also filter unwanted events to achieve less crowded views of the tracing information.

Gantt chart displays are not suitable for displaying statistical information and they are restricted to relatively small intervals of an event history. Statistical data can be visualised by feeding filtered event traces to a general purpose graph plotter. Since new event definition mechanism is a means of selecting information, and the user can visualise this selected information by transferring it to a graph plotter, the user can visualise different aspects of a program in many different forms of visual representation. The

```
#define PATTERN1_EVENT1 eventfind(&e1,"PRIMITIVE",SEND,0,INFINITY,\
      0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN1_EVENT2 nextevent(&e1,&e2,"PRIMITIVE",UNBLOCK,\
      0,INFINITY,0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN1_EVENT3 eventfind(&e3,"PRIMITIVE",RECEIVE,\
      0,e2.t1,0,INFINITY,e1.chan,NULL_HS,NULL_PR,NULL_PS)
#define PATTERN1_EVENT4 nextevent(&e3,&e4,"PRIMITIVE",UNBLOCK,\
      e1.t2,INFINITY,0,INFINITY,NULL_CHAN,NULL_HS,NULL_PR,NULL_PS)

#define PATTERN1_CREATE1   create_event(&new_event,"comm",0,\
      e1.t1,e4.t2,e1.chan,e1.hs,e1.pr,e1.ps)

#define PATTERN1_FILTER1 filter_event(&e1)
#define PATTERN1_FILTER2 filter_event(&e2)
#define PATTERN1_FILTER3 filter_event(&e3)
#define PATTERN1_FILTER4 filter_event(&e4)

eventtype e1,e2,e3,e4,new_event;
compose()
{
while(pattern1());
pr_parallelism(name("comm"));
}
```

Figure 7. Textual definition of a communication pattern

```
find consecutive SEND and UNBLOCK events from process x
find consecutive RECEIVE and UNBLOCK events from process y
   where
      the channel identifications of SEND and RECEIVE events are equal
            and
      UNBLOCK (process y) event is after SEND event
            and
      RECEIVE event is before UNBLOCK (process x)
create an event named ''comm'' (of process x)
   which starts at the same time with SEND event of process x
           and
         ends at the same time with UNBLOCK event of process y
filter SEND, RECEIVE and both UNBLOCK events
```

Figure 8. Structured English definition of a communication pattern

overall structure of the visualisation environment is depicted in figure 9.

This approach requires some work from the users if they want to make detailed queries about their program. However, the advantage of the approach is that the tool has no knowledge about the meanings of the events. The semantic knowledge is defined by the user. When a new event is created a meaning is assigned to the new event. Generic event types, such as the critical path [5], and the events which represent performance metrics, can be defined once and be kept in a library. With this approach, we can provide basic performance visualisation techniques to the novice users. They do not need to define anything but use existing library definitions. Some of the users can customise the libraries to satisfy the changing requirements.

Since a user can formulate queries about a program, (s)he can repeat the same query on several versions of the program, or on the same program under different constraints to identify possible bottle-necks. Today's parallel programs are mainly scientific, and are quiet small compared with prospective parallel programs in industry. Using query facilities can be a comparatively big task for a user who is writing small parallel programs but this task can be proportionally much less expensive to an industrial user, or the others, who are writing large and complex parallel programs.

The language contains special features to handle multiple trace histories. By using these, users can correlate histories that belong to the different executions and/or different versions of the same program. The user can conduct tuning or scale-up experiments, and use the performance visualisation tool to analyse and visualise the results.

4.1. Performance displays

The performance displays are the crucial part of the performance visualisation, and whatever the type of the display it can give insight into the program about the behaviour of a parallel program. In this work, the basic graphical display types are chosen to visualise

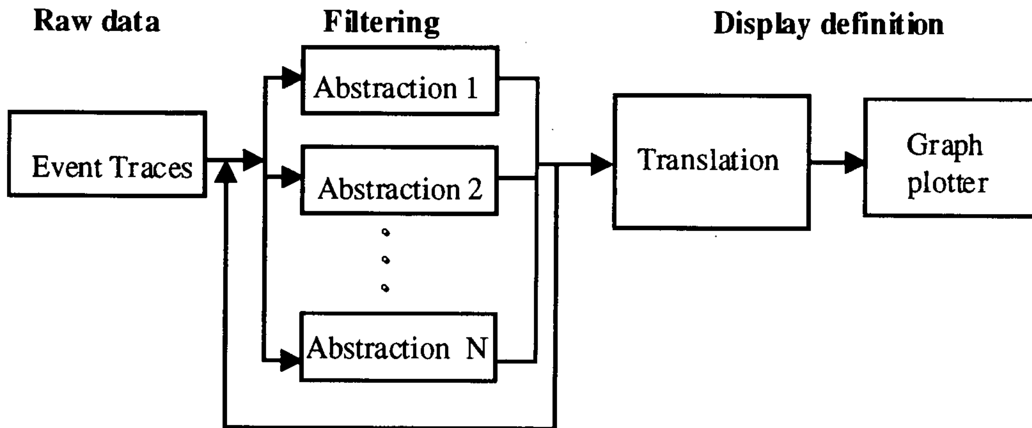


Figure 9. The conceptual structure of the performance visualisation environment

Table 1
Performance visualisation displays

Display Type	Display Dimensions	Cons & Pros
Gantt Chart	Space-Time	+easy to interpret +detailed information -restricted to small intervals -not suitable for visualising large number of processes
3-D display	Space-Time-Metric	+suitable for visualising large programs +many forms of display available -not suitable for detailed behaviour visualisation
Parallelism display	Time-Metric	+provides summary type of information which is not obvious in Gantt chart and 3-D displays
Profile display	Metric-Metric	+summarises resource usage +suitable for any size program -no direct use of locating performance problems

the performance. These display types and the large number of abstract event definitions are combined together to make the performance visualisation tool extensible.

In table 1, a summary of performance displays which are presented in this work is given.

Each of the performance displays can visualise various forms of performance data from a different angle of view. Performance visualisation tools usually accommodate several of these graphical forms and their derivatives because no single display type, as its own, is enough to visualise every aspect of parallel program performance data.

4.1.1. Gantt chart displays

Gantt chart is one of the oldest graphical forms which was used by H.L. Gantt [2] as performance display in industrial management. This display type can visualise detailed performance information in two dimensions: space (processes or processors in performance visualisation case) and time. The activities which occur in a parallel program can be

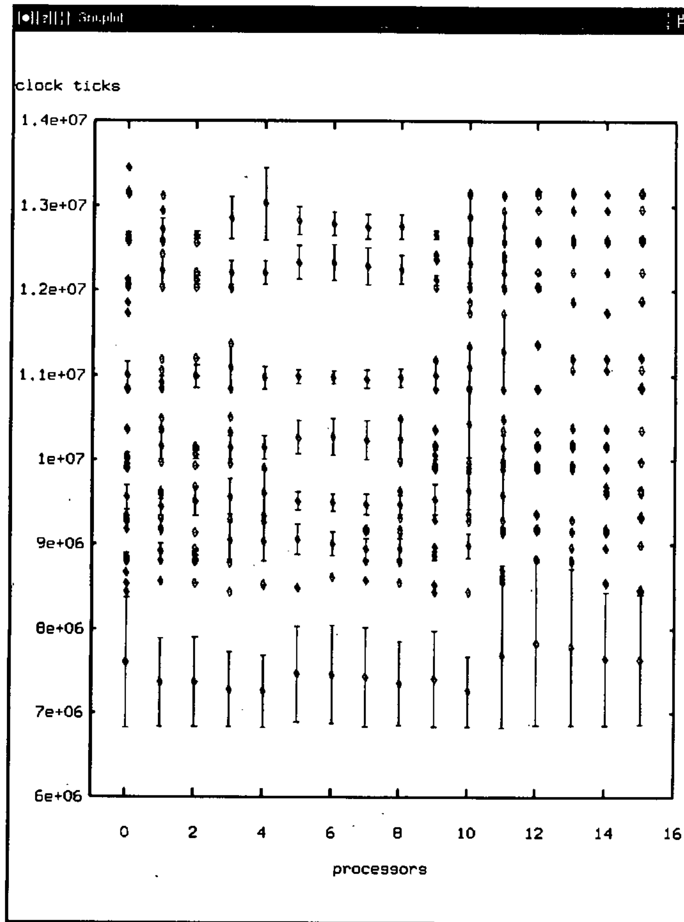


Figure 10. A prototype Gantt chart display using GnuPlot

represented on the chart in several ways: rectangles, icons, lines or textual information, but there is one thing in common. The positions and the lengths of the representations on the chart are proportional to their real life timescales. For performance visualisations the activities to display are the events which are occurred during an execution of a parallel program.

The distribution of events in an event history may present a problem with a Gantt chart. Here, the problem is that there are groups of events, and between the groups, there are considerably long time gaps. In this case, the user cannot fit a sensible interval of history onto the chart: He or she tries either to visualise the interval as it is, which is resulted with overlapping icons, or to zoom in to a smaller interval in which he/she cannot observe activities in a wider context. Without any help, the only way for the user to tackle this problem is to continuously zoom in and out to the chart until he or she explores the details of the event history.

Since the user can combine as many event as he or she wish, the interval size that can be displayed on a Gantt chart can vary from one application to another. In figure 10, for example, basic computational intervals of a parallel program is visualised on a Gantt

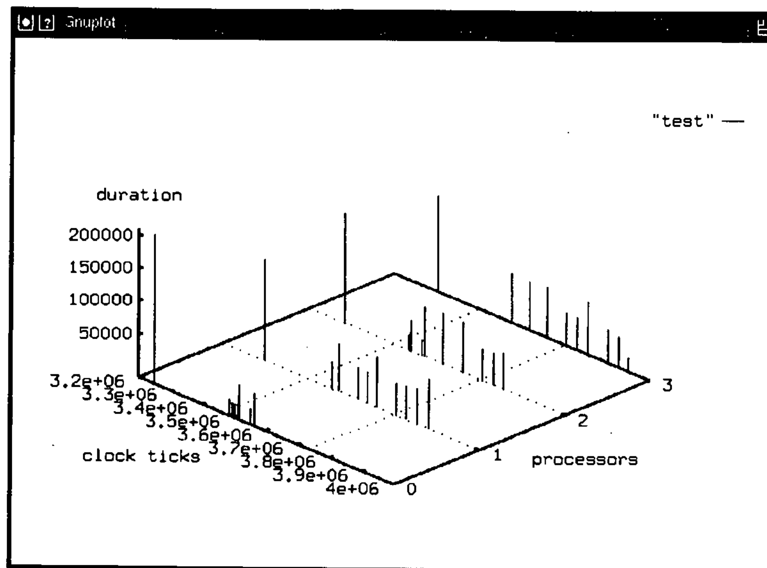


Figure 11. Visualisation of the stages of a program

chart.

4.1.2. 3-D displays

3-D displays which are discussed in this paper are an enhanced form of the Gantt chart display. By adding another dimension to Gantt chart, 3-D displays can be achieved. The third dimension represents the duration of the abstract events which are visualised. This dimension is called metric dimension because it provides a common basis to compare the durations of abstract events. This display type can be seen as a Gantt chart in which events are visualised perpendicular to the X-Y plane. Since one end of each event is aligned with the X-Y plane, the other end serves as a reference point to compare its duration with the durations of the other events. A peak in this kind of graph indicate an exceptionally long activity which might be an indication of a bottle-neck.

In figure 11, a very small data set is visualised as a 3-D graph. There are several different styles of this display type. Depending on the nature of the data that is visualised, the user can switch between these different styles of the same display type. Each of these display styles can be useful in different situations.

The following styles of 3-D graphs are available and can be used for visualisation:

- Surface
- Line
- Scattered (dots in GnuPlot)
- Bars (impulses in GnuPlot)
- Surface contours

- Base contours

3-D surface graph can be useful when large number of processors to be visualised (eg. figure 17). Since small variation between neighbouring data points can make the surface jagged and therefore, difficult to see the general trends of the data, it is possible to improve this type of display by clipping the hidden surfaces from the graph. It is also possible to apply surface smoothing techniques to the surface graphs. This will hide small variations of data and emphasize the important peaks on the graph. By doing this, we loose some of the details but it enables visualisation of large data sets, especially from massively parallel systems.

Line and bar styles can be useful when the user wants to emphasize the variation between data points which are originated from the same processor or the same process. Even though these type of variations in data which usually reflect the behaviour of individual processes are the emphasis, line and bar styles are still suitable to present the data across different processes.

3-D scattered display type is suitable for visualising the distribution of the abstract events. This type of display can be used for monitoring activities throughout very large event histories since the dots which are used for representing data points do not crowd the display. In this type of the display, the duration of the events cannot be seen very easily since the points are floating in a 3-D space of the graph. The whole point of using scattered is to see groups of data points. These groupings can be on a single line of a process. This can manifest a behaviour related to a single process. The groupings can also be in an area or in a volume. This may, for example, represent activities of a group of processes which interact with each other.

3-D Contour graphs are similar to surface graphs. The contour graphs show less detail of the data points but the overall picture of the graph can be more visible since small variations between neighbouring points are filtered. By changing the number of contours which can drawn, the user can control the level of detail that is to be visualised. This technique can be considered as a visual filtering technique since it serves as a device to reduce the amount of unimportant and/or unwanted information.

It is also possible to visualise more than one abstraction on the same graph (i.e. superimposing two graphs of different abstraction levels). These types of mixture graphs may become overcrowded, and the display cannot be interpreted easily. In these cases, visual filtering techniques such as using contours only, surface smoothing are useful. It is also possible to use a different style for each abstraction in a single view.

Similar to Gantt chart displays, 3-D displays also suffer from overlapping, or nearly overlapping data. The data points which are too close each other may look like there is only one data point. Since it is not always easy to zoom in and check if there are any overlapping data points, an alternate 3-D display format was invented to unfold all overlapping data points. The dimensions of the 3-D display is translated from "space,time,metric" to "space, sequence number, metric". Every event in an event stream which belong to a process is given a sequence number starting from the beginning of the stream. Then, these events are visualised using sequence numbers instead of their time-stamps. Since the distance between any two consecutive events is the same, there is no risk of displaying more than one data point as one. In this display, the order between events which originate from different processes is not preserved but the order of events of the same process is

untouched.

Another potential use for 3-D displays whose y-axis is made up from sequence numbers is history comparison. Assuming that the same event abstraction can be achieved for two different event histories, visual comparison of two histories can be achieved. For example, "phase" event which is explained in the section 4.2 is suitable for this purpose. We can compare the template program with an application which uses the template, and monitor the durations of the "phase"s. The same comparison can be carried out on different histories of the application to observe, for example, the effects of the message sizes on the phase durations.

4.1.3. Parallelism displays

Parallelism display is a two dimensional graph type which shows the distribution of parallel activities over time. Unlike the parallelism display of the other performance visualisation tools, the parallelism display which is presented in this paper is not restricted to the parallelism of raw computation. Traditionally parallelism displays show the degree of parallel intervals where processors compute.

The standard parallelism display provides a good metric which explains how the computational resources are used. It does not necessarily mean that keeping processors busy is an indication of a fast parallel program. However, we can conclude from low parallelism that the program is not performing well and therefore, it cannot be fast enough to gain advantage over its sequential implementation. A parallel program may contain duplicate and housekeeping type computations such coding, decoding and interpreting messages, which do not exist in the equivalent sequential implementation. This extra codes can be identified as unuseful computations, and the rest as useful.

Apart from useful/unuseful computation argument, conventional parallelism displays present only very low level information for performance tuning. They just show how much computational resource is used. They do not provide any information about how these resources are used.

Parallelism displays which are presented in this paper are suitable for visualising any kind of resource usage. The definition of the "resource" is not restricted to system resources such as processors and channels. The software resources which use system resources are also considered as abstract resources to be exploited efficiently in order to achieve fast parallel program.

The software resources vary from a process to a small code segment in a process code. We can represent software resources as computational units such as program phases, a step of a loop, turnaround time of a server process, broadcasting time. The list is extensively large since every application has software resources which is specific to its own. Visualising these abstract software resources may reveal valuable information for performance tuning.

In figure 12 four different parallelism displays of a test program (see section 4.2 for the details of the example program) are shown. All of these four displays visualise the same interval of the same program but each display visualise a different aspect of the program. The first from the top is the standard parallelism display which shows how computational resources are used. Since this program is a template, it does not use many computational resources but implements a global data gathering operations using communication related resources.

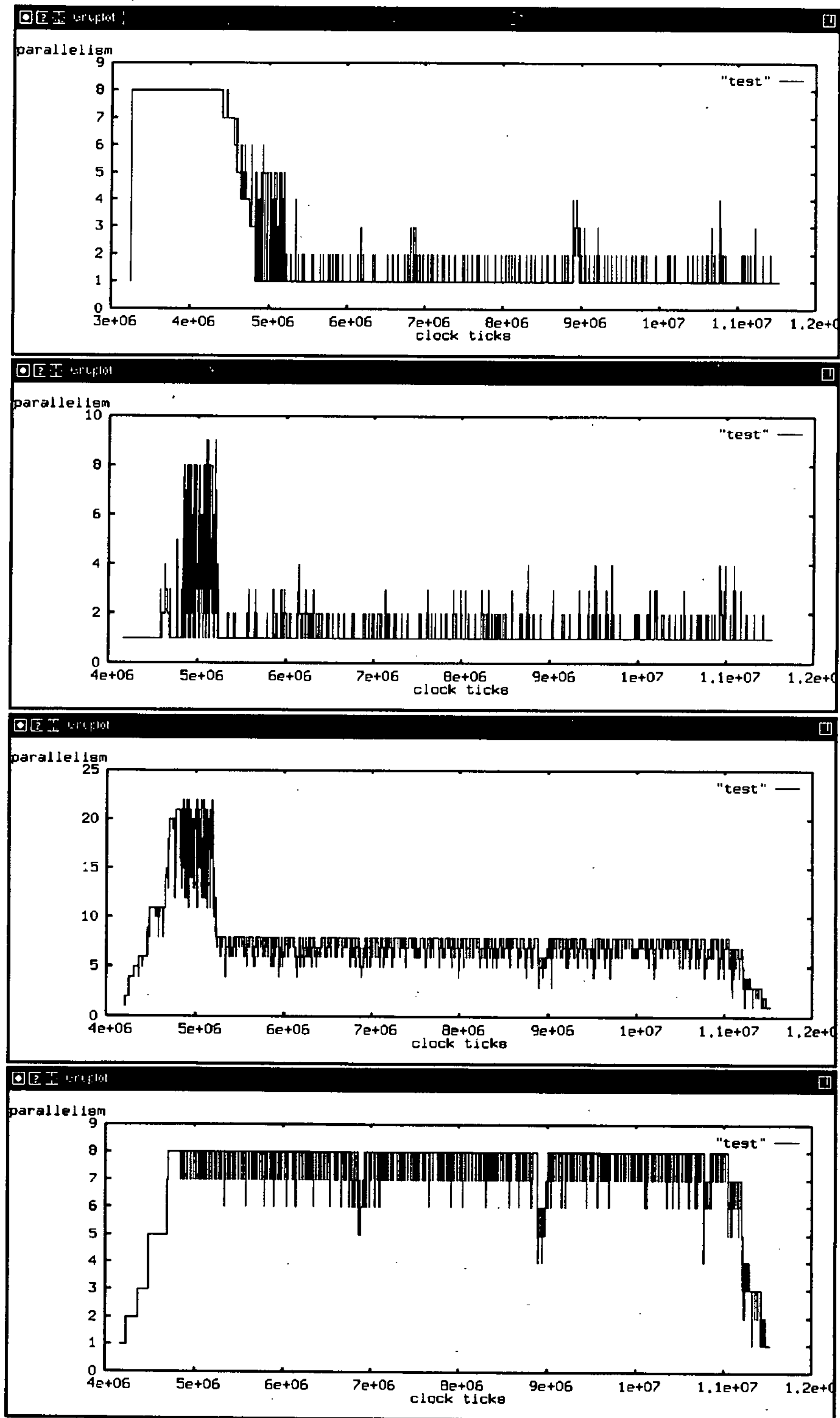


Figure 12. Parallelism views of a program

The second display in figure 12 shows how many processes block on sending messages. Similarly, the third display shows how many processes block on receiving messages. Even though these two displays do not directly show how much communication resources are used, they show the effects of communication on the execution of processes.

The last display in figure 12 visualises how many "broadcast and gather" phases are carried out in parallel at a given time. All these displays are achieved using simple abstractions of primitive events. The first display shows the intervals which are marked by event pairs. These pairs are: ["CREATE", "SEND"], ["CREATE", "RECEIVE"], ["UNBLOCK", "SEND"],["UNBLOCK", "RECEIVE"] and ["UNBLOCK", "TERMINATE"].

The event pair for the second display is ["SEND","UNBLOCK"], and the event pair for the third display is ["RECEIVE","UNBLOCK"]. These simple abstractions are generic abstractions which can be applied to any program without any knowledge of the application, and meaningful performance displays can be achieved. The fourth display differs from the previous three in this respect. The abstraction for this last display only produces meaningful results for the program which is explained in section 4.2.

Since it is possible to map more than one process on a processor, two or more processes will be running on the same process. The scheduler timeslice these processes without processes functionally being affected from this. The user also does not care about the scheduler since system takes care of the scheduling. The only difference a user notices is the execution time. The execution time of a code segment of a process may vary depending on the number of ready to run processes on a processor. Since execution time is the prime concern for the user, (s)he wants to know how many processes are running at a given time. This kind of parallelism is called virtual parallelism. Virtual parallelism is equal to or higher from real parallelism.

Two different versions of parallelism displays are provided: Virtual and real parallelism displays. In the real parallelism displays, the abstract events (of the same processor) which overlap are handled as a single event which is union of the overlapping events. In the virtual parallelism view, every event is handled separately. As a result of this, for example, two intersecting events from the same processes contribute parallelism 1 unit for the non-overlapping and 2 units for the overlapping interval.

The reason behind mapping more than one process on a processor is to take advantage of parallel slackness [7]. When a process blocks for a communication related operation, the processor can be assigned to another process. In the optimum case, only one of the processes is ready to run, and the other processes are waiting for communications. When the running process blocks for a communication and another process completes its communication and and starts running. In this extreme case, neither processor is idle nor any two processes are ready to run at the same time. Since the executions of processes never overlap in this extreme case, the real and the virtual parallelism views of the program will be identical.

The degree of parallel slackness can be determined from the differences between real and virtual parallelism displays. The closer the similarity between the virtual parallelism view and the real parallelism view, the better exploitation of parallel slackness is achieved.

The comparison between the real and the virtual parallelism views can be done for the basic computational blocks but there is a potential for achieving the same results for more abstract levels of a history. For example, we can consider the same comparison for the

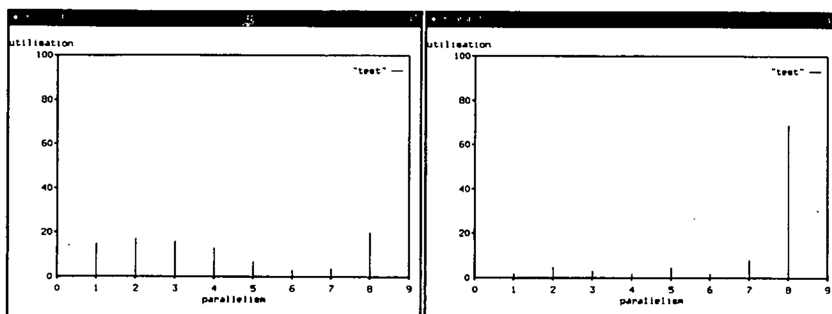


Figure 13. Profile view of "running" and "phase" events

stages of processes.

4.1.4. Utilisation displays

The utilisation display is a summarised version of the parallelism display. This display informs the user about the percentage of the execution time which is spent for each parallelism level.

This display gives information about how a parallel program performs, and does not give any information where and/or when the performance is poor or good.

In figure 13, two utilisation display examples are given. Both of the displays visualise the "broadcast and gather" example (see section 4.2). The first display shows how computational resources (processors) are used in the program. For 8 node topology, the program used computational resources very poorly. It spends its time equally (nearly) using 1,2,3,4 and 8 processors. By looking at the parallelism display (see the first display figure 12), we can also say that 8 processors are mainly used during the initialisation stage of the program.

In figure 13, the second display visualise the "Broadcast and Gather" phases of the same example program. In each phase, a request message is broadcasted to each node, and replies are gathered. The resources which are used by each phase are a network of processes which implements the phases. These processes consume a complex combination of computational and communication resources. When two or more requests are made at the same time, the request will use the same process network and therefore there will be competition between the requests. Here, the utilisation display shows how is the distribution of the multiple requests. For example, from the display we can tell that %70 of the execution time eight requests are processed in parallel. This might be the bad news since multiple phases have to share the same resources.

4.2. A performance visualisation exemplar

The example program which is presented in this section is a skeletal program which consists of a network of processes that implement broadcasting and data gathering mechanisms. Broadcasting node sends the broadcast message sequentially to all nearest neighbours which will forward the message to their neighbours. The topology of the network is a binary n-cube. By exploiting properties of binary n-cubes, it is ensured that every node

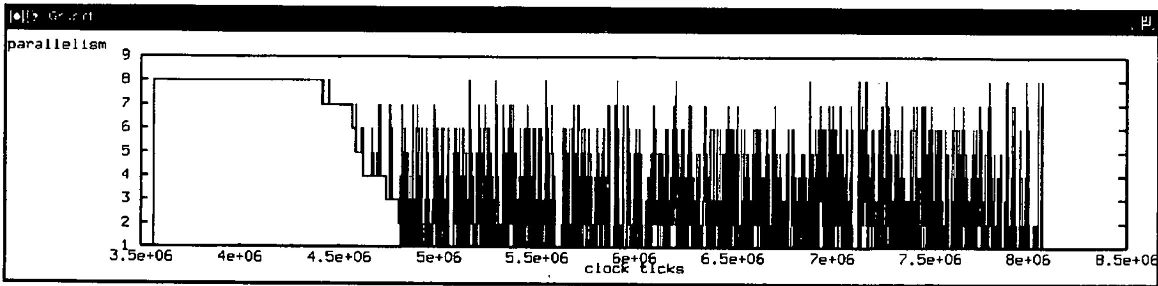


Figure 14. A parallelism view of 8 node "Broadcast and Gather" program

receives the same message only once. Once the request message reaches to a leaf node, leaf nodes' local answer to the request is returned to the node that forwarded the request. A forwarding node waits for answers from the nodes to which it is forwarded the original requests. When the forwarding node gets replies from its all recipients, it combines its answer and all other answers into one message, and returns this combined message to the node that forwarded the original request. This is called "broadcast and gather" operation which collects and combines information from all nodes in the system.

In the implementation, three different processes are used. The first one is the "broadcaster" process which act like a client to distributed data server. The second process is the "receiver" which handles the incoming request/reply messages to a node (processor), and redirects them to the "distributer" process which is responsible for distributing request/reply messages to the nearest neighbours or to the local client ("broadcaster") if the message is a reply to the request which was previously made by the local client. Each broadcaster can make only one request at a time but broadcasters from different nodes can make requests independently from each other. The infrastructure for handling requests (i.e. the network of "receiver" and "distributer" processes) can handle system-wide multiple requests.

The mapping of processes on to the processes is fixed; each processor have one "receiver" and "distributer" pair. The "broadcaster" process is also mapped to the processors from where global operations are going to be required.

In the experiments, 8, 16, 32 and 64 node implementations are monitored and visualised. Figure 14 shows the parallelism display of computational resources. As it can be seen from this display, the level of parallelism varies drastically throughout the program. It is not possible to say how program behaves but this display gives a clear indication of poor computational resource usage.

In Figure 15, four different parallelism views of the example program are given. Each of the displays visualises how program behaves in terms of "broadcast and gather" operations. The main target in visualising this program is to see how these operations are performed, how long they take, and what the overall performance is.

The "broadcast and gather" abstraction represents the intervals in which an operation takes place. The length of the abstract event is relative to the duration of the operation to which it corresponds. The abstraction was achieved by identifying the beginning of the request ("broadcaster" sends a request message) and the completion of the operation

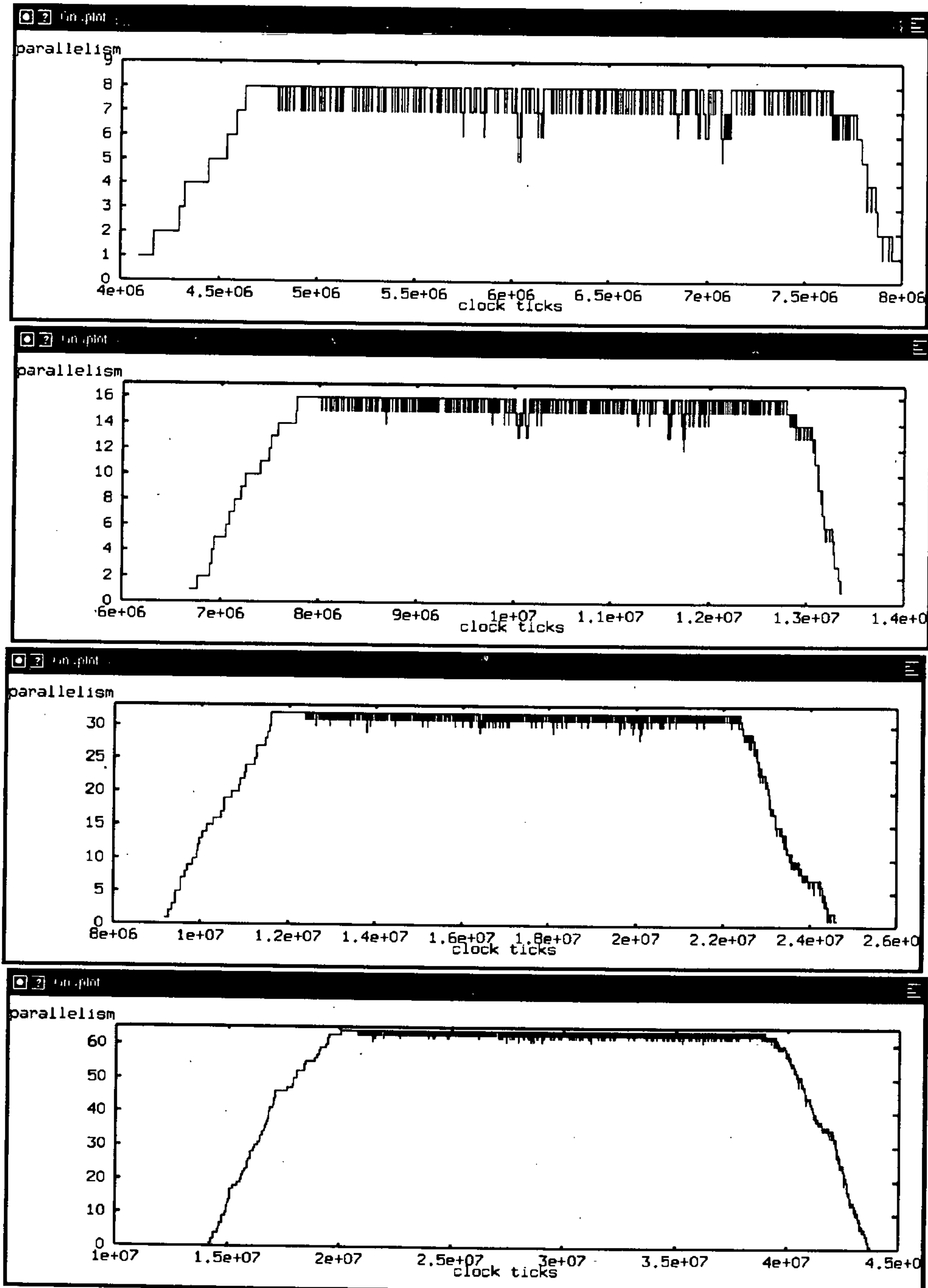
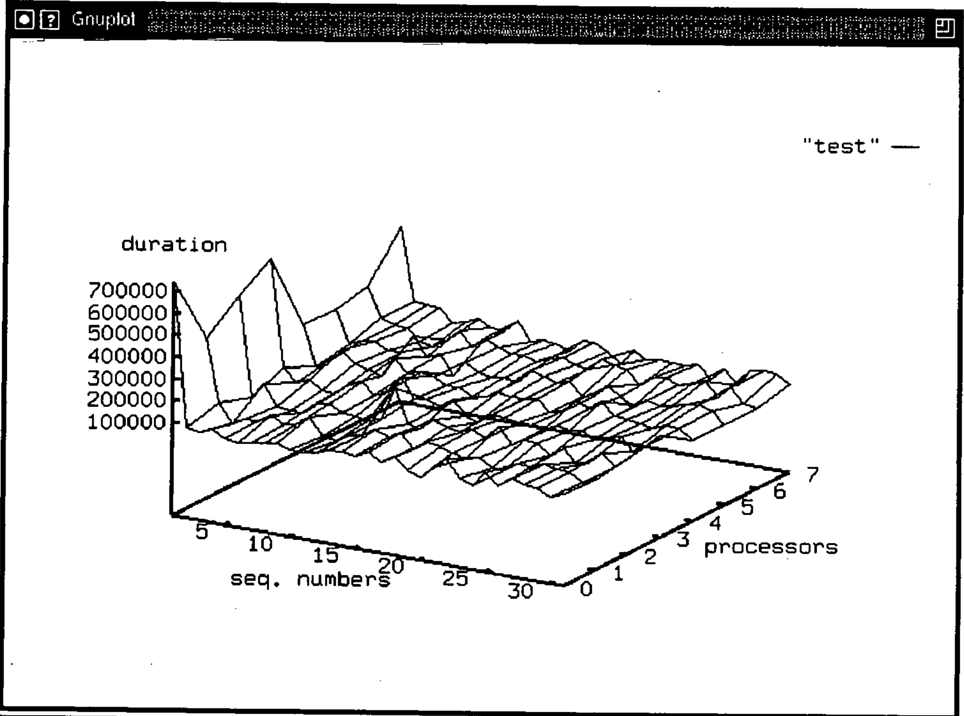


Figure 15. Parallelism views of "phase"s of 8, 16, 32 and 64 node "Broadcast and Gather" program



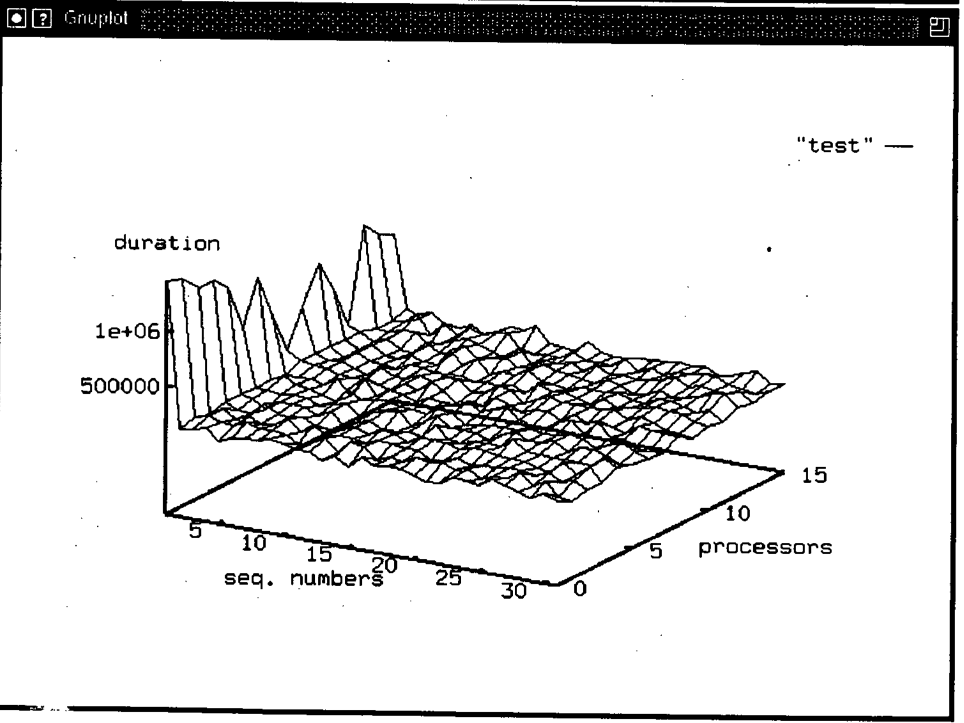


Figure 16. Visualisation of the stages (8 and 16 nodes)

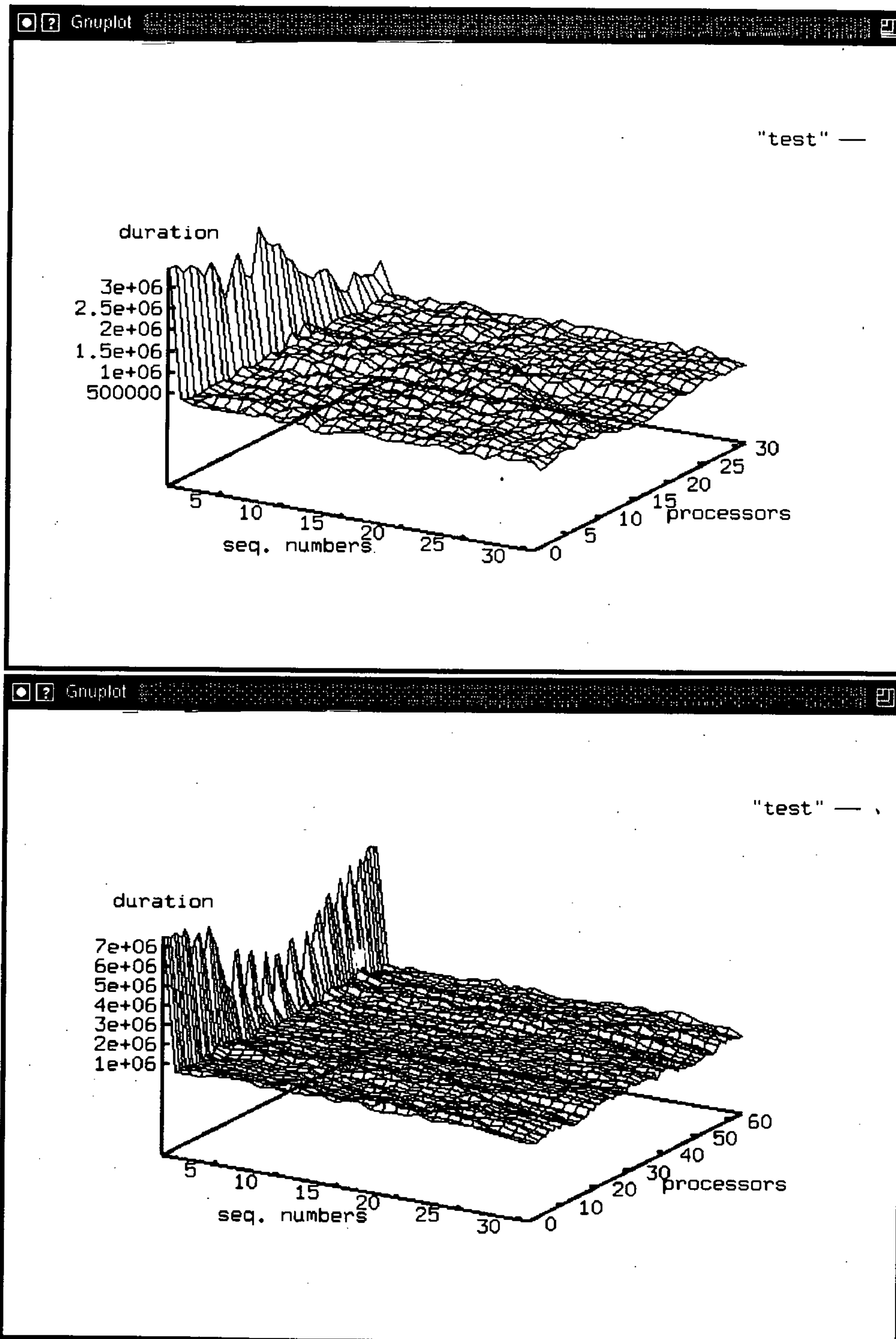


Figure 17. Visualisation of the stages (32 and 64 nodes)

("broadcaster" receives an answer).

In the example programs, each node carries out 30 operations, and then it terminates. Each node also prints the results after 10 operations. The effects of these print operations can be easily recognised in the first display in Figure 15. After one third and two third of the total execution time, the level of parallelism decreases temporarily where the print operations are carried out. The same effects are less visible in the last display in Figure 15 since the number of the processes are high, the individual processes tends to drift. As a result of this, the processes reach to the printing points in different times, and fall in the parallelism is spread over a large interval.

Figure 16 and Figure 17 are the visualisation of the abstraction level whose parallelism displays are shown in Figure 15. In these latest 3-D displays, we can see how long the individual operations last, and easily compare them with each other. As it can be seen from the displays, the first operation of each node takes much longer than the rest of the operations of the same process. Since processes are downloaded to the processors sequentially, some of the processes start running later than the others. This will stop "broadcaster"s from carrying out their operations until all of the processes are running. When all of the processes are available, the system settles down, and the "broadcast and gather" operations are performed at their normal speeds.

In this performance visualisation exemplar, mixture of standard and application specific performance displays was used for monitoring a specific activity of a parallel program.

5. Conclusions

The techniques for monitoring parallel programs are well established compared with the techniques for visualising performance of parallel programs. Large amounts of information can be collected from an execution of a parallel program at a small cost. Using general purpose performance displays a user can visualise these data, and gain limited insight into a parallel program but this is not enough for a user who wants to tune the performance of a parallel program. Detailed information is required about the individual activities of a parallel program. In this paper, a technique to extract information form large volumes of event traces was presented. Using this technique a user can achieve standard performance and application specific performance displays. The technique is also suitable for defining performance metrics, and for relating performance metrics to the application specific information. Another use of application specific visualisation is that performance characteristics of different executions of the same parallel program can be correlated in terms of phases, computational units or any other observable behaviour of a parallel program.

Furthermore, the approach which is presented in this paper is a promising candidate for implementing performance visualisation and analysis tools.

6. References

1 P.C. Bates, and J.C. Wileden, "High-level debugging distributed systems: The behavioural abstraction approach," J. Syst. Soft. 3, Elsevier Science Publishing Co., 1984, 255-264.

2 H.L. Gantt, "Organizing for work," 1919.

3 L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," CACM, Vol. 21, No. 7, July 1978, pp. 558-565.

4 A.D. Malony, D.H. Hammerslag, and D.J. Jablonowski, "Traceview: A Trace Visualisation Tool," IEEE Software, September 1991, pp. 19- 28.

5 B.P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S- S. Lim, and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," IEEE Trans. on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990, pp. 206-217.

6 K.M. Nichols, "Performance Tools," IEEE Software, May 1990, pp. 21-30.

7 L.G. Valiant, "General Purpose Parallel Architectures," Technical Report TR-07-89, Hardward University, Cambridge, April 1989.