A Statistical Investigation of the Factors Influencing the Performance of Parallel Programs, with Application to a Study of Process Migration Strategies

Joseph Phillips

Ph.D.

University of Edinburgh

1994

Abstract

It would be highly desirable for operating systems to take a greater responsibility for process placement and load balancing decisions in parallel machines. This would relieve the programmer of much of the burden associated with fine-tuning an application in order to achieve acceptable performance levels. Before such operating systems can be developed, it is necessary to gain a better understanding of the factors that influence program performance. In particular, it would be useful to be able to identify *classes* of programs which, in a statistical sense, behaved in a similar manner. Then, given an arbitrary program whose class was known, rules and heuristics developed for the program class (in conjunction with program specific information) could be used to make informed placement and load balancing decisions. As a step in this direction, this thesis investigates the application of standard statistical techniques to the performance analysis of particular classes of parallel programs.

Simple CSP-type parallel programs exhibiting loosely synchronous data parallelism are used to illustrate how a common class of programs can be characterised in terms of a relatively small number of parameters representing time-averaged properties. In order to systematically explore parameter space, synthetic programs are used. The execution of these programs is simulated on an accurate performance model of a transputer-based machine. Standard experimental design techniques, such as the analysis of variance, are then applied to develop statistical models relating to the program class. It is shown that useful quantitative predictions can be made for arbitrary class members.

The accuracy of the performance model described above can be improved by taking account of run-time information. The analysis of covariance is a technique which enables this by allowing one to incorporate a number of *covariates* into a model. The covariates investigated in this thesis relate to the dynamic properties

of programs. More specifically, they are a product of the complex interactions which occur at run-time between a program and the underlying machine.

A representative process migration strategy of the type that might be incorporated into an operating system is presented. The covariate-based performance model developed earlier is then used to identify a number of performance measurements which, when optimised, tend to result in improved processor utilisation. A modified migration strategy which makes migration decisions with these optimisations in mind is presented. It is shown that the new strategy can offer significant performance advantages over the original strategy.

Finally, in order to demonstrate the generality of the new process migration strategy, it is tested on a class of dynamically varying programs. Statistical techniques are used to identify the circumstances under which the strategy can offer the greatest performance benefits. The results obtained from the simulation system are validated by showing that they apply to real programs running on a real transputer-based machine.

Acknowledgements

I would like to extend my thanks to my supervisor Rosemary Candlin, for her constant enthusiasm and helpful guidance over the years; to Neil Skilling, for his help in establishing the experimental framework; and to Peter Fisk, for his statistical advice during the early stages of this work. Finally, I would like to thank Caroline, who made this all worthwhile.

This work was funded by a Research Studentship from the Science and Engineering Research Council.

Declaration

I declare that this thesis was composed by myself, and that the work described within is my own except where explicitly stated in the text.

21/5/94

Joseph Phillips

Publications

Some of the material presented in this thesis has already appeared in the following publications:

- J. Phillips and N. Skilling. A modelling environment for studying the performance of parallel programs. In *Proceedings Seventh UK Computer and Telecommunications Workshop*, Edinburgh, July 1991, Springer-Verlag Workshop Series.

- R. Candlin, P. Fisk, J. Phillips and N. Skilling. A statistical approach to predicting the performance of concurrent programs. In *Parallel Computing: From Theory to Sound Practice - Proceedings of the European Workshop on Parallel Computing*, Barcelona, March 1992, IOS Press.

- R. Candlin and J. Phillips. An environment for investigating the effectiveness of process migration strategies on transputer-based machines. In *Proceedings of the World Occam and Transputer User Group Meeting 15: Transputer Systems - Ongoing Research*, Aberdeen, April 1992, IOS Press.

- R. Candlin, P. Fisk, J. Phillips and N. Skilling. Studying the performance properties of concurrent programs by simulation experiments on synthetic programs. In *Proceedings of ACM SIGMETRICS and Performance 1992*, Rhode Island, June 1992, ACM Press.

- R. Candlin and J. Phillips. Statistical modelling as a tool for studying the performance of parallel systems. In *Proceedings of the Leeds Workshop on Abstract Machine Models*, Leeds, April 1993.

• R. Candlin and J. Phillips. A statistical study of the factors that affect the performance of a class of parallel programs on a MIMD computer. In *Proceedings of the International Conference on Decentralized and Distributed Systems*, Palma, September 1993, Elsevier North Holland.

Table of Contents

Chapter 1

Introduction

The von Neumann sequential model of computation has dominated computer architectures for the past 45 years. However, it is becoming ever more difficult to obtain increased performance from conventional machines. One solution to this problem is to use multiple processors in order to exploit the parallelism inherent in many problems; this technique is known as *parallel processing.* Parallel computers are, generally speaking, concerned with the solution of a single problem. This is in contrast to distributed systems, which, while having multiple processors, are usually designed to maximise the throughput of large numbers of relatively independent jobs.

Parallel computers are not new, research into such machines has been taking place for well over 20 years. For example, the ILLIAC IV [7] was constructed in 1968 and the C.mpp [133] in 1971. Historically speaking, parallel computers have tended to be built as research projects, designed to solve specific scientific problems. However, recently these machines have become more widely used in academia, commerce and industry. This is due to advances in technology which have resulted in the cost and size of microprocessors decreasing. Consequently, instead of concentrating on trying to produce faster and larger single processor machines, it has become cost effective to construct parallel computers from many smaller processors.

Parallel computers can be classified in a variety of different ways, the most common classification scheme in use is Flynn's taxonomy [48]. Flynn partitions architectures according to whether single or multiple "streams" of instructions are used, and whether single or multiple "streams" of data are used. Within this scheme, the two dominant classes are known as SIMD and MIMD.

SIMD (Single Instruction Multiple Data) computers are often massively parallel, being constructed of many relatively simple processors called processing elements. These processing elements operate in a lock-step fashion, simultaneously executing the same instructions on different data items. Examples of such machines include the AMT[1] DAP [64], and older models of the Connection Machine [129] from Thinking Machines.

MIMD (Multiple Instruction Multiple Data) computers usually contain a smaller number of processors than SIMD computers, although these processors are generally more powerful. Each processor is capable of executing its own code on its own data. However, often the processors are used in a Single Program Multiple Data (SPMD) mode, where the same program is run on each processor. This differs from the SIMD approach in the fact that the processors do not operate in a lock-step fashion.

MIMD computers can be further divided according to the way processors communicate with one another. If there are a relatively small number of processors, each processor can have access to a shared global memory. Such computers are called *shared memory multiprocessors*, examples include the BBN Butterfly [33] and the Denelcor HEP [43]. Shared memory multiprocessors are reasonably easy to program since well understood operating system techniques such as semaphores can be used to control and synchronise the processors. However, they suffer from

---

[1]Formerly ICL.

one great drawback: as the number of processors is increased, access to the shared memory becomes a bottleneck limiting the speed of the computer.

To overcome this problem, *distributed memory multiprocessors*[2] have been developed; for example, the MEiKO Computing Surface [87], the Intel iPSC2 [1] and the NCUBE-2 [99]. Each processor node typically contains a CPU, a local memory, some sort of context switching mechanism to handle multiple processes, and a communications controller to exchange messages with other processors via high bandwidth links. In older machines, processor nodes are generally sparsely connected using a topology such as a hypercube, mesh or ring. The structure used is sometimes fixed, and sometimes configurable under user control. Messages are passed between processors using *store and forward* techniques, i.e. messages are read in and stored at each intermediate processor node, before being routed onwards towards their destinations. Consequently, these machines exhibit the concept of locality with respect to communications, with some processors being closer together than others. The latest generation of machines generally use *circuit-switched networks* to communicate between processors, thereby reducing communications latency. Locality is also less of an issue, since processors are often equidistant from one another.

Distributed memory multiprocessors have the advantage of being scalable, but they are harder to program than their shared memory counterparts. This is largely due to the added complexity involved in managing inter-processor communications and synchronisations. Process placement and load balancing issues must also be considered. Often a programmer will require a detailed knowledge of the target machine in order to obtain reasonable performance. The main reason for this is that software environments and programming languages for distributed memory multiprocessors have lagged far behind hardware developments. Pro-

---

[2]Sometimes called multicomputers.

grams are often written in dialects of conventional sequential languages, such as C and FORTRAN, which have been extended to support message passing. One notable exception is the language occam [86], which was developed as the native language for the transputer. There have been attempts to relieve the burden on the programmer by providing environments which handle some of the more mundane tasks, and abstract away from the underlying hardware; for example, Linda [36] and Strand [49]. However, even using such environments, it is still a demanding task to utilise the underlying hardware efficiently.

This thesis is concerned with investigating the provision of operating system support for process placement and load balancing decisions in distributed memory multiprocessors. In future, any mention of multiprocessors can be taken to refer to distributed memory machines. Given a program and some information regarding its structure, it would be desirable for the operating system to be able to execute the program efficiently, making the best possible use of the available resources. The work presented here is exploratory, and is a first step in this direction. To limit the scope of the study I concentrate on analysing particular classes of parallel programs, and tend to keep the processor topology and run-time environment constant whenever possible.

1.1 Research Objectives

The study has two main objectives. The first is to investigate in a quantitative manner the relationships which exist between the structure of parallel programs and their performance characteristics. For example, I would like to be able to determine the particular aspects of a program's structure that had the greatest impact on performance. The second objective builds upon this work. It is to undertake a quantitative analysis of the behaviour of programs when executed under the control of a process migration strategy. I would like to be able predict the

circumstances in which a process migration strategy could improve performance, and estimate the likely performance benefits to expect.

These two objectives determine the approach taken in this thesis, and lead me to consider the standard statistical methods of experimental design and analysis.

1.2 Contributions of Thesis

This section highlights the main contributions of this thesis.

- It is shown how a class of loosely synchronous data parallel programs can be adequately characterised in terms of a small number of program parameters describing time-averaged macroscopic properties.

- Standard statistical techniques, such as the analysis of variance and analysis of covariance, are shown to be useful tools with which to, firstly, undertake exploratory performance analysis experiments; and secondly, construct performance prediction models.

- A systematic investigation of the behaviour a class of process migration strategies is presented, for various classes of programs characterised as described above. Quantitative estimates of the relative importance of the factors influencing the performance of the strategies are obtained.

- It is shown how the performance of a process migration strategy can be improved under certain circumstances by attempting to optimise the values of performance metrics identified using an analysis of covariance. These performance metrics characterise the interactions which occur between a program and the machine it is being executed on.

- The design and construction of a novel process migration mechanism for a transputer-based machine is described.

1.3 Chapter Outline

Chapter 2 presents a review of the literature relevant to the areas addressed by this thesis. Performance analysis techniques are surveyed, and the two dominant models of parallel computation are introduced. The scheduling and mapping problems, as found in distributed memory multiprocessors, are described. A discussion of load balancing techniques and process migration strategies is presented.

Chapter 3 describes the tools and environments which have been constructed in order to support the experiments described in this thesis. The statistical techniques used in subsequent chapters are also presented.

Chapter 4 demonstrates how it is possible to characterise the behaviour of a particular class of parallel programs in terms of a small number of parameters describing time-averaged macroscopic properties. Analysis of variance techniques are used to investigate the impact of these parameters on program performance using a number of different metrics.

Chapter 5 investigates a more realistic class of irregular parallel programs. Analysis of variance techniques are used to pin-point program parameters with the greatest predictive powers. A model using these parameters is then derived and shown to perform reasonably well given its simplicity. To improve the accuracy of the model, the analysis of covariance is used to identify suitable covariates to include. The covariates help to characterise the interactions which occur at run-time between a parallel program and the machine it is being executed on.

Chapter 6 examines process migration strategies, as applied to those programs which are mapped in an unbalanced manner, but do not vary over time. A relatively simple strategy is derived, and shown to generally improve the performance of such programs. It is demonstrated that a better strategy can be constructed by attempting to optimise the values of the covariates identified in Chapter 5.

Chapter 7 shows that the improved migration strategy derived in Chapter 6 is suitable for use with programs whose behaviour varies over time. The relationships which exist between the structure of such programs and the performance of the migration strategy are investigated in a quantitative manner.

Chapter 8 summarises, and presents the conclusions resulting from this study. Possibilities for future work are discussed.

Chapter 2

Background

This chapter presents background material relevant to the issues addressed in this thesis. As discussed in the previous chapter, I am interested in a quantitative analysis of the performance characteristics of parallel programs; both with and without process migrations. Consequently, Section 2.1 gives an overview of the various techniques that have been used to analyse and reason about the behaviour of parallel systems, with particular attention to the performance analysis of parallel programs. The two dominant performance models of parallel computation are described in Section 2.2. Section 2.3 introduces the fundamental concepts of scheduling and mapping as applied to parallel programs. These activities greatly influence performance, and are one of the main reasons why parallel machines are harder to program than sequential machines. Section 2.4 presents a general overview of load balancing techniques for parallel machines. Since I am interested in studying the behaviour of programs when executed under the control of a process migration strategy, Section 2.5 surveys the literature relating to this area. Finally, in the light of the preceding material, Section 2.6 sets the scene for the work presented in subsequent chapters.

2.1 Performance Analysis of Parallel Systems

Performance analysis techniques allow us to characterise, model and analyse the complex behaviour patterns which occur within computer systems. Such methods can be used to explore the performance characteristics of existing hardware and software systems, or to explore design alternatives in systems which are yet to be constructed.

Many performance analysis techniques have been developed for conventional sequential machines. Broadly speaking, these can be classified as falling into one of three areas: analytical methods, simulation methods or measurement methods. Although specialised techniques have been developed to handle the extra complexity introduced by parallel systems, the same broad categorisation still holds.

2.1.1 Analytical Methods

The major advantage of analytical methods is that they can be used to construct performance models relatively quickly, and usually with less effort than required using either of the other two techniques. However, their great disadvantage is that they require many simplifications and assumptions to be made. This is because, using such high level techniques, it is very difficult to capture the complex nature of parallel systems to any great level of detail. Consequently, the accuracy of analytical models can be questionable, unless validated using simulation or measurement techniques.

Standard queueing theory methods are often used in performance analysis studies to construct analytical models. A number of tools and techniques have been developed to reason about models constructed in this manner; a useful introduction to these methods is given by King [75]. Queueing models are particularly well

suited to the performance analysis of distributed systems, since they can accurately represent the multiple service sites and independent jobs that characterise such systems. For example, Eager *et al.* [41] use these techniques to investigate the appropriate level of complexity required of a load sharing policy in a distributed system. Queueing models may also be used to model programs executing on multiprocessors, as long as the tasks in the system are relatively independent; see [127] for example. However, queueing networks are unable to account for the complex behaviour patterns that a set of communicating processes exhibit. Consequently, their usefulness is restricted with respect to the performance analysis of parallel programs running on distributed memory multiprocessors.

The Petri net formalism can be used to represent and analyse the performance of parallel programs and machines. Given a Petri net representing the parallel system to be analysed, a *reachability graph* can be constructed defining the system's state space, and thus its *potential* behaviour patterns. This is the approach adopted by Shatz and Cheng [116]. This reachability graph can then be inspected to detect properties of the system under investigation, for example, the possibility of deadlock occurring. A problem with the reachability graph approach is that the state space grows combinatorially with the size of the system being studied; a so called "state space explosion". To overcome this, structural techniques have been proposed: for example, Murata *et al.* [96] describe a technique for detecting deadlock in a Petri net representation of a program without generating the corresponding reachability tree. Such methods are based on the syntactic structure of the Petri net. Even so, it is not clear whether realistically large systems can be analysed using this approach.

There have been a number of attempts to analyse the performance of iterative algorithms running on distributed memory multiprocessors from first principles using analytical techniques. For example, Marinescu and Rice [92] study the effects of communications latency and load imbalance, and suggest schemes for improving performance by reducing synchronisation overheads. Related research

is presented by Brochard *et al.* [19], who investigate the properties of different synchronisation schemes.

It should be noted that both queueing networks and Petri net models can be simulated, as well as being manipulated using analytical methods. Indeed, in the case of Petri nets, simulation may be the only option if the reachability graph of the system under examination becomes too large.

2.1.2 Simulation Methods

Simulation models can incorporate more detail and require less assumptions than analytical models. The penalty to pay for this improvement in accuracy is an increase in the time required to obtain results. This manifests itself both during the construction of the model, and during the execution of the simulation trials themselves.

A number of simulation systems have been implemented for parallel machines and programs. For example, Davis *et al.* [35] describe a system called TANGO which simulates the execution of a parallel program on a multiprocessor. The user is free to select the level of detail that they require a program to be simulated at. The simulation system itself runs on a conventional uniprocessor, but the ordering and timing of events of the simulated multiprocessor is preserved. Contention and interactions between processes are modelled in detail. The system uses an *execution-driven* simulation technique in order to improve execution times; rather than emulating instructions, the source code of the program being simulated is executed whenever possible. A similar tool called HYPERSIM is described by Bain and Shala in [5], the main difference between the two being that, in order to improve performance, the latter tool can itself be executed in parallel on a multiprocessor.

Hayes and Andrews [61] describe a simulation system called ADAS that is particularly suited to early performance analysis studies of integrated hardware and

software systems. In contrast to TANGO, ADAS expects the candidate program to be specified in a graph theoretic manner. From this graph a Petri net is derived, and this is used to simulate the execution of the program running on an arbitrarily chosen number of processors.

The three systems discussed above have one thing in common, they all attempt to simulate the behaviour of the program as well as the underlying machine. There have also been a large number of instruction-level simulation systems constructed for both uniprocessors and multiprocessors. These systems concentrate on simulating the hardware at a detailed level, and are consequently slow, and unsuitable for large-scale experimentation. In addition, they are generally not concerned with program structures, and so are of little interest here.

2.1.3 Measurement Methods

Measurement techniques can be used to collect statistics in order to analyse the performance of existing programs running on existing machines. Such methods are not well suited to exploratory work, since they require the actual implementation of the systems under investigation. They are, however, suited to "fine-tuning" applications. Programmers of parallel machines often have to use such methods in order to achieve acceptable performance levels.

Monitors can be *event-driven*, meaning that measurements are driven by changes in the system's state; for example, process creation, communications or remote-memory accesses. Alternatively *time-driven* techniques can be used, meaning that statistics are collected at pre-defined time intervals. Monitoring can also be achieved with differing degrees of intrusion. Software monitors are the most intrusive, since they have the greatest impact on the execution of the program. Hardware monitors are the least intrusive, however, they require low level architectural support. Hybrid schemes are also possible. For example, Zitterbart [138] describes an event-driven hybrid monitor called NETMON-II for transputer-

based machines; and Zhang *et al.* [136] present a software monitor for the BBN GP1000 multiprocessor. A good discussion of monitoring related issues is given by Marinescu *et al.* in [91].

Recently a great deal of effort has been dedicated to the construction of visualisation tools to help the programmer understand the performance characteristics of their programs in an intuitive manner. Example of such systems include ParaGraph [62] and GRAIL [123].

2.1.4 Comments

This thesis concentrates on the performance analysis of programs, rather than machines. Although a performance model of the machine is still required in order to obtain useful results. I want to carry out large scale systematic exploratory experiments, so measurement techniques are not suitable, since they require the actual implementation of the programs being investigated. On the other hand, analytic methods are not entirely appropriate, since it is desirable to model the complex interactions which occur between a program and the machine that it is executed on as accurately as possible. Consequently, simulation techniques suggest themselves; a suitable simulation environment is presented in Chapter 3. In addition, measurement techniques are used for validation purposes in Chapters 6 and 7.

2.2 Models of Parallel Computation

Generally speaking, an underlying performance model of parallel computation is required before one can reason about the behaviour of a parallel program. The two types of model most commonly found in the literature are *precedence graphs* and *process graphs.*

2.2.1 Precedence Graphs

A precedence graph (sometimes called a dependency graph or task graph) represents a parallel program as a precedence constrained set of tasks. As in [105,117, 127], the graph is generally constrained to be a Directed Acyclic Graph (DAG). The nodes in the graph represent the tasks, and the edges represent data dependencies between tasks. A task may not begin to execute until all the tasks that it is dependent on have finished executing. Obviously some program structures, loops for example, result in non-acyclic graphs. However, this problem can be overcome by un-rolling the graph.

The tasks in a precedence graph are the atomic units of computation i.e. a task will only communicate before it commences execution (in order to receive information), and after it terminates (in order to output information). Execution times are generally associated with individual tasks by allocating weights to the nodes of the graph. A precedence graph can be defined as follows:

> **Definition 2.1** *A precedence graph is a weighted graph $G(V, E)$ whose vertices, $V = \{1, 2, .., N\}$, represent the tasks of the program, and edges, E, represent the data dependencies between those tasks. The relative computational requirements of task i are represented by w_i. The top node in the graph (task 1) has no predecessors. The bottom node in the graph (task N) has no successors. At intermediate levels in the graph, nodes can only be dependent on nodes at higher levels.*

A number of extensions to conventional precedence graphs have been proposed in the literature. For example, Hwang *et al.* [68] add communications costs to the arcs, and Towsley [128] describes how to incorporate information relating to the probability of a communication occurring.

Precedence graphs capture in a detailed manner the interactions and temporal relationships which exist between a set of tasks comprising a particular program.

They are particularly suited to characterising programs written in languages (or executed under environments) which tend to spawn large numbers of relatively short-lived tasks. Parallel functional languages and dataflow languages fall into this category. An alternative technique is to use a process graph to summarise the time-averaged properties of a program.

2.2.2 Process Graphs

A process graph (sometimes called a communication graph or problem graph) represents a parallel program as a graph in which the nodes correspond to processes, and the arcs correspond to communication channels between processes. These graphs tend to emphasise the overall process and communications structure of a computation.

Process graphs have their origin in the work carried out by Stone [124]. He investigates the problem of assigning program modules to a heterogeneous network of processors, where some modules are particularly suited to certain processors. However, Stone's graphs do not represent parallel computations as we know them today, since they do not consider the possibility of modules being able to execute simultaneously.

In [15], Bokhari assumes an unweighted process graph. This approach allows the communications structure of a parallel computation to be defined. Often, however, weights are located to the arcs of a process graph, thereby allowing the volume of communications passing between two processes to be specified, see [16] for example. In its most general form, both the nodes and edges of a process graph are allocated weights representing the time-averaged behaviour of a program; Ercal *et al.* [45] use this approach. The weights associated with nodes specify the computational requirements of the corresponding processes, and, as previously, the weights associated with arcs specify the volume of inter-process communications.

Arcs may either be directed, indicating one way message transfers, or undirected, indicating bidirectional message transfers. A more formal definition follows:

> **Definition 2.2** *A process graph is a weighted graph $G(V, E)$, whose vertices, $V = \{1, 2, .., N\}$, represent the processes of the program, and edges, E, represent the communications channels between those processes. The relative computational requirements of process i are represented by w_i. The volume of communication associated with the channel connecting processes i and j is represented by c_{ij}.*

Process graphs have been extended in a number of different ways. For example, Lo [85] considers compute and communications phases, and shows how the temporal behaviour of the program corresponding to the process graph can be described in terms of these phases. Lee and Aggarwal [82] also use the concept of a phase. Their phase is defined as an interval during which an identifiable and distinct interaction pattern occurs between processes.

Process graphs are best suited to programs which are structured as a set of persistent sequential processes that periodically communicate with one another. Often the process structure will be extractable at compile-time, and the processes will exist for the entire execution period. This paradigm is widely used, and corresponds to the CSP model of parallelism developed by Hoare [63]. The programming language occam [86] was designed as a partial implementation of the CSP model of parallelism; consequently, an occam program can conveniently be represented by a process graph. Many other CSP-type parallel programs are written in conventional languages, such as C or FORTRAN, which have been extended with support libraries to provide an environment within which processes can be created and inter-process communications can occur. CSTools [88] is an example of such a system.

2.2.3 Comments

A process graph for a given program is an aggregate of the corresponding precedence graph. A typical process will consist of a (possibly large) number of tasks, these are represented explicitly in a precedence graph. In contrast, only the cumulative computational demands of the tasks are represented in a process graph. Similarly, a precedence graph gives detailed information about the data dependencies between tasks, whereas a process graph shows only which processes communicate with one another, usually along with an indication of the volumes of inter-process traffic.

As described in Chapter 3, the approach adopted in this thesis is to attempt to characterise parallel program behaviour in terms of a small number of parameters relating to the time-averaged properties of a program. A process graph model of computation is best suited to this task.

2.3 Scheduling and Mapping

The problem of allocating parallel programs to run on parallel machines has received a great deal of attention. The literature generally identifies two distinct problems. The first is called the *scheduling problem*, this is concerned with the assignment *and* ordering of tasks. The second is called the *mapping problem*, this is purely concerned with the assignment of processes to processors. These terms are not in universal use; for example, some authors include task scheduling within their definition of the mapping problem. In their general forms, both problems are known to be NP-hard [46,84,130]. Consequently, approximate methods are generally utilised to generate good sub-optimal solutions.

In the following sections the scheduling and mapping problems are defined and discussed in greater detail. It is assumed that the program graph (whether it be a precedence graph or a process graph) is known in advance.

2.3.1 The Scheduling Problem

The scheduling problem is concerned with the placement and ordering of tasks. A scheduling algorithm generally expects a precedence graph as its input. It produces as its output a schedule for the execution of the tasks which attempts to minimise the run-time of the program. This schedule specifies which processor to execute each task on, as well as the order in which the tasks should be executed on each processor. The problem can be stated a little more formally as follows:

> **Definition 2.3** *Given a set of processors and a set of tasks represented by a precedence graph as specified in Definition 2.1, find a schedule for the execution of the tasks on the processors such that the program's execution time is minimised.*

The scheduling problem has been addressed in numerous publications. For example, El Rewini and Lewis [44] propose a heuristic which considers contention for resources, and schedules based on the level of each task in the precedence graph. In [117], Shirazi *et al.* describe a number of list scheduling algorithms. These include an algorithm that gives priority to computationally intensive nodes, and another that makes use of traditional critical path analysis techniques. Papadimitrou and Yannakakis [105] describe an algorithm which takes communication delays into account and works in an architecture-independent manner (in the sense that the number of processors is assumed to be unbounded).

2.3.2 The Mapping Problem

The mapping problem is concerned with the assignment of processes to processors, and communications channels to physical links. A mapping algorithm generally expects a process graph as its input, and produces as its output a placement satisfying some mapping criteria which depends on the type of strategy being applied. Three types of strategy can be identified in the literature: topological mappings, cost optimisation mappings and adaptive mappings. These are discussed below.

1. **Topological Mappings**

   Topological mapping strategies concentrate on ensuring that communicating processes are placed on adjacent processors; information relating to the volume of inter-process communications or computational intensity of processes is not considered. Example of such strategies include those presented by Bokhari [15], and and Chen and Gehringer [24]. The following formulation of the mapping problem is appropriate:

   > **Definition 2.4** *Given a set of processors and a program represented by an unweighted process graph with no more nodes than there are available processors: find a mapping such that neighbouring processes get assigned to neighbouring processors.*

   The quality of a mapping generated using a topological mapping strategy can be assessed by considering the proportion of process graph edges that fall directly on hardware links. Contraction techniques are generally used if the number of processes in the program graph exceeds the number of processors. The topological mapping problem is known to be NP-hard in its general form [15], although useful results can be obtained for restricted cases, notably when the hardware and process graphs are regular in structure.

2. Cost Optimisation Mappings

Cost optimisation mapping strategies define an *objective function* to characterise the quality of a mapping. This function is then used in conjunction with a heuristic search procedure in order to find a good sub-optimal mapping. One can specify the problem more formally as follows:

> **Definition 2.5** *Given a set of processors, a program represented by a process graph as specified in Definition 2.2, and an objective function characterising the quality of a mapping: find a mapping of processes to processors such that the value of the objective function is optimised.*

Obviously, the intention of the optimisation process is to achieve a mapping that will minimise the execution time of the program. However, due to the complex behaviour patterns that a set of competing and interacting processes exhibit, it is generally considered impractical to attempt to characterise the execution time of a program. In particular, simple additive models are often not appropriate. Consequently, a number of alternative forms for the objective function have been proposed. For example, minimising communications costs [16,82], or minimising communications costs subject to some computational load balancing constraints [13,45]. In the later case, the stated aims conflict with one another to a large extent: to eliminate inter-processor communications costs completely, one could assign all processes to the same processor, but this would result in a very poor computational load balance.

Techniques that have been used to implement the optimisation process are numerous and diverse. For example, Bollinger and Midkiff [16] describe a simulated annealing based approach; such techniques tend to produce good solutions, but are computationally demanding. Genetic algorithms are explored by Kramer and Muhlenbein [94]. They use the methods of replication,

mutation and selection to evolve near optimal solutions. In [103], Mansour and Fox observe that simulated annealing and genetic algorithms take approximately the same time to produce results of a similar quality. Recursive min-cut techniques are investigated by Ercal *et al* [45]. They advocate a two-phase approach whereby, given k processors, the process graph is partitioned into to k clusters so as to minimise *inter-cluster* communications costs. These clusters are then assigned to processors so as to minimise *inter-processor* communications costs. It is reported that this technique produces slightly worse solutions than simulated annealing, but operates several orders of magnitude faster.

3. **Adaptive Mappings**

   There is a class of mapping strategies that are adaptive in the sense that they monitor the execution of the program in order to improve on the (possibly arbitrarily chosen) initial mapping. This approach is typified by the *post-game analysis* techniques proposed by Sunter *et al.* [125], and Ieumwananonthachai *et al.* [67]. In these strategies, performance statistics are gathered and are then processed off-line by heuristics which make suggestions in order to refine the mapping. The program is then re-mapped and re-executed. This process continues iteratively until a satisfactory mapping is achieved.

2.3.3 Comments

The different aims of scheduling and mapping flow directly from the respective characteristics of precedence and process graphs. Precedence graphs present a detailed representation of a program's structure, and there is enough information to attempt to minimise the program's execution time. Process graphs, on the other hand, are a higher level representation, so secondary objectives are used;

the intention being that satisfying these secondary objectives will result in a good execution time.

I am not concerned with formulating new solutions to the mapping problem, indeed, the work described in this thesis utilises rather simple strategies. However, it is known that the particular mapping used can greatly influence performance. Consequently, in Chapter 5, techniques are explored for incorporating information relating to the quality of a mapping into performance prediction models. In Chapters 6 and 7 I investigate how the mapping in force can be improved at run-time using process migration strategies.

2.4 Load Balancing

The scheduling and mapping strategies discussed in the previous section assumed that the program graph was known in advance. Often however, this will not be the case. In such situations the operating system is required to perform "on the fly" scheduling and mapping. This is usually implemented as some sort of *load balancing* heuristic.

Load balancing is used in parallel machines in order to make the best use of the available resources. In a distributed memory multiprocessor running a single application this normally involves ensuring that: the computational load is distributed as equally as possible over all of the processors; and that the communications load is distributed as equally as possible over all of the links.

Ideally, the load balancing strategy (or strategies) will be incorporated into the operating system, and will be able to perform effectively over a wide range of programs. Load balancing techniques are normally classified as falling into one of two categories: *static load balancing* methods or *dynamic load balancing* methods.

2.4.1 Static Load Balancing

Static load balancing techniques allocate processes to processors *before* the program commences execution. This technique is useful for applications whose behaviour is stable over time, and predictable to some extent.

Simple approaches involve distributing processes randomly, or using some sort of cyclical allocation of processes to processors. As an alternative, there is a strategy known as the *scattered spatial decomposition* [102]. This technique operates by dividing a computational domain into a large number of pieces which are then distributed equally among the available processors. The intention being that, on average, processors will have an equal amount of work to do. This method performs well for certain classes of problem, see Section 4.2 for further details.

If one of these strategies is not appropriate, then one of the mapping strategies discussed in Section 2.3.2 generally has to be used. The topological and adaptive mapping strategies are particularly suitable, since they do not necessarily require any information about the computational or communications demands of the processes.

2.4.2 Dynamic Load Balancing

Many problems do not display stable behaviour patterns, instead they vary in unpredicatable and data-dependent ways. This often results in considerable load imbalances. Example of such problems include: particle dynamics applications, parallel discrete event simulations and adaptive mesh calculations. Dynamic load balancing strategies attempt to address these imbalances by moving load from one processor to another as the program is executed. A distinction is sometimes made between load balancing and load sharing, which refers to the less demanding task of attempting to ensure that no node in the system remains idle while work remains to be done [41].

There are two types of dynamic load balancing strategy commonly identified: *load distribution* strategies and *load migration* strategies.

Load distribution tends to occur in systems which spawn large numbers of independent tasks in an unpredictable manner. Instead of attempting to migrate these tasks while they are executing, the operating system attempts to balance the system load by deciding which processor to execute them on as they are spawned. Once a task is executing on a processor, it is never subsequently moved. For example, Lin and Keller [83] propose a load distribution strategy based on the "gradient model". This strategy involves transferring unevaluated tasks to nearby idle processors according to a pressure gradient indirectly constructed by considering requests from those idle processors. Nazief [98] investigates the performance of a number of distribution strategies under different computational models and machine architectures. In [118], Shivaratri *et al.* present a comprehensive survey of load distribution strategies.

A load migration strategy, on the other hand, also considers migrating a task once it has begun to execute. It is suitable both for programs that spawn tasks in an unpredictable manner, and programs that consist of a relatively constant number of communicating modules. Examples of load migration strategies include those presented in [34,56,114,131]. Load migration strategies are discussed in considerable detail in Section 2.5. A special case of load migration worth a brief mention here is *dynamic remapping* [101]. Rather than continually migrating processes in an attempt to achieve a load balance, a dynamic remapping strategy will suspend and remap *all* processes when a sufficient load imbalance is detected.

Load distribution and load migration strategies address many of the same issues. It has been reported that, although both strategies can improve over the no load balancing case, load migration strategies have the capability to significantly out-perform load distribution strategies [80]. In particular, it appears that for programs structured as a set of long-lived modules, load migration is to be preferred [98].

2.4.3 Comments

As already mentioned, this thesis will concentrate on a process graph model of computation. In addition, I will restrict myself to considering load migration as a means of dynamic load balancing. The remainder of this chapter is dedicated to a discussion of load migration strategies. The term *process* is used henceforth as a generic term to represent the migratable unit of work. This unit could take a number of forms: for example, it might be an extremely light-weight independent task. Alternatively, it could be a heavy-weight module which communicates intensively with other modules.

2.5 Process Migration

The ability to migrate processes at run-time can offer a number of attractive benefits. A good example is fault tolerance; in the event of a gradual processor failure, processes can be migrated to alternative homes. Another use is to enable processes to take advantage of special hardware or software capabilities unique to a particular node, or group of nodes. However, in the context of distributed memory multiprocessors, by far the most common use for a process migration facility is for the implementation of dynamic load balancing. This is the application that this thesis concerns itself with. In particular, I concentrate on process migration strategies that could be incorporated into an operating system, rather having to be embedded in the application itself.

It is convenient to think of a process migration facility as comprising of three functionally separate components [2,8]: namely, the statistics collection module, the decision making module, and the migration module. The statistics collection module gathers performance information about the processors and processes in the system. This information is subsequently used by the decision making module in order to decide which (if any) processes to migrate. Once this decision has been

made, the migration module is informed. The migration module is responsible for actually executing the migrations. In reality, the distinction between these three modules may be blurred, and they may be distributed over multiple processors.

In the following sections the decision making and migration modules are discussed. The decision making module is referred to as the *migration policy*, and the migration module is referred to as the *migration mechanism*. The statistics collection module is not of any great interest here, for the purposes of this discussion it is assumed that a suitable monitoring facility exists (whether it be in hardware, software or, some sort of hybrid scheme).

2.5.1 The Migration Policy

A large number of migration policies have been proposed in the literature. Many of the older policies stem from the field of distributed systems; for example, those presented in [57,80,110,121]. There are significant differences in the dynamic load balancing problem between distributed systems and multiprocessors. Distributed systems are usually general purpose multi-user machines, whereas multiprocessors tend to be dedicated to a single application at any one time. Consequently, the processes running on the two types of machine display different characteristics. Processes running on a multi-user machine tend to be relatively independent of one another; for example, separate users' sequential programs. In contrast to this, processes running on a multiprocessor generally need to communicate with each other, since they are attempting to solve the same problem. Consequently, multiprocessors are usually constructed with a reasonably efficient inter-processor communications mechanism in place.

In the light of the above discussion, it is clear why process migration policies for distributed systems have tended to ignore inter-process communications and dependencies when making migration decisions. A consequence of this is that these policies are not always directly applicable to multiprocessor systems. For

example, many programming paradigms produce processes which communicate and interact with each other in complex ways. Policies developed for distributed systems are not suitable without modification in such cases.

It should be noted that there are programming paradigms that do generate relatively independent processes, and migration policies developed for distributed systems can be useful in such cases. A good example is the "divide-and-conquer" paradigm, where a problem is broken down into a number of independent sub-problems which are solved and combined to give a final solution. Parallel implementations of functional languages also tend to generate independent processes.

To summarise: process migration policies developed for distributed systems are generally developed without considering inter-process communications, and are designed to work in environments possibly lacking an efficient inter-processor communications infrastructure. Consequently, care should be taken when applying such policies in multiprocessor systems.

General Characteristics of a Migration Policy

In order to migrate a process a migration policy must make three separate decisions [38][1]:

- *When* to attempt the migration.
- *Which* process to migrate.
- *Where* to migrate the process to.

---

[1]This paper also identifies another decision that must be made, namely *Who* should take responsibility for initiating migrations. We generally assume that this is the responsibility of the operating system.

These decisions are common to all migration policies, although the three steps are not always easily identifiable. For example, in the policy presented in [34], the *Where* decision is not made according to the systems state, instead migrations can only occur between predetermined pairs of processors at any particular iteration of the policy.

There are a number of properties which a good migration policy will display, the most important are listed below:

- Stability

  Processes should not be migrated without reason [121]. Unnecessary migrations can occur for a number of reasons: for example, the use of out of date load information, or in the process of trying to achieve too fine a load balance. This phenomenon is known as *thrashing*.

- Good Convergence

  Once a load imbalance has been detected, a good process migration policy should converge to an improved placement as quickly as possible. The longer the time taken to achieve a good balance, the lower the possible performance benefits.

- Scalability

  In order to be suitable for use in massively parallel architectures, a process migration policy should be able to perform effectively with both large and small numbers of processors.

Process migration policies can be classified in a number of different ways. Below four possible classifications are presented, and examples are given of policies that can be accommodated into each category.

Global vs. Local Information

In order to make informed migration decisions, a migration policy requires information about the state of the machine. This information is said to be *global* in nature if it is gathered from all of the processors, or *local* in nature if it is gathered from only a subset of processors. Both global and local policies have been proposed in the literature, and each have their own advantages and disadvantages.

In the global case, the decision-making processor(s) gather information from all other nodes in the system with the aim of constructing an accurate image of the machine's state. For example, Corradi *et al.* [32] describe a global strategy specific to ring-based processor topologies (this restriction has the effect of minimising the number of communications required in order to distribute information relating to the global state of the machine). In [40], Dragon and Gustafson propose a global mechanism that is particularly suited to particle simulations. In current machines the information gathering process can impose significant communications and coordination overheads for anything other than a small number of processors. This can result in performance statistics being out of date by the time they come to be used in the decision making process. Consequently, global policies are not generally scalable beyond perhaps a few tens of processor at most. Future distributed memory multiprocessors are likely to support very high performance communications mechanisms, and total connectivity of processors. When this occurs, global policies should become more attractive, since, in theory, they can make better informed migration decisions than local policies.

It should be noted that, although the information in a global policy is gathered from all processors, the decision making process can be either centralised or decentralised. In the centralised case, all information is sent to a central point of control, and the migration instructions are issued by this processor; see [79] for example. In the decentralised case, all the processors that can issue migration instructions must know about the state of all other processors in the system; see

[40] for example. In the extreme case, all processors require to know the state of all other processors, obviously the overheads of such a mechanism quickly become intolerable.

In order to reduce information collecting overheads and achieve scalability, distributed memory multiprocessors have tended to exploit local policies. Such policies are decentralised and use information from only a subset of processors in order to make migration decisions. The intention being that repeated local balancing will lead to a satisfactory global balance; this assumption appears to be relatively well founded. For example, Qian and Yang [111] show that for a representative strategy (in which processors average their workload with their nearest neighbours) the expected difference between the average load of an arbitrarily chosen processor and the system wide average is zero. Cybenko [34] investigates the performance of several local migration policies, and uses the eigenstructure of the iteration matrices that arise from dynamic load balancing to prove their convergence properties. Local policies are often found to out-perform global policies for anything but a very small number of processors. However, the major drawback of such policies is that a local view of the machine is inevitably less accurate than a global view. Bad migration decisions can be made because each processor has only a partial view of the machine state.

There are varying degrees of local information that can be used. At one end of the spectrum lie policies which, on each iteration, only consider load balancing between mutually exclusive pairs of processors. The processor pairs used are normally updated in a deterministic and cyclic manner between iterations. These techniques are generally known as *Dimension Exchange Methods.* Cybenko [34] proposes such a policy for hypercube processor topologies. The technique is extended to arbitrary processor topologies using graph colouring techniques by Hosseini *et al.* [65].

There is a relatively large class of policies which divide the machine up into a number of (often overlapping) *balancing domains* [90,111,114,131]. Within each

of the domains, the processors exchange load information and then cooperate in process migration decisions in order to balance the load locally. A common technique is for a processor to exchange load information with its nearest neighbours only, although more distant exchanges are also possible.

Heuristics vs. Physical Analogies

Process migration policies attempt to solve an NP-hard optimisation problem, the aim being to balance the load and in doing so achieve an optimal execution time for the program. The policies proposed in the literature can be classified according to whether they are based on based on heuristics, or some more formal physical analogy.

A large number of policies are based on one or more heuristics around which the decision making process is centred. The intention is that the application of the heuristics will result in an acceptable load balance. Obviously, the choice of heuristic is of prime importance, and not all heuristics will be suitable for all machines or program models. However, a significant number of policies have been constructed in this manner.

A *bidding* heuristic is proposed by Stankovic and Sidhu in [122]. This algorithm works by highly loaded processors distributing requests to receive work to all other processors in the network. Those processors in an under loaded state calculate a bid based on their current situation, and reply to the requesting processor. The highly loaded processor then evaluates its bids, and chooses the best node to migrate a process to. A verification message is sent to the winning bidder to ensure that its state has not changed. If this is the case, then the process is migrated. If the state has changed, then the process is re-started on the highly loaded node.

A well known problem with the bidding algorithm is that highly loaded nodes tend to "gang up" and dump processes on the winner of the bids. The *drafting* algorithm proposed by Ni *et al.* [100] solves this problem, and is believed to result

in a fairer load distribution. The strategy works by each processor maintaining a table recording the state of all other processors in the system (i.e. whether they currently have a high, normal or low load). When a processor changes state it broadcasts this information to all other nodes. In addition, a processor entering the low state will send a *draft-request* to any highly loaded processors. On receiving such a message, the highly loaded processors respond with a *draft-age* message which contains some measure of load with respect to its migratable processes. On receiving all of its replies, the lowly loaded processor will send a *draft-select* message to the chosen highly loaded processor. If that processor is still in a high state, a process is migrated, otherwise the requesting processor has to begin the protocol all over again.

In [134], Xu and Hwang propose and compare the performance of a number of migration heuristics. Two classes of policy are identified, depending on how the target processor is selected (i.e. the *which* decision). The first of these selects target processors in a cyclical manner in order to attempt a fair load distribution, and the second always selects the processor with the minimum load. The policies are applied periodically depending on the system load, and the processes to be migrated are chosen arbitrarily.

As an alternative to heuristics, a number of techniques derived from the natural sciences have been used as models for process migration policies. These physical analogy-based strategies generally revolve around an objective function which represents a cost that the policy will attempt to minimise. Process migrations are used to minimise the costs quantified by the objective function at run-time, in an attempt to achieve a faster execution time. This is analogous to the approach used by the cost optimisation mapping strategies described in Section 2.3.2; the objective functions in both cases contains similar terms.

Policies that fall into this category include those described in [47,50,52,78]. Fox *et al.* [50] use a particle analogy, treating a collection of processes as an ensemble of "particles" of computation, which repel each other if they are situated

on the same processor, and attract each other if they communicate. They define a potential, and dynamically minimise the energy to generate favourable program configurations. Various techniques are proposed for this minimisation process, including neural networks [52], and simulated annealing [47]. It should be noted that simulated annealing, although capable of producing good solutions, is generally considered too computationally demanding to be applied at run-time. In [78], Koller describes the implementation of a load balancer based on the particle analogy.

A criticism often levelled at this class of policies is that the objective functions constructed, being additive in nature, do not accurately account for the complex behaviour patterns exhibited by a set of interacting processes [67]. However, from a pragmatic point of view, such strategies do appear to result in improved performance in certain cases.

Another physical analogy often used is the process of *diffusion*. Boillat *et al.* [12] describe a load migration strategy based on an analogy to the diffusion of heat in a metal bar. Processors are arranged in a pipeline and "particles" of computation dissipate throughout the pipeline in order to balance the load. Further diffusion-based techniques are described in [34,65]. A common characteristic of such policies is that desirable properties, for example optimality and convergence, can be formally proved.

Sender- vs. Receiver-Initiated Policies

Migration policies (and dynamic load balancing policies in general) can be classified according to the party which takes the initiative in the transfer process [42]. Two types of strategy have been identified: sender-initiated, and receiver-initiated (although intermediate strategies are also possible).

In a sender-initiated policy, heavily loaded processors actively seek lightly loaded processors and attempt to off-load processes onto them. Examples of

sender-initiated policies include the bidding heuristic of Stankovic and Sidhu [122], and the distributed threshold based mechanism proposed by Ashraf-Iqbal *et al.* [3].

In a receiver-initiated policy, lightly loaded processors search for heavily loaded processors and offer to receive surplus processes. Examples of receiver-initiated policies include the neighbourhood averaging technique proposed by LeMair and Reeves [131], the drafting algorithm described by Ni *et al.* [100], and the method based on *balancing domains* investigated by Gulati *et al.* [56].

Eager *et al.* [42] conclude that receiver-initiated policies are preferable at high system loads. For light and medium loaded systems, sender-initiated policies are to be preferred. They also point out that sender-initiated policies are to be favoured when load distribution (rather than migration) is used to implement dynamic load balancing.

In [25], Chowkwanyun proposes a hybrid-strategy combining the best of the sender- and receiver-initiated strategies. In the policy described, a processor is able to change from being sender- or receiver- initiated according to its load. The idea being that the most appropriate technique can be used according to the current system load.

Simple vs. Complex Policies

Migration policies exhibit a wide range of characteristics with regard to the complexity of the decision making process, and the amount of load information required. Examples of relatively complex policies include the bidding and drafting algorithms discussed earlier. Simple policies are typified by the random policy described by Grunwald *et al.* in [53].

As a general rule, the migration policy must be sophisticated enough to make good migration decisions, but not so complex that the overheads associated with

the policy out-weigh the possible benefits. Satisfying these two criteria involves making trade-offs between accuracy and minimal intrusion.

Eager *et al.* [41] tend to favour simple policies using small amounts of load information, concluding that such policies can provide good solutions at a fraction of the cost of more complex strategies. However, it should be noted that this study assumes independent processes, so migrants can be chosen arbitrarily from the set of available processes. For many of the programming paradigms used in distributed memory multiprocessors, process interactions must be considered when making migration decisions, thereby increasing the complexity required in order to make good decisions.

Comments

There is no doubt that a well-designed process migration policy is capable of significantly improving performance. However, ensuring that such a policy is robust, efficient, and sufficiently general purpose is still the subject of much on-going research. The above discussion highlights the complex trade-offs that must be considered when designing a policy. Many alternative approaches are possible, and none appears to be best in all circumstances. However, there seems to be a general consensus that in order to be scalable a policy should be distributed; and that to allow the potential benefits to be fully realised, policies should not be overly complex.

The migration policies investigated in this thesis are described in Chapter 6. In terms of the framework presented above, these policies are distributed, heuristic-based, cooperative rather than sender- or receiver-initiated, and relatively simple.

2.5.2 The Migration Mechanism

A number of migration mechanisms have been developed for distributed operating systems. Examples include those provided in the Accent [113], Demos/MP [110], Amoeba [95], Sprite [38] and Charlotte [2] operating systems. In recent years migration mechanisms have also been developed for multiprocessors, see [6,72,73,74, 79] for example. There are no great differences in the requirements of a migration mechanism in the two cases, so no further distinction will be made. It should be noted that application-level migration mechanisms, where the mechanism is built into the code [12], are not considered here. The following section highlights the steps commonly required to be executed in order to migrate a process.

Migrating a Process

The migration mechanism executes migration decisions made by the migration policy. The scheme used to carry out the migration normally conforms to the following structure [2]:

- **Negotiation**

  Both source and destination processors must reserve the appropriate resources and commit themselves to carrying out the migration. If two processors cannot reach agreement, then the migration is abandoned here.

- **Transfer**

  The first stage of the transfer is to suspend the migrating process on the source processor. The process's code and data must then be sent to the destination processor, where it should be copied into a suitable area of memory. If the process is large, the transfer of the process can be a considerable overhead.

- **Establishment**

  The space vacated by the process on the source processor should be marked as being available for reuse. On the destination processor, the process must be prepared for execution. For example, pointers need to be translated, absolute addresses should be transformed, and if necessary, the process should be synchronised with the new local clock. Once all these tasks have been performed, the process can be placed in the active queue.

Desirable Features

There are number of (sometimes conflicting) characteristics that it is desirable for a process migration mechanism to exhibit. These are discussed below.

- **Minimal Residual Dependencies**

  It is said that a process has a *residual dependency* on its original or any subsequent node(s) if the correct operation of the system can only be guaranteed while these nodes remain active [38,74,126]. For example, if a message has to be sent to the original node of a process before it can be delivered to the process's current home, a residual dependency exists on the original node. For the sake of performance, it is desirable to minimise residual dependencies. However, as residual dependencies are reduced, the migration mechanism becomes more complex. This is illustrated below.

- **Transparency**

  A process migration mechanism is said to be *transparent* if, after a process migrates, that process and all others that communicate with it continue as if nothing had happened. Obviously the migration might result in different synchronisation patterns occurring, since communications with the migrant are likely to have been delayed. However, these delays should be

indistinguishable from delays caused by other reasons, for example, a heavily congested network or processor.

Any general purpose migration mechanism must offer transparency, although some systems have to impose restrictions in order to achieve this goal. For example, Baker and Milner [6] limit the use of local clock references.

The main overhead associated with maintaining transparency is ensuring that the correctness of message passing is preserved after migrations. Once a process has migrated, messages to the migrant must be re-routed so that they arrive at the correct location. A number of techniques have been proposed to achieve this. The Accent [113] and Demos/MP [110] operating systems use a simple technique whereby the source processor is relied upon to forward future messages to the migrant's new processor. It is clear that this technique imposes considerable residual dependencies; if a process migrates multiple times, then messages to that process will pass through multiple nodes until they find the migrant's current home. For this reason, this technique is only suitable for systems executing lightly communicating processes.

The Emerald system [74] uses a modified form of message forwarding which informs senders of the correct location of the migrant, so that in future they will be able to route messages correctly. This reduces the residual dependencies, but the old home of the migrant is never sure when it is going to stop receiving incorrectly addressed messages. Joosen *et al.* [73] describe a technique to overcome this. Their migration mechanism displays *time-limited* residual dependencies. When a migration takes place, a *shadow* is left behind on the source processor. The shadow initially contains a record of all the entities that can possibly communicate with the migrant. When an object, A, first communicates with a recently migrated process, B, A's processor becomes informed of the new location of B. In addition, the reference to A is deleted from B's shadow on B's old processor. When all references

have been removed from B's shadow, the shadow can be deleted and the residual dependency will have been removed.

The scheme used by the Charlotte operating system [2] minimises residual dependencies, but has high overheads. Information is maintained at each end of an inter-process channel about the location of the other end. If a process is migrated, all processes communicating with it are told of the new location. This method is more complex than those described above, and entails tables being maintained at each end of a channel. Its great advantage is that it results in no residual dependencies once all processes have been told of the new location of the migrant.

Sprite [38] allocates a *home processor* to each process, which is then consulted whenever a message is sent to the process in question. A residual dependency remains, but it is restricted to the process's home processor. This technique increases the number of communications required in order to achieve a single message transfer, and would seem unsuitable for use in a multiprocessor.

- **Minimal Costs**

Three distinct types of cost can be distinguished:

1. Firstly there are the costs associated with the actual transfer of the process. This consists of the processing time and communications resources required to suspend, transfer and resume the process. Techniques have been developed to reduce this overhead. For example, Zayas [135] describes a *copy on reference* scheme for the Accent operating system. When a process migrates, the code is transferred, but portions of data are only transferred if and when they are needed.

2. There is a cost associated with the migrant process being temporarily unavailable. While the process is suspended, other processes may be blocked waiting on some sort of response from the migrant. The V

system [126] attempts to minimise this time by pre-copying the state of the process before it is suspended. Consequently, only the data that has changed between the time the process was pre-copied and the time it was suspended needs to be subsequently transferred. The system is implemented using a "dirty bit" mechanism.

3. Finally, residual dependencies result in extra communications costs; for example, message forwarding.

Obviously there are trade-offs to be made. If residual dependencies are minimised, the migration mechanism tends to become more complex and time-consuming. Conversely, if a simple mechanism is used, for example one that uses message-forwarding to support transparency, considerable residual dependencies may be generated.

Comments

The above discussion indicates that is is difficult to design a general purpose process migration mechanism that is efficient, transparent and minimises residual dependencies all at the same time. In [38,74,110], it is pointed out that conventional processes complicate matters because they are difficult to contain, and it can be hard to define their boundaries. This is largely because the process state is often distributed throughout a number of operating system structures. It has been suggested that migration can be made easier by restricting the entities that may be migrated. For example, self-contained objects have been proposed as suitable migratable entities [26,73,74]. Such objects have the distinct advantage of having clean, well-defined boundaries.

Two migration mechanisms were constructed as part of the work presented in this thesis, they are described in Chapter 3.

2.6 Summary and Conclusions

The areas of performance analysis of parallel systems, scheduling and mapping, load balancing and process migration have all been introduced in this chapter. These techniques are generally applied in order to improve the performance of parallel programs. All too often the responsibility for this task is placed entirely on the shoulders of the programmer. It is generally accepted that the systems software should assume a more prominent role in this respect.

As a first step in this direction, this thesis explores the relationships which exist between the structure of certain classes of parallel programs and their run-time behaviour, both with and without process migrations. I aim to derive quantitative models by exploring the structure of these relationships. This area has not been addressed to any great extent. Previous research has tended to show that a particular mapping strategy or migration strategy improves performance for a particular application, or set of applications.

In the following chapter the techniques, tools and environments used to carry out this study are described.

Chapter 3

Tools, Techniques and Environments

This chapter describes the tools, techniques and environments that have been adopted in order to support the experiments presented in subsequent chapters. Some of the tools discussed here already existed, and were used in their original form; some were used in a modified form; and others were developed specifically for this work. Section 3.1 discusses the use of synthetic programs, and presents a methodology for their construction. This technique was first suggested by Candlin *et al* [20]. Section 3.2 describes the framework used to define, execute and analyse experiments. The structure of the Experiment Definition Language described here was developed jointly with Neil Skilling. The Experiment Generator and analysis of variance tool described were designed and implemented by Neil Skilling. Section 3.3 summarises the functionality of a simulation system designed specifically to model message passing programs executing on distributed memory multiprocessors. The modifications that were required to this system are discussed in Section 3.3.3; all the work described from this point onwards was carried out by the author. Section 3.4 describes the structure and implementation of a transputer-based process migration mechanism. A number of useful statistical techniques are described in Section 3.5. Finally, a summary is presented in Section 3.6.

3.1 Synthetic Programs

The experiments presented in this thesis use a process graph model of computation. There are a number of reasons for this. First of all, to limit the bounds of the study, only programs obeying the CSP model of parallelism are considered. Such programs are generally structured as a set of long-lived sequential modules which communicate with one another, often in a fixed pattern. In particular, these programs frequently exhibit a strong iterative structure, with processes cycling through computation and communication phases. A precedence graph does not seem a natural representation for these programs, since each iteration would correspond to a new task. Another reason for selecting process graphs is their relatively high level nature. One of the aims of the work presented here is to develop models that allow one to generalise about the behaviour of particular classes of programs. Process graphs are better suited to this task, precisely because they offer a cruder representation of program behaviour than task graphs.

In order to characterise particular parallel programs within a class, a small number (say 5-10) of parameters relating to the time-averaged macroscopic properties of the programs as a whole are used. Possible candidates for parameters include the average grain size, or the average degree of the program graph. There are a number of desirable properties that one would seek in a possible parameter set: for example, the parameters should have useful predictive powers, they should be easily measurable, and ideally, they should be obtainable through compile-time analysis. Parameter sets suitable for characterising various classes of parallel programs are discussed in Chapters 4 and 7.

To enable systematic investigations of particular classes of parallel programs, representative members from those classes are required. One could obtain a number of real programs and parameterise them using profiling tools and/or compile-time analysis. However, this approach poses a number of problems. Firstly, real

parallel programs are rather difficult to obtain, they are often protected by copyright and are of a sensitive nature. Secondly, there is no guarantee that programs gathered in this manner will be evenly spread about parameter space. Indeed, they may well be concentrated in a relatively small area of this space. This would happen because programmers tend to design their parallel programs with a definite machine in mind, and restrain them accordingly. As a result, the structure of a program might bear more relation to the target architecture, than to the structure of the problem.

To overcome these difficulties synthetic programs are used. Synthetic programs allow parameter space to be systematically explored. Values for program parameters can be selected, and subsequently, programs exhibiting those particular values can be generated. By selecting parameter values carefully, one can ensure that real programs fall within the area explored. Synthetic programs have been used elsewhere with success. For example, Bemmerl *et al.* [9] describe a program generator for parallel computers which enables load to placed on the processors and/or the communications network. In [109], Poplawski investigates the generation of synthetic benchmarks for distributed memory multiprocessors. Most previous work has revolved around the generation of real source code, which can then be compiled and executed as one would do a real program. The technique proposed below operates at a slightly higher level, allowing programs to be specified and modelled in a more abstract manner.

3.1.1 Constructing Synthetic Programs

In order to construct synthetic programs conforming to a particular class, one must first of all select a value for each of the program parameters associated with the class. This set of parameter values must then be used to construct a process graph. However, this process graph cannot fully define the behaviour of the corresponding program. To do this a *process template* must be used in order to

specify the microscopic computational and synchronisation properties of processes conforming to the class of programs under consideration. A particular instance of a process template will behave in accordance with the weights associated with the node in the process graph to which it is allocated. The process template assumed in this thesis is described in Chapter 4.

A methodology for generating synthetic programs is now presented:

1. Select values for each of the program parameters in the set.

2. Generate a weighted random process graph consistent with a program exhibiting these parameter values.

3. Take a template describing the behaviour of processes in the class of programs under consideration, and allocate one such template to each node in the process graph.

4. Model the resulting program.

This technique selects a representative program from an infinitely large class of programs exhibiting similar behaviour patterns. If the behaviour of different synthetic programs generated from the same parameter set varies to any great extent, it would indicate that the parameter set does not adequately characterise program behaviour.

Once constructed, a synthetic program has to be modelled.

3.1.2 Modelling Synthetic Programs

A synthetic program constructed as described above could be simulated using some simulation system, or alternatively, it could be executed on a real parallel machine.

The approach taken in this thesis is to simulate the execution of such programs using the simulation system MIMD (which is fully described in Section 3.3). MIMD runs on a conventional sequential machine, and uses discrete event driven simulation techniques to simulate the execution of CSP-type parallel programs on distributed memory multiprocessors. This technique offers a number of practical advantages over actually executing the programs on a real parallel machine. The most important of these is that it enables the provision of non-intrusive and arbitrarily complex monitoring facilities. Also, this approach removes a dependency on the availability of parallel hardware. Experiments can be executed on conventional sequential machines, possibly concurrently. Another advantage is that one is able to capture the global state of the simulated machine and program at any point in time, a very useful feature when undertaking exploratory work.

Throughout this thesis, in order to limit the scope of the study, the synthetic programs generated are interpreted as occam programs, and their execution on a transputer network is simulated. This is a representative environment. Occam fits well into the process graph model, so any results obtained can be expressed in terms of the original process graph, rather than in terms of the simulated program.

The following section describes a suite of tools that have been developed in order to support experiments using synthetic programs.

3.2 The Experimental Framework

An experimental framework has been constructed to aid the definition, execution and analysis of experiments [107]. It consists of three components: an experiment construction tool to enable experiments to be conveniently defined and generated; a user-supplied *modelling engine* to model their execution on a given architecture; and some statistical tools to analyse the results. These three components are described below, and the relationships between them are illustrated in Figure 3–1.

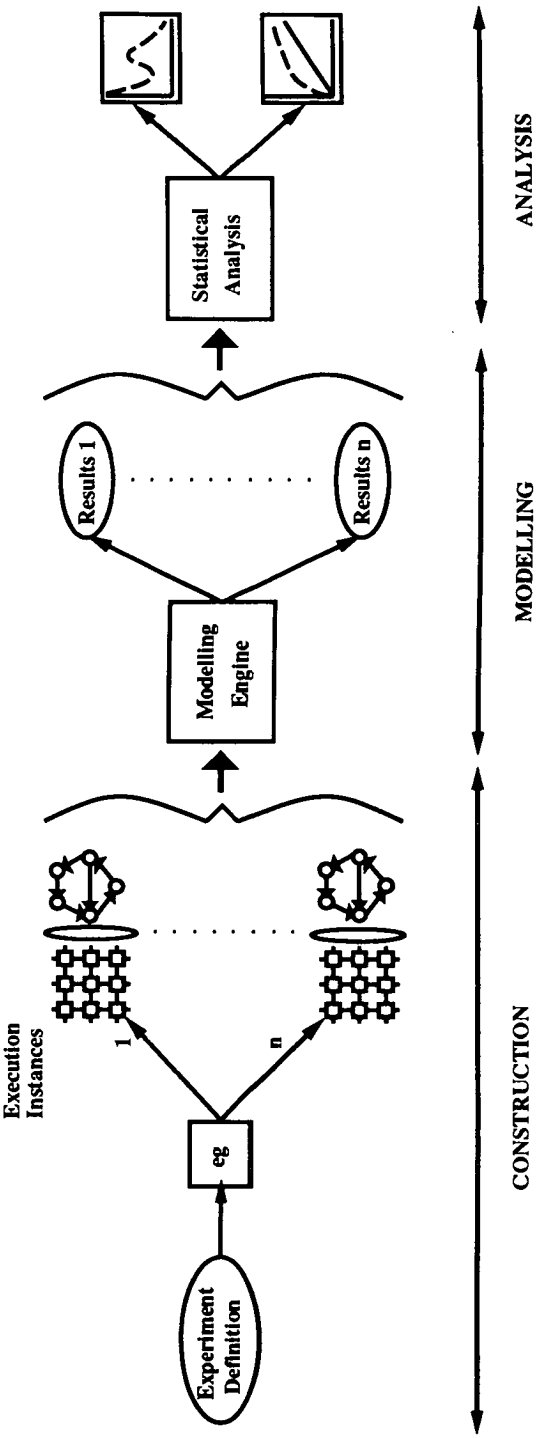

Figure 3–1: The Experimental Framework

3.2.1 Experiment Construction

An experiment is defined as the task of investigating the effect of varying certain parameters characterising a parallel computation on its run time behaviour. In order to construct an experiment, one has to define exactly what it is that one wants to model; this can be done using an *experiment definition.* The experiment definition states in a formal manner which parameters are to be varied, and how they are to be varied within an experiment. A single experiment definition would normally define a set of programs to be modelled. Once an experiment has been defined, the experiment generator is invoked to read the experiment definition and produce a series of *Execution Instances* (EIs), one for each program to be modelled. Each EI contains all the information required to model a single parallel program. These two stages are described more fully below.

Defining an Experiment

The Experiment Definition Language (EDL) allows one to specify at a reasonably high level a set of parallel computations comprising a single experiment. An EDL script is divided into two parts. There is that part which is concerned with specifying the structure of the process graph and processor graph to be modelled, and a mapping between the two. This part of the script specifies the *execution parameters* which define what is to be modelled. The second part of an EDL script is concerned with controlling the actions of the modelling engine. These parameters are referred to as the *modelling parameters*, and they define how to model the programs.

Those execution parameters defining the process graph structure and the mapping can be varied in order to define an experiment. However, to simplify matters the processor graph must remain constant within a single experiment. A variable parameter can be set to a fixed value (e.g. 7), an arbitrary list (e.g. [1.5, 4.75, 5.25]) or a list of equally spaced values (e.g. 3 to 5 step 1). Currently, the lan-

Generation Technique	Degree Range	Node Range
Redfield	3-5	0-160
Random Regular	0-30	0-160
Irregular	0-30	0-160

Table 3–1: Random Graph Generation Strategies

guage can only support full factorial experiments i.e. all possible combinations of all variable parameters are considered. There is no way to restrict the parameter combinations generated. An example EDL script is presented in Figure 3–2.

Lines 3-8 define the graph parameters. First of all the structure of the process graph is specified. A number of techniques for generating regular and irregular random graphs have been implemented; they are listed along with their degree and node ranges in Table 3–1. The Redfield [104] and Random Regular [70] methods produce regular graphs, in the sense that all nodes are of the same degree. The Irregular method produces graphs where, although the nodes have different degrees, the mean degree over all the nodes in the graph can be specified. In this example the Redfield graph generation technique is selected. The degree and size of the process graph to be used is then defined. The number of nodes is varied between 12 and 24 in steps of 4, thereby specifying a number of process graphs. The degree could also be varied if desired. Alternatively, one could supply a user-defined process graph, in which case the number of nodes and the degree of the process graph would be held constant for the duration of the experiment. The hardware is configured to a 4 by 3 mesh of processors. A number of topologies are available; for example, mesh, hypercube, star, ring, pipeline, tree, random or user-defined.

Lines 10-15 specify a number of process definitions. Note that each process definition uses the same process template when modelled, since only a single template can be specified for each individual experiment. A process definition has a name and a set of named numeric parameters which characterise a particular type

```
 1  Begin Experiment
 2
 3  Begin Graph Parameters
 4    Graph Type redfield
 5    Degree 4
 6    Number Nodes 12 to 24 step 4
 7    Hardware mesh 4 3
 8  End Graph Parameters
 9
10  Begin Define Processes
11    proc1 {
12    Int Param1 [500, 800, 1000]
13    Double Param2 12.5 }
14    proc2 { .....
15  End Define Processes
16
17  Begin Define Channels
18    chan1 {
19    Int Param1 1000 to 3000 step 500
20    Int Param2 50 }
21  End Define Channels
22
23  Begin Allocate Processes
24    map proc1 to 70.0 %
25    map proc2 to 30.0 %
26  End Allocate Processes
27
28  Begin Allocate Channels
29    map chan1 to 100.0 %
30  End Allocate Channels
31
32  Begin Placement
33    Algorithm [1,2]
34  End Placement
35
36  Begin Modelling Parameters
37    Text Domain "MIMD"
38    Double Simulation_Time 20E06
39    Int Simulation_Runs 1
40    Boolean Tracing true
41    Double Snapshot_Freq 1E06
42    Int Process_Template 1
43  End Modelling Parameters
44
45  End Experiment
```

Figure 3–2: An Example Experiment Definition

of process behaviour. In the simple case there will be only a single definition. The numeric parameters can be of type `int` or `double`, and can be varied. In this example the process definition `proc1` has two parameters, one of which is varied over the experiment, and one of which remains constant. No meanings are attributed to the parameters at this stage, this is the task of the modelling engine. Channel behaviours are defined in a similar manner in lines 17-21.

Lines 23-26 allocate process definitions to the nodes of the process graph. Process definitions are assigned randomly about the process graph in the proportions specified. In this example 70% of nodes receive the behaviour defined by `proc1`, and 30% receive that defined by `proc2`. Lines 28-30 allocate channel definitions to the arcs of the process graph in a similar manner. In this way a weighted process graph can be created.

Lines 32-34 define the placement algorithm to be used, this can be varied within an experiment. A number of algorithms have been implemented; for example, round-robin or random placements. These algorithms are coded in the C programming language and so can easily be modified, or new ones added.

The modelling parameters are used to control the way in which EIs are interpreted by the modelling engine, they can be varied in the same manner as the execution parameters. The modelling parameters specified in this example (lines 36-43) are tailored towards a MIMD-based modelling engine. They instruct the modelling engine to run each simulation for 20,000,000 clock cycles of simulated machine time; to turn tracing information on; and to take snapshots of the simulated machine every 1,000,000 clock cycles. Finally, the process template that should be used to interpret the process definitions is specified.

The Experiment Generator

The Experiment Generator (**eg**) is a tool which allows full factorial experiments to be generated from a EDL script. It produces a series of EIs, one for each of

the possible combinations of varying parameters defined in an EDL script. For example, the script in Figure 3–2 would produce $4 \times 3 \times 2 = 24$ EIs. The EIs are generated in a way that is independent of any particular modelling domain; they are simply text files describing the computations to be modelled i.e. the process graph, the processor graph, and a mapping between the two. This tool runs on Sun workstations.

3.2.2 Experiment Execution

In order to execute an experiment, the EIs produced by **eg** must be passed to a modelling engine for processing. A modelling engine reads an EI, models the execution of the computation described by the EI, and reports its results. These can then be processed and combined as part of the experiment as a whole. If desired, particular actions of the modelling engine can be controlled by the modelling parameters.

The modelling engine can be a real parallel program, which is able to emulate the parallel computation specified by the EI in a real domain. Alternatively, the modelling engine can be a simulation system which takes the EI and simulates the specified computation. This is the approach used in this thesis, full details are given in Chapter 4.

It is only at the modelling stage that the process and channel definitions are interpreted. For example, some of the values associated with a process could be interpreted as the computational intensity of that process. Some of the values associated with a channel might be interpreted as the mean length of messages passed down that channel.

3.2.3 Analysis of Results

Most of the experiments described in this thesis were analysed using the general purpose statistical package GENSTAT 5 [31]. In addition, the results presented in Chapter 4 made use of a tool designed specifically in order to analyse two level full factorial experiments (these designs are fully discussed in Section 3.5.2).

3.3 MIMD

MIMD [21,55,120] is a Multiple Instruction stream, Multiple Data stream computer simulation system which has been developed at Edinburgh over a number of years. MIMD allows the execution of arbitrary message passing parallel programs to be simulated.

3.3.1 Overview

MIMD is built on top of DEMOS [11] and Simula [108] and runs on Sun workstations. DEMOS is an extension to the Simula programming language providing class definitions for modelling discrete event simulations. MIMD provides new classes which are appropriate for modelling the execution of message passing parallel programs on distributed memory architectures.

The system has been designed to model correct parallel programs, so features such as deadlock detection have not been provided. MIMD *is not* an instruction-level simulator; rather, computations are represented by time delays and interprocess messages are specified by their packet size. In this manner the computational and communications characteristics of a parallel program can be adequately represented. The programs simulated have no "meaning" in the usual sense of the word, they merely represent certain *patterns of activity.* The system works by ensuring that the control structure of the Simula program constructed is the same

as that of the parallel program to be modelled, accordingly the execution of the Simula program corresponds to the simulation of the original parallel program.

A very attractive feature of the approach described here is that non-intrusive monitoring of the parallel program being simulated can easily be implemented. Although the Simula program is simulating a distributed memory multiprocessor, it is in fact just a conventional sequential program, and so can be suspended at any point in time in order to view the state of the simulated machine. This would not be possible in a truly distributed implementation.

At the moment MIMD is tailored towards simulating occam programs on transputer-based machines. However, the range of languages and machines supported could easily be extended by defining new subclasses to handle different processor characteristics, communications protocols and scheduling strategies.

An experimental validation of MIMD for occam programs running on transputer-based machines can be found in [55]. The simulated execution times for the test programs were found to be within 5% of the values measured on the real machine.

3.3.2 Modelling a Computation

Within a MIMD program, objects representing the components of a parallel computation must be declared i.e the hardware, the program and a mapping between the two.

The hardware is represented by an undirected graph of homogeneous processors joined by bidirectional hard links. A processor is characterised by a relatively small number of parameters such as the processor cycle time and hard link speed. Processors and links are treated as resources which may be claimed and released as and when they are required. Objects representing the processors must be declared and wired into the desired configuration. This can be done by using one of the standard topologies provided within MIMD, or by defining an arbitrary topology.

The program is represented by a directed graph of processes joined by unidirectional channels. Each of these processes may have an arbitrary number of parallel sub-processes. Processes can only engage in four types of behaviour: compute, sleep, send a message, and receive a message. A *compute(n)* statement places a process in the CPU scheduling queue with a requirement for n cycles of CPU time; a *sleep(n)* statement sends a process to sleep for n clock cycles; a *send*(C_i, n) statement sends a message of length n bytes down channel C_i; and a *receive*(C_i) statement receives a message on channel C_i. These statements allow different patterns of activity in message passing parallel programs to be adequately modelled.

Processes are mapped to processors by calling procedures that set up suitable objects describing the mapping. The execution of the resulting Simula program corresponds to the simulation of the parallel computation that is to be modelled.

A mechanism has been provided so as to allow synchronous communications between processes positioned at arbitrary positions within the processor domain. This is implemented by explicitly acknowledging each message sent; while this is rather inefficient, it does provide a convenient and safe framework in which to carry out experiments, especially those concerned with process migration.

The actual time taken for a computation or message transfer is usually longer than the value specified, due to competition for resources. The computational power of the processors and the bandwidth of paths between processors are resources which have to be shared between the competing processes of the concurrent computation. This is achieved by time-slicing processors and queuing messages at links. The mechanisms of process scheduling and message passing can be modelled explicitly at a fairly low level. Time-slicing then occurs as it would in a real system, by maintaining a queue of active processes for each processor. However, it should be noted that context switching overheads are not modelled i.e. context switches are assumed to occur instantaneously. Message passing is implemented using a "store and forward" mechanism where inter-processor message hops are explicitly modelled by passing the data through communications processes running

on each processor. A message has to compete for both CPU time on the processors it passes through, and message transfer time on the hard links separating those processors. Message setup and throughrouting costs have been obtained experimentally, the figures used are 20 microseconds and 10 microseconds respectively. The experiments described in this thesis all use MIMD in the mode described above.

Alternatively, faster simulations can be achieved by using simplified scheduling and message passing mechanisms. Elapsed computation times can be calculated by multiplying estimated execution times by the number of processes active on a given processor. In order to estimate message transfer times a formula must be generated relating message transfer times to the number of intermediate nodes and to the message size. One such formula has been obtained from experiments on a Meiko Computing Surface and is described in [23].

The MIMD system collects a wealth of hardware and software statistics which may be examined in a post-mortem manner. They include information on processor usage, link usage, process activity and channel activity. The user can choose to view only a particular subset of the statistics collected.

3.3.3 Modifications Implemented

As part of the work presented in this thesis a number of improvements and additions were made to the MIMD system, and to the underlying DEMOS package. The most significant of these are described below.

An Improved Random Number Generator

The DEMOS package is provided with a rather old random number generator [39], whose properties are not ideal. According to Jain [69], a good random number generator should exhibit the following features:

- it should be efficiently computable
- the period should be large
- successive values should be independent and uniformly distributed

The linear-congruential generator:

$$x_n = 7^5 x_{n-1} mod(2^{31} - 1)$$

exhibits these properties, and is recommended by Park and Miller in [106], who show that it conforms to a minimal standard. The default random number generator in DEMOS has accordingly been replaced with an implementation of this generator.

Periodic Reporting

Facilities have been added to support periodic reporting at user-defined intervals throughout the simulation, rather than just at the end. This facility enables one to gather information relating to the way in which a particular computation evolves. The amount of information generated can easily be controlled.

Priority Scheduling

A scheduling policy with two levels of priority has been implemented, previously all processes were treated equally. Processes are now classified as being either low or high priority; a high priority process can interrupt a low priority process and seize control of a processor at any point in time. This facility allows MIMD to simulate more closely the functionality of the transputer (an overview of the transputer is presented in Section 3.4.1).

Message passing can now be simulated more realistically, since messages that have to pass through intermediate processors to reach their destinations can be

serviced at these processors without having to queue behind ordinary processes. This is how a store and forward message routing system is typically implemented in distributed memory machines [28].

The enhanced scheduling mechanism also allows one to model operating system-type processes, where a single level of priority is inadequate. System calls usually reflect some degree of urgency and demand rapid responses. For example, the process migration mechanism described below runs at high priority so as to respond as quickly as possible to requests.

A Process Migration Mechanism

To support experiments investigating the behaviour of different process migration policies, a process migration mechanism has been constructed for MIMD.

Once a request is made to migrate a process, a series of resource negotiations between the source and destination processors are simulated. At this stage the destination processor can refuse the process for some reason. For example, assuming that there is a notion of process size in the simulation, the process in question might be too large. Suitable delays for these resource negotiations have been measured experimentally on a transputer-based machine, and are incorporated into the simulation system.

Once the destination transputer has decided to accept the process, it, along with its subordinate processes, must be suspended on the source processor. This is implemented by waiting until the process enters or leave the CPU queue. Once suspended, a message of the appropriate size is passed between the two processors to represent the transfer of the process. Some data structures within the simulation are then updated in order that the process in question will subsequently request compute time from the destination processor rather than from the source processor. Garbage collecting can be modelled probabilistically on the source processor if desired.

In order to ensure transparency, message transfers between the migrant and other processes must take place as if the process had never migrated. This is easy to implement within a simulation, since every process has access to global knowledge, therefore messages can always be correctly routed. However, on a real distributed memory machine, this would not be the case. To allow for this, a message redirection system exactly the same as that used in the real migration mechanism described in Section 3.4.3 is simulated. This ensures that message transfers take place in a realistic manner.

Migrations are permitted to take place concurrently, but only one process can be in transit between any two processors at any time. The migration mechanism described here simulates process migrations in a realistic manner and at a reasonably detailed level. A number of migration policies have been constructed on top of this mechanism; they are described in Chapter 6.

3.4 Transputer-based Process Migration

In order to validate the results of simulation experiments involving process migrations, a migration testbed has been constructed [22]. The machine used for this implementation was the Edinburgh Concurrent Supercomputer (ECS), a transputer-based MEiKO Computing Surface based in the Edinburgh Parallel Computing Centre [1].

The T800 transputer does not provide a particularly attractive environment within which to implement an operating system-type application such as a process

---

[1]The Edinburgh Parallel Computing Centre is supported by major grants from the Computer Board, the Department of Trade and Industry, the Science and Engineering Research Council and Industry.

migration mechanism. For example, it lacks any support for memory protection or exception handling. The construction of a general purpose migration mechanism would have required robust and relatively sophisticated implementations of modules handling such things as dynamic memory management, process suspension and resumption, process transfers, message passing consistency and transparency. Such a project would have been a substantial undertaking in its own right, the decision was therefore taken to concentrate on the implementation of an experimental system.

The approach taken involves loading the code and data for every process onto every transputer. This is possible since occam programs have a static process structure which is known at compile-time. Processes can then be enabled and disabled to reflect the initial placement, and subsequently to reflect process migrations. In this way one avoids the need to worry about many of the robustness and correctness issues that would arise in a full implementation of a migration mechanism. The main disadvantage of this approach is that it is not appropriate for larger applications since the whole program must fit onto a single transputer. It is, however, well suited to an experimental system.

An alternative process migration mechanism for transputers is described by Baker and Milner [6]. Their implementation uses a library call to stop and start processes. This has the side effect of enforcing a fixed parameter set on user processes. More specifically, processes are only allocated a single input and output channel. Consequently, some of the purity of the CSP model is lost, since multiple channels must be multiplexed onto these single channels.

3.4.1 Transputer Overview

Before describing the migration system, a brief overview of the transputer's scheduling mechanism and memory organisation is presented.

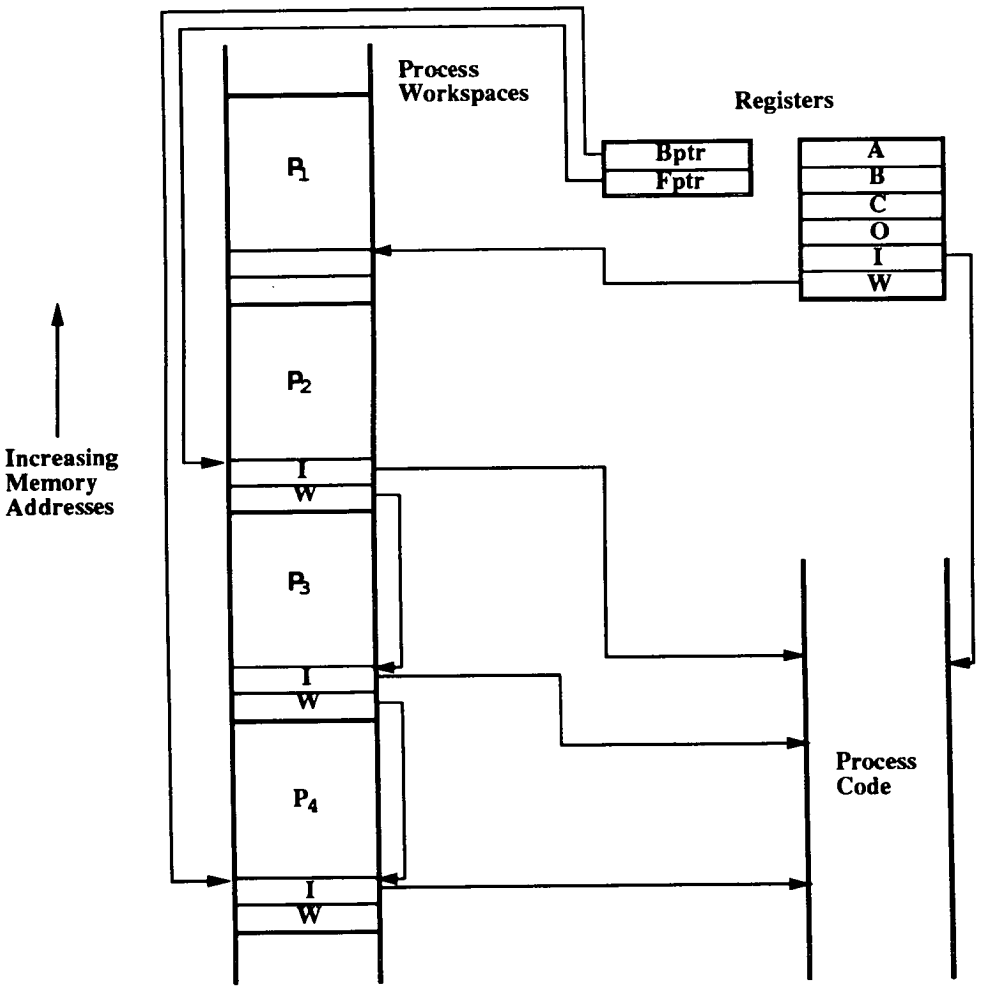

Figure 3–3: A Transputer's Memory and Registers

Processes on the transputer can be run at two levels of priority, high or low. High priority processes run until completion, whereas low priority processes run only when there are no high priority processes ready to run. Low priority processes are descheduled when they block; for example, on a message transfer, or when they have occupied a complete time-slice. To implement this scheme two separate scheduling queues are maintained.

Each process is allocated its own area of memory called its *workspace* to hold local declarations. A process is uniquely identified by its workspace rather than by a section of code, since multiple processes might be executing the same section of code. Subordinate processes are typically created within the parent's workspace, so processes are organised in a nested fashion. Just below each process's workspace there are some words reserved to hold scheduling information. This information consists of the current value of the process's instruction pointer, a pointer to the workspace of the next process in the scheduling queue, and some words used to implement channel transfers and the `ALT` instruction.

The transputer has a small number of registers: a workspace register (W), an instruction pointer (I), an operand register (O), and a three register evaluation stack (A, B, C). In addition to these, there are four registers (Fptr0, Bptr0, Fptr1, Bptr1) pointing to the front and back of each of the respective scheduling queues.

Figure 3–3 shows the transputer executing the processes $P_1 - P_4$ in parallel. It is assumed that all the processes are being executed at the same priority level (either low or high). The process P_1 is currently being executed, so the workspace register is pointing to its workspace, and the instruction pointer register is pointing to its current instruction. The remaining processes are descheduled and waiting in the queue pointed to by Fptr and Bptr. Below the workspace of each of these processes is stored the scheduling information i.e. the process's instruction pointer and a pointer to the next process in the queue. In this diagram the words used to implement channel transfers and the `ALT` instruction are not shown.

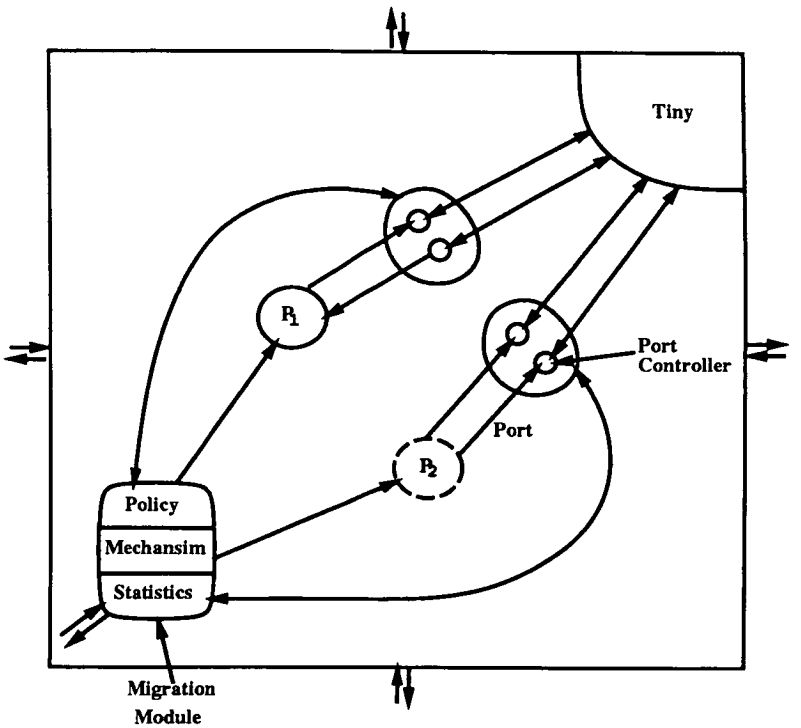

Figure 3–4: An Example Process Configuration

3.4.2 Process Configuration

Figure 3–4 illustrates an example process configuration on a transputer running the migration system. Every transputer in the system will have an identical process structure. However, the transputers will differ in which user processes are enabled and disabled. In this example there are two user processes, P_2 is drawn with a dotted line to indicate that it is disabled on this transputer. Each transputer runs a migration module which cooperates with migration modules on other transputers in order to make and execute migration decisions.

A user process has a number of *ports* over which it can send or receive messages. All messages are sent using the communications harness Tiny [27], and there is a Tiny routing kernel resident on each transputer. Tiny can be used to implement a true virtual channel mechanism if each port is used to communicate with only one other process. Alternatively, Tiny can be used to implement a more general com-

munications structure where messages can be sent to (or received from) arbitrary processes on a single port.

In general, one is free to use any number of processes and an unrestricted communications topology, subject to the constraint that the entire program must fit onto a single transputer.

Messages sent (or received) on a particular port are passed to (or from) Tiny via a dedicated *port controller*. These transfers have been optimised to use pointers rather than ordinary channels. The port controllers are responsible for enforcing a synchronous communications protocol for processes positioned at arbitrary points within the transputer array. This is done by using control signals to explicitly acknowledge messages in a similar way to that described in [112]. The port controllers are also responsible for redirecting messages incorrectly addressed to disabled processes, and for collecting statistics on message transfers. This functionality is fully described in Sections 3.4.3 and 3.4.4.

The synchronous communications protocol used means that deterministic programs will produce the same results, with or without process migrations. However, non-deterministic programs might not, since the detailed ordering of events will be disturbed by migrations. This would be an undesirable feature in process migration experiments, since one would like to be able to compare the performance of programs being executed with and without process migrations. If two executions of the same program gave different results, it would not be possible to make a valid comparison. Consequently, it is assumed that candidate programs do not use the occam `ALT` construct.

The following sections describe the migration and statistics mechanisms implemented. Different migration policies can easily be incorporated into the testbed, and indeed, several have been implemented (see Chapter 6).

3.4.3 The Migration Mechanism

To simplify matters, it is assumed that a candidate program consists of some initialisation code followed by a number of self-contained, top-level processes running in parallel at low priority. In a sense, these processes exhibit many of the characteristics of objects, having clean well-defined boundaries (as noted in Chapter2, objects are generally thought to simplify the construction of a migration mechanism). It is the top-level processes which are the candidates for migration; subordinate processes cannot be migrated independently of their parent processes. It is not necessary to consider the situation where a migration fails, since it is known that there are always adequate resources (due to the fact that the entire program must fit onto a single transputer in order to execute in the first place). In the migration mechanism described here, there are three stages involved in migrating a process. Firstly, the process must be suspended; secondly, it must be transferred to its new home and restarted; and lastly, any messages destined for the process's old home must be redirected. Before describing how these three stages are implemented, the modifications that must be made to a user program before it can be considered a suitable candidate are discussed.

Modifications to User Code

Figure 3–5 illustrates the sorts of modifications that must typically be made to a user's code, they are relatively minor and only affect the top-level processes. It is assumed that the behaviour of each top-level process is encapsulated within a procedure. The steps then involved are described below:

1. Two extra parameters must be introduced into each top-level procedure. A channel `migSignal` must be supplied, down which control signals are sent from the migration module. In addition, an array, `wSpaceInfo`, must be passed. This is used to return information regarding the size and position of the workspace used by this particular process.

```
PROC userCode(<parameter list>)
  INT firstDeclaration:
  ... other declarations
  INT lastDeclaration:
  SEQ
    ... initialisation
    WHILE <condition>
      SEQ
        ... action 1
        ... action 2
           etc ...

                ⇓

PROC userCode(<parameter list>,
              CHAN OF ANY migSignal, []INT wSpaceInfo)
  INT firstDeclaration:
  ... local declarations
  INT lastDeclaration:
  SEQ
    ... report workspace and suspend
    ... initialisation
    WHILE <condition>
      SEQ
        ... suspend if received signal
        ... action 1
        ... action 2
           etc ...
```

Figure 3–5: Modifications Required to User Code

2. Each user process must be modified so that the first statement that it executes is to return its workspace requirements in the parameter `wSpaceInfo`. Although known, this information is not readily available at compile-time. A simple procedure is provided for this purpose; for the example in Figure 3–5 it would be called as follows:

   ```
   report(firstDeclaration,lastDeclaration,wSpaceInfo)
   ```

   For the mechanism to work correctly, one has to ensure that all variables within a process's workspace are stored at contiguous locations in memory. On a Meiko Computing Surface running OPS this involves setting the compiler parameter *separate.vector.space* to False. The area of memory defined by the first two parameters in the above call to `report` can then be assumed to include all variables defining the state of the process.

   Once the workspace information has been supplied, processes suspend themselves using the `STOPP` assembly language command. Each migration module is then responsible for enabling those processes which it initially has control of, this is done with the `RUNP` assembly language command. The initial placement is user-defined.

3. User processes must be modified to periodically check the channel `migSignal` to see if they have been requested to suspend. This technique is fully described below.

Suspending A Process

The problem of suspending a process on the transputer at some arbitrary point is non-trivial [72]. There are a number of operations that a process can be carrying out that make it difficult to capture its state in a straightforward manner. For example, the process could be involved in an off-chip message transfer, it could be dependent on a timer channel, or it could be using the Floating Point Unit.

A solution to this problem is described in [6]. It involves placing the emphasis on the user processes, they must be modified to periodically check whether they have been requested to suspend. If a process detects a suspend signal it must immediately execute the `STOPP` assembly language command in order to suspend itself.

It is the responsibility of the user to embed these checks at suitable positions within the code. For example, the user should avoid attempting to migrate a process which is currently dependent on a local timer. The checks *must* occur in the top-level thread of the process in order to enable execution to be suspended in a clean manner. In Figure 3–5 a single check has been included, although multiple checks are also possible. The frequency of these checks determines the sensitivity of user processes to migration requests, and hence the performance characteristics of the migration mechanism.

Transferring A Process

The state of a suspended process will be completely defined by the contents of its workspace and its instruction pointer. It is known that this is true because a suspended process cannot be executing a user's `ALT` statement, since it has been assumed that a candidate program will not contain any `ALT` statements; and it cannot be engaged in a message transfer, since the process has just suspended itself. To migrate a suspended process, the migration mechanism on the source transputer must transfer the workspace defining the current state of the process to the migration mechanism on the destination transputer, where it can be overlayed in the correct position in memory. In addition, the current value of the instruction pointer must be passed to the destination transputer so that the migrated process will resume execution at the correct instruction. The contents of the process's instruction pointer will be stored one word below its workspace on the source transputer as a consequence of the transputer's scheduling mechanism;

it can therefore be easily transferred. The migration mechanism on the destination transputer must then start the process by using the `RUNP` assembly language instruction.

Message Redirection

Inter-process communications must continue to take place correctly, even after many process migrations, in order to ensure transparency. This is implemented by ensuring that each transputer keeps a record of where it thinks each process in the system is currently positioned. This record is updated every time a message or message acknowledgement is received (by piggy-backing the id of the source transputer on the message or message acknowledgement). In this way a transputer will tend to know the correct positions of those processes with whom its processes communicate; this is sufficient to enable messages to be, in general, correctly addressed. This strategy is similar to the one used by the Emerald system [74]. It exhibits manageable residual dependencies, being neither the most efficient nor the most inefficient in this respect.

When a message is sent, it is forwarded to the transputer where the target process is thought to be. A redirection mechanism is built into each of the port controllers. When a process is enabled, the port controllers are responsible for implementing the synchronous message passing protocol. However, if a process is disabled and one of its port controllers receives a message, it redirects it to the transputer where it thinks the target process is currently enabled. If this proves to be the wrong location, then the message will be repeatedly redirected until it reaches the active instance of the target process. At this point the sender's transputer will be updated with the current location of the target process (via a message acknowledgement) and message passing between these two processes will then take place correctly until one of them migrates.

If processes are allowed to migrate too often, then thrashing might occur and messages will never reach their targets. It is the user's responsibility to ensure that a reasonable period of time is left between process migrations so that the system is allowed to enter a steady state.

3.4.4 Statistics Collection

A statistics collection mechanism has been implemented so that intelligent migration decisions can be made. Information is collected on processor utilisation, process activity and channel activity over a user-defined interval. The transputer does not have any monitoring support built in, so one has to rely on a number of tricks in order to implement these facilities.

The method described in [71] is used to monitor processor utilisation. The technique consists of two stages, a calibration phase followed by a measurement phase, both of which run at low priority. The calibration phase runs in isolation and measures how fast the transputer's scheduling mechanism is. The measurement phase then runs in parallel with the user processes and calculates the percentage utilisation over a given time period. This is achieved by noting how often the low priority process queue is found to be empty.

The approach described in [77] is used to monitor the activity of processes. The processor utilisation monitor described above has been extended so that it also collects information on the amount of computation achieved by user processes. Only the migration candidates (i.e. the top-level user processes) need to be monitored, and it is known that these will be running at low priority. When the utilisation measurement process detects that the low priority queue is not empty, it scans the queue noting which top-level processes the subprocesses in the queue belong to. This is easily done since subprocesses are stored within their parent's workspace, and the area of memory occupied by each top-level process is known. If:

- P is a top-level process,
- T is the elapsed time since the measurement process was last active,
- S is the total number of subprocesses found in the low priority queue in the previous cycle, and,
- S_P is the number of subprocesses of P found in the low priority queue in the previous cycle,

then an estimate for the total amount of computation achieved by P since the measurement process was last active can be written as:

$$T \times \frac{S_P}{S}$$

This is only a crude measure since individual subprocesses may not be active for the same length of time. Some are likely to become descheduled before the end of their time-slice, for example, by blocking on a channel transfer or by terminating. This effect can be minimised by ensuring that the top-level processes have a sequential internal structure as far as possible. Generally, the results will be more accurate if the processes are dominated by computation rather than communications. A detailed analysis of the errors involved can be found in [77].

Each port controller is responsible for collecting information on the number and total volume of messages sent to individual processes. This information can be made available to the statistics collection mechanism whenever necessary.

3.5 Statistical Techniques

This thesis aims to investigate in a quantitative and empirical manner the relationships which exist between the structure of parallel programs and their performance in a given environment. This naturally leads me to consider statistical analysis techniques. This section describes the main statistical techniques used. The discussion is not exhaustive, since I felt that some details and techniques were best presented in context. A general appreciation of the material presented here is necessary in order to facilitate the understanding of subsequent chapters.

3.5.1 Design and Analysis of Experiments

The aim of an efficient experimental design is to gain the maximum information with the minimum number of experiments. Various designs have been proposed, and a number are discussed in Section 3.5.2. In order to draw conclusions from an experiment, appropriate analysis techniques must be applied. Several such techniques are discussed in Sections 3.5.3 and 3.5.5.

The design and analysis of experiments is a subject that has been addressed by a large number of text books, see [18,30,93] for example. These books tend to draw examples from the fields of agriculture, manufacturing and chemical engineering, since this is where these methods have been most widely used. For a discussion of their applicability to the performance evaluation of computer systems see [69]. These techniques offer a number of attractive features that would be difficult to obtain using experiments designed in an ad-hoc manner. Firstly, they allow quantitative estimates to be made of the relative importance of factors affecting performance. Secondly, they allow one to examine the importance of any interactions that might exist between two or more factors. Lastly, they allow one to investigate whether observed differences in behaviour can be explained in terms

of the factors explored within the experiment, or whether they must be attributed to the effects of uncontrolled parameters.

A brief summary of the terminology used in the design and analysis of experiments is now presented.

- An experiment consists of a number of *trials.*
- The *response variable* (or just *response*) is a measure of the outcome of a particular trial. This would normally be a performance metric for studies of computer systems.
- The quantities that are under experimental control are known as *factors*, *parameters* or *predictor variables.* We are generally interested in how the factors affect the response variable.
- The values that a factor can assume are called its *levels* or *treatments.* These levels can be quantitative or qualitative.
- The *effect* of a factor is defined to be the change in the response produced by a change in the level of the factor.
- An *interaction* is said to exist between two factors A and B if the effect of one depends upon the level of the other.
- Experiments are often *replicated* in order to isolate experimental errors.
- In order to analyse an experiment, it is must be assumed that the measurements can be adequately characterised by a particular *model.*
- The differences between the observed values of the response and those predicted by a particular model are known as *errors* or *residuals.*

An experimental design involves specifying a set of factors, a number of levels of each factor, a definition of the particular factor level combinations to be considered, and an indication of the number of replications of the experiment to be carried out. The particular class of experimental designs used in this thesis are known as factorial designs.

3.5.2 Factorial Designs

Factorial designs offer an efficient way of studying the effects of two or more factors. The general principle is to consider the observed response at a number of factor level combinations. The major advantage over just varying one factor at a time, is that this technique allows any interactions that may be present to be detected.

A *full factorial* design considers all combinations of every possible factor level. However, these designs can be costly in terms of the quantity of trials required if the number of factors (or number of factor levels) is not relatively small. A special case of the full factorial design, which is often found to be useful in exploratory experiments, is the two level full factorial design. In such designs, only two levels of each factor are considered. However, a great deal of information can be derived relatively quickly by ensuring that factor levels are chosen correctly. If a factor is quantitative, the levels should be chosen to be extreme values so that the likely possible values of the factor will fall in the range explored. If a factor is qualitative, then only two states can be considered; for example, on and off. A two level full factorial design with n factors using r replications is known as a $2^n r$ design. It should be noted that two level full factorial experiments can only consider linear relationships between the predictor variables and the response. More complex relationships can be investigated if one considers more levels.

A *partial factorial* design can be used if the number of trials required for a full factorial design becomes too large, yet one still wishes to investigate more than two levels for certain factors. In such a design, only a subset of the possible factor level

combinations are considered. This saves time, but less information is obtained. For example, it may not be possible to assess all possible factor interactions.

This thesis uses full factorial designs, with particular emphasis on two level full factorial designs.

3.5.3 Analysis of Variance

The results of a full factorial experiment can be analysed using a conventional analysis of variance (ANOVA) [115]. The analysis of variance is a useful tool for obtaining quantitative estimates of the relative importance of a set of factors, with respect to the observed values of some response variable. The factors in question are assumed to be qualitative in nature, although some may have a correspondence to a numerical scale. However, the analysis is not concerned with these numerical values. Rather, it is concerned with analysing the effects of changing factors from one level to another. For the experiments presented in this thesis, I am generally concerned with investigating the relative importance of a set of parameters characterising the behaviour of a parallel program, with regard to an appropriate performance metric.

The general model underlying an analysis of variance is illustrated below for two factors, A and B:

$$y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_{ij} + \epsilon_{ijk} \tag{3.1}$$

where y_{ijk} represents the observed value in the kth replication with factor A at level i and factor B at level j. The mean response is represented by μ; α_i is the effect of factor A at level i; β_j is the effect of factor B at level j; γ_{ij} is the effect of the interaction between factor A at level i and factor B at level j; and ϵ_{ijk} is the experimental error. This model can be extended in a straightforward manner to an arbitrary number of factors.

There are, however, several assumptions underlying an analysis of variance which must be satisfied in order for the analysis to be valid. These assumptions are listed below:

1. Errors must have constant variance over the entire range of the response i.e. homoscedasticity of variance.

2. Errors must be Independent and Identically Distributed (IID) normal variates with zero mean.

3. The effects of factors and errors must be additive i.e. the underlying model must be structurally adequate.

These assumptions should always be tested before proceeding with an analysis. The first assumption can be tested by constructing a scatter-plot of the residuals versus the predicted response. If a definite trend is visible in such a plot, then one can conclude that the errors do not have constant variance. The second assumption can by tested by preparing a normal quantile-quantile plot of the residuals. In general, a quantile-quantile plot allows one to test whether a set of observations comes from a particular distribution by plotting the observed quantile versus the theoretical quantile. An approximate straight line in such a plot indicates that the observed data does indeed come from the theoretical distribution; see [69] for full details. In the case of an analysis of variance, if the points plotted do not form an approximate straight line passing through the origin, then the errors cannot be described by a normal distribution with a mean of 0. There are a number of situations that would lead one to question whether the third assumption was satisfied. The first of these is if the residuals were observed to be of the same order as the response. Another is if the response covered more than a single order of magnitude.

If any of the assumptions are discovered to have been violated, then a transformation of the response should be considered. Several transformations found to

be useful with respect to the experiments presented in this thesis are described in Section 3.5.4.

The analysis of variance partitions the total variance observed in the response variable into its constituent parts. This variation is quantified using a measure known as the *sum of squares* which, given a set of observations, is defined as the sum of the squares of the differences between the individual observations and the mean value of those observations. The *mean squares* for a particular term in the model can be calculated by dividing the sum of squares by the number of degrees of freedom for that term. In order to test the significance of a particular term, with respect to the variation observed in the response variable, one should divide its mean squares value by the mean squares value for the error term. The corresponding quantity will have an F distribution, so one can then carry out formal statistical tests.

It is clear from the above discussion that replications are essential in order to carry out an analysis of variance, since there would be no error term if there was only one replication. In fact, the estimate of error becomes a basic unit of measurement for determining whether observed differences in the data are really *statistically* different.

In undertaking an analysis of variance, one would hope that much of the variation due to high order interaction terms would turn out not to be significant. This would then enable higher order interactions to be disregarded in further experiments.

3.5.4 Transformations

A transformation of the response variable can often help if the assumptions underlying an analysis of variance have not been satisfied [4] [2]. The analysis can then proceed in terms of the transformed response. The penalty for using a transformation is a loss of clarity in interpretation.

Although any transformation may be applied, there are a number described in the literature that have been found to be generally useful. A selection appropriate to the experiments described in this thesis is now presented.

The Box-Cox Transformation

Box and Cox [17] describe a technique which can be used to estimate the exponent, λ, to use in an arbitrary power transformation. The method uses a family of transformations defined as follows:

$$w = \begin{cases} (y^{\lambda} - 1)/(\lambda g^{\lambda-1}) & \lambda \neq 0 \\ g \ln(y) & \lambda = 0 \end{cases} \tag{3.2}$$

In this definition, y is the original response and g is the geometric mean of all the original responses. To estimate the value of λ which should be used in a power transformation of the form y^{λ}, a standard analysis of variance should be carried out on w, for various values of λ. The maximum likelihood value for λ, is that for which the residual sum of squares from the model fitted to w is minimised. The Box-Cox transformations attempt to satisfy the normality, homoscedasticity of variance and additivity criteria simultaneously. GENSTAT can be used to calculate the optimal value of λ.

---

[2] However this is not always the case, it may be that the simple additive model underlying the analysis variance is not adequate, and that more sophisticated techniques are required.

If the original responses contain some negative values then Definition 3.2 needs to be adjusted by a constant, c, to give:

$$w = \begin{cases} ((y+c)^{\lambda} - 1)/(\lambda g^{\lambda-1}) & \lambda \neq 0 \\ g \ln(y+c) & \lambda = 0 \end{cases} \tag{3.3}$$

Values of c and λ which minimise the residual sum of squares can be estimated simultaneously, the resulting power transformation will be $(y+c)^{\lambda}$.

Atkinson [4] devotes a chapter to the study of transformations for response variables which are expressed as percentages (or proportions). He suggests that the transformation $(1 - y/100)^{\lambda}$ is often useful for responses where the observed values are close to 100%. A suitable value of λ can be estimated using the technique described above.

The Guerrero and Johnson Transformation

The Guerrero and Johnson transformation [4,54] is particularly suited to response variables which are expressed as percentages; it involves the application of the Box and Cox power transformation to $y/(100-y)$ rather than y. A Guerrero and Johnson transform takes the following form:

$$y' = \left(\frac{y}{100-y}\right)^{\lambda} - 1 \tag{3.4}$$

where λ denotes a suitable exponent and y is the original response. The value of λ to be used can be estimated using the following family of transformations:

$$w = \begin{cases} \{(Y/(1-Y))^{\lambda} - 1\}/\lambda G\left(Y^{\lambda-1}/(1-Y)^{\lambda+1}\right) & \lambda \neq 0 \\ \ln(Y/(1-Y))G(Y(1-Y)) & \lambda = 0 \end{cases} \tag{3.5}$$

In this definition, G is the geometric mean operator and $Y = (y/100)$, i.e. Y represents the response expressed as a proportion rather than a percentage. The maximum likelihood estimate of λ, is that for which the residual sum of squares from an analysis of variance carried out on w is minimised.

3.5.5 Analysis of Covariance

An analysis of covariance (ANCOVA) [29,66] allows the predicted value of the response variable to be adjusted for the effects of one or more *nuisance* variables; these are referred to as covariates since they normally vary along with the response. This technique is used in Chapter 5 in order to construct an improved performance prediction model incorporating information relating to the dynamic properties of programs, i.e. the complex interactions which occur at run-time between a program and the underlying machine. In the literature, covariates typically relate to environmental effects, since examples usually come from agriculture or the behavioural sciences, and, as such, are generally considered to be outside the control of the experimenter[3].

Let us consider the case of the full factorial experiment. A single covariate, x, can be incorporated into the model specified in Equation 3.1 to give:

$$y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_{ij} + \delta(x_{ijk} - \overline{x}) + \epsilon_{ijk} \tag{3.6}$$

where x_{ijk} is the measurement made on the covariate x in the kth replication with factor A at level i and factor B at level j. The mean of all the x values is given by $\overline{x}$, and δ is a linear regression coefficient indicating the dependency of y_{ijk} on x_{ijk}. The above model can be extended, in a straightforward manner, to support an arbitrary number of covariates.

As well as the usual assumptions regarding the distribution of the errors, there are a number of extra assumptions underlying Equation 3.6 which should be satisfied in order for an ANCOVA to be valid. Firstly, the true relationship between the response variable and the covariate must be linear and of non-zero gradient. A gradient of zero would imply that the covariate did not vary with the response

---

[3]In fact, as we shall see in Chapter 6, in my case the values of covariates can be controlled to some extent, but this need not concern us at the moment.

at all. Secondly, the regression coefficients associated with individual treatment groups should be equal. This is implicit in the model, since there is only one possible value of δ, which is a pooled estimate across all treatment groups. Finally, for the most unambiguous interpretation of an ANCOVA, the covariate should be relatively independent of the treatments. This condition is not mandatory, however, care should be taken in interpretation when the treatments affect the covariate, since the covariate might either remove or introduce differences in the response that might be misinterpreted as treatment effects.

If the value of a covariate is measured before an experiment begins, then there will be no possibility of the treatments affecting the value of the covariate. For example, in an agricultural experiment, the concentration of fertiliser detected in a plot of land before treatments are applied might be used as a covariate, since this factor would be likely to influence the results obtained between different plots. Alternatively, the covariate might be measured during or after the experiment, in which case there is always the possibility that the treatments will affect the covariate.

The distinction between an analysis of variance, an analysis of covariance and a conventional regression analysis is somewhat blurred. In the general case, an analysis of covariance can be considered to be a combination between an analysis of variance, which relates to a set of qualitative factors, and a straightforward regression analysis, which relates to a set of quantitative factors. However, this classification is not a rigid one, since, as we will see in Chapter 5, an analysis of variance can be extended to handle quantitative factors and produce regression models through the method of orthogonal polynomials.

3.5.6 Paired T tests

A paired t test is a standard statistical technique used to compare two alternatives. The aim of the test is to decide whether the mean values observed under each of

two treatments differ significantly. This technique is used in Chapters 6 and 7 in order to compare the observed performance of a set of programs, executed both with and without a process migration strategy active.

There are two possible types of test: in a two tailed test one is simply testing for a difference in means; in a one tailed test, one is looking to see if one mean is significantly larger than the other. The observations must be paired in the sense that there should be a one to one correspondence between observation i under the first treatment and observation i under the second treatment.

The test statistic can be calculated using the following formula:

$$t = \frac{\overline{D}}{s_D/\sqrt{n}} \tag{3.7}$$

where $\overline{D}$ is the mean of the differences between observations under the two treatments, s_D is the standard deviation of the differences between observations, and n is the number of pairs of observations. The t test assumes that the differences are normally distributed. The critical value, for a given significance level, can be looked up in a table of t values using $n-1$ degrees of freedom.

3.6 Summary

This chapter has discussed a methodology for constructing synthetic programs, and described an experimental framework in which the behaviour of such programs can be investigated. An overview of the functionality of the MIMD simulation system has been given. Process migration mechanisms for MIMD and transputer-based machines were described. Finally, a number of useful statistical analysis techniques were presented.

In the following chapter, I investigate the feasibility of using analysis of variance techniques in order to explore the performance characteristics of a particular class of parallel programs.

Chapter 4

Performance Analysis: A Statistical Approach

This chapter describes a methodology which can be used to investigate the performance of parallel programs from a statistical perspective. To illustrate the application of the technique, a particularly simple class of regular parallel programs is investigated; more realistic programs are studied in subsequent chapters. The proposed methodology makes use of standard statistical techniques in order to ascertain the relative importance of parameters characterising the behaviour of parallel programs. The techniques developed in this chapter are used frequently throughout the remainder of this thesis.

Section 4.1 discusses the motivation behind this work. Section 4.2 describes the particular class of programs investigated here. The experiments themselves are presented in Section 4.3, and some validation results regarding the experimental approach are given in Section 4.4. Finally, a summary and conclusions are presented in Section 4.5.

4.1 Motivation

The standard methods of experimental design and analysis have not been used frequently in computer performance analysis studies, although there are a small number of examples of their successful application, see [89,97] for example. The interactions that occur between a parallel program, the run-time software and the underlying machine are, generally speaking, complex and rather poorly understood. This makes it difficult to construct accurate analytical models. However, it may still be possible to obtain empirical models which are derived purely experimentally. The principles of experimental design were proposed specifically in order to aid such investigations. Therefore, I shall use these techniques, and adopt a conventional experimentalist's approach.

In order to increase our understanding of parallel programs, it would be desirable to be able to identify those program characteristics which have the greatest impact on performance. I am interested in the macroscopic properties of parallel programs, rather than in information relating to the detailed ordering of events. Consequently, I shall use the methodology described in Section 3.1, and will work in terms of program classes characterised by small numbers of easily measured parameters, which summarise time-averaged properties. A particular program instance can be defined by specifying the values of the program parameters. This high level approach tends to lead one to consider the features that programs have in common, as opposed to concentrating on their differences.

The methodology described here uses two level full factorial experiments in conjunction with analysis of variance techniques. Assuming that the necessary conditions underlying the analysis of variance are satisfied, and that the parameters selected adequately characterise performance, this approach allows quantitative estimates of the relative importance of program parameters to be obtained.

The experiments presented in this chapter were inspired by an exploratory experiment conducted by Candlin *et al.* [20]. My approach is similar to theirs in the sense that I use the standard methods of experimental design and analysis; and I use a related program model and similar program parameter set. However, my experiments are considerably more rigorous, and I extend their work in a number of ways. Firstly, as discussed in Section 4.2, I propose a framework for classifying loosely synchronous data parallel programs, and generalise the program model adopted. Secondly, I consider several different performance metrics. Thirdly, I explore the adequacy of the model underlying the analysis of variance in this case; and study the use of transformations of the response variable where this helps to satisfy the conditions for a valid analysis. Lastly, I consider a number of different processor topologies.

4.2 A Class of Parallel Programs

A model of *loosely synchronous data parallelism* has been used in the experiments described in this thesis. This form of parallelism is frequently used for problems defined over some data domain. This domain can often be divided into sections, which can then have similar operations applied to them in parallel. Examples of problems that can be solved in this manner include molecular dynamics simulations, finite element analysis and the solution of partial differential equations. Indeed, in [51] Fox found that 76% of the parallel programs investigated in a survey could be classified as belonging to this class. Typically, in order to calculate the results at one point in the data domain, knowledge is required of other, often neighbouring, points. Consequently, these types of programs generally exhibit iterative behaviour patterns, with the actions of a process being characterised by a computation phase, where local results are calculated, followed by a communications phase, where data is exchanged between a set of neighbours.

Two strategies which are often used to decompose a loosely synchronous data parallel problem are the *geometric decomposition* and the *scattered spatial decomposition.* These two strategies are related, but differ in the granularity of processes that they generate. In a geometric decomposition, processes are usually of a rather large granularity, the problem space being divided up into a relatively small number of regular sections. As the sizes of the processing grains used increases, the amount of communications required decreases relative to the amount of computation that must be achieved between synchronisation points. However, load-imbalance can occur if processing grains grow too large, since some data sections might be heavily populated whereas, others might be relatively empty. In a scattered spatial decomposition, smaller processing grains are used, and they are allocated randomly to each processor in the system. The intention is that, on average, processors will have an approximately equal amount of work to do. However, as the grain size decreases, the communications overheads increase. There is, therefore, a trade-off between the size of processing grain used, and the communications overhead and degree of load imbalance encountered.

The experiments described in this thesis assume a model of loosely synchronous data parallelism, but they are independent of any particular decomposition strategy. Before describing the program model and parameter set used, a classification scheme for loosely synchronous data parallel programs is presented.

4.2.1 Loosely Synchronous Data Parallelism: A Classification Scheme

Unless otherwise stated, one can assume that the programs being discussed in the remainder of this thesis exhibit loosely synchronous data parallelism. Such programs can be classified according to their temporal and spatial characteristics. Those programs whose computational and communications patterns vary significantly over time can be referred to as *time-varying* parallel programs; those whose

patterns do not vary significantly over time can be referred to as *time-invariant* parallel programs. A more formal definition is now given, the constants α and β determine the sensitivity of the classification. It is assumed that an iteration corresponds to that activity which occurs between successive synchronisation points.

Definition 4.1 *Assume that a program consists of a fixed set of processes, PROC, and a fixed set of channels, CHAN. Let* $COMP_{p_i}$ *represent the computational requirements of process p during iteration* i $(i = 1,2,..)$, *and similarly let* $MESS_{c_i}$ *represent the size of message sent down channel c during iteration i. If,* $\forall\ i$ *and* $\forall\ p \in PROC$, $COMP_{p_i}$ *is within the range* $[COMP_{p_1} - \alpha, COMP_{p_1} + \alpha]$ *and,* $\forall\ i$ *and* $\forall\ c \in CHAN$, $MESS_{c_i}$ *is within the range* $[MESS_{c_1} - \beta, MESS_{c_1} + \beta]$, *then a program can be classed as being time-invariant. Otherwise it can be classed as being time-varying.*

Within these two classes, programs can be further sub-divided, with respect to their spatial characteristics, into those displaying *uniform* behaviour and those displaying *non-uniform* behaviour. Programs displaying uniform behaviour generally act on regular problem domains, and so, at any point in time, processes will have approximately equal amounts of work to do and channels will be evenly loaded. On the other hand, non-uniform programs operate on irregular problem domains, so processes will typically have different amounts of work to do, and some channels will be more heavily loaded than others. A more formal definition is now given, the constants γ and δ determine the sensitivity of the classification.

Definition 4.2 *Assume that a program consists of a fixed set of processes, PROC, and a fixed set of channels, CHAN. Let* $COMP_{p_i}$ *represent the computational requirements of process p during iteration* i $(i = 1,2,..)$, *and similarly let* $MESS_{c_i}$ *represent the size of message sent down channel c during iteration i. If,* $\forall\ i$ *and* $\forall\ p \in PROC$, $COMP_{p_i}$ *is within the range* $[COMP_{q_i} - \gamma, COMP_{q_i} + \gamma]$ *for some*

arbitrary $q \in PROC$, *and,* $\forall\ i$ *and* $\forall\ c \in CHAN$, $MESS_{c_i}$ *is within the range* $[MESS_{d_i} - \delta, MESS_{d_i} + \delta]$ *for some arbitrary* $d \in CHAN$, *then a program can be classed as being uniform. Otherwise it can be classed as being non-uniform.*

A general purpose program model for loosely synchronous data parallel programs is now described.

4.2.2 A Program Model for Loosely Synchronous Data Parallel Programs

It is assumed that a program consists of a fixed set of processes, each displaying an iterative compute-communicate structure. This is a relatively simple type of computation to model. A program is constructed from a number of identical processes, each process computes for a length of time and then communicates with all connected processes. The message transfers take place in parallel so as to prevent deadlock from occurring. A program modelled in this way will exhibit the basic characteristics of loosely synchronous data parallelism.

To fully model such a program, one must be able to define the computational requirements of processes, and the lengths of messages sent down channels. This is achieved by allocating functions to the nodes of the underlying process graph. These functions are designed so as to act consistently with the weights associated with individual nodes and arcs. In the program model described here, each node in a process graph has two behavioural functions allocated to it:

- **Computation Function**

  This function defines the amount of computation that must be achieved by a process on each iteration before it can communicate with its neighbours. This quantity determines the granularity of the process.

- **Message Function**

  This function defines the lengths of messages to be sent down the outgoing channels attached to a process. It is assumed that all outgoing channels for a particular process display the same behaviour patterns.

Each of the functions described above takes a single parameter, the current simulation time. They each return a single value which is interpreted according to the context of the function. The values generated will typically be drawn from some statistical distribution, for example, a normal or chi-squared distribution relating to the particular set of program parameters in force. The program parameters assumed in this Chapter are described in Section 4.2.3. If the functions behave constantly over time, time-invariant behaviour will be generated. Alternatively, functions with periodic or phased characteristics can be used to represent time-varying behaviour. If all nodes are allocated identical functions, uniform behaviour will result. Alternatively, if a number of different functions are used, non-uniform behaviour will result.

Figure 4–1 shows the process template corresponding to the program model described above. The behaviour is specified using a Simula-based syntax with calls to MIMD procedures (see Chapter 3). Each process computes for a time defined by its computation function and the current simulation time. Then messages are sent on outgoing channels and received on incoming channels[1]. Message lengths are decided by the appropriate message function and the current simulation time. As mentioned earlier, the simulation time would be ignored if modelling time-invariant programs.

A modelling engine has been constructed to simulate programs obeying the model described here in the MIMD domain, it is fully described in [107]. You will

---

[1]The directions of individual channels are decided randomly before execution commences.

```
class process(compute_func,message_func)
begin
  while true do
  begin
    compute(compute_func(simulation_time));
    for each connected channel C do in parallel
    begin
      if C is outgoing then
        send(C,message_func(simulation_time))
      else
        receive(C);
    end;
  end;
end;
```

Figure 4–1: A Process Template for Loosely Synchronous Data Parallelism

recall that the output from the experiment generator, **eg**, is a set of EIs, each one describing a particular parallel computation. The modelling engine reads an EI and generates suitable MIMD objects representing the processes and processors specified therein. Subsequently, the corresponding Simula program can be executed, and this will be equivalent to simulating the program specified in the EI (as an occam program running on a transputer-based machine in this case).

This chapter investigates the behaviour of loosely synchronous data parallel programs displaying uniform and time-invariant behaviour. A set of program parameters supporting these programs, and time-invariant programs in general, is now described. A program parameter set suitable for modelling time-varying programs is presented in Chapter 7.

4.2.3 Characterising Time-invariant Behaviour

In order to characterise the macroscopic properties of time-invariant parallel programs a set of program parameters are required. I shall use the following parameter set:

$$\{N, c, \mu_{cg}, \sigma_{cg}, \mu_{mg}, \sigma_{mg}, \sigma_c, \sigma_m\}$$

N and c represent the number of nodes and connectivity of the process graph. The parameters μ_{cg} and σ^2_{cg} define a normal distribution describing the distribution of mean compute times (the amount of computation that must be achieved between message transfers) over the nodes of the process graph. Normal distributions were chosen for convenience, although they may not be entirely realistic. However, preliminary investigations indicated that the results obtained were not sensitive to the choice of distribution. Similarly, μ_{mg} and σ^2_{mg} define a normal distribution describing the distribution of mean message lengths over the edges of the process graph. The parameters σ_c and σ_m define chi-square distributions describing the variation, from one iteration to the next, in compute times and message lengths respectively. This variation corresponds to possible different paths through the sequential part of a process. The chi-square distribution is used to describe squares of normal variates, and hence is appropriate for characterising variance. One can see that σ_c and σ_m are concerned with specifying the degree of time-invariance present; they are in fact related to the values of α and β used in the definition of time-invariant parallel programs given in Definition 4.1.

Note that σ_{cg} and σ_{mg} were not contained in the the original program parameter set proposed by Candlin *et al.* [20]. These parameters allow one to characterise non-uniform behaviour by allocating processes and channels different mean computation times and mean message lengths. The behaviour of such programs is explored in Chapter 5.

Once the program parameters have been specified, an arbitrary number of process graphs can be generated. First of all **eg** must be used to construct a random graph using the values N and c. The remaining program parameters are then used to generate and allocate suitable normal distributions to the graph (this is implemented within the modelling engine). These distributions characterise the compute times and message lengths of the processes and channels, and are used to generate the Computation and Message functions used by the process template when the corresponding program is simulated.

A program simulated using a program graph constructed in the manner described above will be a typical program, a representative of the class of programs with the same parameter set and process template. It should be noted that the distributions used to construct synthetic programs from a parameter set are really hidden parameters to the model. It has been assumed that: unweighted graphs are uniformly distributed, means are normally distributed, and variances are distributed as chi-square. The normal distributions used are truncated at zero to prevent negative values being returned.

4.3 A Study of Uniform Time-invariant Programs

Uniform time-invariant programs form a relatively simple class of parallel computation. They therefore represent a suitable starting point for a systematic investigation of program behaviour.

4.3.1 Experiment Description

The execution time of a message passing parallel program is a function of the program structure, the hardware topology it is to be executed on, and the mapping between the two. To simplify matters in the first instance, the hardware and placement strategy were held constant within a single experiment. Various program parameters were then varied in an attempt to gain an understanding of the relationships which exist between the structure of time-invariant uniform parallel programs and their execution times in the given environment.

A two-level full factorial experiment varying four of the eight program parameters specified in Section 4.2.3 was executed. A regular program graph was used, i.e. each node was of equal degree. In fact, regular program graphs are assumed throughout this thesis. The values of the varied parameters were set so as to in-

Program Parameters	
Param	Value(s)
N	{32, 142}
c	{4, 12}
μ_{cg}	{3000, 300,000}
σ_{cg}	0
σ_c	10
μ_{mg}	{10, 5000}
σ_{mg}	0
σ_m	1

Other Parameters	
Param	Value(s)
Hardware	4×4 Mesh
Placement	Round Robin
Trial Length	20,000,000 (1 Sec)
Replications	3

Table 4–1: Parameter Settings for Initial Investigation into Time-invariant Program Behaviour

clude a large subset of possible real programs. The levels used are summarised in the Program Parameters section of Table 4–1.

All of the parameters concerned with specifying the computational characteristics of the processes (i.e. μ_{cg}, σ_{cg} and σ_c) are expressed in terms of clock cycles of the simulated machine. The T800 transputers simulated are assumed to run at 20MHz. Therefore, a computation of 3000 clock cycles corresponds to a relatively fine grained program. A floating point multiply takes approximately 30 clock cycles, so in 3000 clock cycles one could afford to carry out several floating point operations on, say, 20 data items. A computation time of 300,000 clock cycles corresponds to a large grained, compute-bound program.

All of the parameters concerned with specifying message lengths (i.e. μ_{mg}, σ_{mg} and σ_m) are expressed in terms of bytes.

The remaining program parameters were set so as to impose the uniformity and time-invariance conditions upon the programs generated. By setting σ_{cg} and

σ_{mg} to 0, the processes within a program were allocated identical mean compute times, and the channels were allocated identical mean message lengths. By setting μ_c and μ_m to 10 and 1 respectively, the variation in the values generated from one iteration to the next was kept to a minimum. Consequently, processes and channels displayed very similar behaviour patterns, both within a given iteration and between successive iterations.

In addition to the program parameters described above, a number of other factors had to be set defining the hardware to be simulated, the mapping strategy to be used, and the state of various simulation parameters. These factors are specified in the Other Parameters section of Table 4–1. The hardware was set to a 4×4 mesh of transputers with wrap-around connections[2]. A round-robin mapping strategy was used. The length of each of the simulation trials was fixed at 20,000,000 clock cycles of the simulated machine (equal to 1 second of real time). The experiment was conducted with three replications, each replication differing both in the random graph and random number seed used. The entire experiment took approximately 24 hours to complete on a Sun 4 Workstation.

The output of a single simulated execution would ideally be a direct measure of execution time. However, the synthetic programs used here are non-terminating. A suitable performance metric must therefore be chosen. Two metrics will be considered: firstly, the mean amount of computation achieved per process, T, which is expressed in clock ticks; and secondly, the mean percentage utilisation of the simulated processors, U. Both quantities are measured over the entire simulation period. It is clear that T is related to the rate of computation of the processes. U, on the other hand, reflects the utilisation of the underlying machine (which is in turn related to the parallel speedup achieved). One should

---

[2]In fact all the mesh-based processor topologies used in this thesis are assumed to have wrap-around connections.

bear in mind that the calculation of U includes all clock cycles, whether they be directly attributable to the user program or not. This means, for example, that in theory U could be artificially boosted by a poor mapping as a result of including extra throughrouting costs. However, as we shall see, the performance of the programs investigated in this thesis tends to be dominated by computation costs rather than communication costs. Consequently U provides a good indication of program performance.

4.3.2 Results

Two sets of results are presented here, one where the response variable is taken to be program oriented metric, T, and one where it is taken to be the machine oriented metric, U. In each case I investigate whether the assumptions underlying an analysis of variance are satisfied (these are detailed in Section 3.5.3), and derive suitable transformations of the response where appropriate. In addition, a summary of the results obtained by repeating the experiment with two alternative processor topologies is given.

The Metric T

The results of a two level full factorial experiment can be analysed using analysis of variance techniques. However, it is necessary to first check whether the assumptions underlying the analysis have been satisfied. A scatter plot of residuals versus the predicted response can be used to test the homoscedasticity of variance assumption, if this is satisfied there should be no visible trend in such a plot. Figure 4–2 shows a residual scatter plot for this experiment. One can see that the size of error is related to the magnitude of the response variable, with smaller errors occurring at the extreme values of the response variable, and larger errors occurring at intermediate values.

Careful examination of the residuals revealed that closely spaced errors tend to occur when the simulated machine is very highly utilised. This is quite a likely situation with very regular programs, since there are minimal synchronisation overheads. When such an upper bound is approached, i.e. a fully utilised machine, the variation between replicates will naturally tend to decrease. The points on the left of the scatter plot correspond to programs where N is equal to 142. With so many processes to be serviced, it is unlikely that a processor will find itself idle, consequently parallel slackness can be fully exploited. The points on the right of the scatter plot correspond to compute-bound programs with fewer processes, i.e. where μ_{cg} is equal to 300,000 and N is equal to 32. There is minimal variation in these trials because there is little scope for different interaction patterns to emerge.

Greater variability occurs at intermediate values of the response variable, when both μ_{cg} and N are set at their lower levels, i.e. communications-intensive programs with a smaller number of processes. Since replications differ only in the graph shape and random number seed used, one can conclude that in these programs the detailed ordering of events and interactions between processes become more significant. One would expect this sort of behaviour from communications-intensive programs.

The normality of errors assumption can be tested by examining a normal quantile-quantile plot of the residuals. Such a plot is shown in Figure 4–3. One can see that the larger positive and negative residuals do not follow a straight line; this indicates that the residuals have a distribution which has a longer tail than the normal distribution.

The above analysis suggests that at least two of the three assumptions underlying the analysis of variance do not hold for the raw data, so an analysis using T would not adequately describe the relationship between the predictor variables and the response variable. A transformation of the response can help to solve such problems. Direct application of the transformations described in Section 3.5.4 were found to lead to only marginal improvements in this case, due to the

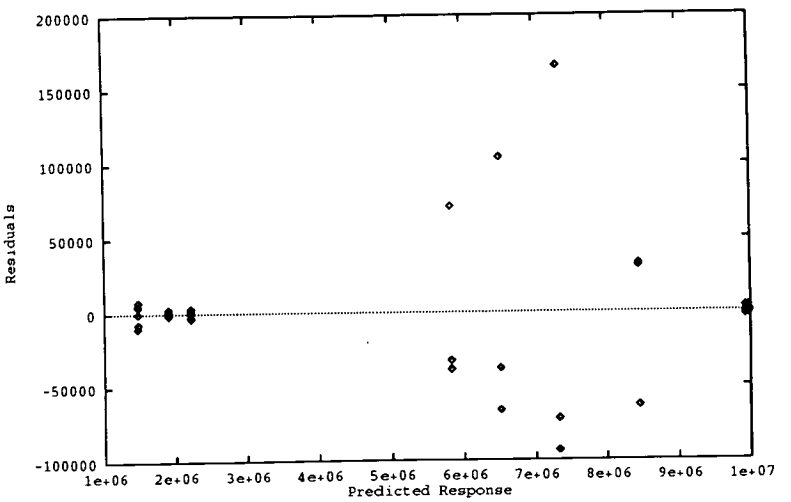

Figure 4–2: Predicted Response vs. Residuals Using Response T

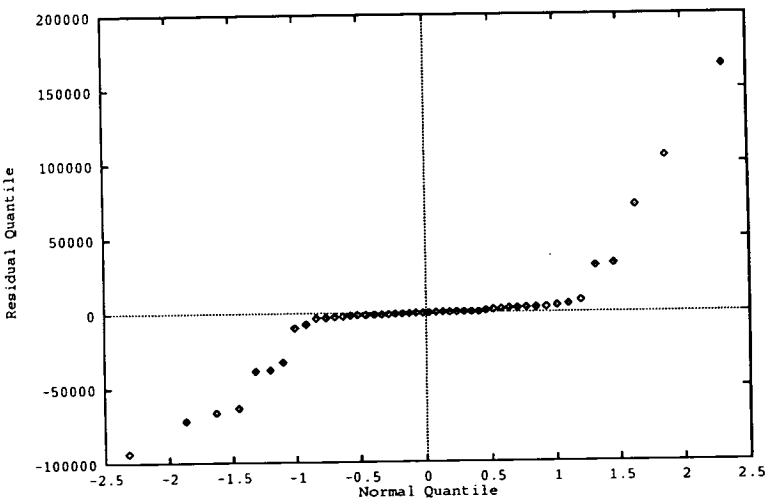

Figure 4–3: Normal Quantile vs. Residual Quantile Using Response T

rather extreme bulging nature of the residual scatter plot. By re-examining Figure 4–2, one can see that the errors at either end of the response variable range need to be exaggerated so that they compare with those found in the middle of the graph. The following sinh transformation was found to be useful in this respect:

$$\sinh\left(\frac{T - 7000000}{1000000}\right) \tag{4.1}$$

The original response is scaled before applying the sinh function so that the transformed responses can be located in a suitable area of the sinh curve. A suitable transformation, T', was then constructed for the original response, T, using Equation 4.1 in conjunction with a Box-Cox transformation (see Section 3.5.4 for full details). It is presented below:

$$T' = \left\{\sinh\left(\frac{T - 7000000}{1000000}\right) + 150\right\}^{2.41} \tag{4.2}$$

Figure 4–4 shows a residual scatter plot and Figure 4–5 shows a residual normal quantile-quantile plot for the transformed variable T'. In both cases the problems identified with the untransformed response seem to have been solved to a great extent. Specifically, the points in Figure 4–4 do not display a significant trend, and the points in Figure 4–5 form an approximate straight line. In addition, the fact that the residuals are of a smaller order than T' indicates that an additive model is structurally adequate (i.e. it can adequately explain the observed variation in the response).

The analysis of the experiment can now proceed in the transformed scale. The analysis of variance table is given in Table 4–2 (the derivation of such a table is discussed in Section 3.5.3). The residual stratum indicates the proportion of the observed variation in the response that cannot be explained by the model underlying the analysis. The F ratios are calculated by dividing the mean squares for the appropriate stratum by the residual mean squares. These values are used to derive the F probabilities. The F probabilities are a measure of the likelihood

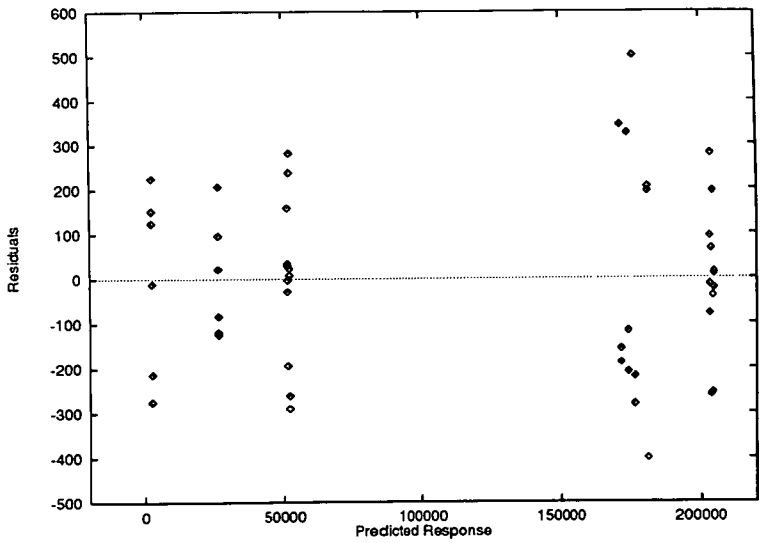

Figure 4–4: Predicted Response vs. Residuals Using Response T'

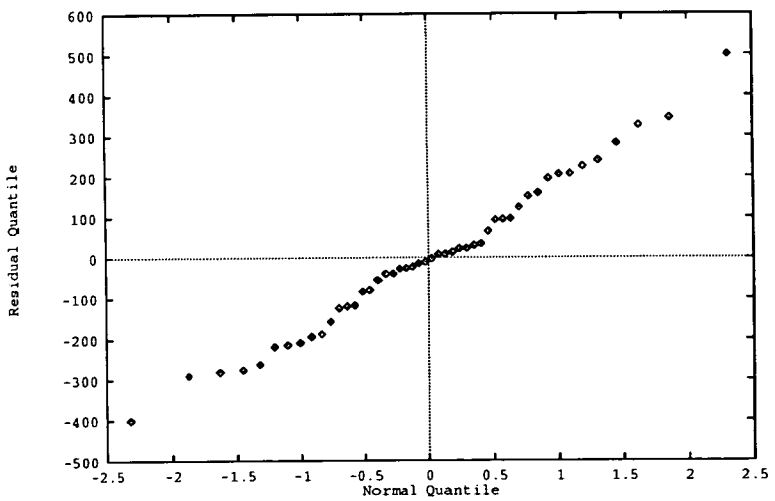

Figure 4–5: Normal Quantile vs. Residual Quantile Using Response T'

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
Between Replicates	2	202674	101337	2.094	0.141
Between Treatments	15	3.09871e+11	2.06581e+10	426850	< .01
Residual	30	1.4519e+06	48396.6		
Total	47	3.09873e+11			

Table 4–2: Analysis of Variance Table Using Response T'

that the F ratios observed could have arisen by chance. The variance observed in the response due to differences in parameter settings (the "treatment" mean squares) is very marked, and much larger than the variance observed between replicates at the same parameter settings. The F probabilities can be used to interpret these results in a formal manner. For example, one can reject the null hypothesis that different parameter settings produce the same results at the 1% level. This result indicates that is very unlikely that such large observed variations in the response variable will have come about by chance. Similarly, one can accept the null hypothesis that different replicates produce the same results at the 1% significance level.

Estimates of the effects of individual parameters and their interactions can be derived from a transformation of the 16 means at each combination of the parameter settings. These values are presented in the second column of Table 4–3, and correspond to the terms present in the model underlying the analysis (see Equation 3.1 in Section 3.5.3). The first line in the table, labelled (gm), specifies the overall mean value of T' (the *grand mean*). Each estimate has the same standard error which is given at the foot of the table. The t-values given in the third column of this table are the ratio of estimates to standard error, they can be used to indicate the relative importance of individual terms in the model. Values of t less than 3 can be taken as a guide to those terms which are of little importance. A less formal technique that can be used to assess the relative importance of various terms in the model is to consider the percentage variation in the response that can be attributed to each term. These figures are

Effect	Estimate	t-value	T' % Var	T % Var
(gm)	111435.12			
c	-3976.92	-125.24	0.245	0.617
N	-78388.06	-2468.67	95.183	88.713
cN	-2179.03	-68.62	0.074	0.226
μ_{cg}	16317.56	513.89	4.125	6.203
$c\mu_{cg}$	3542.19	111.55	0.194	0.567
$N\mu_{cg}$	2311.49	72.80	0.083	3.008
$cN\mu_{cg}$	2303.34	72.54	0.082	0.206
μ_{mg}	-510.04	-16.06	0.004	0.108
$c\mu_{mg}$	138.08	4.35	0.000	0.007
$N\mu_{mg}$	540.02	17.01	0.005	0.110
$cN\mu_{mg}$	-122.89	-3.87	0.000	0.007
$\mu_{cg}\mu_{mg}$	384.31	12.10	0.002	0.100
$c\mu_{cg}\mu_{mg}$	-141.82	-4.47	0.000	0.007
$N\mu_{cg}\mu_{mg}$	-414.72	-13.06	0.003	0.102
$cN\mu_{cg}\mu_{mg}$	141.18	4.45	0.000	0.007
		Total	99.999	99.988
	Standard Error = 31.75			

Table 4–3: Estimates of Effects Using Response T'

shown for T' in the fourth column, and are calculated by expressing the sum of squares attributable to a particular term as a proportion of the total sum of squares. For comparison purposes, the percentage variation observed using the original response is also shown in the fifth column.

The t-values for the N and μ_{cg} effects are markedly high relative to the other terms, together explaining 99.3% of the variation in T'. This suggests that these two parameters dominate, even though other terms also have t-values greater than 3. Comparing the fourth and fifth columns, one can see that the transformation from T to T' has greatly lessened a dependence on the first order interaction between N and μ_{cg}, as well as decreasing the importance of other higher order interactions. This is a desirable property since such interactions could be ignored in future experiments.

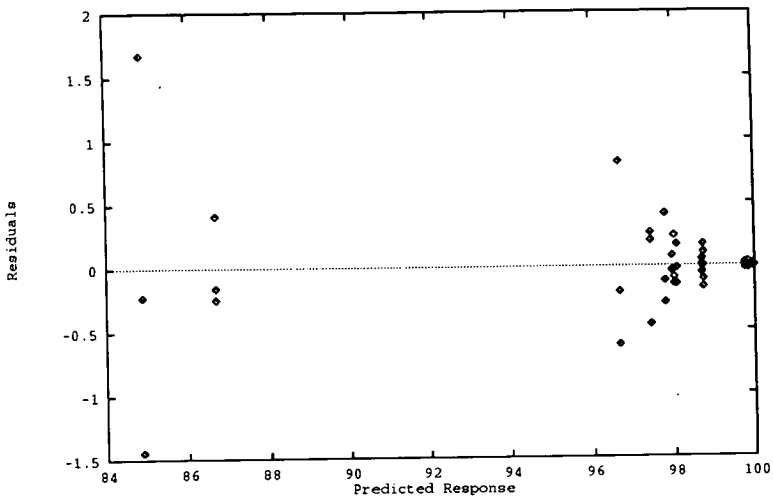

Figure 4–6: Predicted Response vs. Residuals Using Response U

The Metric U

I will now analyse the experiment using the metric U, the mean utilisation of the simulated processors. However, before proceeding, the assumptions underlying an analysis of variance are once again investigated. Figure 4–6 shows a residual scatter plot for U, one can see that there is a funneled tendency in the graph, with values of U closer to 100% giving smaller errors. This is not surprising, since variation will naturally tend to decrease as the simulated machine approaches full utilisation. One can test for the normality of errors by examining Figure 4–7 where a normal quantile-quantile plot is presented. The curvature indicates that the errors follow a distribution with a longer tail than the normal distribution.

The above discussion suggests that once again a transformation of the response should be considered. You will recall from Section 3.5.4 that when the observed values of the response (U in this case) are close to 100%, a transformation of the form $(1 - U/100)^{\lambda}$ can often be helpful. Indeed, this appears to be the case. An exponent of 0.06 was found to minimise the residual sum of squares. Consequently,

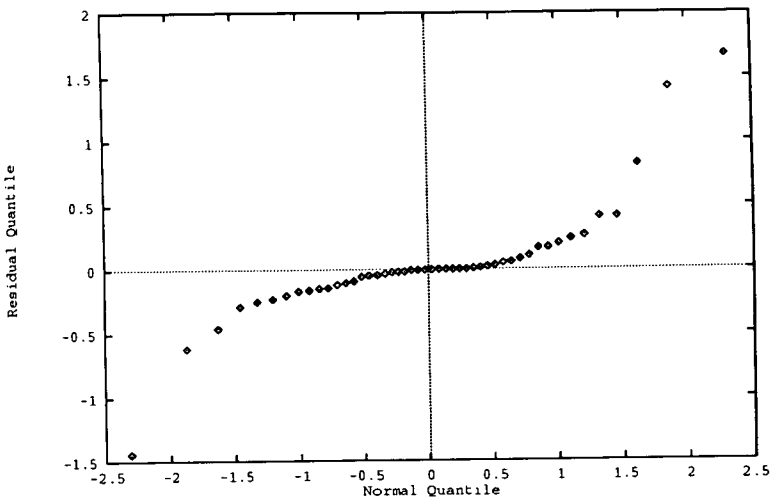

Figure 4–7: Normal Quantile vs. Residual Quantile Using Response U

the following transformation was used:

$$U' = \left(1 - \frac{U}{100}\right)^{0.06} \tag{4.3}$$

Figure 4–8 shows a residual scatter plot and Figure 4–9 shows a normal quantile-quantile plot for the transformed response U'. The problems identified with the untransformed data appear to have been solved, since the points in Figure 4–8 do not display a definite trend, and the points in Figure 4–9 form an approximate straight line. In addition, there is no reason to doubt the structural adequacy of the model, since U' covers only a single order of magnitude, and the residuals are of a smaller order than U'.

The analysis of variance table for U' is given in Table 4–4. The mean squares figures indicate that once again the differences between parameter settings are very pronounced, and much larger than the differences between replicates at the same parameter settings.

Estimates of the parameter effects and their interactions are presented in Table 4–5. The parameter μ_{cg} has the greatest effect on U', followed by the first order interaction between μ_{cg} and N. The terms N, μ_{mg} and their first order interaction

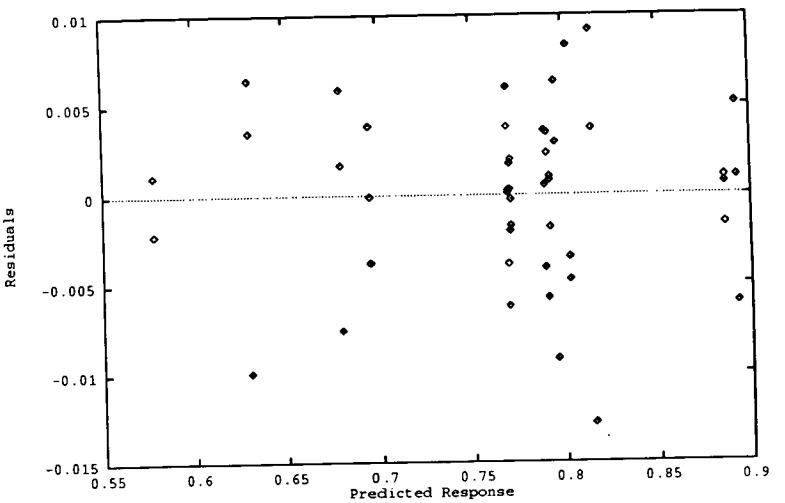

Figure 4–8: Predicted Response vs. Residuals Using Response U'

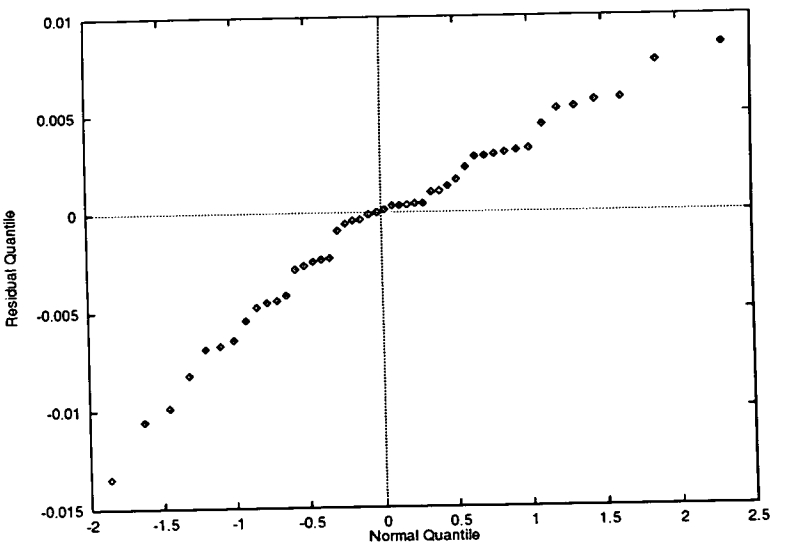

Figure 4–9: Normal Quantile vs. Residual Quantile Using Response U'

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
Between Replicates	2	0.000189	9.47e-05	3.28	0.051
Between Treatments	15	0.31	0.0206	715.64	< .01
Residual	30	0.000865	2.88e-05		
Total	47	0.311			

Table 4–4: Analysis of Variance Table Using Response U'

Effect	Estimate	t-value	U' % Var	U % Var
(gm)	0.7641			
c	0.0066	8.46	0.665	0.006
N	0.0181	23.35	5.061	8.544
cN	-0.0040	-5.15	0.247	0.035
μ_{cg}	-0.0563	-72.56	48.881	28.012
$c\mu_{cg}$	0.0023	2.94	0.080	0.001
$N\mu_{cg}$	0.0441	56.87	30.021	18.216
$cN\mu_{cg}$	-0.0042	-5.42	0.273	0.040
μ_{mg}	0.0206	26.61	6.576	10.815
$c\mu_{mg}$	-0.0046	-5.89	0.322	0.234
$N\mu_{mg}$	-0.0217	-28.03	7.293	11.612
$cN\mu_{mg}$	0.0038	4.86	0.219	0.155
$\mu_{cg}\mu_{mg}$	0.0002	0.26	0.001	10.191
$c\mu_{cg}\mu_{mg}$	-0.0002	-0.19	0.000	0.239
$N\mu_{cg}\mu_{mg}$	0.0008	1.08	0.011	10.956
$cN\mu_{cg}\mu_{mg}$	0.0008	1.04	0.010	0.168
		Total	99.661	99.223
	Standard Error = 0.000775			

Table 4–5: Estimates of Effects Using Response U'

	Mesh	Hypercube	Ring
$T' =$	$\left\{\sinh\left(\frac{T-7000000}{1000000}\right)+150\right\}^{2.41}$	$\left\{\sinh\left(\frac{T-7000000}{1000000}\right)+150\right\}^{1.96}$	$\sinh\left(\frac{T-5000000}{1000000}\right)+150$
$U' =$	$\left(1-\frac{U}{100}\right)^{0.06}$	$\left(1-\frac{U}{100}\right)^{0.15}$	$\left(\frac{U}{100-U}\right)^{-0.17}-1$

Table 4–6: Summary of Transformations Used

are the three next most important. These five explain more than 97.8% of the variation in U'. Higher order interactions, other than those already highlighted, appear to be relatively unimportant. By comparing the fourth and fifth columns, one can see that the transformation of U to U' has simplified matters by removing the dependence on the $\mu_{cg}\mu_{mg}$ and $N\mu_{cg}\mu_{mg}$ interactions.

Alternative Processor Topologies

To increase confidence in the results obtained, further experiments were carried

out using two alternative processor topologies: a 4-dimensional hypercube and a 16 node ring. All of the other experimental parameters remained unchanged. Both T and U were investigated for each of the processor topologies. Table 4–6 shows the transformations that were found to be useful in satisfying the conditions for a valid analysis. The first column summarises the transformations that were derived for the mesh topology. The second column refers to the hypercube topology, one can see that the transformations found to be useful are very similar to those used for the mesh, differing only in the exponent. The third column deals with the ring topology. The T' transformation is similar to those used for the mesh and hypercube, however a different scaling of T was required and no power transformation was needed. The U' transformation is of a different form to those seen so far. The previous U' transformations relied on the observed values being close to 100. However, the values of U observed with the ring were lower than those observed for the mesh and hypercube (as a result of larger synchronisation overheads due to increased contention for communications resources). The particular transformation used is known as a Guerrero and Johnson transformation [4]; it is designed for percentages and proportions, and is fully described in section 3.5.4.

Tables 4–7–4–10 give the analysis of variance tables for the hypercube and ring processor topologies for both T' and U'. In each case the differences in mean squares between parameter settings are far greater than the differences between replicates at the same parameter settings. This agrees with the results obtained earlier for the mesh.

Table 4–11 summarises the percentage variation that can be attributed to individual terms in the model for each of the three processor topologies and each of the two metrics. The first thing to notice is that the results for the mesh and the hypercube are very similar for both T' and U'. However, the ring produces different results. For T', the impact of N is reduced and the importance of the μ_{cg} and $N\mu_{cg}$ terms are both much increased. For U', there are even greater differences, with the effects of c and μ_{mg} becoming more significant and the effect

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
Between Replicates	2	485.9	242.95	0.548	0.584
Between Treatments	15	2.92e+09	1.95e+08	439556	< .01
Residual	30	13299.3	443.31		
Total	47	2.92e+09			

Table 4–7: Analysis of Variance Table for Hypercube Using Response T'

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
Between Replicates	2	5.80e-05	2.90e-05	0.169	0.846
Between Treatments	15	0.798	0.0532	309.41	< .01
Residual	30	0.00516	0.000172		
Total	47	0.803			

Table 4–8: Analysis of Variance Table for Hypercube Using Response U'

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
Between Replicates	2	0.223	0.111	0.56	0.577
Between Treatments	15	58523.4	3901.56	19648	< .01
Residual	30	5.96	0.199		
Total	47	58529.6			

Table 4–9: Analysis of Variance Table for Ring Using Response T'

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
Between Replicates	2	0.00272	0.00136	1.71	0.197
Between Treatments	15	2.21	0.148	185.92	< .01
Residual	30	0.0238	0.0008		
Total	47	2.24			

Table 4–10: Analysis of Variance Table for Ring Using Response U'

Effect	Mesh T'	Hyper T'	Ring T'	Mesh U'	Hyper U'	Ring U'
c	0.245	0.419	1.093	0.665	0.291	4.213
N	95.183	93.384	49.109	5.061	4.291	0.007
cN	0.074	0.208	0.022	0.247	0.002	0.264
μ_{cg}	4.125	4.908	33.280	48.881	50.520	47.380
$c\mu_{cg}$	0.194	0.357	0.329	0.080	0.010	1.471
$N\mu_{cg}$	0.083	0.498	14.851	30.021	26.938	8.887
$cN\mu_{cg}$	0.082	0.216	0.370	0.273	0.285	1.734
μ_{mg}	0.004	0.003	0.434	6.576	8.231	19.347
$c\mu_{mg}$	0.000	0.000	0.102	0.322	0.000	2.258
$N\mu_{mg}$	0.005	0.003	0.012	7.293	8.059	3.656
$cN\mu_{mg}$	0.000	0.000	0.052	0.219	0.009	0.883
$\mu_{cg}\mu_{mg}$	0.002	0.002	0.117	0.001	0.362	6.698
$c\mu_{cg}\mu_{mg}$	0.000	0.000	0.026	0.000	0.007	1.691
$N\mu_{cg}\mu_{mg}$	0.003	0.002	0.043	0.011	0.348	0.013
$cN\mu_{cg}\mu_{mg}$	0.000	0.000	0.150	0.010	0.000	1.315
Total	99.999	99.999	99.989	99.661	99.351	99.816

Table 4–11: Summary of Percentage Variation Figures

of N becoming less so. In addition, many more higher order interactions seem to be important.

4.3.3 Discussion

It is worth recalling that two characteristics of a concurrent program that might be thought to affect performance have been ignored. These are the shape of the program graph, and the detailed pattern of synchronisations between processes as the program is executed. These effects would be picked up by large differences between replicates, since individual trials differ both in the random graph and random number seed used. It has already been shown that the differences between replicates are relatively small, so programs with the same parameter values tend behave in a similar manner. The differences between parameter settings have tended to explain the bulk of the variation in the response variables, so one can be confident that the four parameters explored have useful predictive powers for the

particular class of programs considered. Of course, these properties are dependent on the assumption that all other factors, for example the hardware configuration and placement strategy, remain constant.

The effects of the various parameters can be plausibly interpreted in terms of what is known about the particular program model and machine simulated. Let us first consider the results obtained using a mesh processor topology and the performance metric T', the (transformed) mean computation time achieved per process (see Table 4–3). By far the most important influence comes from N, the number of nodes in the program graph. This seems reasonable since the transputer divides its time equally between the processes it is executing, so, adding extra processes will result in a proportionate slow down in the rate of computation for each process. The other significant parameter is μ_{cg}, the mean amount of computation carried out between message transfers. It is well-known that programmers should attempt to achieve large-scale granularity on message-passing machines, and this result provides a quantitative estimate of the benefits to be obtained relative to the metric T'. The remaining parameters have very little influence. One would expect c, the degree of the program graph, to have a relatively small effect since the processes modelled send and receive messages in parallel, rather than in sequence. This technique results in efficient code on the transputer since inter-processor message transfers can be executed in parallel on all four links, and computation can be overlapped with link transfers. Perhaps a little more surprising is the small impact of μ_{mg}, the mean message length. This indicates that the synchronisation costs of executing a message transfer tend to outweigh the costs of actually moving the data; this is why programmers of distributed memory machines tend to bundle a number of small messages into a single large message whenever possible. This effect is probably exaggerated by the message-passing protocol used, in which messages are explicitly acknowledged in order to enforce a synchronous communications mechanism.

We have seen that the metric T' tends to be dominated by the N, the number

of processes in the program. While this is reasonable for a program oriented metric, it is sometimes desirable to use a metric that is rather more sensitive to the performance of the underlying machine. The (transformed) mean percentage utilisation of the processors, U', is useful in this respect. For the mesh processor topology, the dominant parameter when using U' is μ_{cg} (see Table 4–5). This reflects the fact that, increasing the frequency of message transfers will increase the proportion of time that processes spend blocked (waiting to synchronise with their partners), and so the utilisation of the processors will drop accordingly. The mean message length, μ_{mg}, is also of some importance; this is because longer messages result in increased contention for links, which causes longer blocking times for processes and reduced utilisation of processors. The number of processes also appears to have an influence; the more processes there are, the less likely the chance that a processor will finds itself idle. There are some important interactions, especially that between N and μ_{cg}. This indicates that the impact on performance of increasing the number of processes is dependent on the frequency of message transfers. Obviously, if message transfers occur infrequently then processors will be highly utilised, so adding more processes will have a negligible effect. However, if message transfers occur frequently, then the message passing overheads will be significant, and adding more processes will increase these overheads.

The results for the hypercube are very similar to those obtained for the mesh (see Table 4–11). This is to be expected, since these two topologies have similar interconnection properties for 16 nodes i.e. both degree and diameter are equal to 4 (since the mesh is constructed with wrap-around connections). Accordingly, one would expect similar behaviour to result from executing an arbitrarily chosen random process graph on either of these two topologies, all else remaining equal. In contrast to this, a 16 node ring has degree equal to 2 and a diameter equal to 8. Link contention will therefore be a bigger problem in the ring, since messages have to be mapped onto fewer links, and have further to travel on average. Consequently, one would expect the performance of the ring to be more sensitive

to the frequency and lengths of message transfers. This effect can be seen in the T' results for the ring, where the importance of μ_{cg} and the first order interaction between N and μ_{cg} is larger than that observed for the mesh and hypercube (at the expense of N). This is a reflection of the increased importance that the frequency of message transfers is having on performance. The metric T' is not sensitive enough to pick up the effects of increased message lengths, but it is possible to detect this effect if one uses U'. In the last column of Table 4–11 one can see that μ_{mg}, and almost all terms containing μ_{mg}, are more important for the ring, compared to both the mesh and the hypercube topologies. In addition, the importance of c also increases because it determines the number of message transfers that must be carried out within each iteration. The number of nodes is of very little importance, and the effect of μ_{cg} remains largely unchanged.

It has been shown that the results obtained can vary a great deal according to the choice of performance metric. This choice between T' and U' depends largely on whether one wishes to take a program oriented or machine oriented view. T would be a more useful metric for the applications programmer concerned with program structure, whereas U would be more appropriate if one was attempting to make the best of a program with a fixed structure. In general, T' produces fewer terms of importance but tends to be dominated by N. On the other hand, U' is more sensitive to the other parameters, but generates a model with a larger number of significant terms. This is especially true for the ring, where higher order interactions become more important as a result of the more complex behaviour patterns arising from increased link contention.

4.4 Validation of Experimental Approach

This section presents a number of results concerned with validating the experimental approach. I concentrate on the metric T, but similar results could be obtained for U.

4.4.1 Structure of Synthetic Parallel Programs

The experiments described in this chapter have assumed that random process graphs can be used to represent real parallel programs. It is not clear how the shapes of these graphs relate to the shapes of graphs which might be used to solve real problems. It would therefore be useful to examine how the behaviour of random process graphs compares with the behaviour of program graphs structured in a rather more regular fashion. The results presented here compare random process graphs with mesh and hypercube shaped graphs of the same size and connectivity.

Figure 4–10 illustrates the performance observed using both mesh and random shaped process graphs. The results of three separate experiments are shown, corresponding to three different sized graphs. The connectivity is fixed at 8, since it would not be possible to vary the connectivity for a mesh in a meaningful manner. The 8 edges adjacent to each node in a mesh connect an outgoing and incoming channel to each of the north, south, east and west neighbours. The three experiments executed were essentially the same as the experiment described earlier in this chapter. However, since the graph size and connectivity were fixed within an experiment, only μ_{cg} and μ_{mg} could be varied, so there were four trials in each experiment. For each parameter combination, Figure 4–10 compares the performance achieved by the random graph with that achieved by a mesh of identical

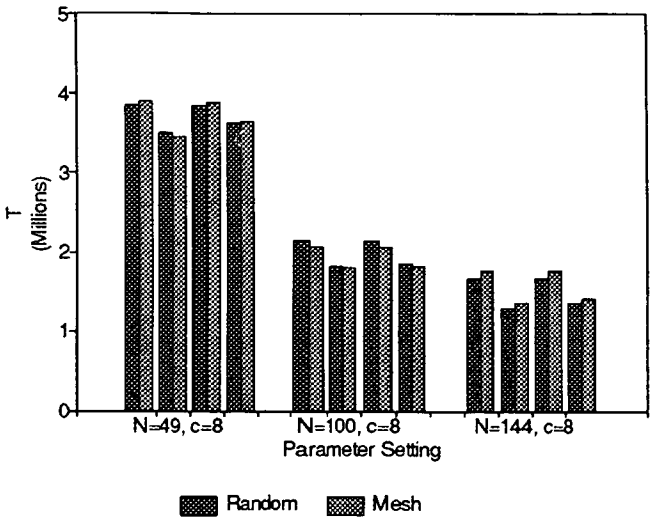

Figure 4–10: Comparison of Mesh and Random Shaped Graphs for T

size and connectivity. One can see that the results are reasonably close, typically within 5%.

Figure 4–11 compares the behaviour of hypercube and random shaped process graphs in a similar way. However, in this case the degree of the graphs can be varied along with the number of nodes. The results for the random graph are close to those obtained with the hypercube. So, it seems that randomly constructed process graphs seem to display comparable behaviour to more regularly shaped graphs of the same size and connectivity (assuming that they are executed under the same conditions).

4.4.2 Simulation Lengths

All simulations so far have been run for 20,000,000 cycles (1 second) of simulated machine time. A question arises as to whether, when compute times are large, this period is long enough for the simulation to reach a stable state. For example, for the experiments described in this chapter, a process might only reach the front

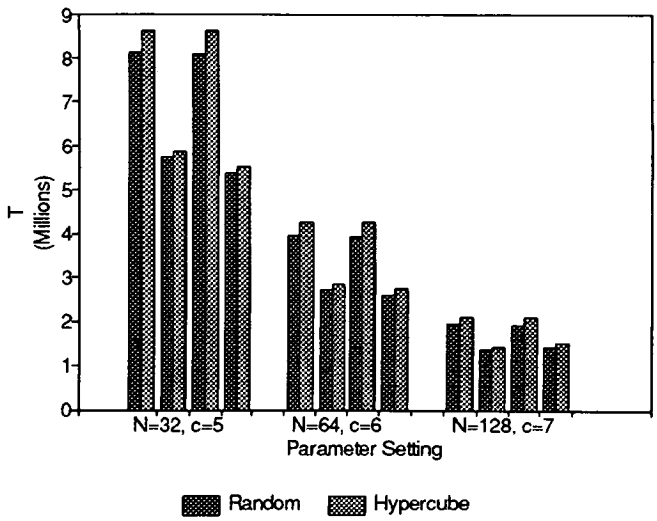

Figure 4–11: Comparison of Hypercube and Random Shaped Graphs for T

c	N	μ_{mg}	10 x 1 sec	10 secs	% diff
4	32	10	99825650	99817856	-0.0078
12	32	10	99474830	99505008	0.0003
4	142	10	22214430	22190130	0.11
12	142	10	22142920	22128880	0.063
4	32	5000	99685500	99662688	-0.023
12	32	5000	99248640	99260512	0.012
4	142	5000	22215220	22187828	-0.12
12	142	5000	22141970	22121962	-0.09

Table 4–12: Comparison of Short and Long Simulation Runs for T

of its processor queue as little as 7 or 8 times if N was set to 142, and μ_{cg} was set to 300,000.

To investigate whether this is a problem, an experiment was carried out using a mesh processor topology. The value of μ_{cg} was fixed at 300,000. Each trial was executed for both 1 second and 10 seconds of simulated machine time. The remaining parameters were varied as normal, the results obtained are summarised in Table 4–12. One can see that the values of T obtained for the 10 second simulations are very close to 10 times the values obtained for the 1 second simulations. This indicates that there is no significant start-up phase, and that the simulation very quickly enters a stable state.

4.5 Summary and Conclusions

This chapter has used standard statistical techniques to develop a methodology for the performance evaluation of parallel programs. A representative program model was described, and the results of a series of simulation experiments were presented. These experiments concentrated on a relatively simple class of parallel programs, namely those displaying uniform time-invariant behaviour. A number of transforms were developed to help in the analysis of these experiments, both in terms of the metric T, the mean computation achieved per process, and the metric U, the mean utilisation of simulated processors. These transformations allowed the assumptions underlying an analysis of variance to be satisfied, at the expense of a certain lack of clarity in interpretation.

The programs investigated in this chapter were deliberately kept simple in order to illustrate the usefulness of the methodology. It has been demonstrated that it is possible to characterise program behaviour in terms of a small number of rather high-level program parameters. Also, it has been shown how the standard methods of experimental design and analysis can be used to obtain quantitative

estimates of the relative importance of these parameters with respect to a number of different performance metrics. In turn, this produces a model which provides a good explanation of the observed performance. The results obtained seem consistent and meaningful, and can be interpreted in terms of what we know about real programs executing on a real machine. It is encouraging to see that such a crude approach produces promising results. In the next chapter non-uniform time-invariant programs are considered, and techniques for constructing performance prediction models are investigated.

Chapter 5

Performance Prediction Models

The experiments described in this chapter have been designed to illustrate how performance prediction models can be constructed for a particular class of parallel programs. Specifically, I concentrate on loosely synchronous data parallel programs exhibiting time-invariant *non-uniform* behaviour. Analysis of variance and covariance techniques are used to provide a framework in which to construct performance prediction models. A particularly interesting consequence of using the analysis of covariance is that the covariates identified can shed considerable light on the problem of understanding the dynamics of program behaviour. This knowledge can then be used to guide the development of process migration strategies; this is illustrated in Chapter 6.

Section 5.1 describes an exploratory experiment designed to discover which program parameters have the greatest predictive powers. Section 5.2 investigates the performance characteristics of these parameters in greater detail, and illustrates how a performance prediction model can be constructed. Section 5.3 shows how the performance of the model can be improved by including covariates containing information relating to the various interactions which occur between the program and the machine it is executed on. Section 5.4 helps to establish the generality of the method presented here, by showing that it can applied successfully to a larger processor domain. A summary and conclusions are presented in Section 5.5.

5.1 Exploratory Analysis

Time-invariant non-uniform parallel programs operate on irregular problem domains. At any point in time individual processes and channels exhibit significantly different behaviour patterns, however, these patterns remain relatively constant throughout the execution period. I concentrate on the performance metric U, the mean utilisation of the simulated processors. In order to construct performance prediction models, this chapter is concerned with investigating the relationships which exist between the structure of a parallel program, and its behaviour when executed on a given parallel machine. Consequently, a machine oriented metric seems a natural choice. Indeed, the metric U will be used throughout the remainder of this thesis.

5.1.1 Experiment Description

The same set of program parameters that was used in Chapter 4 is assumed, namely:

$$\{N, c, \mu_{cg}, \sigma_{cg}, \mu_{mg}, \sigma_{mg}\, \sigma_c, \sigma_m\}$$

When uniform parallel programs were investigated, σ_{cg} and σ_{mg} were set to 0, so that all nodes received identical mean compute times, and all edges received identical mean message lengths. By using non-zero values of σ_{cg} and σ_{mg}, non-uniform patterns of activity can be generated, since individual nodes and edges will be allocated different mean compute times and mean message lengths. The degree of non-uniformity present depends upon the sizes of σ_{cg} and σ_{mg}, relative to their respective means. For example, if $\mu_{cg} = 5000$ and $\sigma_{cg} = 3000$, then mean compute times would vary greatly across the nodes of the process graph. If, however, $\mu_{cg} = 200,000$ and $\sigma_{cg} = 3000$, then the degree of variation would be far less significant, since the standard deviation is small compared to the mean. To

Program Parameters	
Param	Value(s)
N	{32, 142}
c	{4, 12}
μ_{cg}	{3000, 300,000}
$\sigma_{cg}\%$	{10, 80}
σ_c	10
μ_{mg}	{10, 5000}
$\sigma_{mg}\%$	{10, 80}
σ_m	1
Other Parameters	
Param	Value(s)
Hardware	4×4 Mesh
Placement	Round Robin
Trial Length	20,000,000 (1 Sec)
Replications	3

Table 5–1: Parameter Settings for Initial Investigation into Non-uniform Program Behaviour

prevent this ambiguity, the labels $\sigma_{cg}\%$ and $\sigma_{mg}\%$ will be used to express σ_{cg} and σ_{mg} as proportions of μ_{cg} and μ_{mg} respectively. For example, if $\sigma_{cg}\% = 10$, σ_{cg} would be equal to 10% of the value of μ_{cg}.

To investigate the behaviour of non-uniform parallel programs, a two level full factorial experiment was carried out varying six of the eight program parameters. The exact settings for the experiment are summarised in Table 5–1. You will see that this experiment is identical to that presented in Chapter 4, except for the fact that $\sigma_{cg}\%$ and $\sigma_{mg}\%$ are now allowed to vary. As previously, each replication within the experiment differed both in the random graph and random number seed used.

To limit the scope of the study, I only consider mesh-based topologies. It has already been demonstrated that analysis of variance techniques can equally be applied to other processor topologies.

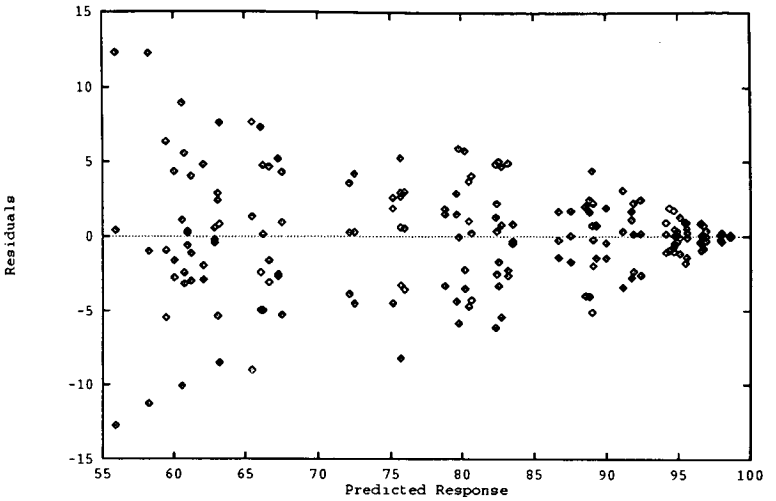

Figure 5–1: Residual Scatter Plot for U

5.1.2 Results

A residual scatter plot for the untransformed response, U, is shown in Figure 5–1. It is clear that the residuals display a funneled tendency, and so do not have constant variance (as is necessary for a valid analysis of variance). To overcome this problem the following Guerrero and Johnson transformation was found to be useful (see Section 3.5.4):

$$U' = \left(\frac{U}{100 - U}\right)^{0.08} - 1 \tag{5.1}$$

Figure 5–2 shows a residual scatter plot and Figure 5–3 shows a residual normal quantile-quantile plot for U'. Both plots give no cause for concern, indicating that the normality and homoscedasticity of variance assumptions hold. Furthermore, U' covers only a single order of magnitude, and the residuals are not of the same order as U'. There is no reason, therefore, to doubt the suitability of an additive model.

The analysis of variance table for U' is given in Table 5–2. The differences in the mean values observed between different parameter levels are large compared to the differences observed between replicates. This is reflected in the F probabilities,

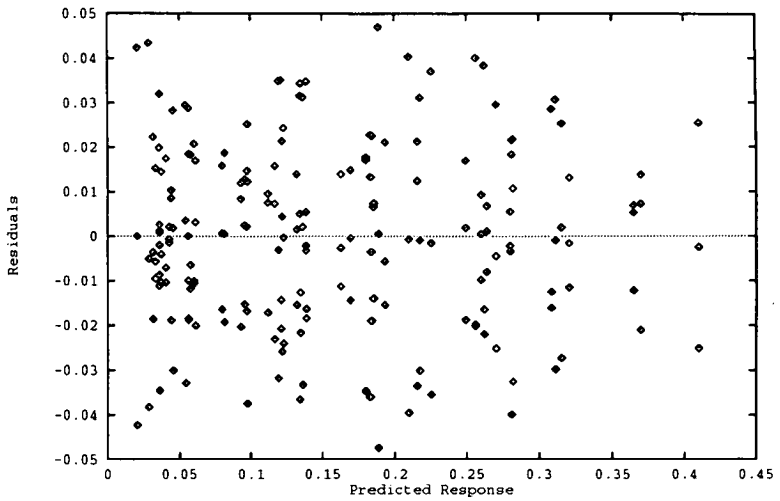

Figure 5–2: Residual Scatter Plot for U'

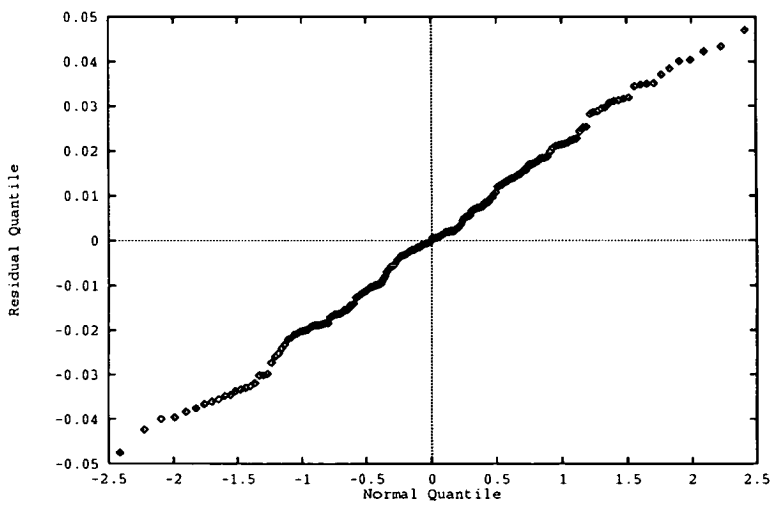

Figure 5–3: Normal Quantile-Quantile Plot for U'

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
Between Replicates	2	0.00124	0.000621	1.013	0.366
Between Treatments	63	1.96	0.0311	50.273	$< .01$
Residual	126	0.0773	0.000613		
Total	191	2.038			

Table 5–2: Analysis of Variance Table Using Response U'

which indicate that the null hypothesis, that different replicates produce the same overall mean value, can be accepted at the 1% significance level. Similarly, the null hypothesis, that different factor levels produce the same mean value, can be rejected at the 1% significance level. One can be reasonably confident, therefore, that the program parameters used adequately characterise performance.

To examine the influence of the program parameters in greater detail, estimates of the effects of individual factors and their interactions were obtained, these are presented in Table 5–3. You will recall that the t values indicate the relative importance of individual terms in the model. To aid readability the % suffix has been dropped from the parameters $\sigma_{cg\%}$ and $\sigma_{mg\%}$. It is clear that the dominant terms are N, the number of nodes, and $\sigma_{cg\%}$, the standard deviation of process compute times (expressed as a proportion of the mean compute time). These, together with their first order interaction, account for 84.1% of the variation in U'. The μ_{cg} and μ_{mg} effects are of less importance, accounting for 1.54% and 1.39% of variation respectively. The remaining two factors, c and $\sigma_{mg\%}$, have very little influence indeed. In total, the program parameters explain 96.15% of the observed variation in U'.

5.1.3 Discussion

Large synchronisation delays in a parallel program will inevitably lower the utilisation of the machine that the program is being executed on, since a processor can do nothing while the processes it has been allocated are blocked. One would

Effect	Estimate	t-value	% Var
(gm)	0.1557	87.09	
c	-0.0023	-1.31	0.05
N	0.0505	28.27	24.05
cN	-0.0051	-2.84	0.24
μ_{cg}	0.0128	7.15	1.54
$c\mu_{cg}$	-0.0114	-6.39	1.23
$N\mu_{cg}$	0.0035	1.95	0.11
$cN\mu_{cg}$	-0.0055	-3.08	0.29
σ_{cg}	-0.0783	-43.83	57.80
$c\sigma_{cg}$	0.0023	1.29	0.05
$N\sigma_{cg}$	-0.0155	-8.65	2.25
$cN\sigma_{cg}$	0.0021	1.16	0.04
$\mu_{cg}\sigma_{cg}$	-0.0140	-7.81	1.84
$c\mu_{cg}\sigma_{cg}$	0.0025	1.38	0.06
$N\mu_{cg}\sigma_{cg}$	-0.0019	-1.09	0.04
$cN\mu_{cg}\sigma_{cg}$	0.0025	1.40	0.06
μ_{mg}	-0.0122	-6.80	1.39
$c\mu_{mg}$	-0.0003	-0.14	0.00
$N\mu_{mg}$	0.0068	3.81	0.44
$cN\mu_{mg}$	0.0013	0.74	0.02
$\mu_{cg}\mu_{mg}$	0.0014	0.78	0.02
$c\mu_{cg}\mu_{mg}$	0.0015	0.83	0.02
$N\mu_{cg}\mu_{mg}$	-0.0067	-3.73	0.42
$cN\mu_{cg}\mu_{mg}$	0.0003	0.15	0.00
$\sigma_{cg}\mu_{mg}$	0.0077	4.30	0.56
$c\sigma_{cg}\mu_{mg}$	0.0005	0.31	0.00
$N\sigma_{cg}\mu_{mg}$	-0.0060	-3.38	0.34
$cN\sigma_{cg}\mu_{mg}$	-0.0005	-0.30	0.00
$\mu_{cg}\sigma_{cg}\mu_{mg}$	-0.0034	-1.90	0.11
$c\mu_{cg}\sigma_{cg}\mu_{mg}$	-0.0007	-0.37	0.00
$N\mu_{cg}\sigma_{cg}\mu_{mg}$	0.0090	5.02	0.76
$cN\mu_{cg}\sigma_{cg}\mu_{mg}$	0.0001	0.08	0.00
σ_{mg}	-0.0033	-1.83	0.10
$c\sigma_{mg}$	0.0003	0.14	0.00
$N\sigma_{mg}$	0.0033	1.83	0.10
$cN\sigma_{mg}$	0.0003	0.17	0.00
$\mu_{cg}\sigma_{mg}$	0.0033	1.84	0.10
$c\mu_{cg}\sigma_{mg}$	-0.0001	-0.06	0.00
$N\mu_{cg}\sigma_{mg}$	-0.0021	-1.17	0.04
$cN\mu_{cg}\sigma_{mg}$	-0.0002	-0.11	0.00
$\sigma_{cg}\sigma_{mg}$	0.0013	0.72	0.02
$c\sigma_{cg}\sigma_{mg}$	-0.0018	-1.01	0.03
$N\sigma_{cg}\sigma_{mg}$	-0.0045	-2.54	0.19
$cN\sigma_{cg}\sigma_{mg}$	-0.0010	-0.58	0.01
$\mu_{cg}\sigma_{cg}\sigma_{mg}$	-0.0021	-1.17	0.04
$c\mu_{cg}\sigma_{cg}\sigma_{mg}$	-0.0008	-0.43	0.01
$N\mu_{cg}\sigma_{cg}\sigma_{mg}$	0.0005	0.30	0.00
$cN\mu_{cg}\sigma_{cg}\sigma_{mg}$	-0.0012	-0.67	0.01
$\mu_{mg}\sigma_{mg}$	-0.0014	-0.81	0.02
$c\mu_{mg}\sigma_{mg}$	-0.0006	-0.33	0.00
$N\mu_{mg}\sigma_{mg}$	0.0028	1.59	0.08
$cN\mu_{mg}\sigma_{mg}$	-0.0002	-0.12	0.00
$\mu_{cg}\mu_{mg}\sigma_{mg}$	0.0092	5.13	0.79
$c\mu_{cg}\mu_{mg}\sigma_{mg}$	0.0000	0.03	0.00
$N\mu_{cg}\mu_{mg}\sigma_{mg}$	0.0008	0.45	0.01
$cN\mu_{cg}\mu_{mg}\sigma_{mg}$	0.0005	0.27	0.00
$\sigma_{cg}\mu_{mg}\sigma_{mg}$	-0.0041	-2.32	0.16
$c\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.0015	0.84	0.02
$N\sigma_{cg}\mu_{mg}\sigma_{mg}$	-0.0034	-1.90	0.11
$cN\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.0017	0.94	0.03
$\mu_{cg}\sigma_{cg}\mu_{mg}\sigma_{mg}$	-0.0051	-2.84	0.24
$c\mu_{cg}\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.0011	0.59	0.01
$N\mu_{cg}\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.0056	3.15	0.30
$cN\mu_{cg}\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.0008	0.44	0.01
		Total	96.15
	Standard Error = 0.00179		

Table 5–3: Estimates of Effects Using Response U'

expect, therefore, processor utilisation to drop as the value of $\sigma_{cg\%}$ grew larger, due to increased synchronisation overheads. This indeed appears to be the case, since the sign of the $\sigma_{cg\%}$ effect in Table 5–3 is negative. The relative unimportance of μ_{cg} indicates that it is the degree of variation in mean compute times, rather than their magnitude, which has the greater impact on processor utilisation. This is in sharp contrast to the conclusions drawn in Chapter 4, where μ_{cg} was always the most important factor, whenever U, or some transformation thereof, was used as the response variable. This apparent anomaly can be explained if one remembers that the experiments in Chapter 4 were executed with σ_{cg} set to 0, resulting in a restricted set of very regular programs and highly utilised processors. In this relatively small area of parameter space, μ_{cg} had a larger impact on performance. However, with more general programs, such as those with non-uniform synchronisation patterns, processor utilisation tends to drop, and σ_{cg} overtakes μ_{cg} as a predictor of performance.

The number of nodes, N, is also of some importance. This is because the larger the number of processes a processor has, the less chance there is of it finding itself idle. Consequently, mean processor utilisation tends to increase as the number of processes increases; this is confirmed by the positive sign of the N effect in Table 5–3.

As noted in Chapter 4, the program model and synchronous message passing protocol used mean that the parameters related to the lengths of messages (i.e. μ_{mg} and $\sigma_{mg\%}$) are of no great predictive value. Also, the degree of the process graph, c, has very little impact on performance. This is due to the parallel nature of message transfers, and the characteristics of the transputer.

It is worth re-examining Figure 5–1, the residual scatter plot for the untransformed data. The magnitudes of the residuals are, in general, considerably larger than those observed in Chapter 4. At certain parameter combinations (typically those exhibiting non-uniform behaviour) replicates with identical program parameters can differ in the observed value of U by more than 25 percentage points.

Source	% Var
seed	6.24
graph	18.30
interaction	75.46

Table 5–4: Comparison of Influence of Random Seed and Graph Shape

The only factors that differ between replicates are the random number seed and random process graph used. The impact of these cannot easily be separated, but it is convenient to think of the random number seed as influencing the detailed patterns of synchronisation, and the graph shape as influencing the characteristics of the mapping being used[1]. In any case, they both affect the form of the interaction between the program and the underlying machine, and this is an important factor which has not so far been accounted for.

To confirm that it is both the random number seed and graph shape which can influence the response variable, a combination of parameter levels which caused large residuals was taken, and a simulation trial was executed at every possible combination of four random number seeds and four random process graphs. The particular levels used were: $c = 4$, $N = 32$, $\mu_{cg} = 3000$, $\sigma_{cg\%} = 80$, $\mu_{mg} = 5000$ and $\sigma_{mg\%} = 80$. Only one replicate of this experiment was possible, since the factors that would have normally varied between replicates were controlled. Consequently, a conventional analysis of variance of table would be of little use, since F values cannot be obtained without replications. As an alternative, Table 5–4 shows the percentage variance in U attributable to the random number seed, the graph shape,

[1] A deterministic mapping strategy, such as a round-robin placement, will always map the same process to the same processor, assuming that the number of processors and processes remain constant between replicates. However, each individual random graph has a different inter-connection structure, so replicates will exhibit different mapping characteristics, even under a deterministic mapping strategy.

and the interaction between the two. These figures were obtained by expressing the sum of squares attributable to a particular source as a percentage of the total sum of squares. One can see that the most important effect, accounting for over 75% of the variation in U, comes from the interaction between the graph shape and random number seed. This supports the assertion above, that the effects of the two cannot easily be separated. The remaining variation is split in favour of the graph shape, with the choice of random number seed also being significant.

5.2 A Model for Performance Prediction

The remaining sections in this chapter illustrate how factorial experiments and standard analysis methods can be used to construct performance prediction models for non-uniform parallel programs.

It would be desirable to be able to construct an equation which, when presented with a set of predictor variables set at randomly chosen levels, would be able to produce a reasonable estimate of the response variable. Generally speaking, the analysis of variance treats factors as being qualitative in nature. However, it is possible to use a designed experiment in conjunction with an analysis of variance in order to treat factors as being quantitative. A polynomial response surface can then be fitted to the observed data using the method of *orthogonal polynomials.* This technique involves a recoding of the original predictor variables in terms of linear combinations of simple polynomials. These recoded variables are said to be orthogonal since they are guaranteed to be uncorrelated with one another. The technique enables one to decompose the effects of individual factors and their interactions into linear, quadratic, cubic and possibly higher order components. The contribution of each component to the overall regression equation can be tested, since individual sums of squares can be produced. The method requires the levels of the factors used in the experiment to be chosen so that they are

equally spaced. The generation of the response surface then follows directly from an analysis of variance, and the results are equivalent to those produced using conventional least squares techniques. The method of orthogonal polynomials is described fully in [93]. It is best illustrated by example, so an experiment is now described which was designed to enable the construction of a regression equation for non-uniform parallel programs.

5.2.1 Experiment Description

In Section 5.1 it was demonstrated that the two dominant parameters influencing the performance of non-uniform data parallel programs, were N, the number of nodes, and $\sigma_{cg\%}$, the standard deviation of compute times (expressed as a percentage of the mean compute time). It would seem reasonable to suppose, therefore, that a performance prediction model constructed using just these two parameters might provide useful estimates of performance, regardless of the levels of the other parameters.

To develop such a model, a full factorial experiment was executed with N set at six equally spaced levels, and $\sigma_{cg\%}$ set at eight equally spaced levels. The values used are summarised in Table 5–5. You will note that the four program parameters that were previously varied have been given arbitrary intermediate values. As usual, a 4×4 mesh of transputers were simulated and trials were executed for 20,000,00 clock cycles of the simulated machine. Three replications were carried out, each one differing in the random graph and random number seed used. However, unlike previous experiments, a round-robin placement was not used. This was because, for non-uniform programs, the mapping used can significantly impact the performance of programs which are identical in all other respects. To allow for an extraneous factor such as this as fully as possible within an experimental design, one should attempt to properly randomise the factor in question. The idea being that its effect will then be "averaged out" over all trials.

Program Parameters	
Param	Value(s)
N	{32, 54, 76, 98, 120, 142}
c	8
μ_{cg}	40000
$\sigma_{cg}\%$	{10, 20, 30, 40, 50, 60, 70, 80}
σ_c	10
μ_{mg}	1000
$\sigma_{mg}\%$	30
σ_m	1
Other Parameters	
Param	Value(s)
Hardware	4×4 Mesh
Placement	Restricted Random
Trial Length	20,000,000 (1 Sec)
Replications	3

Table 5–5: Parameter Settings for Model Construction Experiment Using Non-uniform Programs

The round-robin placement was deterministic, and inadequate in this respect. To overcome this problem a placement strategy known as a restricted random mapping was used. This mapping is restricted in the sense that it will attempt, as far as possible, to load the same number of processes onto each processor in the system (a truly random mapping would be a naive strategy to use).

5.2.2 Results

A residual scatter plot for the untransformed response, U, revealed that the residuals exhibited a funneled tendency, and so did not have constant variance. To remedy this problem the following Guerrero and Johnson transformation was found to be useful:

$$U' = \left(\frac{U}{100 - U}\right)^{0.11} - 1 \tag{5.2}$$

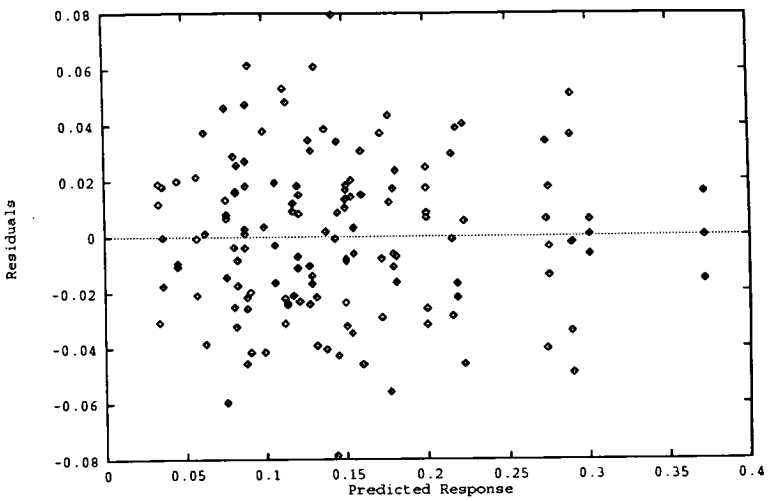

Figure 5–4: Residual Scatter Plot for U'

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
N	5	0.191	0.0383	33.61	< 0.01
$\sigma_{cg\%}$	7	0.546	0.0781	68.57	< 0.01
$N\sigma_{cg\%}$	35	0.0681	0.00195	1.71	0.021
Residual	96	0.109	0.00114		
Total	143	0.915			

Table 5–6: Extended Analysis of Variance Table Using Response U'

Interestingly, the estimate of the exponent in the above equation is very close to the value estimated in the preliminary experiment (see Equation 5.1)[2].

Figures 5–4 and 5–5 show residual scatter plots and residual normal-normal quantile plots for U', in each case there is no cause for concern. There is no reason to suppose that an additive model is not adequate, since, as previously, U' covers only a single order of magnitude, and the residuals are not of the same magnitude as U'. The corresponding analysis of variance table is given in Table

[2]In fact a common value could have used, for example 0.1, and almost all the benefits of the transformations would still have been obtained.

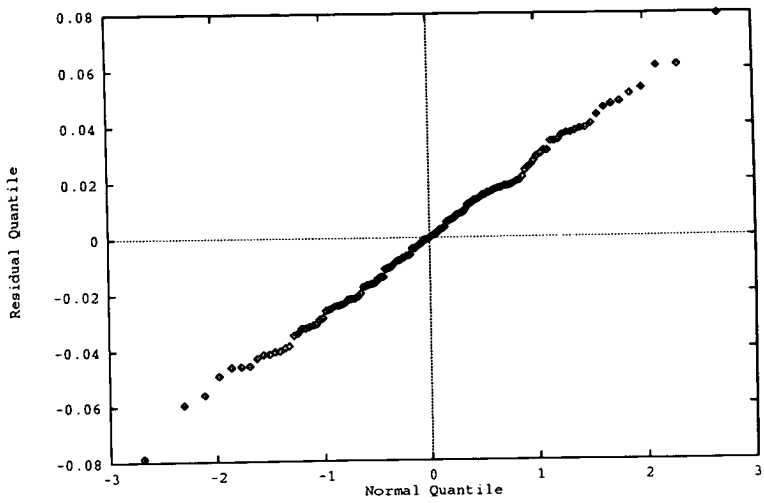

Figure 5–5: Normal Quantile-Quantile Plot for U'

5–6. This is an extended analysis of variance table with the "between treatments" stratum decomposed to show components relating to N, $\sigma_{cg\%}$, and their interaction term. This type of table can be useful when a small number of factors are being investigated. It contains much the same information as can be found in a traditional analysis of variance table, taken in conjunction with a table containing t values and estimates of effects [3]. The F value is equivalent to the square of the corresponding effect estimate, and a F probability less than 0.01 is equivalent to having a t value greater than 3. See page 129 of [76] for a full explanation of the relationship between the F and t distributions in this context.

The relative importance of the terms in the model is consistent with the results obtained in Section 5.1; with $\sigma_{cg\%}$ being most important, followed by N, and then their interaction term. The N and $\sigma_{cg\%}$ effects are significant at the 1% level, and the interaction term is significant at the 5% level. The difference between replicate means (usually shown in the "between replicates" stratum) is small, and has been absorbed into the residual term in this case. One should note that the

[3] In fact, only some information regarding the signs of effect estimates is lost.

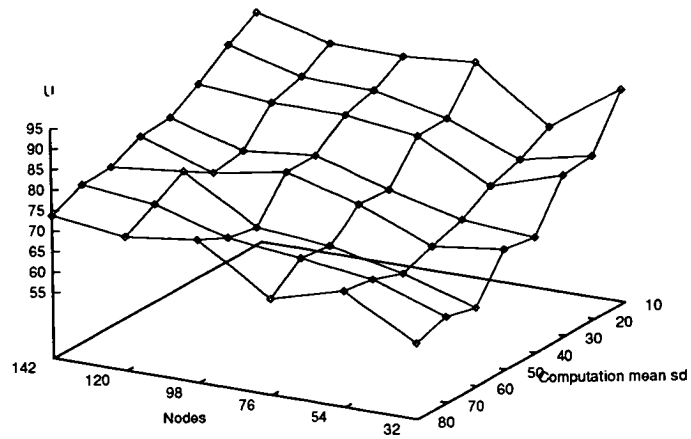

Figure 5–6: Surface Generated by Experiment

model explains over 88% of the variation in U', compared to only 84% for a similar analysis of U (not shown here), so the transformation would appear to have been worthwhile.

Figure 5–6 shows a plot of the response surface generated by this experiment, i.e. the mean values of U, over the three replicates, at each possible combination of N and $\sigma_{cg\%}$. One can see that the surface is rather uneven, although it has a general downward slope as N decreases and $\sigma_{cg\%}$ increases; this is as one would expect. To fit an extremely accurate model to this data would be a difficult and rather pointless task. The surface does not appear to lend itself to an easy interpretation in terms of any of the common functions; for example, a logarithmic, exponential or reciprocal based model. Therefore, a reasonable approach would seem to be to attempt a polynomial approximation, using the method of orthogonal polynomials.

The statistical package GENSTAT has an option which allows an analysis of variance table to be produced with entries decomposed to show the amount of variation due to the linear, quadratic, cubic and quartic components of each effect. Such a table for this experiment is given in Table 5–7. In the $N\sigma_{cg\%}$ stratum, the

Source	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
N					
Lin	1	0.175	0.175	153.98	< .01
Quad	1	0.00223	0.00223	1.96	0.165
Cub	1	0.00218	0.00218	1.91	0.17
Quart	1	0.0115	0.0115	10.08	< .01
Deviations	1	0.000135	0.000135	0.12	0.732
$\sigma_{cg\%}$					
Lin	1	0.466	0.466	409.07	< .01
Quad	1	0.068	0.068	59.77	< .01
Cub	1	0.00451	0.00451	3.97	0.049
Quart	1	0.000314	0.000314	0.28	0.601
Deviations	3	0.0079	0.00263	2.31	0.081
$N\sigma_{cg\%}$					
Lin.Lin	1	0.00127	0.00127	1.11	0.294
Quad.Lin	1	0.0087	0.0087	7.64	< .01
Lin.Quad	1	0.000003	0.000003	0.00	0.959
Cub.Lin	1	0.00141	0.00141	1.24	0.268
Quad.Quad	1	0.00227	0.00227	1.99	0.161
Lin.Cub	1	0.000193	0.000193	0.17	0.681
Deviations	29	0.0543	0.00187	1.65	0.038
Residual	96	0.109	0.00114		
Total	143	0.915			

Table 5–7: Analysis of Variance Table Showing Polynomial Components of Effects

"Lin.Lin" entry refers to that part of the interaction between N and $\sigma_{cg\%}$ which can be explained by a linear function of N and a linear function of $\sigma_{cg\%}$. The entries for "Quad.Lin" etc. can be similarly interpreted. Terms higher than the fourth order are not shown. For each effect, there is an entry marked "Deviations"; this is a measure of how much of the variation in U', due to the effect in question, is not explained by the preceding polynomial terms.

The F value and F probability columns allow one to compare the relative importance of terms in the model. It is clear that most of the variation can be explained by linear terms, although other terms are of some importance, notably the quadratic component of the $\sigma_{cg\%}$ effect. In all cases the "Deviations" compon-

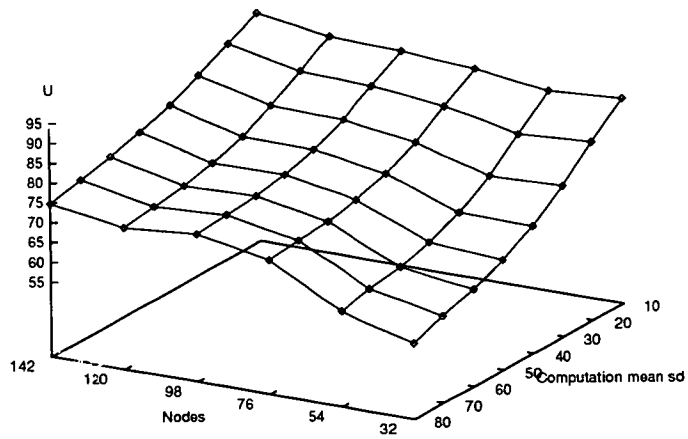

Figure 5–7: Surface Fitted to Observed Data

ents are relatively small. Some selection criteria are required in order to choose which terms should be incorporated into a regression model. The one applied here is to take only those terms which are significant at the 1% level, i.e. those which have F probabilities less than 0.01. This yields 5 terms: the linear and quartic components of the N effect, the linear and quadratic components of the $\sigma_{cg\%}$ effect, and the "Quad.Lin" component of the $N\sigma_{cg\%}$ effect.

Once the terms to be included in the model have been selected, the method of orthogonal polynomials can be used to derive a regression equation. The procedure is mechanical and reasonably straightforward using tables generated by GENSTAT, see [93] for full details. The five terms selected above lead to the following regression equation:

$$\begin{aligned} 10^5 U' = \; & 14782 + 92.9N + 0.001(N-87)^4 - 3.38(N-87)^2 - \\ & 208.47\sigma_{cg\%} + 4.74(\sigma_{cg\%} - 45)^2 - 0.028(N-87)^2(\sigma_{cg\%} - 45) \end{aligned} \tag{5.3}$$

To express the predicted values in the original scale, equation 5.2 can be reversed to give:

$$U = \frac{100(U'+1)^{9.09}}{1+(U'+1)^{9.09}} \tag{5.4}$$

The response surface corresponding to Equation 5.3 is illustrated in Figure 5–7 in terms of the original metric, U. The shape is considerably smoother than the observed surface. This is as one would expect considering that a proportion of the variation in U', attributable to the missing polynomial terms and the deviations, is not accounted for in Equation 5.3.

5.2.3 Discussion

The accuracy of the model given by Equation 5.3 can be tested by comparing the values predicted by it, to those observed in the original experiment. A useful way of displaying the results of such a comparison is to use a histogram, such as that presented in Figure 5–8, of the absolute differences between the observed and predicted values. The differences are expressed in terms of the original scale, U. The labels on the x axis refer to the lower class boundaries of the respective classes. One can see that the values are positively skewed, with many more small differences occurring than large differences. The median absolute difference between observed and predicted values is 3.1, indicating that 50% of the predicted values are within 3.1 percentage points of their corresponding observed values. While this seems reasonable, there are a significant number of larger differences. The upper quartile is equal to 6.0, so 25% of predictions are at least 6.0 percentage points away from their corresponding observed values. The worst prediction is 13.5 percentage points in error.

The errors in the predicted values originate from two sources. Firstly, even if the regression equation predicted exactly the values given by the observed surface presented in Figure 5–6, there would still be errors present due to the differences between replicates (i.e. the residuals). Additionally, inaccuracies arise from the polynomial approximation of the observed surface. The relative importance of these various sources of error can be calculated from Table 5–7, by expressing individual sum of squares as a percentage of the total sum of squares. In this way

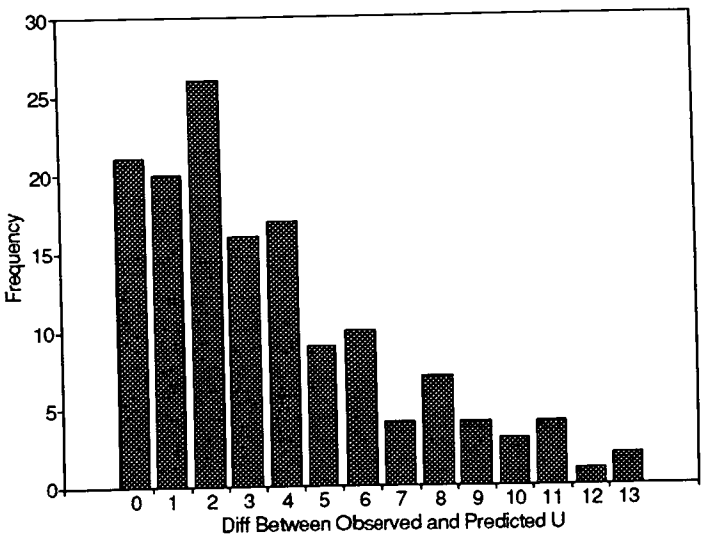

Figure 5–8: Differences In Observed and Predicted Responses for Original Experiment

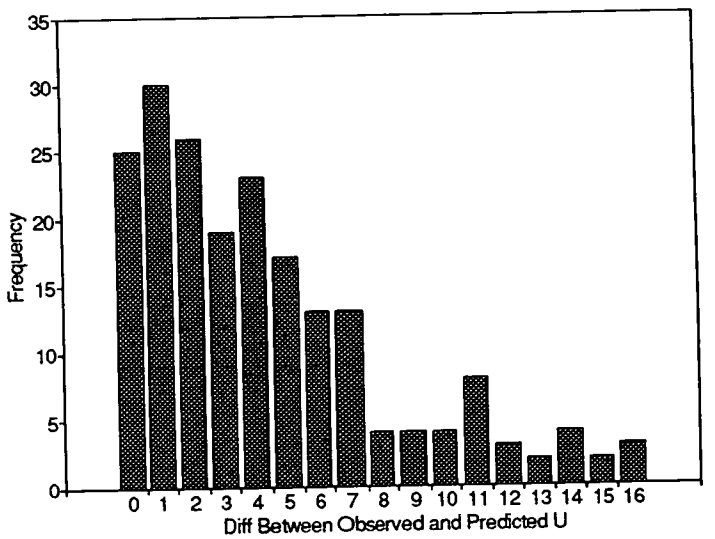

Figure 5–9: Differences In Observed and Predicted Responses for 200 Random Trials

Program Parameters	
Param	Value(s)
N	[32, 142]
c	[4, 12]
μ_{cg}	[3000, 300,000]
$\sigma_{cg}\%$	[10, 80]
σ_c	10
μ_{mg}	[10, 5000]
$\sigma_{mg}\%$	[10, 80]
σ_m	1

Other Parameters	
Param	Value(s)
Hardware	4×4 Mesh
Placement	Restricted Random
Trial Length	20,000,000 (1 Sec)

Table 5–8: Parameter Ranges used to Randomly Generate Programs

it can be calculated that the regression equation explains 79.7% of the variation in U'; the remaining polynomial terms explain 1.5%; deviations from the polynomial approximation account for 6.9%; and the residuals explain the remaining 11.9%.

N and $\sigma_{cg}\%$ were used as predictor variables because they were found to be the most influential parameters from the set:

$$\{N, c, \mu_{cg}, \sigma_{cg}\%, \mu_{mg}, \sigma_{mg}\%\}$$

Accordingly, in the derivation of the regression equation described above, the factors c, μ_{cg}, μ_{mg} and $\sigma_{mg}\%$ were set at arbitrary levels. To test how well the regression equation holds for other values of these parameters, as well as different values of N and $\sigma_{cg}\%$, 200 simulation trials were carried out, where, for each individual trial, the values of the six parameters were selected at random. The parameters were allowed to vary in the range of values explored by the preliminary two level factorial experiment described in Section 5.1.1. The exact values used are summarised in Table 5–8. Note that σ_c and σ_m continue to be set at 10 and 1 respectively, in order to ensure strongly time-invariant behaviour.

Figure 5–9 is a histogram showing the distribution of absolute differences between the observed and predicted values of U. The median value is 4.0, the upper quartile value is 6.8, and the worst prediction is 16.3 percentage points in error. These figures are slightly worse than those observed for the original experimental data (where the mean was 3.1 and the upper quartile was 6.0). However, given the crudity of the model, the results are reasonably encouraging.

A model of the type developed in this section provides a useful means of predicting the performance of an arbitrary program with known parameter values belonging to the class in question. However, so far I have only considered the effects of different parameter settings on the response variable. In the next section, techniques are presented for improving the accuracy of predictions by taking into account factors which differ between programs which have identical parameter settings. This task involves investigating the interactions which occur between a program and the machine it is being executed on.

5.3 Improving the Model: Covariates

The analysis of covariance can be used to reduce the proportion of variability attributed to experimental error within a designed experiment, by making an allowance for factors which are not directly under the experimenter's control (so called nuisance variables). This enables the properties of the response variable to be characterised with greater precision, and so improves the accuracy of a predictive model derived from such an analysis. In my case, the factors not directly under control are the particular initial mapping and random number seed used, these in turn influence the interactions which occur between a program and the machine it is being executed on. If one can incorporate appropriate terms characterising these interactions into the model, it may be possible to go some way

towards being able to account for the observed differences in performance between programs with identical parameter settings.

You will recall from Section 3.5.5 that the model underlying an analysis of covariance for a full factorial experiment with two factors, A and B, and a single covariate x, can be summarised as follows:

$$y_{ijk} = \mu + \alpha_i + \beta_j + \gamma_{ij} + \delta(x_{ijk} - \overline{x}) + \epsilon_{ijk} \tag{5.5}$$

where x_{ijk} is the measurement made on the covariate x in the k^{th} replication with factor A at level i and factor B at level j. The mean of all the x values is given by $\overline{x}$, and δ is a linear regression coefficient indicating the dependency of y_{ijk} on x_{ijk}.

A set of possible covariates is now defined, and their characteristics investigated using the experimental data obtained in Section 5.2. Throughout this section, I continue to use the transformation of the response specified by Equation 5.2, and it can be assumed that all the necessary conditions required for an analysis are satisfied unless otherwise stated.

5.3.1 Selecting a Set of Covariates

Within the simulation framework provided by MIMD, it is possible to collect almost any imaginable performance related data. Therefore, in searching for one or more suitable covariates, I am not restricted by the physical characteristics of the parallel machine; for example, the lack of a global clock or the difficulties presented by distributed memories. Since I am attempting to characterise the form of the interactions which occur between a program and the machine it is executed on, the covariates selected should be concerned with such things as the structure of the mapping, patterns of synchronisation and rates of computation. Listed below is the set of covariates that were investigated. Each was chosen for its possible predictive properties, with respect to the relative efficiency of the execution of a candidate program.

hops_mean

This is the mean, over all channels, of the number of processor hops between sender and receiver. So, if $D(i,j)$ represents the length, in processor hops, of the channel c_{ij} connecting process i and process j; and NC represents the total number of channels, then:

$$hops\_mean = \frac{1}{NC} \sum_{\forall c_{ij}} D(i,j) \tag{5.6}$$

prox

This is a measure of the proximity of processes, a similar metric is used in [14] in the context of a distributed mapping algorithm. The derivation of *prox* is similar to that of *hops_mean*, except that channels which span more than one processor hop are given extra weight, so[4]:

$$prox = \frac{1}{NC} \sum_{\forall c_{ij}} \begin{cases} 1, & D(i,j) \leq 1 \\ 2 \times D(i,j), & D(i,j) > 1 \end{cases} \tag{5.7}$$

prox_len

This is similar to *prox*, except some information regarding the average lengths of messages is included. If $ML(i,j)$ represents the mean length of messages sent down channel c_{ij}, then:

$$prox\_len = \frac{1}{NC} \sum_{\forall c_{ij}} \begin{cases} 1 \times ML(i,j), & D(i,j) \leq 1 \\ 2 \times D(i,j) \times ML(i,j), & D(i,j) > 1 \end{cases} \tag{5.8}$$

---

[4]Since no distinction is made between processes situated on the same processor, and those situated on adjacent processors, an assumption explicit in the definition of *prox* (and *prox_len* for that matter) is that the cost of a context switch is the same as the cost of sending a message to a neighbouring processor. This is unlikely to be the case. For example, on the transputer a context switch takes approximately 5 microseconds. In contrast, the initiation cost of sending a message to a neighbouring processor is approximately 20 microseconds.

sync_mean

This is a measure of the average time taken, across all channels, for the processes at either end of the channels to synchronise. The synchronisation time for a particular message on a given channel is the absolute difference between the time at which the process at one end is ready to send (or receive), and the time at which the process at the other end is ready to receive (or send). If $SYNC(i,j)$ represents the mean synchronisation time for channel c_{ij}, then:

$$sync\_mean = \frac{1}{NC} \sum_{\forall c_{ij}} SYNC(i,j) \tag{5.9}$$

sync_sd

This is a measure of the spread, across all channels, of the time taken for processes to synchronise. It is equal to the standard deviation of the $SYNC(i,j)$ times:

$$sync\_sd = \frac{1}{NC} \sqrt{NC \sum_{\forall c_{ij}} SYNC(i,j)^2 - \left(\sum_{\forall c_{ij}} SYNC(i,j) \right)^2} \tag{5.10}$$

link_cont

This is a measure of how much contention there is for links. If $NUSED(k)$ represents the total number of times link k is used, and $WAIT(k)$ represents the total number of times a message has to wait for link k to become free, then:

$$link\_cont = \frac{\sum_{\forall k} WAIT(k)}{\sum_{\forall k} NUSED(k)} \tag{5.11}$$

ce_ratio_mean

This quantity is related to the average rate of computation achieved by the processes in the system. If $COMP(p)$ represents the amount of computation achieved by process p, $ELAPSED(p)$ represents the actual time taken to

achieve that computation[5] and NP is equal to the number of processes, then:

$$ce\_ratio\_mean = \frac{1}{NP} \sum_{\forall p} \frac{COMP(p)}{ELAPSED(p)} \tag{5.12}$$

ce_ratio_sd

This is a measure of the spread in the rates of computation achieved by the processes in the system. It is equal to the standard deviation of the compute-elapsed time ratios defined above:

$$ce\_ratio\_sd = \frac{1}{NP} \sqrt{NP \sum_{\forall p} \left(\frac{COMP(p)}{ELAPSED(p)} \right)^2 - \left(\sum_{\forall p} \frac{COMP(p)}{ELAPSED(p)} \right)^2} \tag{5.13}$$

A metric similar to *prox_len*, using message counts rather than message lengths to weight channels which are used frequently, might seem attractive. However, the loosely synchronous structure of the program model assumed here ensures that, at any one time, the number of messages that have been sent down any two channels is approximately equal. Therefore, this metric would be of little use in this case.

Table 5–9 shows the residual sum of squares obtained from carrying out an ANCOVA incorporating each of the individual performance measure defined above as a covariate. The lower the residual sum of squares, the greater the proportion of variation in the response variable that has been explained. One can see that the three measures concerned with the distances between processes (namely *hops_mean*, *prox* and *prox_len*) have had very little affect on the residual sum of squares. This is not surprising if one considers the response variable being used. There is no direct relationship between the time taken for message transfers and processor utilisation, since while one process is waiting for a message, another is

[5]This will generally be longer than the actual compute time due to contention for compute resources.

Covariate	Residual Sum Sqs.
None	0.1093
hops_mean	0.1090
prox	0.1088
prox_len	0.1090
sync_mean	0.0342
sync_sd	0.0392
link_cont	0.1086
ce_ratio_mean	0.0755
ce_ratio_sd	0.0741

Table 5–9: Comparison of the Impact of Covariates

likely to be available to use any spare compute resources. The metric *link_cont* also fails to reduce the residual sum of squares by any significant amount, so link contention does not appear to affect performance to any great degree for the program model assumed here. One can conclude that these four measures do not contain enough information (with respect to the efficiency with which a program is being executed) to be used as covariates.

The remaining four measures, *sync_mean*, *sync_sd*, *ce_ratio_mean* and *ce_ratio_sd*, all seem to have a significant impact on the residual sum of squares. A strong positive correlation exists between the observed values of *sync_mean* and *sync_sd*, and a similar although weaker correlation exists between the observed values of *ce_ratio_mean* and *ce_ratio_sd*. The corresponding correlation coefficients are 0.98 and 0.72 respectively. Consequently, it would seem sensible to select a single covariate from each of these two groups, since a second from either group would provide largely redundant information. From Table 5–9 it can be seen that *sync_mean* and *ce_ratio_sd* have the greatest impact on the residual sum of squares, so I will concentrate on these.

The correlation coefficient between *sync_mean* and *ce_ratio_sd* is -0.47, indicating that they are not particularly strongly correlated. This is promising, since

one could expect a model with both variables as covariates to reduce the residual sum of squares to a greater extent than either could do separately.

The question remains as to how one would interpret various values of *sync_mean* and *ce_ratio_sd*. The metric *sync_mean* can be thought of as answering the following question:

> For how long, on average, does a sender (or receiver) have to wait for the remote receiver (or sender) to become ready?

Similarly, the metric *ce_ratio_sd* can be thought of as quantifying an answer to the question:

> To what extent are all processes computing at the same rate?

The smaller the value of *ce_ratio_sd*, the more uniform the rates of computation are across all the processes in the system.

Both of these metrics are a product of the interaction between the program and the underlying machine, and consequently they provide information relating to the relative efficiency with which a program is being executed. However, the metrics are also related in part to the inherent characteristics of the program. For example, a program with badly synchronised processes will tend to exhibit a large value of *sync_mean*, regardless of how efficiently it is executed. Even so, it is likely that the smaller the value of *sync_mean* obtained, the faster the execution will be. The possibility of using both *sync_mean* and *ce_ratio_sd* as covariates in a predictive performance model will now be investigated.

5.3.2 An Improved Model

Before I can proceed with an analysis using both *sync_mean* and *ce_ratio_sd* as covariates, it is necessary to confirm that these two metrics satisfy the assumptions

Source	D o F	Sum of Squares	Mean Squares	F Ratio	Cov. Ef.	F Prob.
N	5	0.158	0.0316	87.62	0.26	< 0.01
$\sigma_{cg\%}$	7	0.088	0.0126	34.88	0.7	< 0.01
$N\sigma_{cg\%}$	35	0.0466	0.00133	3.7	0.99	< 0.01
Covariate	1	0.075	0.075	208.15		< 0.01
Residual	95	0.0342	0.000361			
Total	143	0.915				

Table 5–10: Analysis of Covariance Using *sync_mean* as a Covariate

for a valid analysis (see Section 3.5.5). Tables 5–10 and 5–11 show the full details of separate ANCOVA analyses carried out for *sync_mean* and *ce_ratio_sd*. The first thing to notice is that the treatment sums of squares and the covariate sum of squares do not add up to the total sum of squares minus the residual sum of squares, as one might imagine. This is due to the non-orthogonal nature of the relationship between the treatments and the covariate. A detailed explanation of this phenomenon is given in [31], but in any case, the F ratio values still indicate the relative importance of terms in the model. In both tables, the Covariance Efficiency Factor column (marked "Cov. Ef.") is a measure of the extent to which the effects of individual treatment terms in the model have been masked by the covariate. A value close to zero indicates that the term in question is completely correlated with the covariate, and so once the covariate has been fitted, there is no information left about the corresponding treatment effect. A value that is greater than 0, but significantly less than 0.8, is not particularly desirable. Such a value should be taken as a warning that the measurement used as a covariate has been influenced by the treatment term, which, while not invalidating an ANCOVA, complicates its interpretation. A value of 1 indicates that the treatment term and the covariate are orthogonal.

One can see in Table 5–11 that, when *ce_ratio_sd* is used as a covariate, the covariance efficiency factors give no cause for concern. However, Table 5–10 shows that when *sync_mean* is the covariate, the covariance efficiency factor for the "N" stratum is only equal to 0.26. This indicates that *sync_mean* is relatively strongly

Source	D o F	Sum of Squares	Mean Squares	F Ratio	Cov. Ef.	F Prob.
N	5	0.0764	0.0153	19.6	0.83	< 0.01
$\sigma_{cg\%}$	7	0.143	0.0204	26.12	0.85	< 0.01
$N\sigma_{cg\%}$	35	0.0423	0.00121	1.55	0.99	0.05
Covariate	1	0.0352	0.0352	45.13		< 0.01
Residual	95	0.0741	0.00078			
Total	143	0.915				

Table 5–11: Analysis of Covariance Using *ce_ratio_sd* as a Covariate

Source	D o F	Sum of Squares	Mean Squares	F Ratio	Cov. Ef.	F Prob.
N	5	0.191	0.0381	96.88	0.98	< 0.01
$\sigma_{cg\%}$	7	0.0773	0.011	28.05	0.64	< 0.01
$N\sigma_{cg\%}$	35	0.0404	0.00115	2.93	0.99	< 0.01
Covariate	1	0.0719	0.0719	182.64		< 0.01
Residual	95	0.0374	0.000394			
Total	143	0.915				

Table 5–12: Analysis of Covariance Using Modified Definition of *sync_mean* as a Covariate

correlated with the value of N. This is not surprising, since, once a process is ready to send (or receive) on a channel, the time taken for the remote process to become ready is proportional to the load of the remote processor. Now, as N is increased, the load on the processors also increases. To try and compensate for this, the definition of *sync_mean* can be modified as follows:

$$sync\_mean = \frac{1}{(LOAD \times NC)} \sum_{\forall c_{ij}} SYNC(i,j) \tag{5.14}$$

where $LOAD$ represents the mean number of processes per processor. The results obtained using this modified definition of *sync_mean* as a covariate are summarised in Table 5–12. It is clear that the strong dependence on N has disappeared; also, the residual sum of squares is equal to 0.0374 , which is comparable to the value obtained with the original definition of *sync_mean*. The definition given in Equation 5.14 will be assumed from now on.

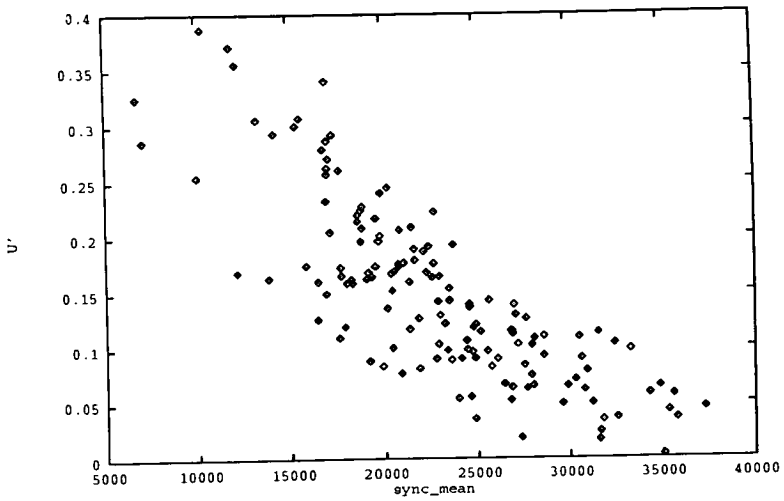

Figure 5–10: U' vs. *sync_mean* Scatter Plot

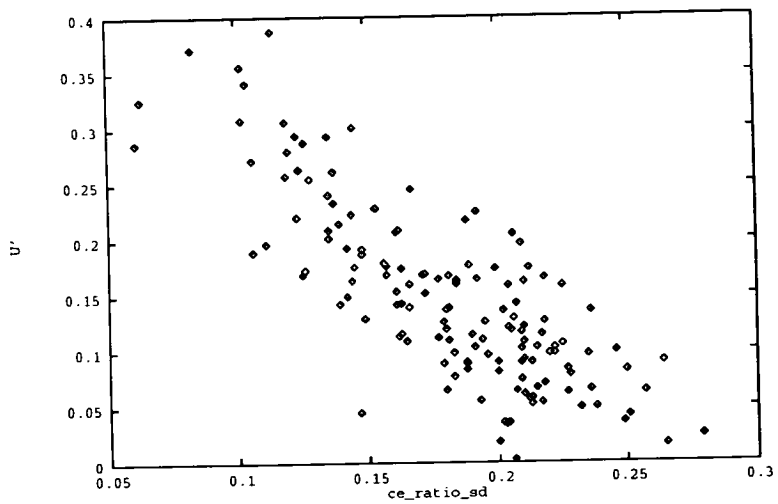

Figure 5–11: U' vs. *ce_ratio_sd* Scatter Plot

Source	D o F	Sum of Squares	Mean Squares	F Ratio
heterogeneous slopes	47	0.0212	0.000451	1.33
homogeneous slopes	48	0.0162	0.000338	
Residual	96	0.0374		

Table 5–13: Test for Homogeneous Regression Slopes for *sync_mean*

Source	D o F	Sum of Squares	Mean Squares	F Ratio
heterogeneous slopes	47	0.0417	0.000887	1.31
homogeneous slopes	48	0.0324	0.000675	
Residual	96	0.0741		

Table 5–14: Test for Homogeneous Regression Slopes for *ce_ratio_sd*

The remaining conditions which should be satisfied for *sync_mean* and *ce_ratio_sd* are: firstly, the linearity of the relationship between covariate and response; and secondly, the homogeneity of regression slopes associated with different treatment groups. Figures 5–10 and 5–11 plot the values of *sync_mean* and *ce_ratio_sd* against the response, U'. It is clear that, in both cases, there is a definite tendency towards linearity; so the first condition is satisfied. The second condition can be verified using a technique described in [66] designed to test the homogeneity of regression slopes between treatment groups. In the experiment being analysed here, there are $8 \times 6 = 48$ different treatment groups, corresponding to the different possible combinations of levels of N and $\sigma_{cg\%}$. Within each treatment group there are three observations, one for each replication of the experiment. Tables 5–13 and 5–14 attribute the residuals obtained, when using *sync_mean* and *ce_ratio_sd* as covariates, to that part which can be explained by a single regression slope, and that part which can only be explained by heterogeneous regression slopes. In both cases the null hypothesis, that treatment regression slopes are equal, can be accepted at the 1% significance level since $F_{(.01,47,48)} = 1.96$.

All the required conditions have been satisfied for the variables *sync_mean* and *ce_ratio_sd* to be used as covariates. The results obtained when incorporating both

Source	D o F	Sum of Squares	Mean Squares	F Ratio	Cov. Ef.	F Prob.
N	5	0.107	0.0215	62.38	0.81	< 0.01
$\sigma_{cg\%}$	7	0.0441	0.0063	18.3	0.62	< 0.01
$N\sigma_{cg\%}$	35	0.0331	0.000947	2.75	0.97	< 0.01
Covariates	1	0.0769	0.0769	111.63		< 0.01
Residual	95	0.0324	0.000345			
Total	143	0.915				

Table 5–15: Analysis of Covariance Using *sync_mean* and *ce_ratio_sd* as a Covariates

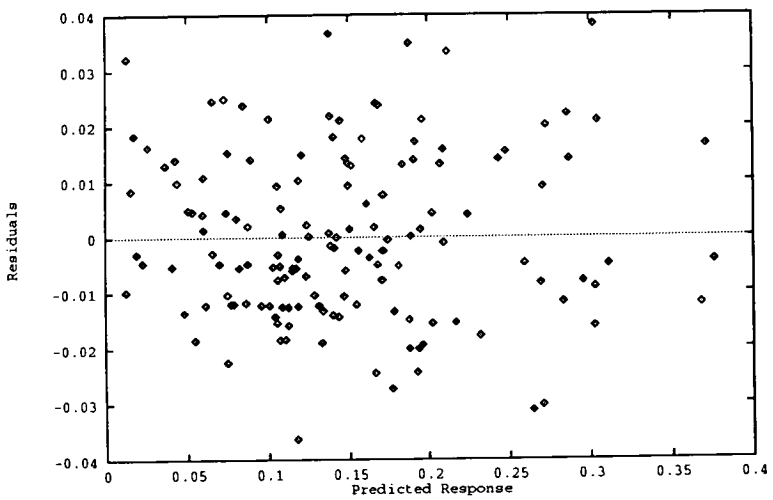

Figure 5–12: Residual Scatter Plot with *sync_mean* and *ce_ratio_sd* as Covariates

sync_mean and *ce_ratio_sd* into an analysis are presented in Table 5–15. It should be remembered that the transformation specified by Equation 5.2 is still being used. The covariance efficiency factors give no great cause for concern, although a little more of the $\sigma_{cg\%}$ effect is masked than would be ideal. The residual sum of squares is equal to 0.0324, which is an improvement over any of the single covariate analyses. It can be seen that the N, $\sigma_{cg\%}$ and $N\sigma_{cg\%}$ effects, as well as the covariate effects, are all significant at the 1% level. The residual scatter plots and residual normal-normal quantile plots associated with this analysis are shown in Figures 5–12 and 5–13; they give no cause for concern.

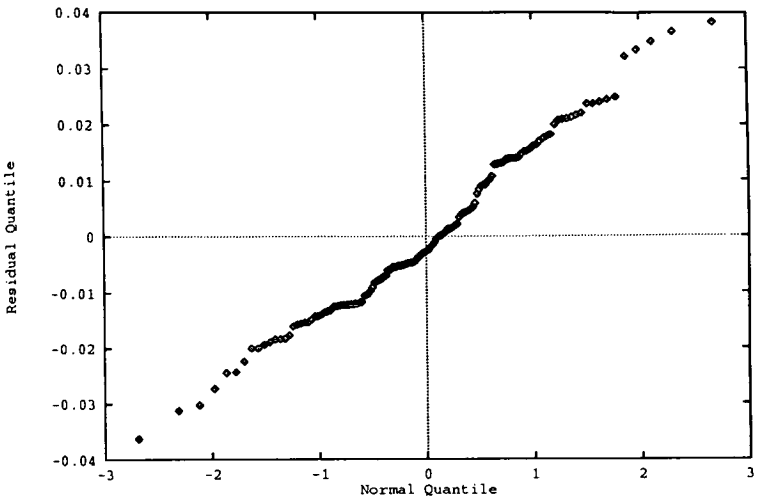

Figure 5–13: Normal Quantile-Quantile Plot with *sync_mean* and *ce_ratio_sd* as Covariates

If U'_{ijk} represents the observed value of U' in the kth replication with N at level i and $\sigma_{cg\%}$ at level j, and *sync_mean* and *ce_ratio_sd* are denoted by x and y respectively, then the model underlying the above analysis can be summarised by the following equation:

$$\begin{aligned} U'_{ijk} &= \mu + P_{ij} + \delta_x(x_{ijk} - \overline{x}) + \delta_y(y_{ijk} - \overline{y}) + \epsilon_{ijk} \\ &= \mu + P_{ij} - 0.0000078(x_{ijk} - 22952) - 0.3(y_{ijk} - 0.18) + \epsilon_{ijk} \end{aligned} \tag{5.15}$$

where P_{ij} represents that part of the variation explained by the program parameters, N and $\sigma_{cg\%}$, set at levels i and j respectively. The regression slopes (δ_x and δ_y) used in the above equation were generated by GENSTAT.

That part of Equation 5.15 represented by P_{ij} can be approximated by a polynomial function using the method of orthogonal polynomials. In this way a general purpose performance prediction equation can be constructed. The details of this technique were discussed in Section 5.2, and the derivation of the function is straightforward. Selecting those polynomial component terms which are significant at the 1% level, an expression to approximate P_{ij} for arbitrary values of N

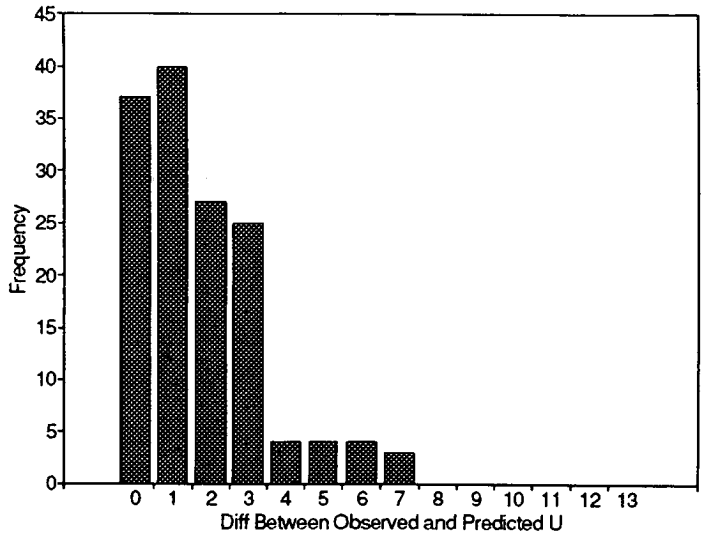

Figure 5–14: Performance of an Improved Model

and $\sigma_{cg\%}$ can be written as follows:

$$\begin{aligned} P = \; & 0.00084(N-87) - 0.00056(\sigma_{cg\%} - 45) + 0.0000398(\sigma_{cg\%} - 45)^2 - \\ & 0.0000117(N-87)(\sigma_{cg\%} - 45) - 0.0209 \end{aligned} \tag{5.16}$$

Combining Equations 5.15 and 5.16 leads to the following regression equation:

$$\begin{aligned} 10^5 U' = \; & 34654 + 136.7N - 312.4\sigma_{cg\%} + 3.98\sigma^2_{cg\%} - \\ & 1.17N\sigma_{cg\%} - 0.78x - 30000y \end{aligned} \tag{5.17}$$

The performance of the model given by Equation 5.17 was tested by comparing the values predicted by it, with the actual data observed in the original experiment (described in Section 5.1.1). These results are summarised in Figure 5–14, and they compare very well to those illustrated in Figure 5–8 (where the model contained no covariates). The median value is 1.9 and the upper quartile is 3.2, compared to 3.1 and 6.0 for the model with no covariates. The worst prediction is only 7.4 percentage points in error.

To test the generality of Equation 5.17, performance predictions were made for the same 200 randomly generated programs used in Section 5.2.3. However,

	sync_mean	*ce_ratio_sd*
c	-0.259	-0.023
μ_{cg}	0.853	-0.506
μ_{mg}	0.018	-0.052
$\sigma_{cg\%}$	0.001	0.013

Table 5–16: Check for Influence of "Lurking Variables" on Covariates

the results obtained were poor, and are not presented here. This failure can be attributed to the influence of so called "lurking variables" on the covariates i.e. there are factors which affect the covariates that have not been accounted for in the model. To confirm this, the relationships between *sync_mean* and *ce_ratio_sd* and the program parameters c, N, μ_{mg} and $\sigma_{mg\%}$ (which were fixed during the derivation of Equation 5.17) were examined. Table 5–16 shows the correlation coefficients calculated between *sync_mean* and *ce_ratio_sd* and the four program parameters, using data from the 200 random trials. The only correlations of note exist with μ_{cg}, the mean compute time of the processes. In the case of *sync_mean*, this correlation is intuitive. One can conclude that the value of *sync_mean* (and *ce_ratio_sd* to a lesser extent) is strongly influenced by the value of μ_{cg}, therefore it seems unlikely that Equation 5.17 will hold for any values of μ_{cg} significantly different from 40000.

Despite this setback, it is still possible to show that the inclusion of covariates in a general purpose performance prediction model can improve accuracy. This can be achieved using conventional regression methods. GENSTAT was used to derive a regression equation using conventional least squares techniques. The data required to construct the model was taken from 200 random trials executed specifically for this purpose. The selection of terms to include in the model was guided by the terms found in Equation 5.17, and the interactions between the covariates and μ_{cg} noted above. Tables of t-values were used to judge the relative importance of the terms included in a particular model. After some experimentation, the following regression equation was found to explain the greatest proportion of

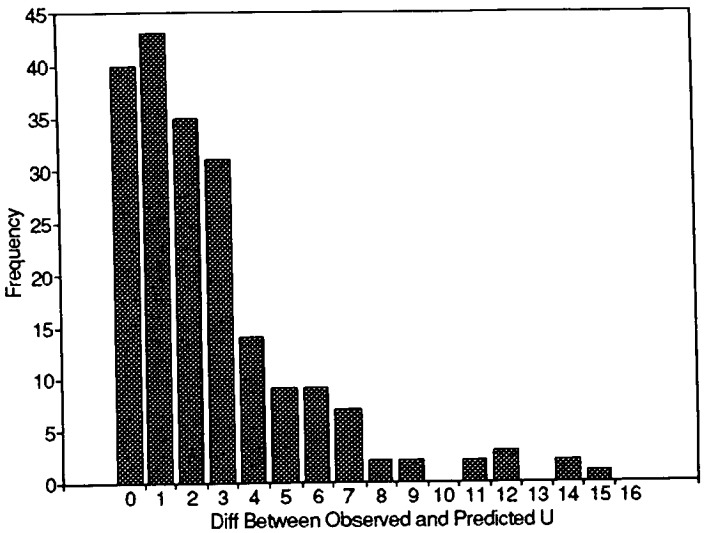

Figure 5–15: Performance of a General Model for 200 Random Trials

variation in the response:

$$\begin{aligned} 10^5 U' = \; & -26906 - 37.1N + 209.7\sigma_{cg}\% - 1.65\sigma^2_{cg}\% + \\ & 0.2x - 51722y - 0.000012\mu_{cg}x + 10^{(-11)} \times 0.23\mu^2_{cg}x \end{aligned} \tag{5.18}$$

where x represents *sync_mean*, y represents *ce_ratio_sd*, and U' was calculated using the following Guerrero and Johnson transformation:

$$U' = \left(\frac{U}{100 - U}\right)^{-0.1} - 1 \tag{5.19}$$

The transformation was necessary to satisfy the assumptions underlying such a regression analysis, which are much the same as those discussed in Chapter 3 with reference to ANOVA and ANCOVA techniques.

Note that most of the terms appearing in Equation 5.17 also appear in Equation 5.18. In addition, there are some terms to characterise the interaction between μ_{cg} and *sync_mean*. The interaction between μ_{cg} and *ce_ratio_sd*, which was already known to be the less important interaction (see Table 5–16), does not appear to have any strong predictive properties in this case.

The performance of Equation 5.18 was tested, once again using the set of 200 random trials introduced in Section 5.2.3. The results of a comparison between

the actual data, and the figures predicted by Equation 5.18 are presented in the form of a histogram in Figure 5–15. The median value is 2.3, the upper quartile is 4.1 and the worst prediction is 15.7 percentage points in error. This is an improvement on the results obtained using the model with no covariates, where the corresponding figures were 4.0, 6.8 and 16.3 respectively.

The general purpose regression methods used to derive Equation 5.18 are not ideal, since they lack the rigour of formal ANOVA and ANCOVA techniques, and rely on the ad-hoc selection of terms to include in the model. The method has been tolerated in order to illustrate the general predictive properties of *sync_mean* and *ce_ratio_sd*.

This section has demonstrated how the predictive power of a small number of carefully selected covariates can be used to improve the accuracy of a general purpose performance prediction model. So far, I have concentrated on a single processor configuration, a 4×4 mesh. Section 5.4 extends these investigations by considering a larger processor domain.

5.4 A 7 by 7 Processor Mesh

To see whether the results obtained in Sections 5.1-5.3 apply to larger processor domains, a similar set of experiments were carried out for a 7×7 mesh of simulated processors; the results are summarised below. Guerrero and Johnson transformations were used whenever necessary, but the exact details of these are not presented here.

Table 5–17 contains the effect estimates for a two level full factorial experiment carried out varying the usual six program parameters; namely c, N, μ_{cg}, $\sigma_{cg\%}$, μ_{mg} and $\sigma_{mg\%}$. Except for N, the levels for these factors were set to be the same as those used in Section 5.1. The high and low values used for N were 98 and 198, this change was necessary in order to ensure that there were at least twice as many

Effect	Estimate	t-value	% Var
(gm)	70.24	316.49	
c	0.61	2.76	0.17
N	3.04	13.71	4.31
cN	-0.27	-1.22	0.03
μ_{cg}	1.58	7.14	1.17
$c\mu_{cg}$	-1.74	-7.84	1.41
$N\mu_{cg}$	-0.58	-2.63	0.16
$cN\mu_{cg}$	-0.21	-0.97	0.02
σ_{cg}	-12.04	-54.27	67.58
$c\sigma_{cg}$	0.43	1.93	0.09
$N\sigma_{cg}$	3.31	14.92	5.11
$cN\sigma_{cg}$	0.11	0.51	0.01
$\mu_{cg}\sigma_{cg}$	-1.76	-7.91	1.44
$c\mu_{cg}\sigma_{cg}$	-0.56	-2.54	0.15
$N\mu_{cg}\sigma_{cg}$	1.82	8.21	1.55
$cN\mu_{cg}\sigma_{cg}$	0.03	0.11	0.00
μ_{mg}	-2.49	-11.24	2.90
$c\mu_{mg}$	0.22	0.98	0.02
$N\mu_{mg}$	1.30	5.84	0.78
$cN\mu_{mg}$	-0.01	-0.03	0.00
$\mu_{cg}\mu_{mg}$	2.09	9.40	2.03
$c\mu_{cg}\mu_{mg}$	-0.17	-0.74	0.01
$N\mu_{cg}\mu_{mg}$	-1.41	-6.34	0.92
$cN\mu_{cg}\mu_{mg}$	-0.06	-0.29	0.00
$\sigma_{cg}\mu_{mg}$	0.75	3.39	0.26
$c\sigma_{cg}\mu_{mg}$	-0.12	-0.53	0.01
$N\sigma_{cg}\mu_{mg}$	-1.29	-5.81	0.77
$cN\sigma_{cg}\mu_{mg}$	-0.01	-0.04	0.00
$\mu_{cg}\sigma_{cg}\mu_{mg}$	-0.91	-4.09	0.38
$c\mu_{cg}\sigma_{cg}\mu_{mg}$	0.26	1.17	0.03
$N\mu_{cg}\sigma_{cg}\mu_{mg}$	1.68	7.58	1.32
$cN\mu_{cg}\sigma_{cg}\mu_{mg}$	0.06	0.27	0.00
σ_{mg}	-0.97	-4.35	0.43
$c\sigma_{mg}$	0.15	0.69	0.01
$N\sigma_{mg}$	1.08	4.87	0.54
$cN\sigma_{mg}$	0.00	0.01	0.00
$\mu_{cg}\sigma_{mg}$	0.55	2.49	0.14
$c\mu_{cg}\sigma_{mg}$	-0.18	-0.82	0.02
$N\mu_{cg}\sigma_{mg}$	-0.69	-3.12	0.22
$cN\mu_{cg}\sigma_{mg}$	0.00	0.02	0.00
$\sigma_{cg}\sigma_{mg}$	0.38	1.71	0.07
$c\sigma_{cg}\sigma_{mg}$	-0.11	-0.51	0.01
$N\sigma_{cg}\sigma_{mg}$	-0.22	-0.98	0.02
$cN\sigma_{cg}\sigma_{mg}$	-0.11	-0.51	0.01
$\mu_{cg}\sigma_{cg}\sigma_{mg}$	-0.92	-4.13	0.39
$c\mu_{cg}\sigma_{cg}\sigma_{mg}$	0.09	0.40	0.00
$N\mu_{cg}\sigma_{cg}\sigma_{mg}$	0.02	0.09	0.00
$cN\mu_{cg}\sigma_{cg}\sigma_{mg}$	-0.01	-0.03	0.00
$\mu_{mg}\sigma_{mg}$	-1.10	-4.93	0.56
$c\mu_{mg}\sigma_{mg}$	0.15	0.68	0.01
$N\mu_{mg}\sigma_{mg}$	1.30	5.88	0.79
$cN\mu_{mg}\sigma_{mg}$	0.04	0.16	0.00
$\mu_{cg}\mu_{mg}\sigma_{mg}$	0.90	4.06	0.38
$c\mu_{cg}\mu_{mg}\sigma_{mg}$	-0.12	-0.54	0.01
$N\mu_{cg}\mu_{mg}\sigma_{mg}$	-0.78	-3.54	0.29
$cN\mu_{cg}\mu_{mg}\sigma_{mg}$	0.08	0.38	0.00
$\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.76	3.41	0.27
$c\sigma_{cg}\mu_{mg}\sigma_{mg}$	-0.15	-0.67	0.01
$N\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.06	0.28	0.00
$cN\sigma_{cg}\mu_{mg}\sigma_{mg}$	-0.02	-0.10	0.00
$\mu_{cg}\sigma_{cg}\mu_{mg}\sigma_{mg}$	-0.67	-3.03	0.21
$c\mu_{cg}\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.18	0.82	0.02
$N\mu_{cg}\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.26	1.16	0.03
$cN\mu_{cg}\sigma_{cg}\mu_{mg}\sigma_{mg}$	0.08	0.36	0.00
		Total	97.08
	Standard Error = 0.222		

Table 5–17: Estimates of Effects for 7×7 Processor Mesh

processes as processors for the lower value of N. One can see from Table 5–17 that, once again, the three most important effects are N, $\sigma_{cg\%}$ and their first order interaction. However, the importance of N has decreased and the importance of $\sigma_{cg\%}$ has increased compared to the results observed for the 4×4 mesh (see Table 5–3). The overall importance of these three terms has decreased, from accounting for 84% of the variation for the 4×4 mesh, to 77% of the variation for the 7×7 mesh.

If one compares the effects of the other terms in Table 5–17 with those observed in Table 5–3, it can be seen that the μ_{cg} effect is of approximately the same importance. However, the influence of μ_{mg} has more than doubled, it now explains 2.9% of the variation in the response. The increased importance of terms involving μ_{mg} is to be to expected, since on a larger processor domain the average distance between processes will tend to increase, assuming that the shape of the process graph is unrelated to the shape of the processor graph. Now, as message sizes increase, the time taken for a message to traverse a link will increase proportionately. The greater the number of links messages have to traverse, the larger the resulting synchronisation delays will be; this is likely to have a negative impact on processor utilisation if these times become too large. The effects of c and $\sigma_{mg\%}$ are largely unchanged between the two experiments. It is interesting to note that higher order interactions are of less importance for the 4×4 mesh than the 7×7 mesh; this suggests the prescence of more complex behaviour patterns in the latter.

As previously, the parameters N and $\sigma_{cg\%}$ were selected for further investigation. The choice is not as clear cut as with the 7×7 mesh, but N and $\sigma_{cg\%}$ still explain over 77% of the variation in the response. A full factorial experiment was carried out with N set at 6 levels and $\sigma_{cg\%}$ set at 8 levels.

A predictive model with no covariates was constructed using the method of orthogonal polynomials, selecting those terms significant at the 1% level. The

	Median	Upper Quartile	Worst Case
Self-test, No Covariates	3.5 (3.1)	5.9 (6.0)	11.7 (13.5)
Random Trials, No Covariates	3.7 (4.0)	6.2 (6.8)	17.2 (16.3)
Self-test, 2 Covariates	1.6 (1.9)	3.1 (3.2)	6.1 (7.4)
Random Trials, 2 Covariates	2.4 (2.3)	4.1 (4.1)	14.8 (15.7)

Table 5–18: Comparison of the Performance of Regression Equations (4×4 Figures in Brackets)

resulting equation was:

$$\begin{aligned} 10^5 U' = \; & -56758 + 894.3N - 8.82(N-148)^2 - 0.01(N-148)^3 + \\ & 0.0048(N-148)^4 - 1098.2\sigma_{cg\%} + 21.4(\sigma_{cg\%} - 45)^2 - \\ & 0.113(\sigma_{cg\%} - 45)(N-148)^2 - 0.233(N-148)(\sigma_{cg\%} - 45)^2 + \\ & 0.0134(N-148)(\sigma_{cg\%} - 45)^3 - 12.4(N-148)\sigma_{cg\%} \end{aligned} \tag{5.20}$$

Comparing the U values predicted by this equation with the observed experimental data resulted in a median absolute difference of 3.5 percentage points. The upper quartile was 5.9 and the worst prediction was 11.7 percentage points in error. To further test Equation 5.20, 200 random trials were executed with the the six program parameters being randomly varied in the range explored by the preliminary two level factorial experiment. A different random number seed was used for each trial. The median absolute difference between observed and predicted values was 3.7, the upper quartile was 6.2 and the worst prediction was 17.2 percentage points in error. These two sets of results are summarised in the first two lines of Table 5–18, the figures in brackets refer to the corresponding figures obtained for the 4×4 mesh. One can see the same trends occurring for both the 7×7 mesh and the 4×4 mesh, with the predictions for the random trials being slightly less accurate than the predictions for the original experimental data.

To improve the accuracy of the model given by Equation 5.20, the covariates *sync_mean* and *ce_ratio_sd* were incorporated into the analysis. The resulting performance prediction equation was:

$$\begin{aligned} 10^5 U' = \; & 913429 - 13781N - 26761\sigma_{cg\%} + 85.7N^2 + 256.3\sigma_{cg\%}^2 + \\ & 362.2\sigma_{cg\%}N - 1.9\sigma_{cg\%}N^2 - 1.6N\sigma_{cg\%}^2 + 0.004\sigma_{cg\%}N^3 + \\ & 0.0107N\sigma_{cg\%}^3 - 0.18N^3 - 1.6\sigma_{cg\%}^3 - 2.65x - 110000y \end{aligned} \tag{5.21}$$

where x represents *sync_mean* and y represents *ce_ratio_sd*. Testing this equation on the original experimental data resulted in a median absolute difference between observed and predicted values of 1.6. The upper quartile was 3.1, and the worst prediction was 6.1 percentage points in error. These figures are a significant improvement over those obtained when no covariates were used, and one can see from the third line of Table 5–18 that, once again, the improvements are comparable to those obtained for the 4×4 mesh.

The performance of Equation 5.21, when presented with the set of 200 random trials, was predictably poor due to interactions between the covariates and μ_{cg} which were not allowed for in the model. To test the usefulness of *sync_mean* and *ce_ratio_sd*, GENSTAT was once again used to generate a general purpose regression equation from a set of random trials executed specifically for this purpose. The equation generated was as follows:

$$\begin{aligned} 10^5 U' = \; & -18974 - 15N + 233.2\sigma_{cg\%} - 1.76\sigma_{cg\%}^2 - 0.00074N^2\sigma_{cg\%} + \\ & 0.11x - 46350y - 0.0000006\mu_{cg}x + 10^{-11} \times 0.11\mu_{cg}^2 x \end{aligned} \tag{5.22}$$

Knowledge of the likely important predictors and interactions was used to guide the initial selection of terms, and t values were then used to discard unimportant terms. The same 200 random trials that were used to test Equation 5.20 were then used in order to test Equation 5.22. The median absolute difference between observed and predicted values was 2.4, the upper quartile was 4.1, and the worst prediction was 14.8 percentage points in error. These results are summarised in the last line of Table 5–18, and one can see that they agree closely with those obtained for the 4×4 mesh.

One can conclude that models with similar predictive powers can be constructed for varying sizes of processor domain. Generally speaking, the models obtained for the 7×7 mesh contained a larger number of higher order terms than their 4×4 mesh counterparts, indicating more complex behaviour patterns. The predictive properties of the covariates *sync_mean* and *ce_ratio_sd* were observed to remain strong as the number of processors was increased. A possible extension of this work would to be to incorporate the domain size into the model by estimating its effect experimentally.

5.5 Summary and Conclusions

In this chapter I have developed a variety of models designed to predict the performance of non-uniform parallel programs, with respect to the machine-oriented metric U. I have considered both small, and larger, processor domains. The models have been shown to produce performance predictions with reasonable error distributions, and the consistency of the results indicates that these techniques could be practically useful. For example, one might use them to arbitrate between two alternative program structures; or to help select the most appropriate program characteristics to concentrate on when performance tuning.

The models derived here can be divided into two classes. Firstly, there are those which are purely a function of a set of program characteristics; in this case the number of nodes and the standard deviation of the compute times across the processes. Secondly, there are those models which, while retaining a component derived from a set of program characteristics, also include information calculated from a set of covariates relating to the dynamic behaviour of the program, i.e. its interaction with the underlying machine. Not surprisingly, the latter type of model proved to be the more accurate, although considering the crudeness of the simpler models, their predictions were reasonably good.

While program parameters could possibly be estimated before execution, the values of covariates certainly could not. One might question, therefore, the usefulness of a model which could only produce a performance prediction after the target program had been executed. While this would be of limited use for performance prediction purposes, these models allow us to understand why programs with the same program parameters can have drastically different performance characteristics when executed. From Equations 5.17 and 5.21 it can be seen that, ignoring the inherent program characteristics, the utilisation of the underlying machine is maximised when the covariates *sync_mean* and *ce_ratio_sd* are minimised (since the signs of these two terms are always negative). Therefore, any adjustments to the program mapping which enables this is to be encouraged. These mapping adjustments could be done in a post-mortem manner, in which case one would tend to iterate towards a better mapping. Alternatively, they could be attempted at run-time using a suitable process migration strategy. The remaining chapters in this thesis are concerned with investigating this idea.

Chapter 6

Analysis of a Class of Process Migration Strategies

This chapter analyses the behaviour of a class of distributed process migration strategies suitable for use in a loosely coupled multiprocessor system. I continue to concentrate on time-invariant non-uniform programs, and investigate the circumstances under which a migration strategy can improve a non-optimal initial mapping. This is analogous to the problem of improving the mapping of a time-varying program which has undergone a phase change and become unbalanced. Time-varying programs are discussed in Chapter 7.

Section 6.1 describes the structure of the class of strategies being studied. Section 6.2 presents a representative migration policy, and investigates its performance characteristics when presented with a selection of unbalanced workloads. In Section 6.3 some exploratory experiments are presented, and drawing on the work concerning covariates presented in Chapter 5, an improved policy is derived. Section 6.4 investigates further the performance characteristics of this improved policy, and presents the results of some validation work carried out on a real transputer-based machine. Finally, a summary and conclusions are presented in Section 6.5.

6.1 Introduction

This section describes a class of process migration strategies suitable for use in a distributed memory multiprocessor. I concentrate on dynamic load balancing; other applications such as fault tolerance, machine reconfiguration or resource sharing, are beyond the scope of this thesis. Also, I do not consider the issues arising from remote execution of processes, or dynamic placement of spawned processes. A suitable migration mechanism is assumed to be available. The MIMD-based mechanism described in Section 3.3.3 is used for simulation trials, and the transputer-based mechanism described in Section 3.4 is used for validation work. These mechanisms both assume that a program consists of a set of communicating, self-contained, long-lived processes. It is these top-level processes which are candidates for migration, subordinate processes are never migrated independently.

The migration strategies investigated here are adaptive, in the sense that migration decisions are made according to the current state of the machine; and distributed, so that there is no global point of control. Migrations take place between immediate neighbours only, accordingly, the communications and computational overheads required to carry out dynamic load balancing are kept to a minimum. Furthermore, as in [114] I shall assume that the operation of a migration strategy revolves around the periodic, synchronous exchange of load information between immediate neighbours. Since processors exchange load information, rather than demanding it of one another, these strategies can be classed as being cooperative, rather than sender- or receiver-initiated. The local nature of information exchange implicitly divides the machine into a set of overlapping neighbourhoods, with each processor sitting at the centre of one such neighbourhood. Consider the mesh-based topology illustrated in Figure 6–1, processor P5 sits at the centre of neighbourhood N5, which encompasses P5's immediate neighbours: P2, P4, P6 and P8. A number of strategies have used this approach, see [90,111,114,131] for

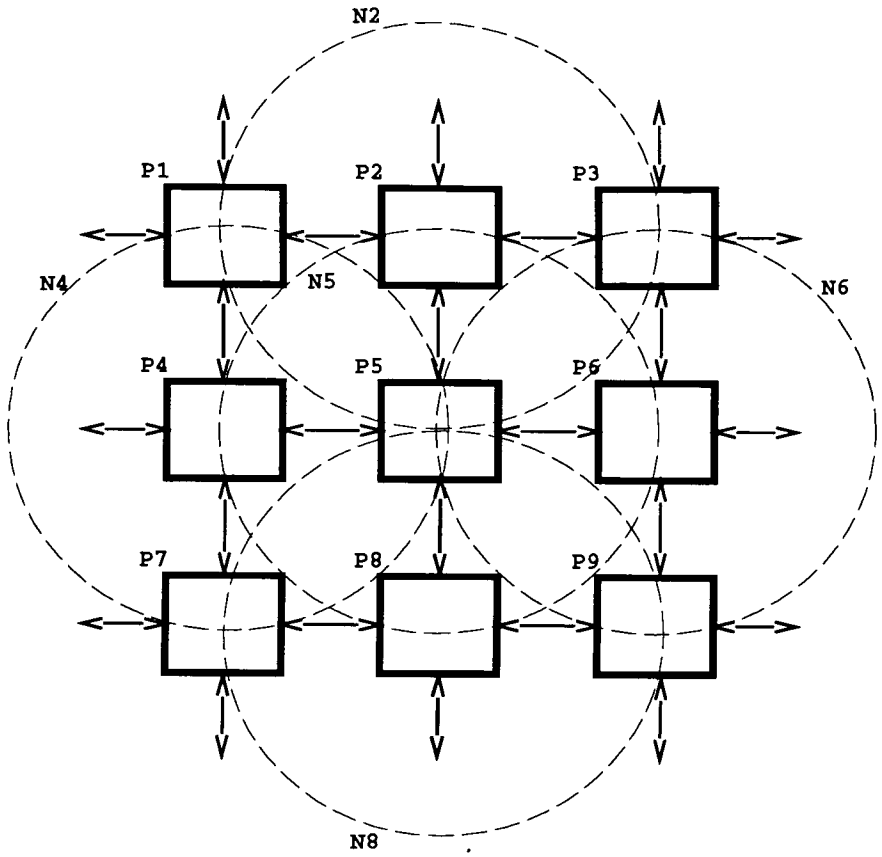

Figure 6–1: Overlapping Neighbourhoods for Mesh Topology

example. A global, although generally non-optimal, balance can be achieved by independently attempting to balance the load within each of the local neighbourhoods [34,111].

I will continue to assume the performance metric U, the mean utilisation of the processors. Many dynamic load balancing strategies have concentrated on maximising U; see [58,83,90,114] for example. U is particularly well suited to dynamic load balancing studies, since processor utilisation is strongly related to the speedup of a parallel system, i.e. in order to approach a linear speedup, all processors must be kept busy at all times.

As well as defining a performance metric, a load metric must also be specified in order to be able to accurately gauge the load of individual processors. This is

a complex issue since it is difficult to summarise load with a single number. In systems with many dynamically spawned, relatively short-lived tasks (for example, multi-user distributed operating systems) the number of active tasks is often used as the load metric. In the context of a single user multiprocessor with persistent processes, this metric is of little use, since it takes no account of the relative weights of processes, or their synchronisation patterns. I shall use the percentage utilisation of the processor as the load metric, since this is a good reflection of the current load. One must always bear in mind that the actions of the migration strategy itself places a load on the processors. However, it will be demonstrated later in this Chapter that this load is not excessive for the cases investigated here, so I can be confident that U truly represents the system load.

Once a processor knows its own load, and that of its neighbours, it can calculate its state relative to those neighbours. Three states are often identified [56,83,100]: low (**L**), medium (**M**) and high (**H**). The thresholds used to distinguish processor states can be fixed, but more often they are adaptive, and are defined as lying a specified distance away from the average load within each neighbourhood [119, 134]. At this point, each processor has a local view of its state, relative to the state of its immediate neighbours.

Migrations can only take place between **H** and **L** loaded processors. To ensure that this condition is satisfied, a further round of negotiations is entered into between each processor which sees itself as being **H**, and those neighbours which it sees as being **L**. The purpose of this is to give those neighbours a chance to confirm whether they are truly in the **L** state, or whether that view was merely a consequence of the prevailing local conditions. In this way, each **H** processor constructs a set of immediate neighbours who are in the **L** state, and are consequently possible targets for migrating processes. At this point the processors have completed the preliminary negotiations phase and are ready to initiate migrations.

As discussed in Section 2.5.1, there are three decision stages associated with a process migration. Firstly, when to initiate the migration; secondly, where

to migrate the process to; and lastly, which process to migrate. In the class of strategies described above, the "when" decision is made automatically on the basis of the state of the processors and the period of load information exchange. The "where" decision is *partly* made after the preliminary negotiations phase, in the sense that each **H** processor knows which of its neighbours are in the **L** state, and so are available to receive processes. The "who" decision, and the remainder of the "where" decision, i.e. the protocols for deciding which processes to migrate (the *selection* policy), and which of the possible processors to migrate them to (the *location* policy), is dependent on the details of the particular migration policy implemented. However, the aim of the selection and location policies should always be to attempt, as far as possible, to ensure that all **H** or **L** processors in the system converge towards the **M** state, thereby achieving equilibrium.

Once the selection and location policies have been executed, migrations will take place between **H** and **L** loaded processors, via the migration mechanism, subject to the *inundation* policy in operation on the **L** processors. The inundation policy is responsible for ensuring that a **L** processor does not accept too many processes. A **L** processor might abandon a particular migration because it does not have enough resources for the migrant, or because it has already accepted enough migrants for it to reach the **M** state, and fears that accepting another will push it into the **H** state.

6.2 Analysis of an Example Process Migration Strategy

A particular member of the class of migration strategies described above can be specified by constructing a migration policy defining the following:

- Threshold values
- Period of application

- Process selection policy
- Process location policy
- Inundation policy

These migration control parameters are defined below for a relatively simple migration policy which is investigated in this section.

1. Threshold values.

   The threshold values categorising **H**, **M** and **L** processors are defined, for each local neighbourhood, to be 5% either side of the mean processor utilisation within the neighbourhood. For example, if the mean utilisation is 75%, then the thresholds specifying the **L**−**M** and the **M**−**H** boundaries would be 71.25% and 78.75% respectively.

2. Period of application.

   The migration strategy is invoked after a fixed time period of 2,000,000 clock cycles of the simulated machine (i.e. every 100 milliseconds).

3. Process selection policy.

   The strategy uses a selection policy based on the activity of processes on each **H** loaded processor in the most recent monitoring period (the period since the migration strategy was last active). An assumption underlying this approach is that future behaviour will repeat past behaviour. This is not necessarily true, especially for time-varying programs. However, the most recent behaviour is the best available guide to future behaviour. The policy attempts to satisfy two objectives. Primarily, it attempts to select as many processes for migration as there are immediate neighbours in the **L** state. If this goal is not achievable, it will try and select the maximum number of processes possible. The secondary aim is to ensure that the processes chosen

are the most computationally intensive possible i.e. those processes which are using the greatest proportion of the available compute resources. Both of these objectives are subject to the restriction that the processor should not move directly from the **H** state to the **L** state as a result of migrating the processes in question. In order to be able to check this, one must be able to estimate the effect of removing a set of processes from a processor.

The problem of assessing the impact on processor utilisation of removing a group of processes is a difficult one. This is due to the interacting and competing nature of processes. If all of the processes on a processor were completely independent of one another, i.e. whenever one process was executing there were never any other processes waiting in the queue, then the effect of removing a group of processes could easily be estimated with reference to the most recent monitoring period. The expected utilisation of the processor, had the processes in question not been present, would be calculated by simply removing the utilisation due to them from the total figure. This approach actually gives a lower bound on the expected utilisation. In reality, however, processes are likely to interfere with one another, so the actual utilisation is likely to be higher than that estimated, since those processes left behind will be able to mop up a proportion of the spare compute time previously used by the migrating processes. The selection policy described here uses the lower bound to estimate whether the destination processor is likely to move from the **H** state to the **L** state as a result of migrating a set of processes.

4. Process location policy.

Given a set of N immediate neighbours in the **L** state, and a set of no more than N candidate processes, then the process location policy will randomly map the processes to the processors in a one to one fashion. In any one mi-

gration cycle, therefore, no more than one process can be migrated between any two processors.

5. Inundation policy.

   A null inundation policy is used, so that a **L** processor will never reject any migrating processes sent to it. It is therefore assumed that there are always adequate resources to service an incoming process[1], and the strategy does not atiempt to prevent a processor moving directly from the **L** state to the **H** state as a result of accepting migrants.

The policy defined above is now investigated with reference to non-uniform time-invariant programs, to see whether it can improve their performance. In a sense, these programs correspond to single phases of time-varying programs. If the strategy can be shown to generally converge to a better placement, then it is reasonable to assume that it would be able to do the same for a particular phase of a time-varying program.

6.2.1 Slightly to Moderately Unbalanced Workloads

This experiment was designed to show how well the migration policy described above performs when presented with non-uniform time-invariant programs producing slightly to moderately unbalanced workloads.

Experiment Description

I shall assume the usual set of program parameters:

$$\{N, c, \mu_{cg}, \sigma_{cg}, \mu_{mg}, \sigma_{mg}\, \sigma_c, \sigma_m\}$$

---

[1]In fact, this is always true for the transputer-based migration mechanism described in Section 3.4, and one can easily insist on it for the MIMD-based migration mechanism.

Program Parameters	
Param	Value(s)
N	{32, 142}
c	8
μ_{cg}	40000
$\sigma_{cg}\%$	{10, 80}
σ_c	10
μ_{mg}	1000
$\sigma_{mg}\%$	10
σ_m	1

Other Parameters	
Param	Value(s)
Migration Strategy	{On, Off}
Process Size	2000
Hardware	4×4 Mesh
Placement	Restricted Random
Trial Length	100,000,000 (5 Secs)
Replications	6

Table 6–1: Parameter Settings for Exploratory Experiment Using Slightly to Moderately Unbalanced Workloads

It was observed in Chapter 5 that the two most important program parameters, with respect to the metric U, were: N, the number of nodes; and $\sigma_{cg\%}$, the standard deviation of compute times expressed as a percentage of the mean compute time. So, in order to test the performance of the migration strategy, a two level full factorial experiment was carried out varying N, $\sigma_{cg\%}$, and the status of the migration strategy. The remaining program parameters were set to the same intermediate values that were applied in Chapter 5. The exact values used are summarised in Table 6–1. The mean compute time across the nodes of the process graph, μ_{cg}, was set to 40000, corresponding to a small to medium grained program. The degree of the program graph, c, was set to 8, allowing 4 incoming and 4 outgoing channels per process on average. The mean message length, μ_{mg}; and standard deviation of message lengths, $\sigma_{mg\%}$, were set to 1000 and 10 respectively. Finally, to enforce time-invariance, σ_c and σ_m were set to 10 and 1. Note that I could, in fact, have explicitly defined the values of σ_{cg}, since the mean compute time is fixed within this experiment. However, the percentage notation is retained here for consistency.

A number of other parameters had to be fixed in order to fully define the experiment. Firstly, a migrating process was assumed to occupy 2000 bytes of space (this assumption also holds for all subsequent experiments involving process migration). As usual, the hardware was set to a 4×4 mesh of transputers. A restricted random placement was used to ensure that, as far as possible, each processor received the same number of processes[2]. Each simulation run was executed for 100,000,000 clock cycles (5 seconds) of simulated machine time, in the hope that this would allow sufficient time for the migration policy to operate. In order to increase confidence in the results obtained, and look for smaller effects, six replications were executed rather than the usual three. This did not require

[2]In a real system a more sophisticated mapping strategy might be used, however that is not of concern here.

an excessive amount of experimental effort due to the relatively small number of parameters being explored. Each replicate differed both in the random graph and random number seed used.

Results

A residual scatter plot of U produced a funneled shape, indicating the need for a transformation. As was often the case in Chapter 5, a Guerrero and Johnson transform was found to be useful, specifically the transformation:

$$U' = \left(\frac{U}{100 - U}\right)^{0.18} - 1 \tag{6.1}$$

This transformation allows the assumptions underlying the analysis of variance to be satisfied. Table 6–2 presents a table of effects for the response U'. The presence of the label s in the first column indicates that the migration strategy is active. It is clear that all three main effects have a significant impact on performance. Furthermore, the sign of the s effect is positive, indicating that having the migration strategy turned on increases the estimated value of U', and so improves performance (since U' is a monotonically increasing function of U). The various interactions of s with the other two parameters are also important, accounting for more than 10% of the variation in U'. Their signs are also positive.

The above analysis seems to suggest that, generally speaking, the migration strategy improves performance. It would be useful to know the extent of this improvement, and the circumstances under which the possible gains are greatest. This can be achieved by examining the data in greater detail using paired t tests (see Section 3.5.6).

One can use a one tailed paired t test in order to test whether the values of U' observed with the migration strategy turned on, are significantly larger than those observed with the migration strategy turned off. The test is paired, since, ignoring the activities of the migration strategy, corresponding observations will

Effect	Estimate	t-value	% Var
(gm)	0.407		
s	0.079	9.79	9.53
N	0.136	16.84	28.19
sN	0.037	4.6	2.10
$\sigma_{cg\%}$	-0.174	-21.64	46.57
$s\sigma_{cg\%}$	0.063	7.8	6.05
$N\sigma_{cg\%}$	0.01	1.2	0.14
$sN\sigma_{cg\%}$	0.049	6.02	3.61
		Total	96.19
	Standard Error = 0.0081		

Table 6–2: Estimates of Effects for Exploratory Experiment Using U'

be identical in all others factors under experimental control. The assumption regarding the normality of the distribution of the differences underlying the t test is automatically satisfied as a result of the assumptions underlying the analysis of variance being satisfied (see page 110 of [81]). As well as considering the data as a whole, similar tests can be carried out for various subsets of the data. For example, by fixing N and/or $\sigma_{cg\%}$ at certain values, one can partition the data in a number of different ways. This approach allows one to gain an understanding of the exact conditions under which performance can be improved.

Table 6–3 contains the results of a number of t tests of the type described above for various partitions of the data. The first column specifies which particular subset of the observed data values is being tested. For example, the t test referred to in the first line considers all observations, the second considers only those where $\sigma_{cg\%}$ was set to 10 etc. The second column gives the observed mean values of U across the six replications when the migration strategy was inactive. The third column gives the corresponding figures when the strategy was active. The fourth column specifies the number of pairs of observations that each test is based upon. The final column gives the results of the t tests, which are carried out in terms of the transformed metric U'. The null hypothesis (H_0) is always that, the mean value of U' across all six replications for the observations where the migration strategy was

Data	$\overline{U}$	$\overline{U}_s$	n	Reject H_0: $\overline{U}' \geq \overline{U}'_s$?
overall	76.22	86.35	24	Yes 1%
$\sigma_{cg\%}$=10	91.37	92.48	12	No
$\sigma_{cg\%}$=80	61.07	80.22	12	Yes 1%
N=32	70.95	78.53	12	Yes 1%
N=142	81.49	94.17	12	Yes 1%
$\sigma_{cg\%}$=10 N=32	87.79	89.83	6	No
$\sigma_{cg\%}$=80 N=32	54.11	67.23	6	Yes 1%
$\sigma_{cg\%}$=10 N=142	94.96	95.14	6	No
$\sigma_{cg\%}$=80 N=142	68.02	93.20	6	Yes 1%

Table 6–3: Results of Paired t Tests for Exploratory Experiment using U'

inactive, $\overline{U}'$, is greater or equal to the mean value of U' for the observations where the migration strategy was active, $\overline{U}'_s$. That is, the migration strategy does not improve performance. So, in order to be able to conclude that any improvements in performance are due to the migration strategy, and not due merely to sampling fluctuations, one must be able to reject H_0. For various subsets of the data, the final column indicates whether H_0 can be rejected, and if so, at what significance level (I only consider the 1%, 5% and 10% significance levels). Examining the data as a whole, one can accept the alternative hypothesis, that the migration strategy improves performance, at the 1% significance level. A more detailed examination reveals that the performance benefits are not significant for reasonably uniform programs, i.e. those trials when $\sigma_{cg\%}$=10. For all other partitions of the data however, including those which are only partly composed of trials where $\sigma_{cg\%}$=10, one can be confident that the migration strategy improves performance. Although, no doubt, this is due to the presence of data gathered from trials where $\sigma_{cg\%}$=80.

Discussion

So, I have shown that a simple migration strategy appears to improve performance significantly (with respect to a processor utilisation based metric) for a subset of time-invariant programs exhibiting non-uniform behaviour which have been mapped in a well balanced manner. It should be noted that in the cases where the strategy does not significantly improve performance, there is no evidence to suggest that it degrades performance.

Several interesting questions can be asked about the results presented above. Firstly, is the increased utilisation observed mainly attributable to the improved performance of the user program, or do the migration overheads occupy a significant proportion? Remember that these overheads are modelled explicitly in MIMD. Secondly, the results concerning the covariates *sync_mean* and *ce_ratio_sd* obtained in Chapter 5 suggest that, for a given program, improvements in performance resulting from an improved placement will produce lower values of *sync_mean* and *ce_ratio_sd*. Is this still the case if the improvements occur at run-time as a result of process migrations? These questions can be answered by considering alternative metrics to U, and by carrying out paired t tests to see whether the results obtained are consistent with those observed for U. The results of a number of such tests are presented in Table 6–4, and are explained below. Transformations were used where necessary to satisfy the assumptions underlying the tests, these transformations were straightforward and are not discussed here.

To confirm that the increased utilisation figures observed really are the result of improved performance, a metric M, directly related to the rate of computation of the user program was examined. M is defined as the mean number of messages sent down a program's channels during its execution. For the loosely synchronous program model assumed here, the number of messages sent down any two channels at any moment in time should differ by at most one or two. Therefore, this metric provides a good measure of how much work the program has done, since it is

Data	n	Reject H_0: $\overline{M} \geq \overline{M}_s$?	% Improvement	Reject H_0: $\overline{S} \leq \overline{S}_s$?	Reject H_0: $\overline{C} \leq \overline{C}_s$?
overall	24	Yes 1%	11.6%	Yes 1%	Yes 1%
$\sigma_{cg\%}$=10	12	No	1.9%	No	No
$\sigma_{cg\%}$=80	12	Yes 1%	27.5%	Yes 1%	Yes 1%
N=32	12	Yes 1%	10.7%	Yes 1%	Yes 5%
N=142	12	Yes 1%	15.2%	Yes 1%	Yes 1%
$\sigma_{cg\%}$=10 N=32	6	No	2.3%	No	No
$\sigma_{cg\%}$=80 N=32	6	Yes 1%	24.8%	Yes 1%	No
$\sigma_{cg\%}$=10 N=142	6	No	0.2%	No	No
$\sigma_{cg\%}$=80 N=142	6	Yes 1%	37.6%	Yes 1%	Yes 1%

Table 6–4: Results of Paired t Tests for Exploratory Experiment Using Alternative Metrics

directly related to the number of iterations achieved[3]. The results of the t tests for M are given in the third column of Table 6–4. One can see that the results are identical to those obtained for U'. This seems to confirm that a processor utilisation based metric accurately reflects the performance of the program being executed, even when process migrations are being carried out. Indeed, the values of U and M are highly correlated; for example, if the data is partitioned into two sets according to the value of N, the corresponding correlation coefficients are 0.955 and 0.974 respectively. I can therefore conclude that the migration overheads are not excessive, and that increases in U due to the activity of the migration strategy can be attributed mainly to improved program performance. The fourth column of Table 6–4 gives, for each data set, the percentage improvement in M, and hence in program performance, achieved by the migration strategy. The overall figure

[3]The metric M is not suitable for general use since it relies very strongly on the details of the particular program model used.

	Rep 1	Rep 2	Rep 3	Rep 4	Rep 5	Rep 6
$\sigma_{cg\%}$=10 N=32	0	0	0	0	0	0
$\sigma_{cg\%}$=80 N=32	3	1	3	4	4	9
$\sigma_{cg\%}$=10 N=142	10	6	4	2	12	0
$\sigma_{cg\%}$=80 N=142	43	34	32	70	40	53

Table 6–5: Total Number Migrations Executed

is 11.6%, with larger improvements generally being observed for higher values of $\sigma_{cg\%}$.

The fifth and six columns of Table 6–4 contain the results of t tests carried out for the covariates *sync_mean* and *ce_ratio_sd*. For the sake of convenience, $\overline{S}$ is used to represent the mean value of *sync_mean* over the six replications; and similarly, $\overline{C}$ is used to represent the mean value of *ce_ratio_sd*. Note that, in this case, I am looking to see whether the actions of the migration strategy decrease, rather than increase, the values of the covariates. Therefore, the signs used in the null hypotheses are the reverse of those seen so far. It is clear that the results for *sync_mean* agree exactly with those observed for U and M, so one can conclude that whenever the migration strategy significantly improves performance, the value of *sync_mean* obtained is significantly reduced. The results for *ce_ratio_sd* are reasonably close to those observed for U and M, the main difference occurring when there are a small number of nodes and a high standard deviation of compute times. However, generally speaking, improved performance also results in significantly lower values of *ce_ratio_sd* being observed, as one would expect.

The final issue that should be addressed is the dynamic behaviour of the migration strategy. Table 6–5 lists the total number of migrations carried out at each combination of N and $\sigma_{cg\%}$, for each of the six replications. When N=32 and $\sigma_{cg\%}$=10, no migrations are carried out whatsoever. This is because the workload

is already balanced due to the uniform nature of the programs, and the fairness of the placement strategy. A small number of migrations occur when N is increased to 142 and $\sigma_{cg\%}$ remains set at 10. These migrations are triggered because the increased numbers of processes mean that relatively small units of work can be moved from processor to processor. Consequently, processes can be migrated from **H** processors without sending them directly into the **L** state, so the migration strategy will attempt this in an effort to achieve a finer balance. However, we have already seen that these migrations do not bring any significant performance improvements. A small number of migrations occur when N=32 and $\sigma_{cg\%}$=80. We have seen that these migrations, although small in number, do result in significant performance benefits. This is because the non-uniform nature of the programs means that the initial placement is unlikely to be optimal. Larger numbers of migrations take place when N=142 and $\sigma_{cg\%}$=80. Once again, the non-uniform nature of the programs mean that these migrations achieve significant performance improvements. The increased numbers of migrations observed are a consequence of having a large number of processes, and hence smaller migratable units of work. This means that a finer balance is possible, but the strategy is required to carry out more migrations in order to converge to a stable state.

It is interesting to study how the performance of the migration strategy evolves with time, in order to see how efficiently it converges to a good balance. For this purpose, imagine that the 5 second simulation time is divided up into 10 $\times$ 0.5 second time frames. The cases where N=32 and $\sigma_{cg\%}$=80, and where N=142 and $\sigma_{cg\%}$=10 are straightforward, since all of the migrations are completed within the first two time frames. Figure 6–2 illustrates how the number of migrations and mean utilisation of the processors evolve with time for the situation where N=142 and $\sigma_{cg\%}$=80. The labels on the x axis refer to the upper class bounds of each of the time frames. The labels on the y axis refer to the number of migrations and percentage utilisation respectively. For both sets of data, the points plotted correspond to the mean value taken over the six replicates, 95% confidence

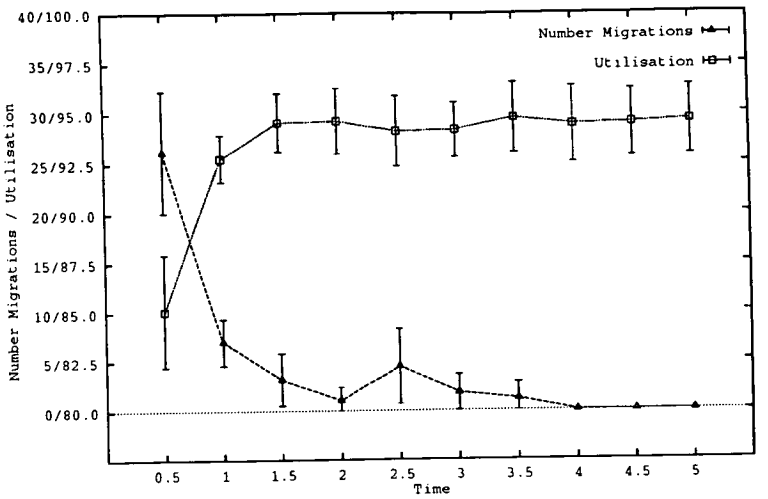

Figure 6–2: Performance of Migration Strategy Over Time When N=142 and $\sigma_{cg\%}$=80

intervals are also shown for each of the points, contracted where necessary to prevent negative migration count values being displayed. One can see that the migration strategy performs most of its work within the first 1.5 seconds, with a good mean utilisation, and the majority of migrations, being achieved within that period. To give an idea of how quickly the strategy can improve performance, the mean utilisation in the first half second, had the migration strategy not been active, would have been just 68.03%. So a 20 percentage point increase in performance is achieved, just within the first half second. The time between 1.5 seconds and 3.5 seconds is spent settling down, with no significant performance benefits being obtained. After 3.5 seconds a completely stable global balance is achieved. So, it seems that the migration strategy acts in a stable manner, and converges to an improved placement in an acceptable time for non-uniform programs mapped in a relatively well balanced manner.

Effect	Estimate	t-value	% Var
(gm)	65.84		
s	11.68	16.01	43.97
N	12.34	16.9	49.05
sN	0.84	1.15	0.23
$\sigma_{cg\%}$	0.15	0.21	0.01
$s\sigma_{cg\%}$	-0.12	-0.17	0.00
$N\sigma_{cg\%}$	-0.3	-0.41	0.03
$sN\sigma_{cg\%}$	-0.15	-0.2	0.01
		Total	93.3
	Standard Error = 0.73		

Table 6–6: Estimates of Effects for Strongly Unbalanced Workloads Using U

6.2.2 Strongly Unbalanced Workloads

To see how efficiently the migration strategy performs when presented with strongly unbalanced workloads, a very similar experiment to that described above was carried out. The only difference between this experiment, and the one defined by Table 6–1, was that a truly random, rather than restricted random, placement strategy was used. Programs mapped in this manner have the potential to be very unbalanced, since one processor might be responsible for many more processes than another. The problem of improving the performance of these programs corresponds to the problem of improving the performance of a time-varying program that has evolved to a particularly unbalanced state.

Diagnostic plots were examined and they indicated that there was no need to consider using a transformation of the response. The table of effects derived from an analysis of variance on U is shown in Table 6–6. The only two significant terms in the model are the main effects of s and N, together accounting for 95.7% of the variation in U. The relative unimportance of $\sigma_{cg\%}$ conflicts with the results obtained up to now, but can be explained if one considers the impact of using a completely random placement. The unbalanced behaviour due to a high value of

Data	$\overline{U}$	$\overline{U}_s$	n	Reject H_0: $\overline{U} \geq \overline{U}_s$?
overall	54.15	77.52	24	Yes 1%
$\sigma_{cg\%}$=10	53.88	77.49	12	Yes 1%
$\sigma_{cg\%}$=80	54.43	77.55	12	Yes 1%
N=32	42.66	64.34	12	Yes 1%
N=142	65.65	90.70	12	Yes 1%
$\sigma_{cg\%}$=10 N=32	42.23	63.86	6	Yes 1%
$\sigma_{cg\%}$=80 N=32	43.08	64.81	6	Yes 1%
$\sigma_{cg\%}$=10 N=142	65.52	91.12	6	Yes 1%
$\sigma_{cg\%}$=80 N=142	65.77	90.29	6	Yes 1%

Table 6–7: Results of Paired t Tests for Strongly Unbalanced Workloads Using U

$\sigma_{cg\%}$ becomes overshadowed by the unbalanced behaviour caused by the placement strategy.

The size and positive sign of the s effect suggests that the migration strategy has a very beneficial impact on performance. As in the previous experiment, this can be examined in greater detail using paired t tests. Table 6–7 shows the results of t tests carried out on various subsets of the data. One can see that the improvements in performance due to the migration strategy are definitely worthwhile, and are always significant at the 1% significance level.

To confirm, once again, that the improvements observed in Table 6–7 come about as a result of improved performance rather than as a result of migration overheads, and that the expected decreases in the values of the covariates *sync_mean* and *ce_ratio_sd* are obtained, further t tests were carried out using alternative metrics. The results of these tests are summarised in Table 6–8. The results obtained for all three metrics are very close to those seen for U, the only differences being that several of the hypotheses can only be accepted at the 5% significance level,

Data	n	Reject H_0: $\overline{M} \geq \overline{M}_s$?	% Improvement	Reject H_0: $\overline{S} \leq \overline{S}_s$?	Reject H_0: $\overline{C} \leq \overline{C}_s$?
overall	24	Yes 1%	47.5%	Yes 1%	Yes 1%
$\sigma_{cg\%}$=10	12	Yes 1%	48.0%	Yes 1%	Yes 1%
$\sigma_{cg\%}$=80	12	Yes 1%	46.9%	Yes 1%	Yes 1%
N=32	12	Yes 1%	50.7%	Yes 1%	Yes 1%
N=142	12	Yes 1%	38.2%	Yes 1%	Yes 1%
$\sigma_{cg\%}$=10 N=32	6	Yes 1%	51.1%	Yes 5%	Yes 1%
$\sigma_{cg\%}$=80 N=32	6	Yes 1%	50.3%	Yes 1%	Yes 1%
$\sigma_{cg\%}$=10 N=142	6	Yes 1%	39.1%	Yes 1%	Yes 1%
$\sigma_{cg\%}$=80 N=142	6	Yes 5%	37.2%	Yes 1%	Yes 1%

Table 6–8: Results of Paired t Tests for Strongly Unbalanced Workloads Using Alternative Metrics

	Rep 1	Rep 2	Rep 3	Rep 4	Rep 5	Rep 6
$\sigma_{cg\%}$=10 N=32	5	6	6	7	8	5
$\sigma_{cg\%}$=80 N=32	4	10	5	6	5	4
$\sigma_{cg\%}$=10 N=142	64	49	67	62	35	46
$\sigma_{cg\%}$=80 N=142	44	25	53	55	62	55

Table 6–9: Total Number Migrations Executed for strongly Unbalanced Workloads

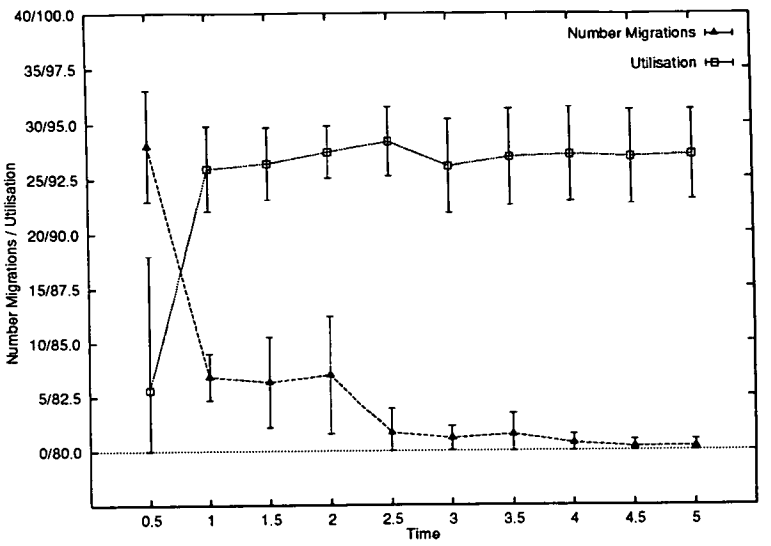

Figure 6–3: Performance of Migration Strategy Over Time for Strongly Unbalanced Workloads When N=142 and $\sigma_{cg\%}$=10

rather than at the 1% significance level. The fourth column of Table 6–8 gives, for each of the data sets, the percentage increase in performance attributable to the use of the migration strategy. Overall, the performance of the strategy is very encouraging, increasing the rate of computation achieved by an average of almost 50%.

Table 6–9 lists, for each replication, the number of migrations carried out for each of the parameter combinations. It is clear that the number of migrations required to achieve a good balance increases as the number of nodes increases. This is consistent with the results observed for lightly to moderately unbalanced workloads.

When N=32, the migrations are always completed before the simulation time reaches 0.5 seconds, so the strategy very quickly converges to an improved placement. The behaviour of the strategy when N=142 is illustrated in Figures 6–3 and 6–4. These graphs show the mean number of migrations and the mean processor utilisation (with 95% confidence intervals) across the six replications. The values plotted refer to half second time frames throughout the simulation period. Both

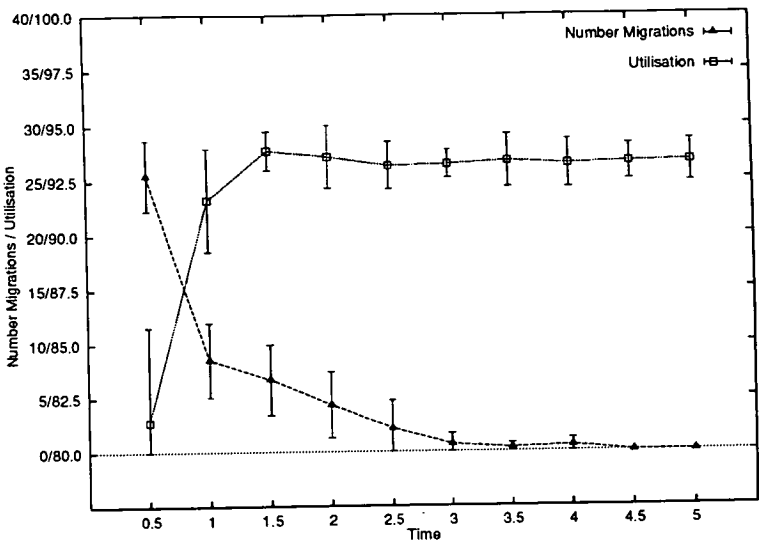

Figure 6–4: Performance of Migration Strategy Over Time for Strongly Unbalanced Workloads When N=142 and $\sigma_{cg\%}$=80

graphs give a similar picture to that seen already in Figure 6–2. A good balance is generally achieved after 1.5 seconds, the time between 1.5 and 3.5 seconds is spent settling down, and after 3.5 seconds a stable global balance is achieved.

6.2.3 Summary

We have seen how a representative process migration strategy can be used to improve the performance of a variety of time-invariant non-uniform programs. This is encouraging, since the choice of control parameters defining the behaviour of the strategy, i.e. the selection policy, location policy etc., was not given a great deal of consideration.

The extent of the improvement achieved by the strategy is dependent on the degree of unbalance present in the workload, due either to the placement used, or the structure of the program. For slightly to moderately unbalanced workloads, the benefits increase as the degree of non-uniformity present in the program increases. For strongly unbalanced workloads, the benefits seem relatively independent of the

structure of the program. The time taken to achieve a balance is dependent on the number of nodes in the user program, with more migrations being required for larger numbers of nodes. This characteristic is a direct consequence of limiting to one the number of processes that can be transferred between any two processors within a single migration cycle. The strategy has been shown to possess good stability and convergence properties, generally reaching a reasonable placement within 1.5 seconds.

The overheads imposed by the migration strategy are acceptable, and the metric U would seem to accurately reflect the performance of the program being executed. In addition, the migration strategy was shown to decrease the values of the covariates *sync_mean* and *ce_ratio_sd* as performance was improved.

6.3 An Improved Migration Strategy

You will recall that a particular member of the class of migration strategies being studied can be defined by specifying its migration control parameters, i.e. the threshold values, period of application, process selection policy, process location policy and inundation policy. This chapter investigates each of these in turn, with the aim of constructing an improved strategy. In order to achieve this, an incremental approach has been adopted. The migration control parameters have been divided into three groups which will be explored in sequence, thereby allowing the results of one experiment to be used in subsequent experiments. The first two migration control parameters to be considered are the quantitative factors concerned with the method of application of the policy, i.e. the migration threshold values and migration period. Next, the control parameters concerned with process selection and location are studied, and finally the effect of using an inundation policy is examined.

To simplify matters, the program parameters were not varied within these experiments. Instead, a representative workload was chosen, and the effects of altering the migration strategy were examined with reference to it. The results obtained are generalised for different programs and processor domains in Section 6.4.

The program parameters N and $\sigma_{cg\%}$ were fixed at 100 and 40 respectively, since these are reasonable intermediate values. With the exception of the placement strategy used, the remaining parameters were set to the values listed in Table 6–1. A truly random placement strategy was used to generate strongly unbalanced workloads, which have the most to gain from the application of process migrations. However, this placement strategy was applied in a different manner than has been seen previously. So far, when using a random placement strategy, a different mapping has been generated for each individual simulation trial within a replication. So, for example, two trials within a replication which differed in some parameter not affecting the shape of the program graph, would still receive different random mappings. This has been the correct approach to take up to now, since it has allowed me to generalise across the entire population of random placements. However, there is now a need for greater precision since I wish to examine how the performance of the migration policy is influenced by the parameters controlling it. If the initial placement were to vary within a single replication, the effects of the migration control parameters might be confused with the effects of the different placements. Since N and c are not varied in these experiments, the same program graph applies to an entire replication, so it is possible to use exactly the same initial mapping within that replication. This allows any observed effects to be attributed to the factors under experimental control, rather than to different mappings. However, individual replications are still allocated different program graphs and random number seeds.

6.3.1 Method of Application

The definition of the migration threshold values affects the likelihood of processes being considered for migration. The further away the thresholds become from the mean utilisation within a neighbourhood, the less susceptible processes will be to migrations, since processors will be more likely to be classified as being in the neutral **M** state. Defining the **L**−**M** and **M**−**H** boundaries as being fixed percentages away from the mean utilisation within a neighbourhood leads to a convenient stability property. For example, imagine that the threshold was set at $x\%$, now, since $x\%$ of 40% is smaller than $x\%$ of 90%, it is harder for processors to achieve the **M** state at lower utilisations (since **M** spans a smaller interval). At higher utilisations the **M** state is easier to achieve, and so migrations are less likely to occur. This is a desirable property, since, due to the asymptotic nature of the metric U, it is not sensible to waste too much effort trying to improve performance at high processor utilisations. However, at lower utilisations improvements in performance are easier to achieve, and so should be attempted whenever possible.

The migration period defines the interval at which migrations are *considered.* If this period is too short, the monitoring information gathered will be inadequate, since the computation itself will not have time to stabilise between migration cycles. If the period is too long, then the convergence to an improved placement will take an unnecessarily long time.

In order to find the best levels for these two migration control parameters, a full factorial experiment was carried out. Each of the two parameters was set at six levels, i.e. the migration period was set at 1,000,000, 1,500,000, 2,000,000, 2,500,000, 3,000,000 or 3,500,000 clock cycles; and the migration threshold was set at 1%, 3%, 5%, 7%, 9% or 11%. The remaining migration control parameters remained unchanged from the initial migration policy described in Section 6.2, i.e. process selection was based on selecting the largest number of computationally in-

tensive processes possible, processes were located randomly, and a null inundation policy was used.

The optimal performance of the migration strategy was observed to occur when the threshold was set to 5% and the period was set to 2,000,000 clock cycles (i.e. 100 milliseconds). By chance, these are the values that have already been used. At thresholds above 5%, performance tended to tail off because processors were finding it too easy to achieve the **M** state, and so migrations were being inhibited. At thresholds below 5% migrations were also observed to be inhibited, but for a different reason. At low thresholds one might expect many migrations to be triggered, because it would be relatively difficult for processors to achieve the **M** state. However, this is in fact not the case due to the characteristics of the selection policy used. You will recall that processes are only considered as migration candidates if it is estimated that their migration will not cause the source processor to move directly from the **H** state to the **L** state. As the threshold is reduced, it becomes more difficult to find processes that satisfy this criteria, and so migrations are inhibited.

Obviously, the choice of migration period is very sensitive to the characteristics of the program being executed, as well as other external factors. Indeed, in a general purpose implementation of process migration, it would be desirable to activate the migration policy adaptively according to the current system load, rather than at fixed intervals. This approach is investigated in [134,137]. However, it is necessary to restrict the scope of this study, so I will assume a fixed period. Other migration strategies have used fixed periods with success. In [114], Saletore uses a value of 100 milliseconds, and in [32], Corradi *et al.* investigate periods in the range 50-150 milliseconds. A period of 2,000,000 clock cycles is consistent with these values.

A migration threshold of 5% and a migration period of 2,000,000 clock cycles are assumed for the remaining experiments in this chapter.

6.3.2 Process Selection and Location Policies

To examine the extent to which the choice of selection and location policies influence performance, a full factorial experiment was carried out using four different process selection policies and four different process location policies. Some of these policies were designed to minimise the values of the covariates *sync_mean* and *ce_ratio_sd*. This seems a sensible approach to take, since it has already been established that lower values of these covariates are associated with improved performance. It is unclear, however, whether purely local decision making will enable the values of the covariates to be optimised at a global level. Before presenting the results of an exploratory experiment, the particular policies used are defined. In both cases, the policies are presented in approximate increasing order of complexity. Suitable computation delays have been incorporated into the simulation system to allow for the decision making process.

Process Selection Policies

1. Maximum number of most computationally intensive possible (**select−1**).

   This is the same policy as used previously. It attempts to satisfy two objectives. Primarily, it tries to find as many processes as there are immediate neighbours in the **L** state. The secondary aim is to ensure that the processes selected are the most computationally intensive possible. This policy, and indeed all the other selection policies, are subject to the restriction that the processes removed should not result in the processor moving directly from the **H** state to the **L** state. The method used achieve this was described in Section 6.2.

2. Maximum number of least computationally intensive possible (**select−2**).

   This policy is very similar to **select−1**, the only difference being that instead of selecting the most computationally intensive processes possible, the

least computationally intensive ones are chosen. This means that processes are less likely to be inhibited from migrating as a result of the mechanism detecting that the source processor might enter the **L** state.

3. Minimisation of covariate *sync_mean* (**select−3**).

   This policy attempts to select processes so as to minimise the value of the covariate *sync_mean*. The policy necessarily operates locally, trying to select those processes which tend to delay synchronisations.

   The policy is based on a piece of information called the *blocking factor*, which is maintained for each process. When a process communicates on a channel, two outcomes are possible: either the partner in the message transfer is ready, in which case the communication can proceed; alternatively, the message transfer will block because the partner is not ready. This information can be gathered for the channels attached to a particular process, assuming a synchronous communications mechanism. The ratio of the number of times a message transfer does not block, to the number of times it does block, for a given process, is defined as its blocking factor.

   If the blocking factor for a process is greater than 1, then it indicates that the process tends not to block more often than it does block, and vice versa if the blocking factor is less than 1. The selection policy works on the premiss that a process with a blocking factor greater than 1 tends not to block, and so could possibly benefit from being executed faster. If these processes are executed faster, then the global *sync_mean* value should be reduced. Accordingly, the processes with the highest blocking factors are selected to be migrated. The policy attempts to select one process for each of the **L** immediate neighbours.

4. Minimisation of covariate *ce_ratio_sd* (**select−4**).

   This policy attempts to select processes so as to minimise the value of the covariate *ce_ratio_sd*. Once again the policy necessarily operates locally, trying to choose those processes which will decrease the variability in the ratio of compute to elapsed times.

   The policy works by ranking the processes on a processor according to their absolute distance from the mean *ce_ratio* observed on that processor. The processes which have *ce_ratio* values furthest away from the mean (either above or below) are duly selected for migration, the intention being to minimise the *ce_ratio_sd* on the source processor (and hopefully reduce its value globally). As usual, it is attempted to select one process for each of the **L** immediate neighbours.

Process Location Policies

1. Random allocation (**locate−1**).

   This is the policy used previously. Processes are mapped to the available processors in a random fashion, subject to the restriction that each processor can only receive a single migrant from any particular processor within a single migration cycle.

2. Minimisation of communications overheads (**locate−2**).

   This policy uses the communications characteristics of the processes to guide the mapping. A process will be attracted to the processor connected to the link down which it has transferred the most messages during the preceding monitoring period. A problem can arise if two processes want to migrate to the same processor. Such conflicts are resolved by comparing a message count for each process, the loser has to attempt to migrate to its second

choice processor. The aim of this strategy is to minimise the communications overheads by reducing the number of links which messages have to traverse.

3. Minimisation of covariate *sync_mean* (**locate−3**).

   This policy attempts to minimise *sync_mean* by mapping processes to the available processors according to their blocking factors. The basic idea is that the process with the highest blocking factor could most benefit from being executed more quickly. It should therefore be sent to the available processor with the lowest utilisation in the preceding monitoring period. The process with the next highest blocking factor should be migrated to the processor with the next lowest utilisation, and so on.

4. Minimisation of covariate *ce_ratio_sd* (**locate−4**).

   Locating processes so as to minimise *ce_ratio_sd* is problematic. This is because the *ce_ratio* value associated with a process is intimately linked to the processor on which the process was executing, since the *ce_ratio* is an indicator of the extent to which the process interacts with the other processes resident on the processor. The only way to use the information is to assume that a process which has been involved in contention for processor resources on one processor (i.e. low *ce_ratio* value) will continue to be involved in contention if migrated to another processor. Similarly, a process that has been involved in very little contention will continue to execute in the same manner if migrated. These assumptions may not be very realistic, but they are necessary if a migration location policy based on local information is to be derived that attempts to minimise *ce_ratio_sd*.

   The policy attempts to minimise the *ce_ratio_sd* value within a neighbourhood by mapping processes so as to bring the processors whose local *ce_ratio_mean* values are furthest away from the mean value closer to the mean. Therefore, processes with low *ce_ratio* values are mapped to pro-

cessors whose local *ce_ratio_mean* values are higher than the neighbourhood average. Similarly, processes with high *ce_ratio* values are mapped to processors whose local *ce_ratio_mean* values are lower than the neighbourhood average.

Results

The relative performance of the different selection and location policies can be examined by studying the mean values of U obtained over the six replications. Table 6–10 shows the mean values of U obtained for each of the four selection policies. Each of these means is based on 24 values i.e. six replications for each of the four location policies. The mean values for **select−2** and **select−3** are very similar to those observed for the original selection policy, **select−1**. The use of **select−4** however, seems to result in better performance. As usual, a paired t test can be used to test whether this improvement is statistically significant[4]. The value of the test statistic is 1.79, so, consulting a table of t values, one can conclude that **select−4** out-performs **select−1** at the 10% significance level.

Table 6–11 contains a similar table for the location policies. The behaviour of **locate−1**, **locate−2** and **locate−4** are very similar; **locate−3** performs a little better than the other three, but, statistically speaking, this improvement is not significant.

Table 6–12 gives a more detailed breakdown of the mean values obtained at each combination of the selection and location policies. Each of the values in the table is the mean of six observations. It is clear that all of the combinations, with the exception of **select−3/locate−4**, produce higher mean values of U than

---

[4]We can assume here, and in subsequent tests, that the assumptions underlying the t test have been satisfied, appropriate transformations will have been used whenever necessary.

select−1	select−2	select−3	select−4
87.50	86.86	87.84	89.45

Table 6–10: Observed Mean Values of U for Each Selection Policy

locate−1	locate−2	locate−3	locate−4
87.72	87.80	88.79	87.34

Table 6–11: Observed Mean Values of U for Each Location Policy

	locate−1	locate−2	locate−3	locate−4
select−1	85.07	87.46	88.81	88.65
select−2	86.25	88.64	86.43	86.13
select−3	89.60	88.25	88.92	84.58
select−4	89.95	86.85	90.99	89.99

Table 6–12: Observed Mean Values of U for Each Combination of Selection and Location Policy

the combination used in earlier experiments (i.e. **select−1/locate−1**). The best performance is obtained using **select−4/locate−3**, this is consistent with the figures observed in Tables 6–10 and 6–11, where these two policies gave the best performance. A t test can be used to test whether the value obtained using **select−4/locate−3** is significantly higher than that obtained using **select−1/locate−1**. The corresponding test statistic is 2.72. So, with reference to a table of t values, one can conclude that the performance of **select−4/locate−3** is better than that of **select−1/locate−1** at the 5% significance level. This conclusion cannot be reached for any of the other policy combinations.

The objectives of **select−4** and **locate−3** are to minimise the values of *ce_ratio_sd* and *sync_mean* respectively. To confirm whether this objective has been achieved, paired t tests can be used to show whether using **select−4/locate−3** rather than **select−1/locate−1** results in lower values of *ce_ratio_sd* and *sync_mean*. With reference to *ce_ratio_sd*, the observed values are 0.143 for

select−1/locate−1 and 0.124 for **select−4/locate−3**. The corresponding test statistic is 1.9. For *sync_mean* the values are 123927 and 115134 respectively, leading to a test statistic of 3.76. Consequently, it can be concluded at the 10% significance level that **select−4/locate−3** reduces the value of *ce_ratio_sd*. Similarly, it can be concluded at the 5% significance level that **select−4/locate−3** reduces the value of *sync_mean*.

Discussion

It is clear that using a migration selection policy based on minimising the covariate *ce_ratio_sd*, and a migration location policy based on minimising the covariate *sync_mean*, leads to significant performance benefits for this representative workload. The improvements over using **select−1/locate−1** are all the more impressive if one bears in mind the asymptotic nature of the metric U. It also seems that local decision making is effective in reducing the values of *ce_ratio_sd* and *sync_mean* at a global level. It is not surprising that a process selection policy based on minimising *ce_ratio_sd* turns out to be better than one based on minimising *sync_mean*. This is because the *ce_ratio* values for the processes on a given processor contain a lot of local information relating to the competing processing demands of those processes. So, when a processor is selecting which of its processes to migrate, this information is very relevant. Similarly, process location based on minimising *sync_mean*, rather than *ce_ratio_sd*, is more effective since *sync_mean* values contain globally oriented information relating to the interaction between processes on different processors.

One might expect the location policy based on minimising the communications overheads (**locate−2**) to have a significant impact on performance. However, its unspectacular performance is consistent with the program model being simulated. Since each process communicates on all of its channels at each iteration, any attempt to reduce the communications overheads by moving certain processes closer

together will be negated by the extra overheads associated with the communications which each process must still maintain with its other partners. Under a different program model, this policy could well operate more efficiently. For example, one could imagine the situation where two processes suddenly started communicating heavily and exclusively with one another; it would then make good sense for them to move closer together.

6.3.3 Inundation Policy

This section presents an experiment to test whether the use of an inundation policy can improve performance. So far, a null policy has been used, so migrants are never rejected. However, to prevent processors moving directly from the **L** state to the **H** state, it might be desirable for a receiving processor to refuse a process permission to migrate. Such a policy has been implemented, and is investigated here. It works by a processor including the processing time achieved by the migrating process (in the previous monitoring period) in an estimate of what its utilisation would have been had the process in question been present. If a particular process results in the estimate crossing the **M−H** border, then that process is refused permission to migrate. The experiment described below compares the performance obtained using this inundation policy, to that obtained using a null policy.

The remaining migration control parameters were configured in an optimal fashion, i.e. a migration period of 2,000,000 clock cycles, a threshold of 5%, selection policy **select−4** and location policy **locate−3**. The usual representative workload was used, and six replications were executed.

In the following discussion, let the variable *lh_count* denote the number of times that a processor moves directly from the **L** state to the **H** state. When the inundation policy is inactive (i.e. a null policy) the mean *lh_count* value observed is 15, this is a very small number if one considers the maximum possible. With 16

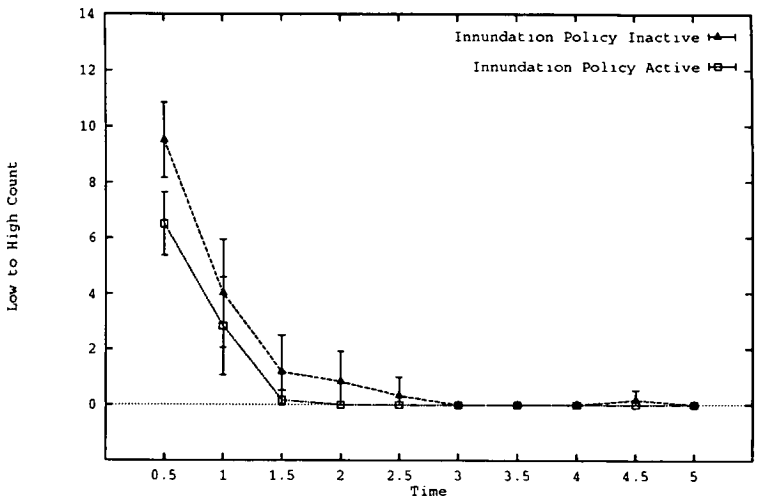

Figure 6–5: Evolution of Low to High Count With Inundation Policy Active or Inactive

processors and a migration period of 2,000,000 clock cycles, there are approximately 800 opportunities for a processor to move directly from the **L** state to the **H** state as a result of processes migrating. When the inundation policy is active, the mean *lh_count* value is reduced, as one would expect, to 8.83. Using a t test one can conclude that this reduction is significant at the 1% significance level.

Figure 6–5 illustrates how the mean *lh_count* values evolve with time when the inundation policy is active and inactive, 95% confidence intervals are given. One can see that the non-zero values of *lh_count* occur early in the simulation, as the migration strategy is converging to an improved placement. Once this placement is obtained, processors rarely move directly from the **L** state to the **H** state. As one would expect, the *lh_count* declines more sharply when the inundation policy is active.

So, it seems that the inundation policy can reduce the value of *lh_count*. However, it is not yet clear how this influences performance. When the inundation policy is inactive, the mean value of U obtained is 88.85%. In contrast, when the inundation policy is active, the mean value of U obtained is 80.39%. It appears

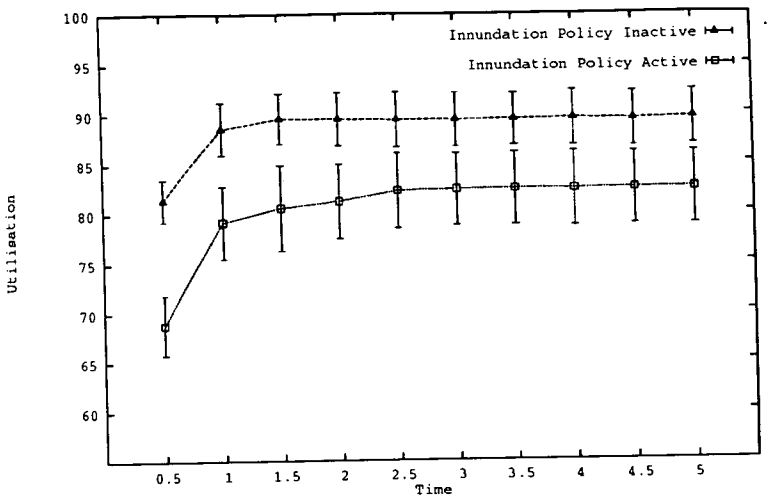

Figure 6–6: Evolution of Utilisation With Inundation Policy Active or Inactive

that the inundation policy has a negative effect on performance, indeed, statistically speaking, one can conclude at the 1% significance level that the inundation policy reduces performance.

Figure 6–6 illustrates how utilisation evolves with time when the inundation policy is active and inactive. It is clear that the inundation policy results in lower utilisations being achieved early in the simulation. It is then not possible for the migration strategy to fully recover from the sub-optimal placement, and so a lower overall utilisation is obtained. This phenomenon could be explored further by studying the geography of process movements at a microscopic level. However, that level of detail is not appropriate for the work presented here.

So, it seems that inhibiting the dissemination of processes about the processor domain in order to reduce the *lh_count* value is not a good idea. This is because it disrupts the convergence properties of the migration strategy, reducing the rate at which processes can be migrated. In any case, instances of processors moving directly from the **L** state to the **H** state are relatively rare, and tend to disappear altogether when a good placement is achieved. This is a consequence of the stability properties already inherent in the migration strategy, i.e. a processor can

never accept more than four processes in a single migration cycle (one per link), and a processor can never move directly from the **H** state to the **L** state as a result of migrating processes.

6.4 Generality of Results

In Section 6.3 optimal settings for the migration control parameters were derived, i.e. a migration period of 2,000,000 clock cycles, a threshold of 5%, selection policy **select−4**, location policy **locate−3** and a null inundation policy. This was done with respect to a representative workload executed on a 4×4 mesh of processors. In this section it is shown how well the results obtained can be generalised for a larger hardware topology, and different types of workload. In addition, the results of some validation work carried out on a real machine are presented.

6.4.1 A Larger Processor Topology

To test the scalability of the results obtained so far, with respect to the number of processors, several simulation experiments were carried out using a 7×7 mesh. With the exception of N, the remaining program parameters were set to the values used for the 4×4 mesh experiments presented in Section 6.3. N was increased to 200, allowing an average of approximately four processes per processor. As previously: six replications were carried out; the random number seed and program graph were varied between replications; simulations were run for 5 seconds of simulated machine time; and a truly random initial placement was used. Within each replication, three trials were executed. The first used no migration strategy, the second used **select−1/locate−1** and the third used **select−4/locate−3**. For the latter two trials, the migration period was always set to 2,000,000 clock cycles, the threshold to 5%, and a null inundation policy was used.

The mean value of U obtained using no migration strategy was 40.61%, compared to 71.26% using the **select−1/locate−1** strategy. Statistically speaking, this result is significant at the 1% significance level. In addition, it is consistent with the results obtained for a 4×4 mesh of processors in Section 6.2.2, where it was observed that a simple migration strategy could always significantly improve performance for programs mapped in a truly random fashion.

The mean value of U obtained using **select−4/locate−3** was 74.67%. Consequently, one can conclude that **select−4/locate−3** results in better performance than **select−1/locate−1** at the 10% significance level. This result is encouraging, since it indicates that the performance of the migration strategy based on minimising the covariates *ce_ratio_sd* and *sync_mean* is scalable. Refering to Table 6–12, it can be seen that the corresponding figures for the 4×4 mesh were 85.07% and 90.99% respectively. These utilisations are higher, because in the 4×4 mesh case there were more processes available, relative to the number of processors.

So, for this example workload, the results observed for a 4×4 mesh of processors have been shown to scale to a 7×7 mesh of processors. Namely, a simple migration strategy is better than none at all, and a strategy based on minimising *ce_ratio_sd* and *sync_mean* seems better than a simple one.

6.4.2 Alternative Workloads

The experiments described in this section were designed to gather more information about the exact conditions under which the covariate-based migration strategy (**select−4/locate−3**) out-performs the simple strategy (**select−1/locate−1**) for a 4×4 processor mesh. Two experiments are presented, the first uses a truly random placement (a strongly unbalanced workload), and the second uses a restricted random placement (a lightly to moderately unbalanced workload). In order to detect the effects of the migration strategies as accurately as possible, within each replication, program graphs of the same shape were are allocated the same initial

placement. Two level full factorial experiments were used to undertake this exploratory work. The program parameters varied were N, which was allocated the value 32 or 142, and $\sigma_{cg\%}$, which was allocated the value 10 or 80. The remaining program parameters were given their usual intermediate values. Six replications were executed, and the results are presented below.

Strongly Unbalanced Workloads

Table 6–13 gives the results of t tests carried out for various subsets of the data. $\overline{U}_{1/1}$ is used to represent the mean value of U over the six replicates when the simple migration strategy was used. Similarly, $\overline{U}_{4/3}$ is used for the covariate based migration strategy. Considering the data as a whole, one can conclude at the 10% significance level, that the covariate strategy performs better than the simple strategy. A closer examination reveals that this improvement is only realised when there are a smaller number of nodes in the program graph, this is especially true when the standard deviation of compute times is also high. For high numbers of nodes, the two migration strategies give very similar performance. However, it has already been shown that the covariate-based migration strategy out-performs the simple one when $N = 100$, so the point of convergence occurs at some value of N higher than 100.

Lightly to Moderately Unbalanced Workloads

Table 6–14 gives the results of t tests carried out for various subsets of the data. It is clear that, for lightly to moderately unbalanced workloads, the behaviour of the two migration strategies is very similar.

Data	$\overline{U}_{1/1}$	$\overline{U}_{4/3}$	n	Reject H_0: $\overline{U}_{1/1} \geq \overline{U}_{4/3}$?
overall	72.55	75.33	24	Yes 10%
$\sigma_{cg\%}$=10	74.68	76.27	12	No
$\sigma_{cg\%}$=80	70.41	74.4	12	Yes 10%
N=32	54.60	59.89	12	Yes 5%
N=142	90.49	90.78	12	No
$\sigma_{cg\%}$=10 N=32	57.61	60.96	6	No
$\sigma_{cg\%}$=80 N=32	51.60	58.81	6	Yes 10%
$\sigma_{cg\%}$=10 N=142	91.75	91.58	6	No
$\sigma_{cg\%}$=80 N=142	89.23	89.98	6	No

Table 6–13: Results of Paired t Tests Comparing Migration Strategies on Strongly Unbalanced Workloads

Data	$\overline{U}_{1/1}$	$\overline{U}_{4/3}$	n	Reject H_0: $\overline{U}_{1/1} \geq \overline{U}_{4/3}$?
overall	84.99	84.87	24	No
$\sigma_{cg\%}$=10	90.92	90.26	12	No
$\sigma_{cg\%}$=80	79.10	79.48	12	No
N=32	77.81	77.82	12	No
N=142	92.22	91.92	12	No
$\sigma_{cg\%}$=10 N=32	88.74	88.74	6	No
$\sigma_{cg\%}$=80 N=32	66.87	66.89	6	No
$\sigma_{cg\%}$=10 N=142	93.11	92.78	6	No
$\sigma_{cg\%}$=80 N=142	91.32	92.06	6	No

Table 6–14: Results of Paired t Tests Comparing Migration Strategies on Lightly to Moderately Unbalanced Workloads

Discussion

We have seen that the more unbalanced the workload, and the lower the number of nodes in the program graph, the better the covariate-based migration strategy performs in relation to the simple migration strategy. This is most likely because the accuracy of each migration decision becomes more important as the number of processes decreases, since the processes in question will correspond to relatively large units of work. For large numbers of processes, each process corresponds to a relatively small unit of work, so the improvements to be had from migrating one process, as opposed to another, are not particularly great.

The covariate-based migration strategy never performs significantly worse than the simple strategy. So, even considering the fact that it only improves performance under certain circumstances, it still seems a worthwhile choice.

6.4.3 Validation

In order to confirm that the results obtained in this chapter are valid, some experiments were carried out using the transputer-based migration mechanism described in Section 3.4. The programs executed were synthetic, and of exactly the same structure as those used in the simulation experiments described in this chapter. As previously, the program parameters N and $\sigma_{cg\%}$ were set at 100 and 40 respectively. The program graph was structured as a 10×10 mesh of processes, since this is a valid member of the set of random program graphs. The remaining program parameters were set to the values assumed throughout this chapter (see Table 6–1). Each process in the program carried out arithmetic operations corresponding to its computation weight, before exchanging dummy messages with neighbours.

The migration control parameters were initially configured as one would expect, i.e. a threshold of 5%, a period of 2,000,000 clock cycles (every 0.1 seconds) and no inundation policy. The selection and location policies were set to **select−1/**

Rep	No Migrations	Overheads Included	Simple Strategy	Covariate Strategy
1	53.95	54.78	42.85	39.51
2	51.27	52.03	39.57	38.02
3	65.55	57.51	42.29	39.97
4	50.77	51.29	43.3	40.88
5	54.05	54.82	42.12	40.34
6	56.01	56.71	41.46	40.1
Overall	53.77	54.52	41.93	39.8

Table 6–15: Execution Times Observed Using A Truly Random Placement

locate–1 or **select–4/locate–3**, according to whether a simple or covariate-based migration strategy was being investigated. However, a migration period of 2,000,000 clock cycles turned out to be unsatisfactory, since thrashing was observed to occur. A value of 4,000,000 was found to be acceptable i.e. every 0.2 seconds. This anomaly can be easily understood if one considers the differences between the migration mechanism used for the simulation experiments, and that used on the transputer. In the simulations, when a particular process is marked for migration, MIMD suspends it the next time it either enters or leaves the processor queue. In contrast, the transputer-based migration mechanism can only suspend a process when the process itself decides to check whether it has been requested to suspend. Therefore, the time for a suspend instruction to be realised is dependent on the program itself, and in any case is longer than the MIMD mechanism, since processes must reach the front of the CPU queue in order to be suspended.

A 4×4 mesh of transputers was used, and each trial was run for 1000 iterations. Table 6–15 shows the results observed using a truly random initial placement. Six replications were executed, each differing in the initial placement and mapping of weights to the nodes and arcs of the process graph. For each replication, six trials were executed, and it is the means of these execution times (in seconds) which are presented in Table 6–15. In addition, the overall mean values are displayed in the final line.

The second column contains the execution times observed with no migration strategy. The third column contains the times observed with the statistics collection and selection/location policies of the migration strategy active. However, in this case instead of migrating processes, dummy messages were sent. The small difference between the second and third columns confirms that the overheads of operating the migration strategy are relatively small. It can be seen from columns four and five that both the simple and covariate-based strategies dramatically improve performance. If one considers the overall figures, using **select−1/locate−1** leads to a 21.9% improvement in performance, and using **select−4/locate−3** leads to a 26% improvement in performance. Paired t tests can be used to give a formal comparison, the corresponding test statistics are 11.27 and 14.4 respectively. Consequently, one can conclude that both strategies improve performance at the 1% significance level.

Examining the performance of the two migration strategies, it can be seen that **select−4/locate−3** out-performs **select−1/locate−1** as the analysis in Section 6.3.2 would suggest. Once again, this comparison can be formalised using a paired t test. The corresponding test statistic is 7.1. With reference to a set of t values, one can conclude that **select−4/locate−3** out-performs **select−1/locate−1** at the 1% significance level. This is an even stronger result than that reported in Section 6.3.2 (where the test was only significant at the 5% level).

Table 6–16 shows a corresponding set of results to those discussed above, obtained using a restricted random placement. Once again six replications were used, and the figure given for each replicate is the mean of six observations. The first point to notice is that the difference between columns two and three is small, again indicating that the overheads of executing the strategy are acceptable. An examination of columns four and five reveals that the migration strategy still brings performance improvements. However, these are not as striking as previously, since the restricted random placement will usually provide a better initial mapping than a truly random placement. This agrees with the observations made in Section 6.2.1,

Rep	No Migrations	Overheads Included	Simple Strategy	Covariate Strategy
1	42.46	43.06	39.98	39.03
2	38.84	39.49	37.6	36.58
3	43.89	44.39	39.55	39.34
4	42.14	42.93	39.37	39.17
5	40.58	40.99	37.77	37.99
6	41.7	42.33	38.38	38.14
Overall	41.6	42.2	38.77	38.37

Table 6–16: Execution Times Observed Using A Restricted Random Placement

i.e. that the use of a migration strategy can improve the performance of programs mapped using a restricted random placement, as long as the programs themselves display a degree of non-uniformity. This was the case in this experiment, since a $\sigma_{cg\%}$ value of 40 was assumed. As usual, t tests can be used to confirm that both migration strategies improve performance. The corresponding test statistics are 6.74 for a comparison between no strategy and **select−1/locate−1**; and 9.79 for a comparison between no strategy **select−4/locate−3**. Accordingly, one can conclude at the 1% significance level that both strategies improve performance.

The investigation presented in Section 6.4.2 into the circumstances in which **select−4/locate−3** out-performs **select−1/locate−1** suggests that the performance of both strategies are very similar for programs mapped using a restricted random placement strategy. If one examines Table 6–16, it can be seen that this indeed appears to be the case. The performance of the two strategies are very similar. Indeed, using a paired t test and a corresponding test statistic of 2.04, one can conclude at the 10% significance level that the execution times observed under the two strategies are the same.

An investigation of the convergence properties of the migration strategies using the transputer-based migration mechanism gave no cause for concern. The stability properties observed in the simulation experiments were also found to hold, as long as the migration period did not fall below 0.2 seconds. Convergence was

generally reached after 6-8 seconds, with the vast majority of the migrations occurring in the first 3-4 seconds or so. This is a longer time than that observed in the simulations, however, this is consistent with the use of a longer migration period.

The results obtained from the validation experiments described here give no reason to doubt the results of the simulation work presented earlier. Indeed, it seems that we can be confident that the simulation environment provides an acceptable approximation of the behaviour of a real machine.

6.5 Summary and Conclusions

It has been demonstrated how a relatively simple migration policy can be used to, generally speaking, improve the performance of a class of time-invariant, non-uniform parallel programs mapped in an unbalanced manner. It was shown that the policy exhibited good convergence and stability properties. The parameters controlling the operation of the migration strategy were explored, with reference to a representative workload, in order find their optimal settings. Process selection and process location policies based on minimising the covariates *ce_ratio_sd* and *sync_mean* were shown to significantly improve the performance of the policy. Further experiments revealed that the performance of the covariate-based strategy was best for strongly unbalanced workloads (as long as the number of nodes was not too high). In any case, the covariate-based strategy never performed significantly worse than the simple strategy. Some validation work was carried out on a transputer-based machine, and the simulation results were found to be accurate.

So, we can see that the covariates identified using the analysis of covariance in Chapter 5 can be used to guide the construction of a process migration strategy. It appears to be possible to use local decision making in order to globally optimise the values of these covariates at run-time, and so improve performance. These

techniques presented here are general in nature, and could equally be applied in alternative environments.

Chapter 7

Time-varying Programs and Process Migration

This chapter presents a preliminary investigation into the behaviour of the covariate-based migration strategy derived in Chapter 6, when presented with time-varying programs. The motivation behind this study is, given a particular program, to be able to predict whether the process migration strategy can improve its performance. It would also be desirable to be able to predict the extent of any possible performance improvements.

Section 7.1 shows how the program parameter set assumed up to this point can be extended to support time-varying behaviour. Section 7.2 describes an experiment designed to explore the nature of the relationship existing between the structure of time-varying programs, and the performance of a process migration strategy. Section 7.3 presents the results of some validation work carried out on a real transputer-based machine. Finally, a summary and conclusions are presented in Section 7.4.

7.1 Characterising Time-varying Behaviour

You will recall that I am concentrating on loosely synchronous, data parallel programs. Many programs conforming to this model display time-varying behaviour, in the sense that they proceed through a number of phases characterised by distinct patterns of activity. Furthermore, these behaviour patterns are often data-dependent, and it may not be possible to predict them in advance.

Time-varying behaviour could be modelled in a number of different ways. For example, computation times and message lengths could follow time-dependent patterns of activity defined by sine waves or step functions. Such techniques are used to characterise the workloads of multi-user systems in [10]. However, in the interests of simplicity and continuity, an extended form of the existing program model will be used to represent time-varying patterns of activity.

So far, the following program parameters have been assumed:

$$\{N, c, \mu_{cg}, \sigma_{cg}, \mu_{mg}, \sigma_{mg}, \sigma_c, \sigma_m\}$$

It is clear that both σ_c and σ_m are related to the time-varying behaviour of a program, since they characterise the differences between one iteration of the program and the next. However, simply using large values of σ_c and σ_m in order to generate significantly different compute times and message lengths between successive iterations of each process is not an appropriate approach to take. This is because the behaviour of programs constructed in this manner would fluctuate wildly, and consequently, it would be very difficult for a process migration strategy to keep up with the evolution of the program. In addition, such unstable programs would appear to have little correspondence with real programs (for the loosely synchronous program model assumed here).

Rather than fluctuating wildly, loosely synchronous parallel programs tend to exhibit phases where, generally speaking, each process will either be in a steady

state or a transitional state. Within a steady state, processes will display similar behaviour patterns from one iteration to the next. According to the characteristics of the individual program, the relative lengths of steady and transitional phases will vary. At one end of the spectrum lie programs whose behaviour patterns are constantly evolving; in such cases the program is permanently in a transitional state. Alternatively, the transitional phases might be of negligible length, so that processes pass through a series of distinct states. Of course, the behaviour of most programs will lie somewhere between these two extremes.

The program parameter set used in previous chapters can be extended with a number of extra parameters enabling time-varying parallel programs to be adequately represented. I am interested in generating relatively straightforward time-varying patterns of activity in order to carry out exploratory experiments. Accordingly, some simplifying assumptions have been made. Firstly, it is assumed that transitional phases are of negligible length, so processes move directly between consecutive steady states i.e. the processes modelled display sharp phase changes. Secondly, I only attempt to characterise the time-varying behaviour of compute times, and not message lengths. This would seem reasonable, since it has already been established that the particular program structure and message passing protocol used in this thesis results in message passing costs being dominated by synchronisation times, rather than by actual message transfer times. Message lengths, therefore, do not have a major impact on performance.

So, in order to model time-varying behaviour, four new parameters are now introduced defining the phase characteristics of processes. The new parameters are:

$$\{\mu_{pg}, \sigma_{pg}, \sigma_p, \sigma_{pc}\}$$

The parameters μ_{pg} and σ^2_{pg} define a normal distribution describing the distribution of phase lengths over the nodes of the program graph, and σ^2_p defines a chi-square distribution describing the variance in phase lengths between successive phases. Note, that for the sake of consistency, $\sigma_{pg\%}$ may be used to express σ_{pg} as a

proportion of μ_{pg}. The parameter σ_{pc} describes the degree of variation in compute times between successive phases of the computation (i.e. the magnitude of phase changes), it is drawn from a chi-square distribution. The parameters described above allow the following behaviour to be characterised:

1. The mean phase lengths of the processes (μ_{pg}, $\sigma_{pg\%}$).

2. An overall measure of the degree of variation in phase lengths between successive phases (σ_p).

3. An overall measure of the degree of variation in compute times between successive phases (σ_{pc}).

The choice of a normal distribution to represent phase lengths is a simplifying assumption. The approach described here is consistent with that used previously to allocate compute times to processes, and message lengths to channels.

To give an idea of the sorts of time-varying behaviour that can be generated using the new program model, two simple simulation trials were executed. The parameter values that were used are summarised in Table 7–1. The values of $\sigma_{cg\%}$ and $\sigma_{mg\%}$ were set so that there was relatively little variation in the initial mean compute times and mean message lengths across the nodes and arcs of the process graph. Also, by setting σ_c and σ_m to 0, I ensured that there was little variation in behaviour between successive iterations within a single phase. The four new program parameters were used to specify the time-varying behaviour. A σ_{pc} value of 10,000 resulted in a reasonable amount of variation in compute behaviour between successive phases, considering that the mean compute time for the processes was set to 40,000 clock cycles. The mean phase length of the processes was set to 20,000,000 clock cycles. Two trials were carried out, one with $\sigma_{pg\%}$ set to 1, and one with $\sigma_{pg\%}$ set to 20. The first of these trials represented a program displaying synchronised global phase changes, since phase lengths were defined to be of a very similar length for all processes. The second trial introduced more locally

Program Parameters	
Param	Value(s)
N	32
c	4
μ_{cg}	40000
$\sigma_{cg}\%$	10
σ_c	0
μ_{mg}	1000
$\sigma_{mg}\%$	10
σ_m	0
μ_{pg}	20,000,000
$\sigma_{pg}\%$	{1,20}
σ_p	10
σ_{pc}	10000

Other Parameters	
Param	Value(s)
Hardware	4×4 Mesh
Placement	Restricted Random
Trial Length	100,000,000 (5 Secs)
Replications	1

Table 7–1: Parameter Settings for Preliminary Experiment

oriented behaviour, with processes undergoing phase changes at different times. To simplify matters, the phase length displayed by each process, once allocated, was kept almost constant by setting σ_p to 10.

Only one replication was used, since the purpose of this experiment was merely to illustrate the different types of activity which can be represented using the extended parameter set. The patterns of activity generated by the first trial, where $\sigma_{pg}\%$ was defined to be 1, are shown in Figure 7–1. This graph shows the compute requirements of six randomly chosen processes over the entire simulation period. One can clearly see the phase changes occurring every second (20,000,000 clock cycles). The phase changes are of a global nature, occurring at approximately the same time for each process because of the small value of $\sigma_{pg}\%$. Successive phases are of similar length because of the small value of σ_p. Figure 7–2 shows a similar graph for the second trial. The patterns of activity are far more interesting in this

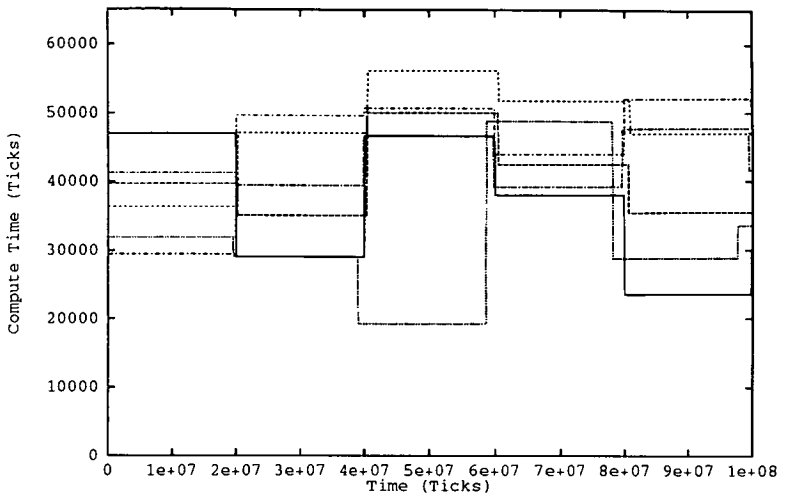

Figure 7–1: Evolution of Compute Times With Global Phase Changes for Six Arbitrarily Chosen Processes

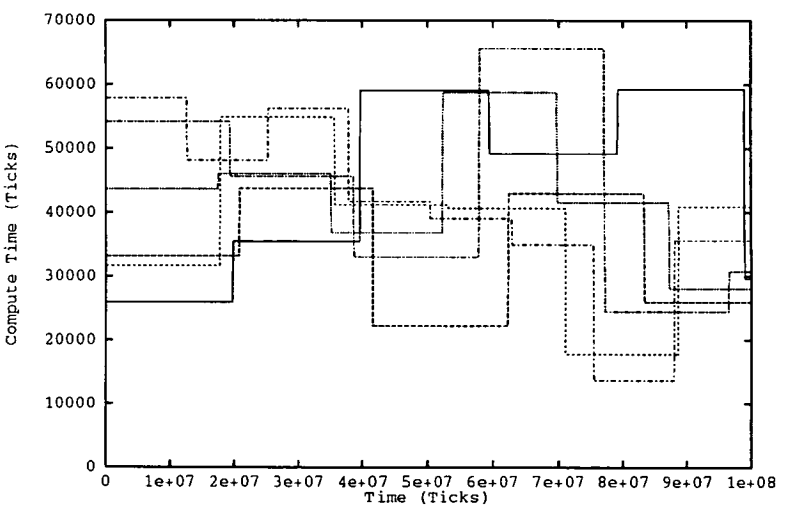

Figure 7–2: Evolution of Compute Times With Local Phase Changes for Six Arbitrarily Chosen Processes

case, because the larger value of $\sigma_{pg\%}$ has resulted in phase lengths varying over the nodes of the program graph. Consequently, phase changes occur at different times for individual processes.

This section has demonstrated how the addition of four extra parameters to the original program parameter set has allowed reasonable, although admittedly simple, time-varying patterns of activity to be generated. I now investigate the behaviour of these time-varying programs when executed under the control of a process migration strategy.

7.2 Process Migration and Time-varying Programs: Preliminary Investigations

In Chapter 6 we saw how process migration strategies could be used to improve the performance of time-invariant parallel programs by improving a non-optimal initial mapping. The strategies were observed to display consistent and stable behaviour patterns. The experiment described in this section explores the behaviour of time-varying programs when executed using the covariate-based process migration strategy.

7.2.1 Experiment Description

A full factorial experiment was carried out varying three of the twelve program parameters. At each of these parameter combinations, two trials were executed, one with the migration strategy turned on, and one with the migration strategy turned off. The exact parameter values used are summarised in Table 7–2.

This experiment was designed to study the influence of the structure of a time-varying program on the performance of the migration strategy. As a result,

program parameters concerned with such things as the shape of the graph and the sizes of messages remained fixed at reasonable intermediate levels. Some random noise was introduced between successive iterations within the same phase by setting σ_c to 1000. This was done in order to mimic the situation often found in real programs, where computation demands vary between successive iterations, as well as between successive phases (although to a lesser extent). The phase characteristics of the programs were defined by setting σ_{pc}, μ_{pg}, $\sigma_{pg\%}$ and σ_p to appropriate values. The value of σ_{pc} was varied from 2000 to 32,000 in steps of 10,000, thereby defining phases changes of increasing magnitude. The mean phase length of the processes (in clock cycles) was varied from 1,000,000 to 22,000,000 in steps of 7,000,000. The standard deviation of the phase lengths over the nodes of the graph was set to either 1% or 20% of the mean, corresponding to purely global or more local phase changes. By setting σ_p to 1000, the phase lengths of processes were defined to remain relatively constant once decided upon.

The migration strategy was applied according to its optimal settings derived in Chapter 6. You will note that each trial was run for 30 seconds of simulated machine time (i.e. 600,000,000 clock cycles). This period is longer than that used in previous experiments in order to allow time-varying behaviour to be observed over a reasonable period. Replicates differed both in the random graph and random number seed used. Those trials within a replication which differed only in the state of the migration strategy, were allocated the same initial placement in order that any differences in performance could be attributed solely to the actions of the migration strategy.

7.2.2 Results

Table 7–3 shows an extended analysis of variance table using U as the response variable (diagnostic plots revealed no need for a transformation). The letter s is used to represent the state of the migration strategy, which can either be active

Program Parameters	
Param	Value(s)
N	100
c	8
μ_{cg}	40000
$\sigma_{cg}\%$	10
σ_c	1000
μ_{mg}	1000
$\sigma_{mg}\%$	10
σ_m	1
μ_{pg}	{1e7, 8e7, 1.5e8, 2.2e8}
$\sigma_{pg}\%$	{1,20}
σ_p	1000
σ_{pc}	{2000, 12000, 22000, 32000}

Other Parameters	
Param	Value(s)
Migration Strategy	{On, Off}
Process Size	2000
Hardware	4×4 Mesh
Placement	Restricted Random
Trial Length	600,000,000 (30 Secs)
Replications	6

Table 7–2: Parameter Settings for Investigation into Time-varying Program Behaviour

Effect	D o F	Sum of Squares	Mean Squares	F Ratio	F Prob.
s	1	15420.2	15420.2	7371.1	<.01
σ_{pc}	3	7012.3	2337.4	1117.3	<.01
μ_{pg}	3	246.35	82.12	39.25	<.01
$\sigma_{pg\%}$	1	7.64	7.64	3.65	0.057
$s\sigma_{pc}$	3	3042.2	1014.1	484.73	<.01
$s\mu_{pg}$	3	698.24	232.75	111.26	<.01
$\sigma_{pc}\mu_{pg}$	9	223.93	24.88	11.89	<.01
$s\sigma_{pg\%}$	1	4.73	4.73	2.26	0.134
$\sigma_{pc}\sigma_{pg\%}$	3	22.25	7.42	3.55	0.015
$\mu_{pg}\sigma_{pg\%}$	3	5.28	1.76	0.84	0.472
$s\sigma_{pc}\mu_{pg}$	9	223.4	24.82	11.87	<.01
$s\sigma_{pc}\sigma_{pg\%}$	3	10.62	3.54	1.69	0.169
$s\mu_{pg}\sigma_{pg\%}$	3	0.76	0.25	0.12	0.948
$\sigma_{pc}\mu_{pg}\sigma_{pg\%}$	9	35.23	3.91	1.87	0.056
$s\sigma_{pc}\mu_{pg}\sigma_{pg\%}$	9	21.86	2.43	1.16	0.32
Residual	320	669.44	2.09		
Total	383	27644.42			

Table 7–3: Extended Analysis of Variance Table

or inactive. You will recall that the F ratio column allows the relative importance of the main effects and their interaction terms to be compared. The F probability column shows the level at which each of the terms is statistically significant.

The model fitted by the analysis of variance explains 97.6% of the variance in U. This value was obtained by expressing the sum of squares attributable to the various terms as a percentage of the total sum of squares. The remaining variation can be attributed to experimental error. From the F ratio column it can be seen that the migration strategy effect, s, is clearly the dominant factor, itself explaining over 55% of the variation in U. The next most important factors are the σ_{pc} effect, and the $s\sigma_{pc}$ interaction term (you will recall that σ_{pc} represents the magnitude of phase changes). To confirm that the effects of s and σ_{pc} operate in the expected directions, one can examine the mean values of U observed after partitioning the

data according to the values of s and σ_{pc}[1]. When s was active the mean value of U observed was 90.29, compared to a value of 77.61 when s was inactive. This confirms that the migration strategy has a beneficial effect on processor utilisation. Similarly, the values of U observed when $\sigma_{pc} = 2000, 12000, 22000$ and 32000 were 88.91, 86.26, 81.03 and 78.05 respectively. This is as one would expect; as phase changes increase in magnitude, synchronisation delays are greater and processor utilisation decreases.

The other terms in the table seem relatively unimportant in comparison to those discussed above. However, of those remaining, the mean phase length, μ_{pg}, seems to have the greatest influence. The standard deviation of phase lengths, $\sigma_{pg\%}$, has a relatively small impact on processor utilisation.

7.2.3 Discussion

The above analysis, while indicating that the migration strategy can improve performance, reveals very little about the dynamic behaviour of the strategy. For example, are there any significant interactions between the strategy and programs? What are the circumstances in which the strategy performs at its best? To answer questions like these I will now study in some detail the dynamic behaviour of the migration strategy. This is essential in order to understand the relationships which exist between the program parameters and the performance of the strategy.

A large number of statistics are collected by MIMD during the execution of simulation trials. As well as providing total figures relating to the entire simulation period, the statistics can be collected at user-defined intervals throughout the simulation period. For the experiment described here, the 30 second simulation

---

[1]This is necessary since an extended analysis of variance does not contain any information regarding the signs of effects.

period was divided into 60 × 0.5 second time frames. In each of the graphs that follow, I plot three data sets (consisting of 60 observations each) over the entire simulation period. These three data sets are: the values of U observed with the migration strategy inactive; the values of U observed with the migration strategy active; and the number of migrations executed. The analysis is divided into two sections. First of all, those trials exhibiting synchronised global phase changes ($\sigma_{pg\%} = 1$) are examined. I then turn my attention to those trials exhibiting more locally oriented behaviour, where processes undergo phase changes at different times ($\sigma_{pg\%} = 20$).

Global Phase Changes

Figures 7–3 and 7–4 present graphs of the form described above for each of the possible combinations of the parameters σ_{pc} and μ_{pg}, when $\sigma_{pg\%}$ is equal to 1. Each of the points plotted is the mean value taken over the six replications; confidence intervals are not shown since they would make the graphs too difficult to read.

Graphs (a)-(d) in Figure 7–3 illustrate the dynamic behaviour of the migration strategy when presented with time-varying programs displaying phase changes of small magnitude i.e. $\sigma_{pc} = 2000$. As one would expect, the computational demands of processes change little between successive phases. By studying the behaviour observed when the migration strategy is turned off, one can see that processor utilisation is almost constant. This is to be expected from programs of this type. By examining the observed processor utilisation figures with the migration strategy turned on, it can be seen that in all all four cases improved figures are obtained. In addition, these improvements are sustained for the entire simulation period. The graphs in Figure 7–3 also show the actions of the migration strategy over time. It is clear that relatively small numbers of migrations are required (usually less than 20) in each 0.5 second observation period. This is of the order shown to be acceptable in Chapter 6, so I can be confident that

the increased utilisation observed under the migration strategy can be attributed to improved program performance, rather than to migration overheads. However, this can be confirmed by studying the $\sigma_{pc} = 2000$ section of Table 7–4. This table includes the results of paired t tests carried out on the data corresponding to each of the graphs in Figure 7–3. For each parameter combination, the results of two t tests are presented. The tests compare the mean values of two different metrics, U and M, obtained with the migration strategy inactive (denoted by $\overline{U}$ or $\overline{M}$) to those obtained with it active (denoted by $\overline{U}_s$ or $\overline{M}_s$). Each test is based on six pairs of observations, since there were six replications of the experiment. You will recall from Chapter 6 that M is defined as the mean number of messages sent down a channel. This metric is specific to the program model used here, and is very closely related to the rate of computation. For both metrics, one can always conclude that the migration strategy improves performance at the 1% level. This confirms that the increased processor utilisation can be attributed to improved program performance. It seems that, for programs exhibiting phase changes of small magnitude, the migration strategy is capable of obtaining significant performance improvements with relatively small numbers of migrations.

Graphs (e)-(h) in Figure 7–3 illustrate the behaviour observed as phase changes are increased in magnitude i.e. $\sigma_{pc} = 12000$. Ignoring Graph (e) for the moment, Graphs (f)-(h) show the migration strategy working well for programs displaying increasing phase lengths. The phase changes are clearly visible in the processor utilisation plots. When the migration strategy is active, sustained performance improvements of the order of 10% or so are consistently achieved. Each time a phase change occurs, performance begins to drop off as the mapping used for the previous phase becomes inappropriate. At this point the migration strategy detects an unbalanced situation developing and activates itself. An improved mapping is generally obtained very quickly. One can see that the actions of the migration strategy are stable, and good convergence properties similar to those observed in Chapter 6 are displayed. These conclusions are confirmed by the

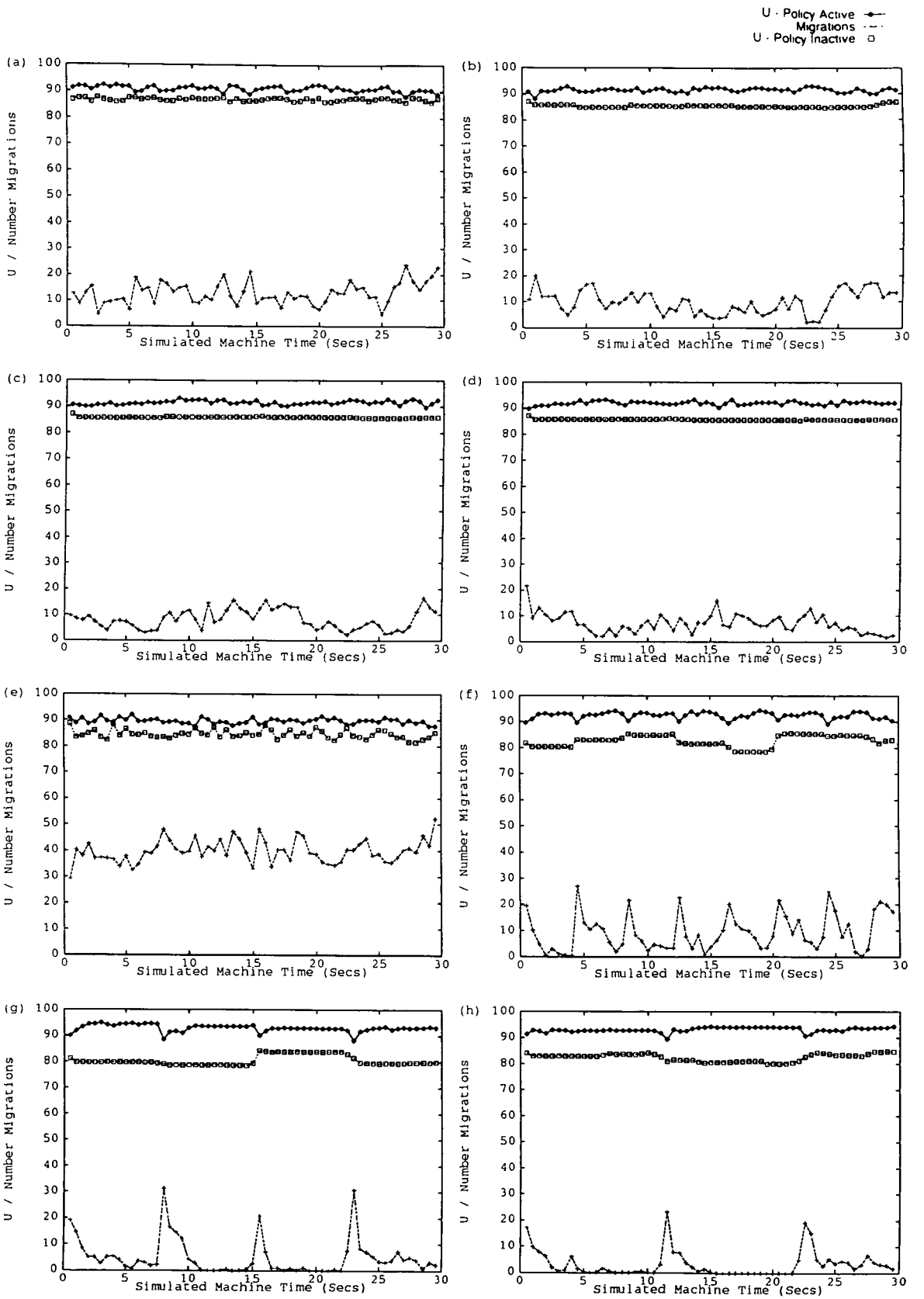

Figure 7–3: Dynamic Behaviour Where $\sigma_{pg}\% = 1$ and $\sigma_{pc} = 2000$ or 12000:
(a) $\sigma_{pc} = 2000$, μ_{pg}=1e7 (0.5 secs) (b) $\sigma_{pc} = 2000$, μ_{pg}=8e7 (4 secs)
(c) $\sigma_{pc} = 2000$, μ_{pg}=1.5e8 (7.5 secs) (d) $\sigma_{pc} = 2000$, μ_{pg}=2.2e8 (11 secs)
(e) $\sigma_{pc} = 12000$, μ_{pg}=1e7 (0.5 secs) (f) $\sigma_{pc} = 12000$, μ_{pg}=8e7 (4 secs)
(g) $\sigma_{pc} = 12000$, μ_{pg}=1.5e8 (7.5 secs) (h) $\sigma_{pc} = 12000$, μ_{pg}=2.2e8 (11 secs)

Data	$\overline{U}$	$\overline{U}_s$	Reject H_0:	Lev	$\overline{M}$	$\overline{M}_s$	Reject H_0:	Lev
$\sigma_{pc} = 2000$								
$\mu_{pg} = 1e7$	87.28	90.98	$\overline{U} \geq \overline{U}_s$	1%	2030	2114	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 8e7$	85.68	91.46	$\overline{U} \geq \overline{U}_s$	1%	2003	2170	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 1.5e8$	86.36	91.48	$\overline{U} \geq \overline{U}_s$	1%	2027	2150	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 2.2e8$	85.71	92.01	$\overline{U} \geq \overline{U}_s$	1%	2005	2158	$\overline{M} \geq \overline{M}_s$	1%
$\sigma_{pc} = 12000$								
$\mu_{pg} = 1e7$	85.57	89.67	$\overline{U} \geq \overline{U}_s$	1%	1940	2041	$\overline{M} \geq \overline{M}_s$	5%
$\mu_{pg} = 8e7$	82.8	91.45	$\overline{U} \geq \overline{U}_s$	1%	1970	2180	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 1.5e8$	80.94	92.96	$\overline{U} \geq \overline{U}_s$	1%	1903	2192	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 2.2e8$	82.43	93.05	$\overline{U} \geq \overline{U}_s$	1%	1938	2202	$\overline{M} \geq \overline{M}_s$	1%
$\sigma_{pc} = 22000$								
$\mu_{pg} = 1e7$	74.11	82.11	$\overline{U} \geq \overline{U}_s$	1%	1705	1803	$\overline{M} \geq \overline{M}_s$	10%
$\mu_{pg} = 8e7$	72.52	91.19	$\overline{U} \geq \overline{U}_s$	1%	1695	2123	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 1.5e8$	74.03	92.31	$\overline{U} \geq \overline{U}_s$	1%	1697	2164	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 2.2e8$	73.66	92.83	$\overline{U} \geq \overline{U}_s$	1%	1694	2141	$\overline{M} \geq \overline{M}_s$	1%
$\sigma_{pc} = 32000$								
$\mu_{pg} = 1e7$	69.67	79.41	$\overline{U} \geq \overline{U}_s$	1%	1508	1601	$\overline{M} \geq \overline{M}_s$	10%
$\mu_{pg} = 8e7$	67.15	90.21	$\overline{U} \geq \overline{U}_s$	1%	1489	1924	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 1.5e8$	69.17	91.57	$\overline{U} \geq \overline{U}_s$	1%	1525	1970	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 2.2e8$	67.64	90.48	$\overline{U} \geq \overline{U}_s$	1%	1509	2000	$\overline{M} \geq \overline{M}_s$	1%

Table 7–4: Results of Paired t Tests for Different Values of σ_{pc} and μ_{pg} when $\sigma_{pg\%} = 1$

$\sigma_{pc} = 12000$ section of Table 7–4, which indicates that one can conclude at the 1% significance level that the migration strategy improves performance for both the U and M metrics in all three cases.

However, if Graph (e) in Figure 7–3 is examined, it is possible to see a rather different picture emerging for programs with very short phase lengths. In such cases the activity of the migration strategy seems unstable. This is because the strategy is unable to keep up with the programs it is operating on. Just as it is beginning to adapt to one phase, another phase change occurs, and it has to start all over. This results in a phenomenon known as *thrashing*, i.e. processes tend to migrate back and forward between processors without ever settling in one place. The overheads imposed by this activity mean that I need to investigate the extent

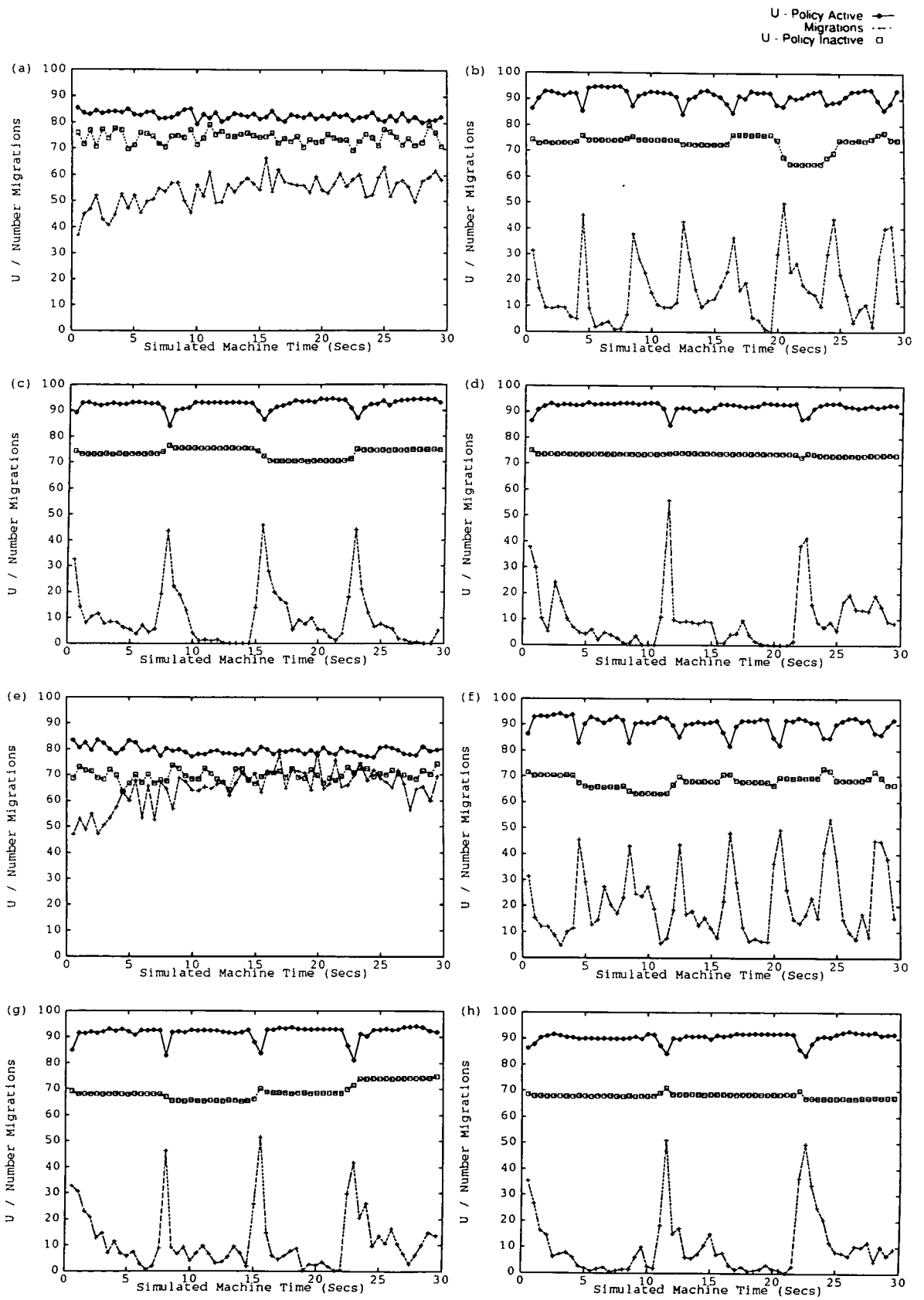

Figure 7–4: Dynamic Behaviour Where $\sigma_{pg\%} = 1$ and $\sigma_{pc} = 22000$ or 32000:
(a) $\sigma_{pc} = 22000$, μ_{pg}=1e7 (0.5 secs) (b) $\sigma_{pc} = 22000$, μ_{pg}=8e7 (4 secs)
(c) $\sigma_{pc} = 22000$, μ_{pg}=1.5e8 (7.5 secs) (d) $\sigma_{pc} = 22000$, μ_{pg}=2.2e8 (11 secs)
(e) $\sigma_{pc} = 32000$, μ_{pg}=1e7 (0.5 secs) (f) $\sigma_{pc} = 32000$, μ_{pg}=8e7 (4 secs)
(g) $\sigma_{pc} = 32000$, μ_{pg}=1.5e8 (7.5 secs) (h) $\sigma_{pc} = 32000$, μ_{pg}=2.2e8 (11 secs)

to which the increased utilisation observed under the migration strategy can be attributed to migration overheads. In Table 7–3 one can see that when $\sigma_{pc} = 12000$ and $\mu_{pg} = 1e7$ it is still possible to conclude that the migration strategy improves U at the 1% level. However, it is possible to conclude that the migration strategy increases the value of M only at the 5% level. This indicates that the migration overheads are beginning to occupy a sizeable proportion of the increased processor utilisation generated by the actions of the migration strategy.

Figure 7–4 presents activity graphs for trials exhibiting phase changes of increasing magnitude. Graphs (a)-(d) relate to those trials where $\sigma_{pc} = 22000$, and Graphs (e)-(h) relate to those trials where $\sigma_{pc} = 32000$. For the moment I will ignore the trials with the shortest phase lengths i.e. Graphs (a) and (e). Consider the remaining graphs. It can be seen that the actions of the migration strategy consistently improve processor utilisation. The larger the value of σ_{pc}, the greater this improvement appears to be. This is because phase changes of increased magnitude result in more unbalanced programs, therefore the potential benefits of a migration strategy are greater. The activity of the strategy seems stable, and good convergence properties are displayed, similar to those observed in Chapter 6. Slightly more migrations are required to reach an improved mapping than observed in Figure 7–3, where phase changes were of smaller magnitude. This is because phase changes of larger magnitude mean that successive phases have less in common with one another, so a larger number of migrations are required to reach an improved placement each time round. Table 7–4 confirms that, at the 1% significance level, the migration strategy improves performance for both U and M in the cases discussed above.

Returning to Graphs (a) and (e), we can see that the trend noted in the discussion of Graph (e) in Figure 7–3 continues. Namely, for short phase lengths the activity of the migration strategy becomes unstable, and convergence is never reached. The entries in Table 7–4 corresponding to Graphs (a) and (e) show that one can accept at the 1% significance level that the migration strategy increases

Data	$\overline{U}$	$\overline{U}_s$	Reject H_0:	Lev	$\overline{M}$	$\overline{M}_s$	Reject H_0:	Lev
$\sigma_{pc} = 2000$								
$\mu_{pg} = 1e7$	86.51	91.53	$\overline{U} \geq \overline{U}_s$	1%	2024	2141	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 8e7$	85.3	91.93	$\overline{U} \geq \overline{U}_s$	1%	2002	2176	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 1.5e8$	86.34	91.65	$\overline{U} \geq \overline{U}_s$	1%	2013	2148	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 2.2e8$	86.65	91.74	$\overline{U} \geq \overline{U}_s$	1%	2032	2133	$\overline{M} \geq \overline{M}_s$	1%
$\sigma_{pc} = 12000$								
$\mu_{pg} = 1e7$	85.93	89.59	$\overline{U} \geq \overline{U}_s$	1%	1987	2065	$\overline{M} \geq \overline{M}_s$	5%
$\mu_{pg} = 8e7$	82.32	92.13	$\overline{U} \geq \overline{U}_s$	1%	1954	2159	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 1.5e8$	81.84	92.28	$\overline{U} \geq \overline{U}_s$	1%	1919	2147	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 2.2e8$	82.1	92.51	$\overline{U} \geq \overline{U}_s$	1%	1907	2160	$\overline{M} \geq \overline{M}_s$	1%
$\sigma_{pc} = 22000$								
$\mu_{pg} = 1e7$	73.73	81.82	$\overline{U} \geq \overline{U}_s$	1%	1703	1805	$\overline{M} \geq \overline{M}_s$	10%
$\mu_{pg} = 8e7$	72.79	89.89	$\overline{U} \geq \overline{U}_s$	1%	1693	2104	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 1.5e8$	72.33	90.92	$\overline{U} \geq \overline{U}_s$	1%	1699	2115	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 2.2e8$	70.73	91.62	$\overline{U} \geq \overline{U}_s$	1%	1644	2120	$\overline{M} \geq \overline{M}_s$	1%
$\sigma_{pc} = 32000$								
$\mu_{pg} = 1e7$	69.55	78.48	$\overline{U} \geq \overline{U}_s$	1%	1506	1554	$\overline{M} \neq \overline{M}_s$	10%
$\mu_{pg} = 8e7$	67.57	88.52	$\overline{U} \geq \overline{U}_s$	1%	1476	1917	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 1.5e8$	68.43	90.17	$\overline{U} \geq \overline{U}_s$	1%	1475	1928	$\overline{M} \geq \overline{M}_s$	1%
$\mu_{pg} = 2.2e8$	70.1	90.78	$\overline{U} \geq \overline{U}_s$	1%	1523	1925	$\overline{M} \geq \overline{M}_s$	1%

Table 7–5: Results of Paired t Tests for Different Values of σ_{pc} and μ_{pg} when $\sigma_{pg\%} = 20$

U. In contrast, one can accept that the migration strategy increases M only at the 10% significance level. This indicates that I cannot be as confident that the increased utilisation provided by the migration strategy can be attributed to increased program performance, rather than migration overheads.

Local Phase Changes

This section concentrates on those trials where $\sigma_{pg\%}$ was set to 20. These trials do not display the synchronised global phase changes which were observed when $\sigma_{pg\%} = 1$. Instead, phase changes occur at different times for different processes, thereby resulting in gradual transitions in behaviour over the processes of the program.

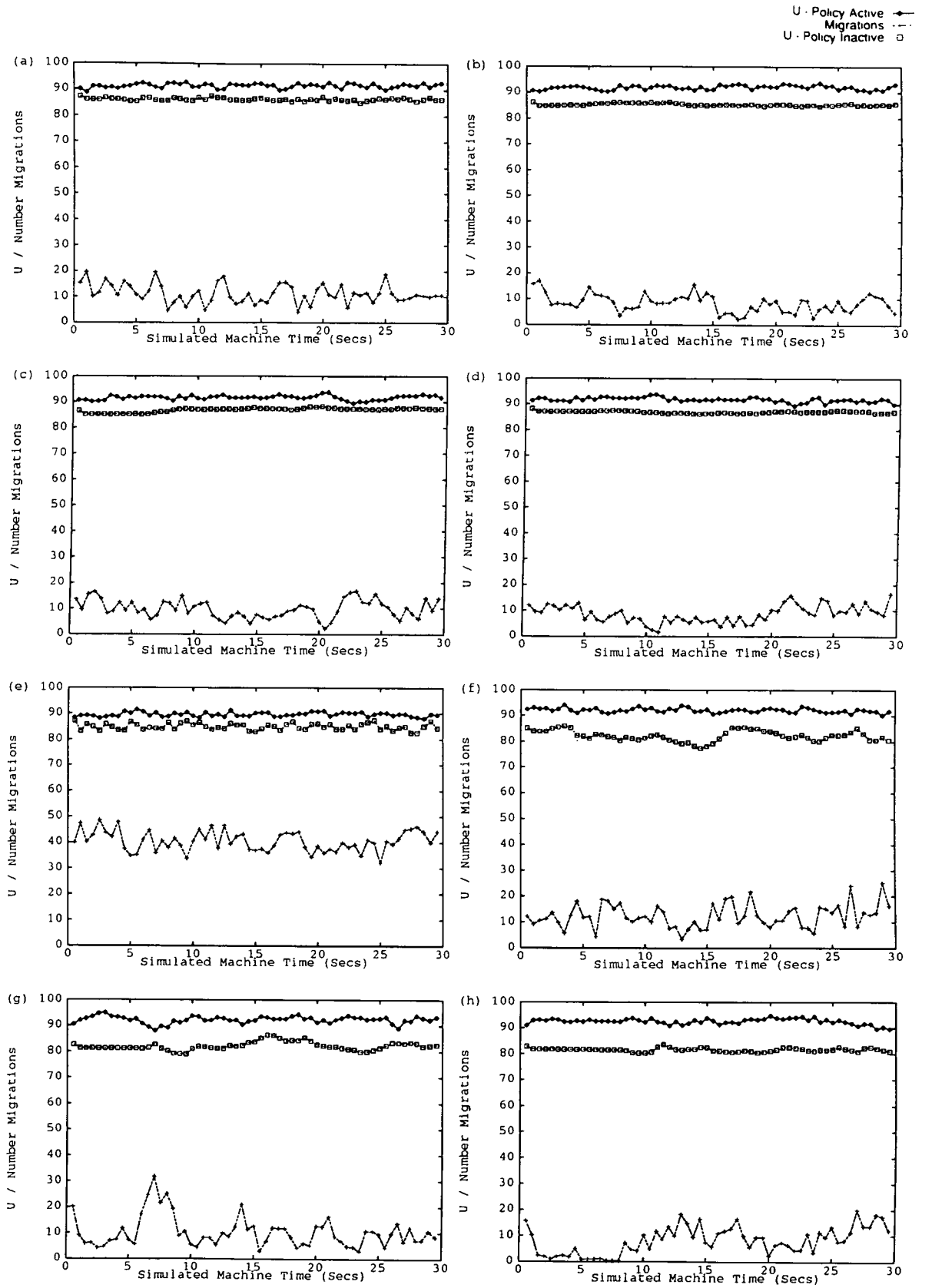

Figure 7–5: Dynamic Behaviour Where $\sigma_{pg\%} = 20$ and $\sigma_{pc} = 2000$ or 12000:
(a) $\sigma_{pc} = 2000$, μ_{pg}=1e7 (0.5 secs) (b) $\sigma_{pc} = 2000$, μ_{pg}=8e7 (4 secs)
(c) $\sigma_{pc} = 2000$, μ_{pg}=1.5e8 (7.5 secs) (d) $\sigma_{pc} = 2000$, μ_{pg}=2.2e8 (11 secs)
(e) $\sigma_{pc} = 12000$, μ_{pg}=1e7 (0.5 secs) (f) $\sigma_{pc} = 12000$, μ_{pg}=8e7 (4 secs)
(g) $\sigma_{pc} = 12000$, μ_{pg}=1.5e8 (7.5 secs) (h) $\sigma_{pc} = 12000$, μ_{pg}=2.2e8 (11 secs)

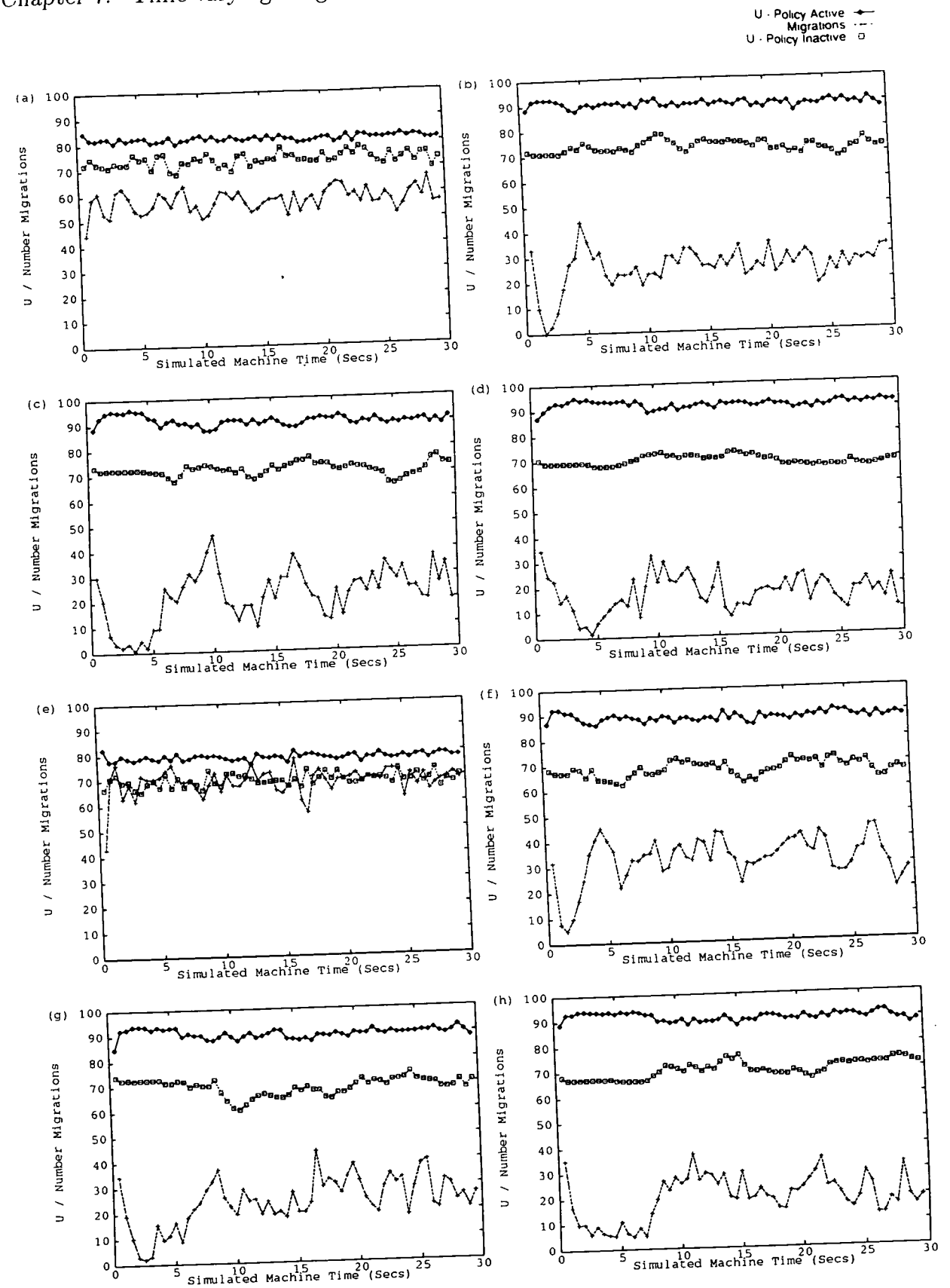

Figure 7–6: Dynamic Behaviour Where $\sigma_{pg}\% = 20$ and $\sigma_{pc} = 22000$ or 32000:
(a) $\sigma_{pc} = 22000$, μ_{pg}=1e7 (0.5 secs) (b) $\sigma_{pc} = 22000$, μ_{pg}=8e7 (4 secs)
(c) $\sigma_{pc} = 22000$, μ_{pg}=1.5e8 (7.5 secs) (d) $\sigma_{pc} = 22000$, μ_{pg}=2.2e8 (11 secs)
(e) $\sigma_{pc} = 32000$, μ_{pg}=1e7 (0.5 secs) (f) $\sigma_{pc} = 32000$, μ_{pg}=8e7 (4 secs)
(g) $\sigma_{pc} = 32000$, μ_{pg}=1.5e8 (7.5 secs) (h) $\sigma_{pc} = 32000$, μ_{pa}=2.2e8 (11 secs)

Graphs (a)-(d) of Figure 7–5 illustrate the behaviour observed for relatively time-invariant programs i.e. where $\sigma_{pc} = 2000$. These behaviour patterns are very similar to those shown in Figure 7–3 for trials exhibiting global phase changes. This is to be expected. The small magnitude of the phase changes means that their exact timing does not have any great influence on the behaviour observed. The $\sigma_{pc} = 2000$ section of Table 7–5 confirms that the increased processor utilisation can be attributed to improved program performance, rather than migration overheads.

Graphs (e)-(h) of Figure 7–5 are concerned with trials where $\sigma_{pc} = 12000$. One can observe similar performance improvements in Graphs (f)-(h) to those seen in Figure 7–3 for trials where $\sigma_{pg\%} = 1$. However, the activity of the migration strategy in Figure 7–5 seems markedly different to that observed in Figure 7–3. The local nature of the phase changes has resulted in the migrations being spread throughout the entire simulation period, and the sharp peaks of activity noted earlier are no longer present. The number of migrations executed still remains relatively small. Table 7–5 shows that for Graphs (g)-(h) one can conclude at the 1% significance level that the migration strategy improves performance for both the U and M metrics.

A different picture emerges in Graph (e) of Figure 7–5, which relates to programs displaying short phase changes. As was the case for trials in Figure 7–3, short phase lengths result in unstable behaviour and increased numbers of migrations. In Table 7–5, it can be seen that when $\sigma_{pc} = 12000$ and $\mu_{pg} = 1e7$, the significance level at which it is possible to conclude that the metric M increases is 5%. This indicates that the actions of the migration strategy are beginning to result in non-negligible overheads.

Figure 7–6 presents activity graphs for programs displaying phase changes of increasing magnitude. It can be seen that the migration strategy provides similar performance improvements to those observed in Figure 7–4 for trials where $\sigma_{pg\%} = 1$. Due to the local nature of the phase changes, the actions of the migration

strategy are now spread across the entire simulation period. For Graphs (b)-(d) and Graphs (f)-(h) the migration overheads are tolerable due to the large performance gains achieved. This is confirmed by the relevant entries in Table 7–5. However, for Graphs (a) and (e) the migration overheads occupy a sizeable proportion of the increased processor utilisation. Indeed, by examining the entry in Table 7–5 where $\sigma_{pc} = 32000$ and $\mu_{pg} = 1e7$, it can be concluded at the 10% significance level that the value of M observed with the migration strategy active is not significantly different to that observed with the migration strategy inactive.

Conclusions

The experiment and subsequent analysis presented here have enabled us to develop a detailed understanding of the relationships which exist between a class of loosely synchronous time-invariant parallel programs, and the performance of a representative migration strategy. It has been demonstrated that the strategy is capable of significantly improving the performance of programs, as long as phase lengths are not too short. The actions of the strategy are generally stable, especially for programs exhibiting global phase changes. In such cases, phase changes are detected accurately and convergence to an improved placement is obtained rapidly, just as was observed in Chapter 6. For programs exhibiting more locally oriented phase changes, the actions of the migration strategy are spread more evenly across the entire simulation period. In this way the strategy is able to react to local fluctuations in behaviour in order to improve performance.

The results of paired t tests using the two performance metrics U and M have indicated that the overheads imposed by the strategy seem acceptable, as long as the phase changes do not occur too frequently. However, for short phase lengths, the actions of the migration strategy become unstable, thrashing tends to occur, and the overheads associated with the migrations become non-negligible. The notable exception to this is the case where $\sigma_c = 2000$. For these trials there is

not a great deal of imbalance present in the programs, so migrations are naturally inhibited.

It is interesting to note that this detailed examination of the behaviour of the migration strategy has provided conclusions that were not at all obvious from the preliminary analysis of variance table. For example, it is clear that programs with short phases lengths are not ideal candidates, yet the μ_{pg} and $s\mu_{pg}$ terms in Table 7–3 give little indication that these factors might play important roles in determining performance.

7.3 Validation

To confirm that the simulation results presented in this chapter are applicable to real programs, some validation experiments were carried out on a transputer-based machine using an example loosely synchronous time-varying program. The program used was a well known ocean evolution simulation called WATOR, first described in [37].

The problem domain of WATOR is a 2-dimensional ocean with wrap-around borders. The population of this ocean consists of sharks and fish whose behaviour is governed by a number of simple rules which are applied at each simulation step. During each cycle, a fish or shark may move to a randomly chosen adjacent location (North, South, East or West), provided that the location in question is not already occupied by a member of the same species. Fish exhibit simple behaviour patterns, tending to move to empty locations. Sharks exhibit more complex behaviour since they need to eat fish in order to survive, and accordingly, they tend to move to locations occupied by fish in preference to empty locations. Sharks and fish both give birth to a single offspring after a fixed number of simulation steps, defined by the two parameters *sbreed* and *fbreed* respectively. Finally, a shark will die if it fails to eat a fish for *starve* consecutive simulation cycles.

In order to begin a WATOR simulation five parameters must be initialised. Three of these, *sbreed*, *fbreed* and *starve* were introduced above. In addition, the parameters *nsharks* and *nfish* define the number of sharks and fish present at the beginning of the simulation. It is assumed that these are distributed randomly and uniformly across the ocean.

A simulation obeying the rules outlined above would not be particularly close to any real population process. However, the WATOR program is interesting because, although simple, it exhibits many of the characteristics of more complex dynamic algorithms typically implemented on parallel machines. For example, fish and sharks tend to aggregate quickly into schools, regardless of the initial distribution. This suggests load balancing problems, since for any particular simulation cycle, the shark and fish population is unlikely to be uniformly spread across the ocean. In addition, the behaviour of WATOR cannot be predicted in advance; in order to know the population distribution after n simulation cycles, it is necessary to execute all $n - 1$ previous cycles. This indicates that static load balancing techniques are unlikely to be suitable.

For the reasons outlined above, it is clear that WATOR can be classified as a non-uniform time-varying program. It is therefore not surprising that it has been used a number of times as a testbed program for investigations into dynamic load balancing techniques; see [51,59,132] for example.

7.3.1 Implementation

First of all the ocean must be divided into a number of equally sized sub-domains, each of which is allocated to a process. A standard data parallel approach is adopted, with processes being cyclic in nature, computing their own internal data movements, before exchanging information with their neighbours in order to update boundary locations.

Using this technique in a naive manner, it is possible for data items to collide with one another at domain borders, i.e. exist in the same location at the same point in time. This breaks one of the rules of the WATOR simulation as originally stated. Fox [51] describes a technique based on *rollback* to prevent such collisions. This technique involves keeping a record of every data movement made so that it is always possible to rewind data items to a previous step if a collision occurs. However, in undoing one collision, another might be caused, so multiple rollbacks are possible. Indeed, there is no upper bound on the number of rollbacks that might need to be performed.

To overcome this problem, Hanxleden and Ridgway-Scott advocate a slight modification of the WATOR algorithm so that it is possible for two data items to temporarily share the same ocean location [60]. However, the general trend for data items to repel one another is maintained (except for sharks when they find a fish). The characteristics of this modified simulation were demonstrated to be almost identical to the original WATOR simulation. It is this version of the WATOR simulation that is implemented here.

One other point of note is that, in order to maintain determinism under the migration strategy, each process has its own random number generator.

To enable phases of differing lengths to be created artificially, an extra parameter, *nreps*, was added to the simulation. This parameter specifies the number of times that each process should repeat a cycle (where each repetition corresponds to a complete iteration, including communications). So, when $nreps = 1$ the simulation behaves as normal. As the value of *nreps* is increased, the lengths of phases increase, because the simulation spends time repeating the same computation over and over.

nrep	Migrations Inactive	Migrations Active
1	187.55	218.72
10	188.28	171.29
20	183.85	153.32
30	181.51	140.21

Table 7–6: Execution Times Observed for WATOR with Repeated Cycles

7.3.2 Results

An 80 × 80 ocean was assumed, and this was partitioned among 100 processes. The simulation was executed on a 4 × 4 mesh of transputers and the migration strategy parameters were set to the optimal values derived in Chapter 6. The WATOR parameters were set as follows: $nfish = 320$, $nsharks = 60$, $fbreed = 10$, $sbreed = 6$ and $starve = 5$.

The results obtained are presented in Table 7–6, the figures refer to execution times in seconds. In each case, the program was run for as many iterations as were necessary to give an execution time of between 180 and 190 seconds for the case where the migration strategy was inactive. One can see that when $nreps = 1$, the migration strategy actually degrades performance. This is a more extreme effect than was seen in the simulation experiments. As the value of *nreps* is increased, however, the migration strategy begins to improve performance. The magnitude of the improvement (in percentage terms) seems proportional to the value of *nreps*, which is, in turn, proportional to the phase length. These results agree with the simulation results observed earlier, i.e. the likely performance improvements of a migration strategy increase as the phase length increases. Accordingly, it appears that we can have a reasonable degree of confidence in the simulation results.

Figure 7–7 illustrates the dynamic behaviour of the machine and the migration strategy for different values of *nreps*, graph (a) refers to the program with the shortest phase lengths, and graph (d) refers to the program with the longest

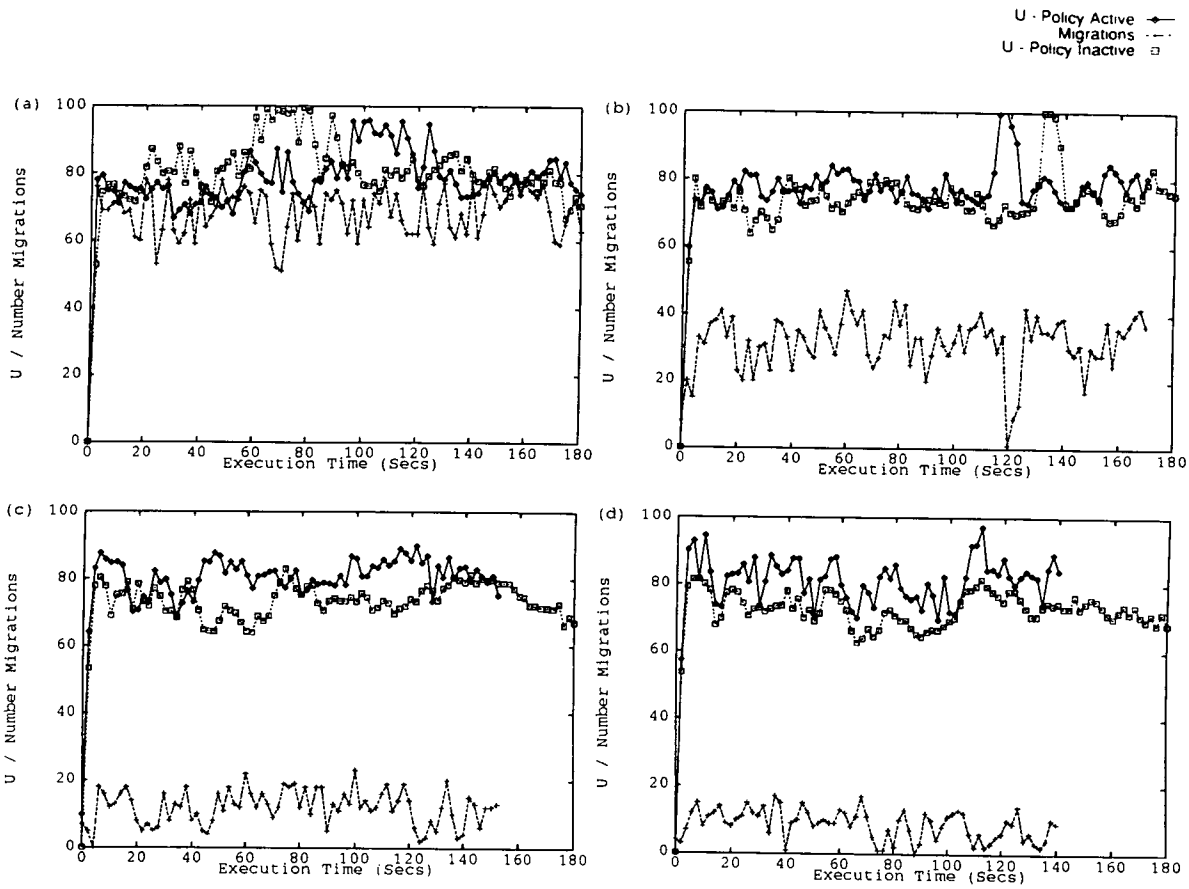

Figure 7–7: Dynamic Behaviour of WATOR with Repeated Cycles

(a) 1 Repetition (b) 10 Repetitions

(c) 20 Repetitions (d) 30 Repetitions

phase lengths. The features observed are very similar to those seen in Figures 7–5 and 7–6, which contained graphs relating to simulation trials exhibiting locally oriented phase changes.

7.4 Summary and Conclusions

This chapter has demonstrated how time-varying patterns of activity can be adequately characterised and generated using a relatively simple program model. This model has been used to examine the behaviour of a migration strategy when presented with programs displaying time-varying behaviour. We have been able to gain an understanding of the conditions under which the migration strategy performs well, and those under which it does not. Consequently, given an arbitrary program with known program parameters, it is possible to make statements regarding the suitability of this particular migration strategy. In addition, approximate estimates of the improvement in performance to expect can be made.

The results of some validation experiments using a real program running on a real transputer-based machine were presented. These results broadly agree with those obtained using simulation techniques.

Chapter 8

Summary and Conclusions

This thesis has demonstrated how the standard methods of experimental design and analysis can be applied to performance analysis studies of parallel programs. In particular, the techniques were shown to be appropriate for a study of the performance characteristics of a class of process migration strategies.

Chapter 3 discussed the tools, techniques and environments that were utilised in order to carry out this work. A methodology was described enabling the construction of synthetic parallel programs. An experimental framework and a simulation system called MIMD were then presented. These tools were designed specifically to specify, construct and model synthetically generated programs. The environment was tailored to simulate occam programs and transputers. Two process migration mechanisms were described, one for MIMD, and one for real occam programs running on transputer-based machines. Finally, the main statistical techniques used in this thesis were discussed.

Chapter 4 illustrated how simple two level full factorial experiments could be used, in conjunction with analysis of variance techniques, in order to undertake quantitative performance analysis experiments. A classification scheme and program model for loosely synchronous parallel programs were defined. It was decided to concentrate on a particularly simple class of regular programs in this chapter

in order to illustrate the methodology. A program parameter set designed to characterise the time-averaged properties of this class of programs was proposed.

Using an analysis of variance, it was shown that programs displaying similar macroscopic parameters (i.e. having the same program parameters), but differing in their microscopic details (i.e. having different synchronisation properties due to different program graphs and random number seeds), displayed very similar performance characteristics. This was illustrated for a number of different processor topologies, and for two different performance metrics: the mean computation achieved by the processes, and the mean utilisation of the underlying machine. Various transformations of the response were shown to be necessary in order to satisfy the assumptions underlying an analysis of variance. I concluded that the program parameter set chosen adequately represented performance. In addition, it was demonstrated that quantitative estimates of the relative importance of program parameters could be determined. These estimates seemed consistent and meaningful, bearing in mind what is known about the program model and machine that were simulated.

Chapter 5 investigated the construction of performance prediction models. A more realistic class of programs was assumed, in which processes and channels could display different behaviour patterns. However, these patterns were constrained to remain relatively constant over time. As in Chapter 4, an analysis of variance revealed the program parameters with the greatest impact on performance with respect to U, the mean utilisation of the underlying processors. An exploratory experiment was carried out varying just these parameters, and the method of orthogonal polynomials was then used to fit a response surface to this data. Given an arbitrary program with parameters chosen within the range explored, 75% of predicted values were found to be within 6.8% of the corresponding observed values of U.

The analysis of covariance was then used in an attempt to characterise the relationships existing between the predictor variables and U with greater preci-

sion. This technique is generally used in order to take account of factors which are not under the experimenter's immediate control. In this case, these factors explained the differences in performance observed between programs with identical program parameters. Consequently, a number of covariates characterising the interaction patterns occurring between the program and the underlying machine were explored. Two useful covariates were identified, and it was shown that the assumptions underlying an analysis of covariance were satisfied. The first covariate *sync_mean* can be thought of answering the following question:

> For how long, on average, does a sender (or receiver) have to wait for the remote receiver (or sender) to become ready?

The second metric *ce_ratio_sd* can be thought of as quantifying an answer to the question:

> To what extent are all processes computing at the same rate?

A model constructed using these covariates, as well as the more conventional program parameter terms, resulted in 75% of predicted values of U being within 4.1% of the corresponding observed values of U, given an arbitrary program with parameters chosen within the range explored.

I concluded that the covariates *sync_mean* and *ce_ratio_sd* contained useful information relating to the interactions which occur between a program and the machine it is being executed on, at least for the program model investigated here. The performance prediction equations revealed that lower values of the two covariates were associated with improved performance.

Chapter 6 investigated the performance of a class of process migration strategies when applied to the problem of improving a non-optimal initial mapping. A representative strategy was described, and it was shown that this strategy exhibited good stability and convergence properties for a variety of different workloads. Estimates of the likely performance benefits obtainable were presented. An improved

migration strategy that attempted to minimise dynamically the values of the covariates *sync_mean* and *ce_ratio_sd* was then derived. This strategy was shown to perform as well as the simple strategy, and out-perform it in certain circumstances (typically for smaller numbers of processes and unbalanced workloads). Some validation results were presented using real occam programs running on a transputer-based machine. I concluded that it seems possible to use local actions to reduce the values of *sync_mean* and *ce_ratio_sd* globally at run-time, and so improve performance.

Chapter 7 investigated the performance of the improved strategy when presented with programs which varied over time. A set of program parameters suitable for characterising such behaviour patterns was described. An exploratory experiment was carried out, and the actions of the strategy were shown to lead to improved performance as long as phase lengths were not too short. For short phase lengths thrashing was observed to occur, and the performance of the strategy deteriorated. Some validation work was presented using a real program that conformed to the program model simulated. The experiments described in this chapter allowed us to pinpoint the situations when the strategy investigated was able to improve performance; and they also allowed quantitative estimates to be made of the likely performance benefits to expect.

The work presented here has demonstrated that the standard methods of experimental design and analysis can be used to increase our understanding of the performance characteristics of parallel programs. This area had not been addressed to any great extent previously. I have shown that it is feasible to search for simplified program models that are characterised in terms of a small number of parameters representing time-averaged properties. Quantitative performance prediction models can be derived in terms of these program parameters, using designed experiments and synthetic programs. Such models could be used to aid program configuration decisions. For example, the initial performance prediction equations developed in Chapter 5 indicate the factors that one should concentrate

on when trying to fine-tune the performance of a program obeying the particular program model under consideration. Given a choice between reducing synchronisation overheads and reducing message lengths, one should concentrate on the former, since it is known that message lengths do not have a great impact on performance in this case. It is also possible to obtain quantitative estimates of the likely performance benefits available by carrying out these optimisations.

It has also been demonstrated how these models can be extended to take account of environmental factors, using the analysis of covariance. This allows the impact of different program mappings and communication patterns to be explored, thereby providing considerable insight into the dynamics of program behaviour. As an example of a practical application of these techniques, we have seen how information derived from an analysis of covariance can be used to improve the performance of a representative process migration strategy.

The results obtained in Chapters 6 and 7 demonstrated how the performance of programs running under a particular run-time environment (typified by a process migration strategy) can be estimated quantitatively. Such techniques allow us to better understand the circumstances under which a migration strategy can improve performance. Models of the type developed here could possibly be incorporated into an operating system. Then, given a program with known parameter values, the operating system would be able to decide whether it was worth applying a given process migration strategy.

8.1 Future Work

There are a number of directions in which future work could proceed. It would be interesting to extend these investigations to alternative program models; for example, divide and conquer or task farm models. It is not clear how well other classes of programs could be characterised in terms of time-averaged parameters,

or what those parameters would be. Since some programs are more naturally represented in terms of precedence graphs rather than process graphs, techniques for characterising the time-averaged behaviour of precedence graphs need to be explored.

Even retaining the program model assumed in this thesis, there are a number of avenues to explore. I have only considered a single machine, and generally speaking, have held constant the processor topology. It would be interesting to investigate whether the details of the particular machine and topology used could be adequately incorporated into performance prediction models. The results presented in Chapters 5-7 relating to covariates and process migration were derived exclusively using mesh based topologies. It would be informative to study how easily these results could be extended to alternative machines and topologies. Also, although I have illustrated the plausibility of my results using real programs, it would be desirable to undertake a more extensive validation study.

Another area that requires further work is the problem of deciding whether a given program belongs to a particular program class. It needs to be determined to what extent this information can be derived automatically by analysing the source code and/or executing the program.

The particular covariates identified in Chapter 5 are, to a certain extent, tied to the program model used. However, the information that they provide, i.e. that synchronisation overheads should be minimised and computation rates equalised, is general in nature and applies to all programs. Alternative program models should be explored, and suitable covariates derived. It would then be interesting to see if the essential essence of these covariates could be extracted and expressed in general terms. Ideally, one would like to be able to define covariates that were universally applicable.

This thesis has concentrated on optimising the values of covariates dynamically at run-time in order to improve performance. An alternative approach would

be to attempt this in a post-mortem manner. This idea is related to the post-game analysis adaptive mapping strategies described in [67,125]. These strategies execute a program collecting monitoring information, and then use heuristics in an attempt to propose an improved mapping. The process is then repeated, the idea being that one will eventually converge to an acceptable mapping. It would be an interesting application of the analysis of covariance to provide a formal framework in which to develop suitable heuristics. For example, for the class of programs investigated in this thesis, the heuristics should propose alterations to the mapping such that the values of the covariates *sync_mean* and *ce_ratio_sd* were minimised. Note that, unlike the situation found when constructing a distributed migration strategy, global knowledge could be used in this case.

With regard to process migration strategies, there is a large volume of work remaining to be done before their use can be adopted on a widespread scale. For example, instead of using simplistic mapping strategies, it would be informative to study the effect of using more sophisticated mapping heuristics. A greater understanding needs to be obtained of the precise conditions under which particular process migration strategies can improve performance, for a wide variety of different program models. It would be only at this stage that one could consider incorporating such strategies into a general purpose multiprocessor operating system. To support this, there is a requirement to develop general purpose performance prediction models capable of estimating the performance benefits to expect, given an arbitrary program with known program parameters, and a particular process migration strategy.

The work described in this thesis has illustrated the application of the standard methods of experimental design and analysis to the performance analysis of parallel programs. This is just a first step, and there is a requirement for further systematic studies in order to establish the extent to which the approach proposed here is applicable to other types of programs and machines. In the longer term, it would be desirable to be able to construct operating systems that, given an

arbitrary program, could determine what class the program belonged to, and consequently make suitable mapping and run-time decisions in order to execute the program as efficiently as possible.

Bibliography

[1] R. Arlauskas. iPSC/2 system: A second generation hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 38–42, 1988.

[2] Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, 22(9):47–56, 1989.

[3] M. Ashraf-Iqbal, J. Saltz, and S. Bokhari. A comparative analysis of static and dynamic load balancing strategies. In *Proceedings of International Conference on Parallel Processing*, pages 1040–1047, 1986.

[4] A. Atkinson. *Plots, Transformations and Regression.* Oxford Statistical Science Series 1. Oxford University Press, 1987.

[5] W. Bain and S. Shala. HYPERSIM: A hypercube simulator for parallel systems performance modelling. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 792–800, 1988.

[6] S. Baker and K. Milner. A process migration harness for dynamic load balancing. In *Proceedings of World Transputer and Occam User Group Technical Meeting 14: Occam and the Transputer - Current Developments*, pages 52–61. IOS Press, 1991.

[7] G. Barnes, R. Brown M. Kato, et al. The ILLIAC IV computer. *IEEE Transactions on Computers*, 17(8):746–757, 1968.

[8] T. Bemmerl, A. Bode, O. Hansen, and T. Ludwig. A testbed for dynamic load balancing on distributed memory multiprocessors. ESPRIT Project 2701, PUMA Working Paper Nr. 14, WP 4.5, T.U. Munchen, 1990.

[9] T. Bemmerl, P. Hofstetter, and T. Ludwig. LOTOP - A load generator for parallel computers. In *Hypercube and Distributed Computers : Proceedings of the First European Workshop on Hypercube and Distributed Computers*, pages 83–91. Elsevier Science, 1989.

[10] R. Berry and J. Hellerstein. Characterizing and interpreting periodic behavior in computer systems. Research report, IBM T.J. Watson Research Center, 1992.

[11] G. Birtwistle. *Discrete Event Modelling on Simula.* Macmillan, 1986.

[12] J. Boillat, F. Bruge, and P. Kropf. A dynamic load-balancing algorithm for molecular dynamics simulation on multiprocessor systems. *Journal of Computational Physics*, 26(1):1–14, 1991.

[13] J. Boillat, N. Iselin, and P. Kropf. MARC: A tool for automatic configuration of parallel programs. In *Transputing '91*, pages 311–329. IOS Press, 1991.

[14] J. Boillat and P. Kropf. A fast distributed mapping algorithm. In *Proceedings CONPAR 90*, pages 405–416, 1990.

[15] S. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, 30(3):207–214, 1981.

[16] S. Bollinger and S. Midkiff. Heuristic techniques for processor and link assignment in multicomputers. *IEEE Transactions on Computers*, 40(3):325–333, 1991.

[17] G. Box and D. Cox. An analysis of transformations. *Journal Royal Statistical Society*, B 26:211–246, 1964.

[18] G. Box, W. Hunter, and J. Hunter. *Statistics for Experimenters.* Series in Probability and Mathematical Statistics. John Wiley and Sons, 1978.

[19] L. Brochard, J. Prost, and F. Faurie. Synchronization and load unbalance effects of parallel iterative algorithms. In *Proceedings of International Conference on Parallel Processing*, pages 153–160, 1989.

[20] R. Candlin, P. Fisk, and N. Skilling. A statistical approach to finding performance models of parallel programs. In *Proceedings Seventh UK Computer and Telecommunications Workshop*, Springer-Verlag Workshop Series. University of Edinburgh, 1991.

[21] R. Candlin, T. Guilfoy, and N. Skilling. A modelling system for process-based programs. In *Proceedings of the European Simulation Congress*, 1989.

[22] R. Candlin and J. Phillips. An environment for investigating the effectiveness of process migration strategies on transputer-based machines. In *Proceedings of* 15^{th} *World Occam and Transputer User Group Meeting: Transputer Systems - Ongoing Research*, pages 13–23. IOS Press, 1992.

[23] R. Candlin and L. Qiang-Yi. Communications performance in occam programs on the MEiKO Computing Surface. Technical Report CSR-6-90, Dept. Computer Science, University of Edinburgh, 1990.

[24] W. Chen and E. Gehringer. A graph-oriented mapping strategy for a hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 200–209, 1988.

[25] R. Chowkwanyun and K. Hwang. Multicomputer load balancing for concurrent lisp execution. In *Parallel Processing for Supercomputers and Artificial Intelligence*, pages 325–366. McGraw-Hill, 1989.

[26] A. Ciampolini, A. Corradi, and L. Leonardi. How to manage object migration in massively parallel architectures. In *Parallel Computing: From Theory to Sound Practice - Proceedings of the European Workshop on Parallel Computing*, pages 312–315. IOS Press, 1992.

[27] L. Clarke. Tiny 1.0: Discussion and user guide. Technical Report UG-9, Edinburgh Concurrent Supercomputer Project, University of Edinburgh, 1989.

[28] L. Clarke and G. Wilson. Tiny: An efficient routing harness for the Inmos transputer. Technical report, Edinburgh Concurrent Supercomputer Project, University of Edinburgh, 1989.

[29] W. Cochran. Analysis of covariance: Its nature and uses. *Biometrics*, 13(1-4):261–281, 1957.

[30] W. Cochran and G. Cox. *Experimental Designs.* Second Edition. John Wiley and Sons, 1957.

[31] GENSTAT 5 Committee. *GENSTAT 5 Reference Manual.* Clarendon Press, 1988.

[32] A. Corradi, L. Leonardi, and F. Zambonelli. Load balancing strategies for massively parallel architectures. *Parallel Processing Letters*, 2(2-3):139–148, 1992.

[33] W. Crowther, J. Goodhue, E. Starr, et al. Performance measurements on a 128-node butterfly parallel processor. In *Proceedings International Conference on Parallel Processing*, pages 531–540, 1985.

[34] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, 1989.

[35] H. Davis, S. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using TANGO. In *Proceedings International Conference on Parallel Processing*, pages 99–107, 1991.

[36] A. Deshpande and M. Schultz. Efficient parallel programming with Linda. In *Proceedings Supercomputing 1992*, pages 238–244, 1992.

[37] A. Dewdney. Computer recreations. *Scientific American*, December 1984.

[38] F. Douglis and J. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience*, 21(8):757–785, 1991.

[39] D. Downham and F. Roberts. Multiplicative congruential pseudo-random number generators. *Computer Journal*, 10(1):74–77, 1967.

[40] K. Dragon and J. Gustafson. A low-cost hypercube load-balance algorithm. In *Proceedings Fifth Distributed Memory Computing Conference*, pages 583–589, 1990.

[41] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, 1986.

[42] D. Eager, E. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated load sharing. *Performance Evaluation*, 6(1):53–68, 1986.

[43] J. Kowalik (Ed). *Parallel MIMD Computation: HEP Supercomputer and its Applications.* MIT Press, 1985.

[44] H. El-Rewini and T. Lewis. Scheduling parallel programs onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9(2):138–153, 1990.

[45] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 210–221, 1988.

[46] D. Fernandez-Baca. Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, 1989.

[47] J. Flower, S. Otto, and M. Salama. A preprocessor for irregular finite element problems. Technical Report C^3P-292B, California Institute of Technology, 1987.

[48] M. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21:948–960, 1972.

[49] I. Foster and S. Taylor. *Strand : New Concepts in Parallel Programming.* Prentice-Hall, 1990.

[50] G. Fox. A review of automatic load balancing and decomposition methods for the hypercube. Technical Report C^3P-385, California Institute of Technology, 1986.

[51] G. Fox et al. *Solving Problems on Concurrent Processors, Volume 1.* Prentice Hall International, 1988.

[52] G. Fox and W. Furmanski. Load balancing loosely synchronous problems with a neural network. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 241–278, 1988.

[53] D. Grunwald, B. Nazief, and D. Reed. Empirical comparison of heuristic load distribution in point-to-point multicomputer networks. In *Proceedings Fifth Distributed Memory Computing Conference*, pages 984–993, 1990.

[54] V. Guerrero and R. Johnson. Use of the box-cox transformation with binary response models. *Biometrika*, 69:309–314, 1982.

[55] T. Guilfoy. A modelling system for POSIE. Master's thesis, Dept. Computer Science, University of Edinburgh, 1988.

[56] S. Gulati, J. Barhen, and S. Iyengar. The pebble crunching model for load balancing in concurrent hypercube ensembles. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 279–284, 1988.

[57] A. Hac and X. Jin. Dynamic load balancing in a distributed system using a decentralized algorithm. In *Proceedings 7th International Conference on Distributed Computing Systems*, pages 170–177, 1987.

[58] A. Hac and T. Johnson. A study of dynamic load balancing in a distributed system. In *Proceedings ACM SIGCOMM Symposium on Communications, Architectures and Protocols*, pages 348–356, 1986.

[59] R. Hanxleden and L. Ridgway-Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13(3):312–324, 1991.

[60] R. Hanxleden and L. Ridgway-Scott. Correctness and determinism of parallel monte carlo processes. *Parallel Computing*, 18:121–132, 1992.

[61] P. Hayes and A. Andrews. Multiprocessor performance modelling with ADAS. In *AIAA Computers in Aerospace V11 Conference*, pages 335–343, 1989.

[62] M. Heath and J. Etheridge. Visualizing the the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[63] C. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, 1985.

[64] R. Hockney and C. Jesshope. *Parallel Computers.* Adam Hilger Ltd., 1981.

[65] S. Hosseini, B. Litow, M. Malkawi, et al. Analysis of a graph coloring based distributed load balancing algorithm. *Journal of Parallel and Distributed Computing*, 10(2):160–166, 1990.

[66] B. Huitema. *The Analysis of Covariance and Alternatives.* John Wiley and Sons, 1980.

[67] A. Ieumwananonthachai, A. Aizawa, et al. Intelligent mapping of communicating processes in distributed computing systems. In *Proceedings Supercomputing 1991*, pages 512–521, 1991.

[68] F. Anger J. Hwang, Y. Chow et al. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal Computing*, 18(2):244–257, 1989.

[69] R. Jain. *The Art Of Computer Systems Performance Analysis.* John Wiley & Sons, 1991.

[70] M. Jerrum and A. Sinclair. Fast uniform generation of random graphs. Technical Report CSR-281-88, Dept. Computer Science, University of Edinburgh, 1988.

[71] G. Jones. Measuring the busyness of a transputer. *Occam User Group Newsletter*, 12:57–64, 1990.

[72] P. Jones and C. Hojung. On the feasibility of run-time process migration in multi-transputer machines. In *Proceedings of Occam User Group Technical Meeting 13: Real-Time Systems with Transputers.* IOS Press, 1990.

[73] W. Joosen, Y. Berbers, et al. Transparent object migration in adaptive parallel applications. In *Parallel Computing: From Theory to Sound Practice - Proceedings of the European Workshop on Parallel Computing*, pages 300–311. IOS Press, 1992.

[74] E. Jul, H. Levy, et al. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[75] P. King. *Computer and Communication Systems Performance Modelling.* Prentice-Hall International, 1990.

[76] D. Kleinbaum, L. Kupper, and K. Muller. *Applied Regression Analysis and Other Multivariable Methods.* PWS-Kent Publishing Company, 1988.

[77] E. Koenders. A measurement system for post-game analysis on transputers. Report EL/BSC90R160, University of Twente, 1990.

[78] J. Koller. A dynamic load balancer on the Intel hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 279–284, 1988.

[79] J. Koller. The MOOS II operating system and dynamic load balancing. In *Proceedings Fifth Distributed Memory Computing Conference*, pages 599–602, 1990.

[80] P. Krueger and M. Livny. A comparison of preemptive and non-preemptive load distributing. In *Proceedings IEEE International Conference on Distributed Computing Systems*, pages 123–130, 1988.

[81] P. Lane, N. Galwey, and N. Alvey. *GENSTAT 5 An Introduction.* Clarendon Press, 1987.

[82] S. Lee and J. Aggarwal. A mapping strategy for parallel processing. *IEEE Transactions on Computers*, 36(4):433–441, 1987.

[83] F. Lin and R. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38, 1987.

[84] V. Lo. Heuristic algorithms for task assignment in distributed systems. *IEEE Transactions on Computers*, 37(11):1384–1397, 1988.

[85] V. Lo. Temporal communication graphs: Lamport's process-time graphs augmented for the purpose of mapping and scheduling. *Journal of Parallel and Distributed Computing*, 16:378–384, 1992.

[86] Inmos Ltd. *Occam 2 Reference Manual.* Prentice Hall International, 1988.

[87] Meiko Ltd. *Computing Surface Reference Manual.* Meiko Ltd, Bristol, UK, 1989.

[88] Meiko UK Ltd. *CS (Communicating Sequential) Tools for SunOS.* Meiko Ltd, Bristol, UK, 1990.

[89] G. Lyon, R. Snelick, and R. Kacker. Time-perturbation tuning of mimd programs. In *Computer Performance and Engineering 1992 - Proceedings of Sixth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 211–224, 1992.

[90] G. Macharia. CLD: A novel approach to dynamic load balancing. *Microprocessing and Microprogramming*, 28(1-5):43–47, 1990.

[91] D. Marinescu, J. Lumpp, T. Casavant, et al. Models for monitoring and debugging tools for parallel and distributed software. *Journal of Parallel and Distributed Computing*, 9(2):171–184, 1990.

[92] D. Marinescu and J. Rice. Synchronization and load imbalance effects in distributed memory multiprocessor systems. *Concurrency: Practice and Experience*, 3(6):593–625, 1991.

[93] D. Montgomery. *Design and Analysis of Experiments.* John Wiley and Sons, 1991.

[94] H. Muhlenbein, M. Gorges-Schleuter, and O. Kramer. New solutions to the mapping problem of parallel systems: The evolution approach. *Parallel Computing*, 4(3):269–279, 1987.

[95] S. Mullender. Process management in a distributed operating system. Technical Report CS-R8713, Amsterdam Centre for Mathematics and Computer Science, 1987.

[96] T. Murata, B. Shenker, and S. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Transactions on Software Engineering*, 15(3):314–326, 1989.

[97] R. Nance, R. Moose, and R. Foutz. A statistical technique for comparing heuristics: An example from capacity assignment strategies in computer network design. *Communications of the ACM*, 30(5):430–442, 1987.

[98] B. Nazief. *An Empirical Study of Load Distribution Strategies on Multicomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1991.

[99] *NCUBE-2 6400 Series Supercomputer Technical Overview.* 1989.

[100] L. Ni, C. Xu, and T. Gendreau. A distributed drafting algorithm for load balancing. *IEEE Transactions on Software Engineering*, 11(10):1153–1161, 1985.

[101] D. Nicol and P. Reynolds. Optimal dynamic remapping of data parallel computations. *IEEE Transactions on Computers*, 39(9):206–219, 1990.

[102] D. Nicol and J. Saltz. An analysis of scatter decomposition. *IEEE Transactions on Computers*, 39(11):1337–1345, 1990.

[103] N.Mansour and G. Fox. Allocating data to multicomputer nodes by physical optimization algorithms for loosely synchronous computations. *Concurrency: Practice and Experience*, 4(7):557–574, 1992.

[104] E. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs.* InterScience Series. John Wiley & Sons, 1985.

[105] C. Papadimitriou and M. Yannakakis. Towards an architecture independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19(2):322–328, 1990.

[106] S. Park and K. Miller. Random number generators : Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[107] J. Phillips and N. Skilling. A modelling environment for studying the performance of parallel programs. In *Proceedings Seventh UK Computer and Telecommunications Workshop*, Springer-Verlag Workshop Series. University of Edinburgh, 1991.

[108] R. Pooley. *An Introduction to Programming in SIMULA.* Blackwell Scientific Publications, 1987.

[109] D. Poplawski. Synthetic models of distributed memory parallel programs. *Journal of Parallel and Distributed Computing*, 12(4):423–426, 1991.

[110] M. Powell and B. Miller. Process migration in DEMOS/MP. *ACM Operating Systems Review*, 17:110–118, 1983.

[111] X. Qian and Q. Yang. Load balancing on generalized hypercube and mesh multiprocessors with LAL. In *IEEE International Conference on Distributed Computing Systems*, pages 402–409, 1991.

[112] L. Qiang-Yi. An improved virtual channel mechanism for occam programs on the Meiko Computing Surface. Technical Report CSR-6-90, Dept. Computer Science, University of Edinburgh, 1990.

[113] R. Rashid and G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings Eighth ACM Symposium on Operating System Principles in ACM SIGOPS Review*, pages 64–75, 1981.

[114] V. Saletore. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium grain tasks. In *Proceedings Fifth Distributed Memory Computing Conference*, pages 994–999, 1990.

[115] H. Scheffe. *The Analysis of Variance.* John Wiley and Sons, 1959.

[116] S. Shatz and W. Cheng. A Petri net framework for automated static analysis of Ada tasking. *The Journal of Systems and Software*, 8(5):343–359, 1988.

[117] B. Shirazi, M. Wang, and G. Pathak. Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributed Computing*, 10:222–232, 1990.

[118] N. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, 1992.

[119] W. Shu and L. Kale. Chare Kernel - A runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11(3):198–211, 1991.

[120] N. Skilling. MIMD. Internal report, Department of Chemical Engineering, University of Edinburgh, 1990.

[121] J. Stankovic. Stability and distributed scheduling algorithms. *IEEE Transactions on Software Engineering*, 11(10):1141–1152, 1985.

[122] J. Stankovic and I. Sidhu. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Proceedings of the Fourth International Conference on Distributed Systems*, pages 49–59, 1984.

[123] S. Stepney. GRAIL: Graphical representation of activity, interconnection and loading. In *Proceedings of World Transputer and Occam User Group Technical Meeting 7: Parallel Programming of Transputer Based Machines*, pages 272–280. IOS Press, 1987.

[124] H. Stone. Multiprocessor scheduling with the aid of network flow diagrams. *IEEE Transactions on Software Engineering*, 3(1):85–93, 1977.

[125] J. Sunter, E. Koenders, and A. Bakkers. Post-game analysis on transputers. In *Transputing '91*, pages 330–341. IOS Press, 1991.

[126] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the V-System. In *Proceedings Tenth ACM Symposium on Operating System Principles in ACM SIGOPS Review*, pages 2–12, 1985.

[127] A. Thomasian and P. Bay. Analytic queuing network models for parallel processing of task systems. *IEEE Transactions on Computers*, 35(12):1045–1054, 1986.

[128] D. Towsley. Allocating programs containing branches and loops within a multiple processor system. *IEEE Transactions on Software Engineering*, 12(10):1018–1024, 1986.

[129] L. Tucker and G. Robertson. Architecture and applications of the connection machine. *Computer*, 21(8):26–38, 1988.

[130] J. Ullman. NP-complete scheduling problems. *Journal of Computer and Systems Sciences*, 10:384–393, 1975.

[131] M. Willebeek-LeMair and A. Reeves. Distributed dynamic load balancing. In *Proceedings Fifth Distributed Memory Computing Conference*, pages 609–612, 1990.

[132] W. Williams. Load balancing and hypercubes: A preliminary look. In *Proceedings of the Second Conference on Hypercube Concurrent Computers and Applications*, pages 108–113, 1987.

[133] W. Wulf and C. Bell. C.mpp - a multi-mini-processor. In *Proceedings AFIPS Fall Joint Computer Conference*, pages 765–777, 1972.

[134] J. Xu and K. Hwang. Heuristic methods for dynamic load balancing in a message-passing supercomputer. In *Proceedings Supercomputing 90*, pages 888–897, 1990.

[135] E. Zayas. Attacking the process migration bottleneck. In *Proceedings Eleventh ACM Symposium on Operating System Principles in ACM SIGOPS Review*, pages 13–24, 1987.

[136] X. Zhang, X. Qin, and B. Bian. Design of an instrumentation monitor for visuializing interconnection network multiprocessor performance. In *Parallel Computing: From Theory to Sound Practice - Proceedings of the European Workshop on Parallel Computing*, pages 92–95. IOS Press, 1992.

[137] S. Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, 1988.

[138] M. Zitterbart. Monitoring and debugging transputer networks with NETMON-II. In *Proceedings CONPAR 90*, pages 200–209, 1990.