A Statistical Approach to Performance Evaluation of Parallel Systems with Reference to Chemical Engineering

Neil Skilling

Ph.D.

University of Edinburgh

1995

Abstract

Distributed memory multicomputers appear to offer a cost effective general purpose parallel computing resource. Unfortunately these multicomputers have not always delivered the processing performance promised from a summation of individual processor speeds. A lot of time and effort can be expended trying to close this performance gap.

Detailed dynamic simulations of chemical processing equipment can be naturally and robustly modelled as a set of communicating sequential processes where the information flow accurately mirrors the material flow in the real equipment. These programs have a static, time invariant process graph which is suited to execution on a distributed memory MIMD machine.

There are many factors that affect the performance of a parallel program. The programmer, usually with the aid of profiling tools, is faced with a trial and error tuning up process. This thesis addresses the issue of performance evaluation of parallel systems by presenting a methodology that enables rapid identification of performance limiting factors. In particular the study of static placement strategies as performance factors can be readily investigated for a range of programs.

Through the use of standard statistical design of experiments, synthetic program graphs and a general purpose multiprocessor simulation system, placement strategies and other performance factors can easily be identified and their precise effect quantified. Through statistical analysis predictive performance models can also be constructed.

The approach presented is general and can be applied to an arbitrary parallel program. Results are presented for a common class of parallel programs called *structured spatial decomposition* and for process systems simulations.

Acknowledgments

I would like to thanks my two supervisors Rosemary Candlin and Jack Ponton for their guidance, particularly to Rosemary for her enthusiasm and encouragement.

Thanks also go to all the sensitive people around me who mostly knew when and when not to ask – "How is the thesis coming along?". Thanks to Jennifer for being extra patient and for agreeing to marry me. Mother, it may be time to get your hat on !

I would like to thank Jane Hillston for giving me the idea of an experiment generator and even a language to describe the experiments. Also Joe Phillips who was brave enough to try my software and not be too critical when it crashed. Peter Fisk for his invaluable help and advice with the statistics. All the people I have met through the Edinburgh Parallel Computing Centre who have inspired and argued with me and latterly tried to kick me around the football field. To all the people in Chemical Engineering who have been good friends.

Paul Hewson – I couldn't have done it without you.

Statement

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.

Neil Skilling

Publications

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

- Rosemary Candlin, Qiangyi Luo, and Neil Skilling. The investigation of communication patterns in occam programs. In John Wexler, editor, *Developing Transputer Applications*, Proceedings of the 11th Occam User Group Technical Meeting, pages 99–108. IOS Press, ISBN: 90 5199 020 0, September 1989.

- Rosemary Candlin, Thomas Guilfoy, and Neil Skilling. A modelling system for process-based programs. In *Proceedings of the 3rd European Simulation Congress*, pages 251–255. Edinburgh, September 1989.

- Rosemary Candlin, Thomas Guilfoy, and Neil Skilling. A modelling system for process based programs. *Fifth UK Computer and Telecommunications Performance Engineering Workshop*, June 1990.

- Rosemary Candlin and Neil Skilling. A modelling system for the investigation of parallel program performance. *Fifth International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, ISBN 0-444-88989-2*, February 1991.

- J W Ponton, E S Fraga, R C McKinnel, and N Skilling. Parallelisation strategies for process modelling on mimd computers. *IChemE Research Event*, February 1991.

- J W Ponton, E S Fraga, R C McKinnel, and N Skilling. Simulation of nonlinear chemical processes and control systems using transputers. In Fleming Irwin, editor, *Transputer Control.* Research Studies Press, March 1991.

- Rosemary Candlin, Peter Fisk, Joe Phillips, and Neil Skilling. A statistical approach to finding performance models of parallel programs. In J Hillston R Pooley P King, editor, *Proceedings of the Seventh UK Computer and Telecommunications Performance Engineering Workshop*, page 180. Springer Verlag, ISBN: 3-540-19733-8, Edinburgh, July 1991.

- Rosemary Candlin, Peter Fisk, Joe Phillips, and Neil Skilling. A modelling environment for studying the performance of parallel programs. In J Hillston R Pooley P King, editor, *Proceedings of the Seventh UK Computer and Telecommunications Performance Engineering Workshop*, page 180. Springer Verlag, ISBN: 3-540-19733-8, Edinburgh, July 1991.

- Rosemary Candlin, Peter Fisk, Joe Phillips, and Neil Skilling. Studying the performance properties of concurrent programs by simulation experiments on synthetic programs. *Performance Evaluation Review, Special Issue, 1992 ACM Sigmetrics and Performance 92 International Conference on Measurement and Modelling of Computer Systems*, 20(1), June 1992.

- Rosemary Candlin, Peter Fisk, Joe Phillips, and Neil Skilling. A statistical approach to predicting the performance of concurrent programs. *Proceedings of EWPC'92, the European Workshops on Parallel Computing*, page 616, March 1992.

- Rosemary Candlin, Peter Fisk, Joe Phillips, and Neil Skilling. Studying the performance properties of concurrent programs by simulation experiments

on synthetic programs. *EPCC Technical Report*, EPCC–TR91–24, 1992.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One friend in life is much; two are many; three are hardly possible. Friendship needs a certain parallelism of life, a community of thought, a rivalry of aim

– Henry Brooks Adams,
The Education of Henry Adams

This thesis is concerned with *parallel computing*, the process of solving problems on parallel computers. Parallel computing is a relatively young field, the Illiac IV, a processor array came into operation in 1968 [BBK68]. Very large-scale integration (VLSI) made personal computing possible and it also made the development of large scale computing devices containing hundreds or even thousands of processors feasible. General purpose distributed memory MIMD machines began to become available in the early 1980's.

Programming general purpose machines became the next topic of serious interest. At first it was very difficult, with few software engineering tools compared with those available on uniprocessor systems. Early parallel computers did not support common programming languages, often requiring to be programmed in their unique language. Added to that was the fact that good algorithms for sequential computers are not necessarily good algorithms for parallel computers. Parallel computers demand a new way of thinking and programming if the full power of the machine is to be realised.

Chemical Engineering and in particular *process systems* have a lot of computationally intensive applications. Process systems practitioners were quick to realise and exploit the opportunities that parallel computers presented. In particular process systems simulation of a process plant can be *naturally* parallelised through the use of a process based programming approach where parallel processes are used to represent plant equipment and information flow corresponds to material flow through the real equipment. Such a parallel programming approach is possible with a *communicating sequential process* (CSP) programming system.

There is a major problem presented by CSP programs [Hoa84] and distributed memory multicomputers: the *mapping problem* is how best to place the *processes* of the program onto the *processors* of the machine. Usually the programmer would like to minimise the execution time of the program by choosing a good *mapping.*

The need to investigate *good* mappings for process systems programs led to the investigation of mapping strategies in more general terms. During this investigation it was noted that there was no rigorous or convenient way that a mapping strategy could be tested against a program in general. Allied with this was the fact that there was no convenient way of analysing the performance characteris-

tics and performance limiting factors of a parallel program.

This thesis presents a methodology for performing performance evaluation reviews of arbitrary parallel programs. It allows the user of the system to identify performance limiting factors quickly for an arbitrary parallel program. The methodology uses statistical experiments. The particular interest of *mapping strategies* as performance factors is investigated and it is shown that the methodology is adequate for such an investigation.

It is also shown that synthetic programs can be used to describe parallel computations rather than having to use real parallel programs. This allows the investigator to study a far larger range of programs than would be ordinarily possible. From the point of view of deriving performance estimates it is shown that carefully validated simulation and real execution of synthetic programs can be equally useful.

1.1 Contributions of Thesis

- The application of the methodology of statistical design of experiments to study arbitrary parallel programs.
- The use of synthetic programs to give a wider study of parallel programs.
- The employment of these techniques in a systematic investigation of static placement strategies.
- A demonstration of the utility of these techniques in producing performance models for both synthetic and real programs.

1.2 Chapter Outline

Chapter 2 gives an introduction to parallel computing through the literature available. The possible hardware classifications of a parallel computer are presented and the different ways that it is possible to program parallel machines. The two major models of parallel computing are described which are sufficiently detailed for performance evaluation studies to be performed. The task scheduling and process placement problems are introduced.

Chapter 3 gives a background to the art of computer systems performance analysis and presents a systematic approach to this due to Jain [Jai91a]. The two most important statistical tools necessary for the performance evaluation are also introduced.

Chapter 4 describes the methodology for evaluating parallel program performance in full. The performance metrics, the use of synthetic programs and the need for tools to aid the programmers are demonstrated. The experiment generator is presented and is shown to fulfill some of the requirements. The MIMD modelling system is also presented as it is also used in the performance evaluation studies.

Chapter 5 presents an experiment that was used to validate the MIMD modelling system against a process systems program. Following this is a complete experiment using the methodology that investigates the parameters used to describe a synthetic program and validates the program model used.

The final chapter of results, Chapter 6 presents a complete example experiment conducted on a set of placement strategies. This demonstrates how the methodology can be used to investigate this particular performance factor over a wide range of programs. The use of the predictive capabilities of models produced

from the statistical experiments is also shown. Following this is a case study of a set of placement strategies being applied to a process systems program.

Chapter 7 summarises and presents the conclusions from the study and presents proposals for future work.

A glossary of terms used throughout the thesis is provided after the final chapter and before the bibliography.

Chapter 2

Parallel Computing – Providing Solutions and Problems

Many hands make light work.
Too many cooks spoil the broth.
– Proverbs

2.1 Precis

Parallel computing is a very wide field and there are many active research areas within it. These range from the design and construction of novel computer architectures, through network and language design to parallelism applied to an

individual problem. This introduction to parallel computing is not complete; it is tailored towards the work presented in this thesis, though not exclusively so.

The structure of the chapter is as follows: first a general problem statement is given (§2.2); then an introduction to parallel computing is given (§2.3), including the terminology commonly used (§2.4) and the classifications of different parallel hardware (§2.5). Following this is a major section on the different ways in which parallel computers can be programmed (§2.6). In order to reason about parallel computations it is necessary to create *models* to represent those computations. The two major representations for parallel computations are described in Section 2.7 along with two of the common problems associated with these models. A survey of some profiling and performance evaluation tools is given in Section 2.8. The chapter finishes with a description of the approach taken in this work.

2.2 Problem Statement

Distributed memory multicomputers appear to offer a cheap and powerful computing resource. Unfortunately these multicomputers have not always delivered the processing performance promised from a summation of individual processor speeds. It is this performance gap that programmers spend long periods of time trying to bridge. This can be done in all the normal ways available to programmers of sequential computing (code optimization or reordering), but parallel computing also offers the choice of rearranging the program on the machine. The work described in this thesis is concerned with reducing the performance gap by providing more information about good placements to the programmer so as to avoid a lengthy tuning process. The intent is to find placement strategies and evaluate them to give an idea of their applicability to certain distributed program

types. If the best strategy can be applied automatically to a distributed program then it should save the program developer a period of costly program tuning.

The application programs of particular interest are those from Chemical Engineering. Some of these programs have a very regular geometric parallelism and regular communication patterns. Examples of this are the dynamic simulation of items of chemical processing equipment or even a complete chemical plant. Faster than real time detailed dynamic simulation is possible [McK94], but demands a lot of computing resource.

There is one major caveat emptor with parallel computing, exemplified as follows. In "The Design and Analysis of Parallel Computing"[Smi92] Smith states

> *Certain applications of computers require much more processing power than can be provided by today's machines.*

This statement is true and has been true since the start of the computer age. Like motorways and the motor car, the provided capacity lags behind the requirement. So *today's* computers will never be able to provide all the processing power asked of them, because they are today's computers. Parallel processing is seen as a promising way of providing large amounts of computing power to a processing hungry community.

The idea of performing tasks in parallel is not new. It can easily be traced back to Victorian times where large banks of "computers" were used to calculate tide and other mathematical tables. In this instance the computer just happened to be a desk clerk.

2.3 Introduction to Parallel Computing

A parallel computer is a machine with more than one processing unit working together to produce some final computation. The important aspect is that the processors work together in some sense, and thus have to be coordinated, either by hardware or by software.

There is no need for the processors of a parallel computer to be physically close together. A parallel computer could be a group of processor boards connected by a very high bandwidth back plane; alternatively you could describe all the machines connected to the Internet as the largest loosely connected parallel computer in this Solar system. There is certainly a lot of computing power in these machines though in general they are all working on different problems.

Even before the practical realization of multicomputer systems there was a lot of interest in predicting the performance of and potential speedups from parallel computers. As Casavant puts it "The notion that a loosely coupled collection of processors could function as a more powerful general-purpose computing facility has existed for quite some time" [CK88].

In the early 1980's the advances in semiconductor manufacturing technology made the realization of affordable parallel computers possible. Cheap, relatively powerful processing chips became available in high volumes, turning the notion into reality. This renewed the interest of theoretical performance modellers, and practical systems people in the field.

Another consideration is that it is getting increasingly difficult to manufacture faster semiconductors with more transistors in the same space, and we are rapidly approaching the physical limits of semiconductor technology. It is relatively ex-

pensive to manufacture a very large scale integration chip to run at high speeds, whereas a set of lower scale integration chips can be manufactured much more quickly and efficiently and turned into a parallel machine.

This availability of cheap processors has meant the development of a whole range of parallel computing architectures and a complementary range of parallel programming models and languages. Often the machine and language designs existed long before their physical realization was possible. This was the case with the parallel programming language Occam and the Inmos transputer [Inm84, Inm89a]. Occam was an implementation of Hoare's Communicating Sequential Process system [Hoa84] and was only available in a simulation system before the advent of the transputer made a parallel implementation possible. In fact the transputer was almost purpose built for the Occam programming language.

2.4 Parallel Computing Terminology

Parallel computers are designed to make computations proceed more quickly. So we must first define what we mean by "more quickly". A number of definitions have been proposed in the literature. We can take as our first principle from Smith the *Principle of Unitary Speedup* [Smi92].

Definition 1 *Unitary Speedup*

Suppose that the fastest sequential algorithm for performing a parallel computation with parameter n has execution time of $T(n)$. Then the fastest parallel algorithm with m processors (each comparable to that of the sequential computer) has execution time $\geq T(n)/m$

It is important to note that we must compare the fastest sequential algorithm against the fastest parallel algorithm. This definition tells us that it is impossible to get a parallel program that achieves a greater than unitary speedup. If this is the case then it is unlikely that you are comparing like with like. A faster than unitary speedup is not possible because you could then take the parallel algorithm and simulate it on the uni-processor and get a faster sequential algorithm. This would contradict the fact that the given sequential algorithm is the fastest possible.

Hatcher and Quinn [HQ91] define *Speedup* to be

Definition 2 *Speedup*

Speedup is the ratio between the time needed for the most efficient sequential program to perform a computation and the time needed for a parallel program to perform the same computation.

In this case we assume that the sequential program executes the complete computation on a single processor of a parallel computer and the parallel version executes on one or more processors. The difference is subtle but important. They restrict the sequential program to the memory of one node of the parallel machine so that large, unrealistic problems that could not fit comfortably in a uniprocessor memory are not used when calculating speedup. Instead they define another quantity *Scaled Speedup*

Definition 3 *Scaled Speedup*

Scaled Speedup is the ratio between how long a given optimal sequential program would have taken, had it been able to run on a single processor of a parallel com-

puter, and the length of time that the parallel program requires, when executing on multiple processors of the same computer.

Another measure of the performance of a parallel program indicates the reduction in execution time achieved as processors are added. *Parallelizability* is the ratio between the execution time of a parallel program on one processor and its execution time on multiple processors [HQ91]. Quite often *Parallelizability* is confused with speedup.

Hatcher and Quinn go on to say that perhaps the least confusing and controversial measure of program performance is the wall clock speed of the program. This is the most relevant measure from the point of view of a programmer who wants to improve the performance of a parallel program. This is of course a machine dependent measurement, unlike the other measures discussed.

2.5 Multiple Processor Computer Systems

There are many classifications of multiple processor computer systems. The most popular is due to Flynn [Fly66]. This compares the different types of instruction stream and data stream that a multicomputer could use. There are the two types of instruction stream, a single stream *SI* and a multiple instruction stream *MI*. There are two different data streams, *SD* a single data stream and *MD* a multiple data stream. This leads to four different processor architectures shown in Figure 2.1 and detailed below.

SISD This is the classic Von Neumann model. A single stream of instructions operate on a single stream of data. This type of machine is also referred to

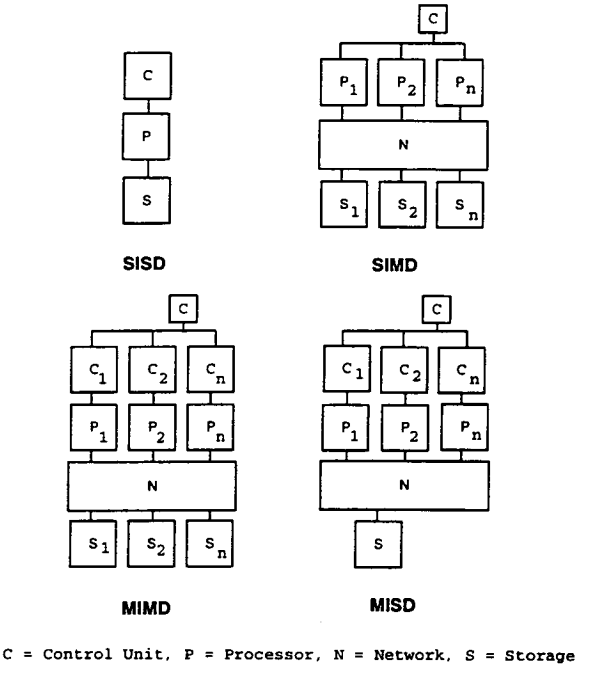


Figure 2.1: Classification of Multicomputers

as a uniprocessor. Examples include almost any computer up to the early nineteen eighties including almost all microcomputers and workstations.

SIMD All the processors execute the same instructions on different data simultaneously. Examples include Thinking Machines' Connection Machine CM1 and CM2, the latter having between 16384 and 65536 processing elements. All the processors are under the control of a single control unit.

MIMD This is an extension of SIMD where each processor can have independent programs that are read from common or local storage. The processors usually do not operate in lock step. These machines can range from tightly coupled asynchronous elements such as a multiprocessor workstation, using shared memory, to a less tightly coupled Meiko Computing surface using distributed memory [Mei92], through to a loosely coupled workstation network. MIMD machines have the possibility to be heterogeneous processing environments.

MISD This case is often compared to computation that uses *Systolic Arrays*[1]. These are arrays of processors that are developed to solve specific problems – usually on a single VLSI chip. A clock coordinates the data movement between processors and output from some processors is pipelined into others. For a discussion of systolic computers see Schreiber [Sch80] or Kung [Kun82].

All but the first classification are deemed parallel or multiple processor computers. Distributed memory MIMD and shared memory SIMD are the two most popular classes of machines for parallel computation.

Flynn's classification scheme was refined by Händler in 1977 [Han77]. This scheme is much more detailed than Flynn's but it still leaves much to be desired. For example there is a new important class of parallel computer that are SIMD-MIMD hybrids that the scheme is unable to describe. The first announced commercial machine was the Connection Machine CM-5 [Cor91]. This is essentially a MIMD machine that has hardware features that allow precise synchronization of processes and processors.

The different classes of parallel machines are best suited to different application areas. SIMD machines are most useful for highly data centric applications in which some regular computations have to be performed. MIMD distributed memory machines are proving to be the most scalable and general purpose parallel computers.

[1] **systole** noun, contraction of the heart during which blood is pumped into the aorta and the arteries that lead to the lungs – **systolic**.

2.6 Parallel Programming Paradigms

Many different parallel *programming models* or *paradigms* have been proposed in the literature. I will use the word paradigm to describe programming models to avoid the over use of the word model throughout this thesis.

The clear classifications of parallel computing hardware are muddied by the parallel programming paradigms that are presented to the programmer and user. SIMD and MIMD machines are both capable of supporting programming paradigms that are at odds with their physical construction. Performance penalties are usually paid if a SIMD programming paradigm is implemented on a distributed memory MIMD system.

In the programming of parallel computers there are different programming paradigms. The way in which a parallel machine is programmed may require the programmer to have a full knowledge of the underlying machine but there have also been attempts to insulate the programmer from the hardware of the machine to a greater or lesser extent.

In the early days of parallel computing it was common for the programming language to be very tightly linked to the underlying hardware. An example of this is Connection Machine Lisp [Cor87]. This was a result of the fact that some of these initial machines had no "real" operating system and also to the lack of an available language with any useful parallel constructs. This led to a host of vendor specific languages often very different from those languages being used on traditional sequential machines. These languages were inherently non-portable and had all the compatibility and portability problems equivalent to those that existed on sequential computers before standards like Fortran emerged. Programming early parallel computers often involved taking a step backwards in

software engineering technology.

Since the late nineteen eighties these problems have diminished quickly. Standard operating systems and (augmented) programming languages have become available on many parallel systems. This has made the software engineering task easier as standard development tools can be used to speed the software development cycle.

Claims are now made for genuinely portable programming languages. The problem of portability of the program is taken away from the programmer and given to the compiler writer and run-time communications library. Consider the difference in support software that is necessary to provide simple communication between two processes in a parallel program in the different cases of a shared memory and distributed memory machine. Totally different communications strategies must be used – but must appear identical to the programmer.

Proposals for genuinely portable parallel programming languages include Fortran-90 and Seymour [Ame92, MS89]. Hatcher and Quinn have also proposed the Data Parallel Programming language [HQ91] which gives the programmer a simple model of computation which can be used on several architectures. A similar claim of portability could now also be made for programs written with the Parallel Virtual Machine (PVM) software [GBD+93]. The PVM software provides a library of message passing primitives. These primitives and those in the emerging Message Passing Initiative (MPI) [Mes94] have been ported to many machines. The main disadvantage of these approaches is that the programs will run more efficiently on some machines rather than others.

In the next sections we look at common approaches to software development for parallel computers. This starts with conventional programming languages

and then moves on to languages with low and high level parallel constructs and concludes with architecture independent languages.

2.6.1 Conventional Programming Languages

This is the simple solution as far as the programmer is concerned, but the most difficult way from the compiler writers point of view. Take a standard imperative programming language, such as FORTRAN or C, and let a parallelizing compiler detect and exploit the parallelism in the program. This presents the programmer with a simple Von Neumann SISD model of computation. This is easy from the programmer's point of view because it shifts all the work onto the parallelizing compiler. It also has one very large benefit that existing programs (the well known "dusty decks") can take advantage of the machines.

Whilst writing a sequential program, the programmer will have sequentialized sections of the program which were inherently parallel. The programmers may not notice that some parts of the program are computationally independent. Parallel programmers have to be taught to look again at the way they create algorithms to release the inherent parallelism.

Any parallelism in an existing program will be hidden in a set of `DO` loops and control structures. The job of the compiler would then be to discover which can be performed in parallel and this is a difficult task which is probably why not many parallelizing compilers exist [ZBG88]. It may be possible when a vector or array processor is used and the compiler looks purely at the loop constructs. Loop constructs are the easiest to analyse and vectorize as has been shown by Scarborough and Kolsky [SK86]. A parallelizing compiler may only be suitable

for certain types of programs and may fail completely to detect any parallelism.

2.6.2 Languages with Low-Level Parallel Constructs

This is the most common and historically developed way of programming parallel computers. An imperative language is taken and augmented with constructs to permit parallel programming. Traditionally there have been two approaches based on what type of hardware the language has been aimed at. These are for multiprocessor or shared memory machines and multicomputers.

Multiprocessor Programming Languages

This set of programming languages provides the programmer with a shared memory programming model and so has traditionally been provided on multiprocessor machines. The language should have the ability to spawn and terminate parallel processes, manage synchronization between processes and distinguish between private and shared data.

An example of this is Sequent's Parallel C [Ost89] for their Sequent Symmetry machines [Seq87]. In this language the `m_fork` function forks off a set of parallel processes to execute a function. The processes suspend execution when they reach the end of the called function. Data is passed between cooperating processes by implicit access to shared variables.

These are very low level constructs for the programmer. They can be difficult to debug and it is difficult to eliminate timing errors. It has been shown by several

people that even short programs can have troublesome bugs [AO85, Sto88].

Multicomputer Programming Languages and Tools

These programming languages augment a traditional language with a set of message passing primitives. This may be done by extending the language itself and providing a new compiler for that language or by providing a set of library routines that can be called from a standard programming language. Data is transferred between processes by explicit communication statements. This requires the programmer to keep the state of each separate process in mind when writing the program.

An example of an extended language is Intel's nCUBE C and an example of the use of a library of message passing routines is Meiko's CS-TOOLS or the MPI proposal [Mei92, Mes94]. Each of these systems provides a set of message passing primitives which allow direct process to process communication. They provide send and receive operations, often with blocking and nonblocking variants. This provides asynchronous communication between multiple processes. Less common is support for broadcast or scatter and gather operations.

The most common programming problem with these types of languages is that of deadlock. Deadlock occurs when a cycle of communications is waiting to occur with each process in the cycle waiting on the next one. None of these programming environments can detect deadlock. Detecting deadlock before running a program is as difficult as running the whole program.

The major advantage of these programming languages is that you can write a program consisting of multiple processes which behave differently, but this is

also one of the problems. The programmer has to be able to keep different multiple states in their head whilst constructing the program. It is easiest, if it is possible, to implement SPMD (Single Program Multiple Data) or scattered spatial decomposition algorithms where the data is distributed over the memories of individual processors each executing the same program. This approach is useful for problems where the same operation is performed on different pieces of data independently with little boundary swapping such as for example image processing.

2.6.3 Languages with Higher-Level Constructs

Parallel computing provides the programmer with a wide variety of architectures. Software portability has been a problem with the earlier proposed parallel programming languages, with the languages being tied to one vendor's hardware/software combination. Parallel languages with higher-level constructs present the programmer with models of computation that can encompass a wide variety of architectures.

In these languages the compiler and programmer work together as a team to provide an efficient translation of the program. In this respect these languages lie somewhere between programming languages with low-level parallel constructs and the use of a traditional programming language with a parallelizing compiler.

There are many examples of this type of language in the literature. For example, Thinking Machines Corporations C* [RJ87] for their Connection Machine, Coherent Parallel C [FO88]. Both these languages augment the C programming language with vector operations. Linda [CG89b, CG89c, CG89a] provides the

programmer with a virtual tuple space into which data can be inserted. Tuples are retrieved from the tuple space, usually in a blocking fashion. This is just a different way to view a message passing system. Occam [Inm84] is the original communicating sequential process programming language. Channels are used as communication conduits. The major concept of sequential (SEQ) and parallel (PAR) statements identified by their level of indentation originate with this language. Parallel Pascal is a simple extension to Pascal with vector operations [RB84, RB87]. Poker [Sny84, GS87] is a visual programming system that has been parallelized.

2.6.4 Architecture Independent Programming Languages

A new set of parallel programming languages are emerging which provide the programmer with a consistent programming model and are aimed at a wide range of parallel architectures. Again at one end we have the sequential programming language and the parallelizing compiler. Other approaches have been the DataParallel C language from Hatcher and Quinn [HQ91]. This provides the programmer with a SIMD virtual parallel machine, the virtual parallel processors are simulated on as many processors as are available at run time.

Another approach is that of functional and logic programming languages, which take a traditional functional or logic programming language and provide a parallel implementation of this. Parallel implementations of Prolog [Cam87, FT88] and new languages such as SISAL [AO85, LSF88] and Strand [FT90] have been developed.

With all these programming languages the parallelism of the machine is hidden

from the programmer and so the efficiency of the program is entirely reliant on the implementation of the compiler and the run time communications system. The compiler and the run time system are responsible for the placement of the processes onto the available processors. The number of processors available to the program are given at run time and the run time system determines how the processes will be divided. Hatcher and Quinn claim speedups approaching 50 percent of the processors available when compiling the same programs on Intel iPSC/2, nCUBE 3200 and a Sequent Symmetry S81 [HQ91]. There are also programs for which speedup is a lot worse. Good speedups are achieved for programs that are close to the architecture of the machines.

2.7 Modelling Parallel Computers and Programs

In performance studies of parallel computations it is convenient and efficient to work with a *model* of a program rather than the real program. The model should be a realistic representation of the parallel program for the purposes of the study; in that it abstracts out those features which are relevant for performance measurement. This is in comparison to performing a study by direct measurement.

Performance models of parallel computations can be used to predict which implementation and which configuration of software elements for a certain computer are best in some sense. Performance models can involve hardware and software elements, though it is unusual for people to investigate both at the same time. Hardware designers like to have models that predict the performance of their machine designs and consider different network interconnects, for example [BDQ86]. Software engineers like to be able to find the best configuration of a particular

piece of software - they do not usually consider changing the hardware in some fundamental way, although they may be able to alter the topology of the machine. The software engineer may need to consider how the hardware is connected together and the design of interconnects is a large topic on its own. A complete description of computer performance methodologies can be found in [Hl84].

2.7.1 Representation of Parallel Programs

In the modelling and analysis of parallel programs there are two common formalisms for representing the programs. These are the task based model and the process based model of computation. Each of these formalisms has a set of performance problems associated with it, but in each case the main requirement is to minimise the execution time of the program.

2.7.2 Task Based Model of Parallel Computation

The task based model represents a parallel computation as a directed acyclic graph (otherwise known as a DAG) called the *task graph*, $G = (V, E)$, where V is a set of vertices and E a set of edges. Each vertex and edge has an associated weight. The weight of an edge usually represents the size of the communication packet which is sent between the two vertices. The weight of the vertex is the amount of computation that has to be performed. A vertex v has a set of predecessor vertices and a set of successor vertices. The vertex v cannot proceed with its computation until all of its predecessor vertices have finished their computation and have sent a message down the edge which links the vertices. This produces a temporal precedence relation on the task graph. An introduction is

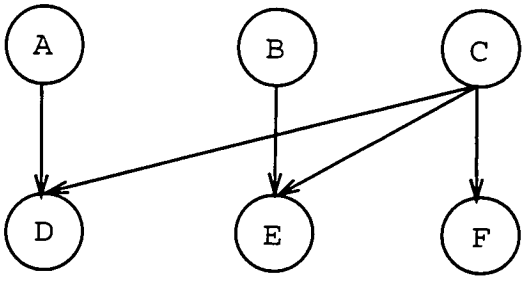


Figure 2.2: A Sample Task Graph

PE1	A	C	D	F
PE2	B		E	

Schedule 1, Makespan = 4

Time →

PE1	C	B	E	
PE2	A	D	F	

Schedule 2, Makespan = 3

Table 2.1: Two Possible Schedules for the Task Graph of Figure 2.2

given by El-Rewini or Sarkar, [ERL90, Sar87].

It is quite common for the edges to have zero weights in which case they request a synchronization rather than a communication event.

2.7.3 The Task Scheduling Problem

The main performance problem associated with the task graph is that of finding a *schedule* for the tasks in the graph. The schedule gives for any task the time when it will start execution and on which processor. This takes a multiple processor system consisting of P processors and tries to find an assignment of the tasks to the processors in P such that the temporal precedence relation is not invalidated and that the total execution time for the DAG is a minimum. This is known as the *makespan* for the given task graph.

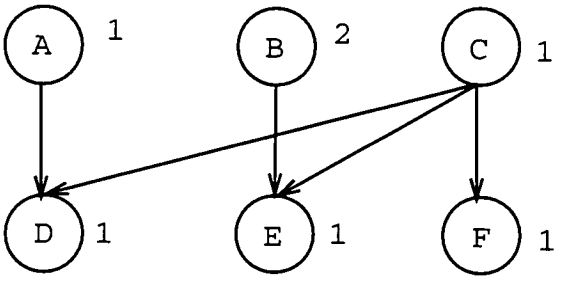


Figure 2.3: A Sample Task Graph with Weights

PE1	A	C	D	F	Schedule 1, Makespan = 4
PE2	B	B	E		

Time →

PE1	C	B	B	E	Schedule 2, Makespan = 4
PE2	A	D	F		

Table 2.2: Two Possible Schedules for the Task Graph of Figure 2.3

Figure 2.2 shows a simple task graph. Each task is assumed to have an equal weight. Two possible schedules for the graph when two processing elements (PE1 and PE2) are available are shown in Table 2.1. The second schedule has a makespan of 3 compared to the first of 4, the first schedule leaves the processing elements idle for a quarter of the time. The second schedule is one of the optimal schedules on two processors for the task graph shown.

If the weight of the vertex B is doubled to produce a task graph as shown in Figure 2.3 then the makespans become identical. The makespans for the same schedules are shown in Table 2.2. The two schedules are now optimal for the task graph.

Finding the minimum makespan is an NP-complete problem and is analogous to the Graph Isomorphism problem, though particular solutions for certain classes of problem have been found [Bok81b].

The task graph usually represents fine grained parallel computation. For example

each vertex could represent a handful of instructions in a program. This means that the task graph is a very detailed model and has problems in representing large programs, particularly those with iterative loops. Much work has been done in the area of DAGs and their properties [ERL90, Sin87].

A lot of the work in this field ignores communication costs or assumes a fixed cost communication irrespective of the size of the packet or the bandwidth of the associated communications network. This simplification may be appropriate for certain types of problem but it is not adequate for programs where significant amounts of communication are performed between distant processes. More detailed and realistic models are being investigated [Lo88, ML82, NCTI92].

2.7.4 The Process Based Model of Parallel Computation

The process based model of parallel computation again represents the program as a weighted graph (usually called the *process graph*). This graph is not necessarily either directed or acyclic. Each node represents a communicating sequential process and each edge represents the fact that there is at least one communication event between the two associated processes during the computation. This model was best described in a seminal paper by Bokhari [Bok81a]

The process graph weights represents the time averaged properties of the program. For example, edge weights could represent the average packet size sent along the particular communication channel. The vertex weights may have several interpretations: they could for example represent the size of the process in terms of memory required for storage, or the size of the main loop of the process.

It is common for process graphs to have a set of weights associated with each edge and vertex.

The process graph therefore gives a less detailed view of the computation than the task graph but allows much larger and coarse grained program structures to be modelled. The task graph can be thought of as a detailed description of some temporal part of the process graph. The task graph details the communication and computation events in detail, including the order in which events occur, whereas the process graph gives time-averaged properties.

2.7.5 The Process Placement Problem

Associated with the process based model of parallel computation is the *process placement* or *process mapping* problem. As with the task allocation problem this is a NP-complete problem and is computationally equivalent to the graph partitioning and multiprocessor scheduling problem [Bok81a].

The problem is to find a mapping between the processes and the processors such that some optimality function (criterion) is satisfied. Usually this criterion is to minimise the execution time of the program, though other alternative criteria could be the maximal utilization of the processor resources or the network bandwidth.

A sample process graph is shown in Figure 2.4 with three possible placements. Assume that the process graph has equal computational weights at each node and the weights on the arcs represent the size of the messages that have to be sent frequently between the nodes in a long running program. There are four

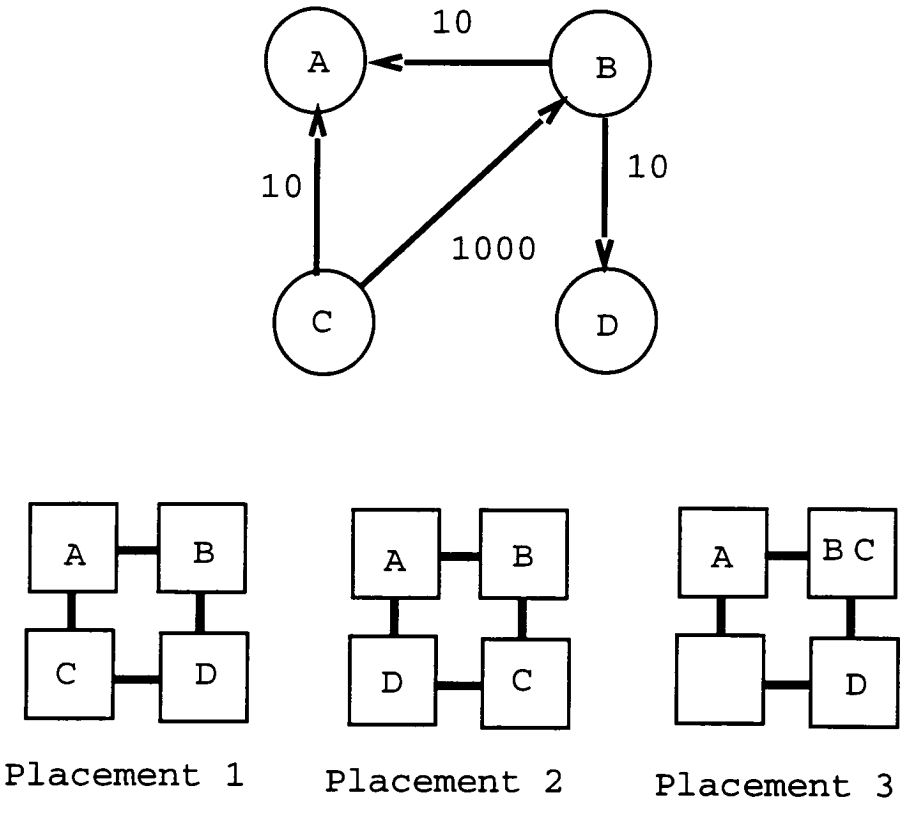


Figure 2.4: Simple Process Graph and Three Placements

processors available each is connected to two neighbours. Placement 1 is an obvious way to allocate the processes onto the processors; this may be obvious because there are four processes and four processors. The problem with this mapping is that the large communications between processes B and C have to traverse across two links, hence loading two links when one could suffice and probably disrupting the computation on A's processor to forward the messages. The second placement corrects this problem by swapping processes D and C so that B and C only have to send messages across one link. If it were possible to load both processes B and C onto the same processor then an even better placement might be Placement 3. Communication costs have been severely reduced in this placement at the cost of slowing the computation of processes B and C since they now have to share the same processor. This is just a small illustration of the complexities of process placement.

A complete taxonomy of scheduling mechanisms in general purpose distributed computing systems is given by Casavant and Kohl [CK88]. This provides a hierarchical naming scheme of task and process scheduling techniques. The structure of this hierarchy is shown in Figure 2.5. Their classification is firstly based on the way that the mechanism collects its data (local/global), whether the mechanism itself is distributed and then subdivides into different algorithmic approaches.

A static process placement strategy is exercised once before execution of the program. This computes an initial placement which may remain unchanged throughout the lifetime of that program. The initial placement is calculated from the process or task graph. Dynamic placement strategies make initial static placements and then periodically reevaluate the performance of the placement and make changes if necessary. The data required for these decisions has to come from some system on each processor. These are often referred to as dynamic load balancing techniques, and require complex decision making at run time.

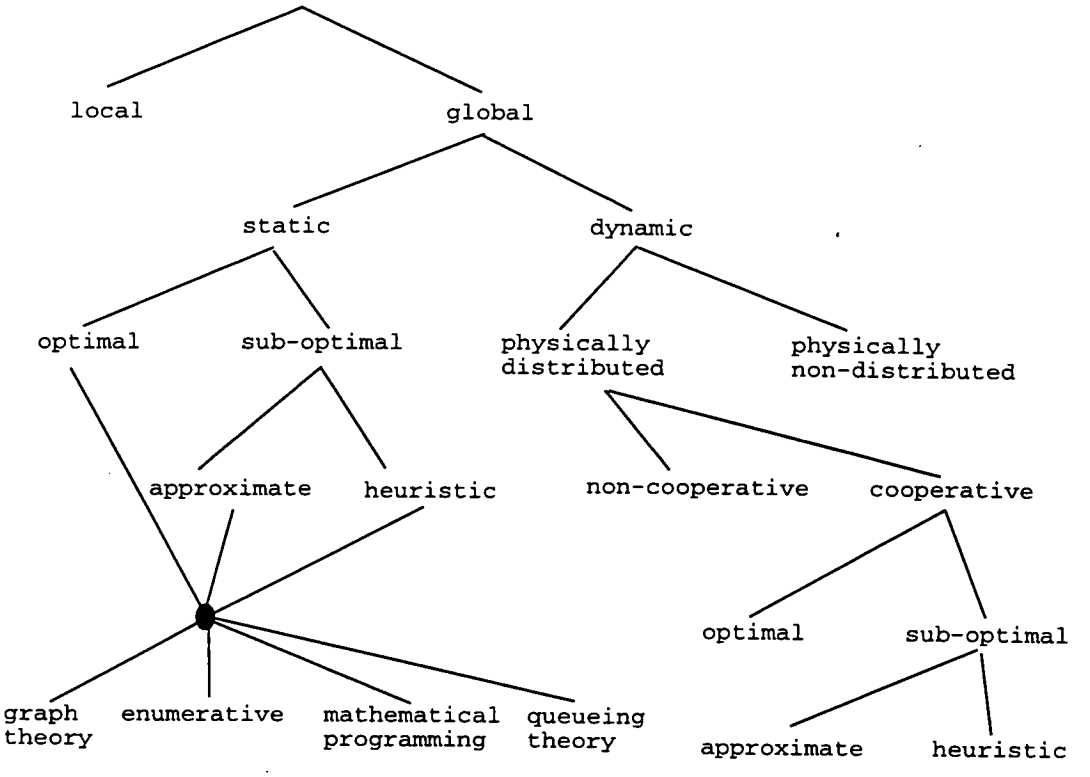


Figure 2.5: Task/Process Scheduling Characteristics

Dynamic placement algorithms collect their information in different ways and vary in effectiveness accordingly [AJ88]. Global system status information is usually difficult or costly to obtain in a multiprocessor system and so most dynamic placement strategies rely on local optimization procedures to effect a balanced load. As would be expected there is also an extensive literature on the subject of dynamic load balancing, for example [CC87, FB89, MTM88, PD89] and process migration [Can88, Gai90]. Phillips has investigated different dynamic load balancing/migration strategies on a range of program types [Phi94].

Dynamic load balancing is necessary for time variant programs where the computational load of processes varies over time or the structure of the program graph changes dynamically.

Static Placement Strategies

Unlike the task graph analysis which is usually far too detailed to be more than a theoretical tool the process placement strategy field has seen much experimental work and evaluation. The use of heuristic placement strategies for process based programs usually breaks down into two approaches: those that are static based strategies and those that are dynamic strategies. A more complete formalism has been suggested by [AJ88] in which the data collection of the strategy is also taken into account.

The whole area of task graph scheduling and process placement strategies is reviewed by Thanisch and Norman [Tha90].

Many placement strategies have been suggested by various authors. We can take as a sample set those presented by Donnet and Skillicorn [DS88]. They describe the problem as a graph partitioning problem and present several algorithms which they compare against a simulated annealing algorithm. The algorithms presented include Random Partitioning and Heaviest Edge First which attempts to reduce the communication costs by allocating processes with the heaviest edges to the same processor. Both of these algorithms are used later in Chapter 6.

The algorithm Find Local Communication Minima tries to minimize the total weight of edges radiating from each processor. This is essentially the dual of Heaviest Edge first. Donnet and Skillicorn then go on to develop an algorithm based on the technique of simulated annealing. They compare all the algorithms only by the estimated cost of the mapping or partitioning produced. These are mostly graph based algorithms and a lot of work is devoted to graph techniques. In particular complex branch and bound tree search techniques have been used by Sinclair to produce mappings for small programs and processor setups [Sin87].

Other standard graph techniques have been proposed by Fernandez-Baca amongst others [FB89].

More esoteric graph partitioning based approaches have been proposed for example by Muhlenbeim et al. [MGSK87] in which techniques from the natural sciences such as competition for resources and evolution are applied to the problem. This is based on the reasoning that nature has performed some of the best speciation[2] that is available so why not copy the techniques. This of course is taken a lot further by algorithms which ignore the structure of the problem altogether and rely purely on the genes or parts of the solution. This means that Genetic Algorithms can and have been applied to this problem. See for example a parallel genetic algorithm for process placement by Pettey and Leuze [PL88].

Summaries of several techniques can be found in for example [HK72] and [KM88].

Placement for Real Programs

Of specific interest is the work by the author and colleagues detailed in [PFMS91]. This addresses the particular problems and simplifications that occur with Chemical Engineering parallel computations. These range from the simple geometric and obvious decomposition of a distillation column simulation program to the large exponential and computationally intense search space required for synthesis procedures for chemical plants.

The distillation column example is simple because each tray can be modelled in detail by a process connected in a chain of similar processes. Information flow is

[2] **speciation** noun, the evolutionary development of a biological species [C20: from SPECIES + ATION].

closely related to physical material flow in the real distillation column. This is described in more detail by M^{c}Kinnel in his thesis [McK94].

2.7.6 Modelling Parallel Programs – Summary

The two approaches to the modelling of parallel programs described above offer two distinct and complementary views of parallel computation. Whilst the task graph offers a very fine grained instruction level view, the process graph can be used to provide a higher level, time-averaged view of the parallel computation. These two representations are also commonly used as paradigms at the programming level, and are appropriate for different types of language. These describe two different programming models which are also complementary. The task graph parallel program model is best used to describe the shared memory programming model. Here tasks represent computation and the precedence of tasks implies a synchronization or barrier point. The process based representation describes best the communicating sequential process model of computation. This is more usually described as a message passing system.

2.8 Performance Analysis Tools

Performance analysis or profiling tools for parallel programming are often incorporated within more extensive software engineering support environments. A clear example of this is the Crystal system supplied with Thinking Machines CM-2 machine [CCL88]. This system allows users to interactively debug their program, which may be executing on up to 16,535 processors, viewing the values

and most commonly accessed parts of data structures as well as being able to see the percentage of time spent on each individual line of code. This is an exceptional example of such a tool and unfortunately is restricted to this one particular platform.

Other profiling tools with good visualization facilities enable programmers to study the performance properties of particular programs and to detect undesirable features such as load-imbalance and communications bottlenecks. Measurements may be made directly on the running program using a hardware or software monitor, or a performance model of the program may be constructed and executed on a simulation model of the machine. Examples of these three approaches are respectively: INCAS [NHM+87], IPS [MY87], and TRANSIM [Har89b] in conjunction with GECKO [Har89a]. GECKO shows the processors and links of a transputer array and uses colour to show the hot spots. A simulated system has many advantages from the point of view of flexibility and ease of use, whereas hardware monitors are specific to a particular type of machine and are expensive to implement. Software monitors consume machine resources and may distort the behaviour of the program under investigation.

Yang and Miller [YM89] suggest that there should be an underlying program model with which to interpret the performance metrics measured. They suggest that a *program activity graph*, representing the duration of activities and their precedence relationships. Support is provided to help the programmer find the critical path of the program activity graph. A reduction in the critical path will lead to a performance increase.

Examples of analysis systems based on task graphs are given by Task Grapher [ERL90] and ADAS [FFI85]. Task Grapher is a high-level tool which allows a user to set up a dataflow model and investigate performance under a number

of different scheduling heuristics and machine topologies. ADAS is particularly targeted at early performance evaluation of hierarchical, integrated hardware and software systems. The designer works in terms of a dataflow model from which a Petri net [Pet81] is then derived. Performance metrics can then be obtained by simulation or, if the net is not too big, analytically.

Kitajima and Plateau [KP94] have used process-based program skeletons to perform performance analysis on a distributed memory parallel computer with the ANDES system. This system also used the MIMD simulator described in Chapter 4.

The systems described above are only a subset of those that have been reported in the literature, though they are typical in mostly being based on precedence graphs, in providing comprehensive statistics and in leaving the decisions about what should be tuned up to the programmer.

When it comes to dynamic scheduling strategies a typical system for the study is described by Hemery and Geib [HG94]. This combines a simulator and a graphical front end in which to compare different scheduling strategies on a hypercube system. A program is written in the event generation language GENESE and interpreted by the system. There are a limited number of strategies that can be employed based on different information distribution patterns.

Traditionally computer systems have been modelled analytically using queueing networks. It is not feasible to model the complex interactions in parallel systems by queueing networks. On the other hand, *Petri Nets* can be used to model parallelism accurately. Petri nets were developed by C A Petri in 1962 to define the coordination of asynchronous events (see [Pet62] and [Pet81] or [Brä93]). Petri nets allow the modeller to clearly describe interdependence between and

synchronization of parallel processes. A petri net is a *directed, bipartite graph* with *tokens* (markings). Nodes are either a *place* or a *transition.* The *place* is often called a state and can be empty or occupied with a token: *marked.* A *transition* is often called an action. Edges are *arcs* which are directed connections between places and transitions. By associating a time delay with a transition it is possible to estimate the times required for a system to move from one state to another.

The size and complexity of many parallel systems precludes the complete use of analytical modelling techniques, though hybrid models where the hardware resources are modelled as queueing networks embedded within a generalised stochastic Petri Net or Markov process representing the control structure of the program have been created [BBC86] and [KME89]. In this way, the size of the state space is reduced compared to what it would be if the entire system were modelled as a Petri Net. It is not clear whether realistically large systems can be handled by this approach, which suffers also from the disadvantage that changes to program or hardware may require substantial redesign of the model.

All of the above systems aim at providing quantitative performance predictions for *specific* programs. None of the systems allows the user to evaluate a set of related programs easily. Neither do they show how performance factors may interact. The programmer must proceed on a trial and error basis to tune up the program.

2.9 Process Systems Parallel Programs

In the introduction to this thesis it was explained that the impetus for this work came from parallel computations being performed in process systems. Detailed, dynamic simulation of chemical processes requires significant amounts of computer resource. Uniprocessor machines do not have the power to provide this so the need for parallel processing arose.

Simulations of physically connected chemical processing equipment are achieved naturally in parallel processing by representing each unit by a program process. This gives a robust solution structure because any calculation errors are localised to the unit and allows individual components to be changed easily. The shape of the program graph is fixed and the process loads are relatively time invariant. This means that the amount of computation to be performed at time t is roughly equivalent to that to be performed at time $t + 1$. There are situations when this is not the case, such as when there are large disturbances within the system due to, for example, feed stock changes. These changes may remain localised, giving an unbalanced computational load or the effects may spread throughout the system, balancing the computation once more. In general though, these periods of disturbance will be small compared to the complete processing time.

With this type of program in mind, I have limited my study to process based programs that have a fixed structure and which are mostly time invariant. I have also concentrated on static placement strategies, because the type of program described above does not require dynamic load balancing strategies to be employed. The work has also been limited to using general purpose distributed memory MIMD machines. This was partly based on the general availability and popularity of these machines and the specific availability of such machines in Edinburgh.

2.10 Design of Approach

Though many placement strategies have appeared in the literature there has been little work (with a few exceptions [DS88, HP92]) to rigorously quantify the efficiency of a set of placement strategies for a set of parallel programs. Most work either describes a single placement strategy applied to a small set of (similar) test programs or shows the application of different strategies to the same program. The research described in this thesis, on the other hand is directed towards the improvement of parallel applications by any appropriate method, rather than a specific interest in the placement strategies themselves. This requires an abstraction of the parallel program into certain core characteristics so that the behaviour of a class of parallel programs under various placement strategies can be investigated.

A methodology for screening for appropriate program characteristics in a systematic way was needed. This led to the decision to build such a special purpose software system that would make use of standard statistical methods in order to quantify the merits of the various strategies. It was expected that different strategies would perform better for different types of programs. For example the strategy called Heaviest Edge First (described above) would have little effect on a program with small communications costs.

The decision for a rigorous comparison of the strategies was advocated by the work of Nance and Moose [NMF87] in the field of network evaluation. This entails taking the view that the program or system to be investigated is a black box. Known parameter settings are fed to the black box and responses are received. The parameters of a distributed computer system could be the hardware, software and placement strategy. Through the analysis of these parameters the performance of a particular placement strategy in relation to the program can be

investigated.

This kind of parameter analysis necessitates the ability to create programs which have desired characteristics. This has led to the use of synthetic or automatically generated programs. This gives much more flexibility and control in model analysis.

2.11 Summary

This chapter has given a general introduction to parallel computing. It has shown the different classifications for parallel computing hardware and the different parallel programming paradigms that exist. The two major models of parallel computations, the *task graph* and *process graph* were introduced and the associated problems of *scheduling* and mapping.

This chapter concluded with the restrictions that have been placed on the study due to the particular interests of the author and to the particular approach to performance evaluation that is presented in this work.

Chapter 3

Performance Analysis and Statistical Techniques

I keep six honest serving men. They taught me all I knew. Their names are What and Why and When and How and Where and Who

– Rudyard Kipling

3.1 Precis

This chapter is in two distinct parts. Firstly *computer systems performance analysis* is introduced (§3.2) and a systematic approach is presented in Section 3.2.1.

Performance analysis is discussed in the particular context of parallel computing and any special problems are outlined in Section 3.2.2. The novel way in which this systematic approach to performance analysis is applied to parallel systems forms the basis of the methodology described in this thesis. The particular details of this are outlined in Chapter 4.

The important statistical tools and techniques that form a necessary part of the performance analysis are presented in Section 3.3.

3.2 Performance Analysis

Jain describes computer performance analysis as an art rather than a science [Jai91a]. Like a work of art a successful performance evaluation can not be produced automatically. Every evaluation requires an intimate knowledge of the system being modelled and a careful selection of the methodology, workload and tools.

Performance evaluation has always been a distinct part of Computer Science. Historically performance evaluation studies have centered on large multi-user systems. A common study would be aimed at identifying the performance bottleneck of the system and factors considered could be the disk subsystem, memory paging or scheduling algorithm. The interest in performance evaluations of parallel systems has grown alongside the development of the systems themselves. The driving force, again, is to identify bottlenecks in certain areas of the system to close the performance gap between power and execution time.

All computer users are interested in performance evaluation. Usually this is

phrased in the statement "I would like my computer to go faster" or in the question "Why is my computer not as fast as I thought it would be?". Like any consumer the user wants to get the highest value (performance) possible at the lowest cost. Computer performance evaluation can be applied at every stage of production and use of a machine, from a logic simulation of a processor board at the design stage through to final performance testing of a disk drive on a standalone system. Commonly though, it is the performance of the whole system in terms of job execution speeds that the user is most interested in.

In this thesis, I am particularly interested in making performance improvements from a programmers point of view. Changing hardware is not an every day decision taken by programmers, whereas software provides many more opportunities for change. For instance the way data is distributed, synchronous or asynchronous computations and more traditional code optimization can all be factors affecting performance.

Parallel systems pose performance analysis questions which do not arise in uniprocessor systems. The most common performance criterion associated with parallel computers is the concept of speedup, which was introduced in Section 2.4. It was sometimes naively assumed that linear speedup of software should always be achievable. This was not always practically achievable due to poor process placement, slow communications software and slow networks as well as to the inherent structure of the computation in some cases.

- State the goals of the study and define the system boundaries
- List system services and possible outcomes
- Select performance metrics
- List system and workload parameters
- Select factors and their values
- Select evaluation techniques
- Select the workload
- Design the experiments
- Analyze and interpret the data
- Present the results. Start again, if necessary.

Table 3.1: Steps for a Performance Evaluation Study

3.2.1 A Systematic Approach to Performance Evaluation

Jain [Jai91b] has outlined a systematic approach to performance evaluation. His points are general and can be applied to any performance evaluation study. They are shown in Table 3.1 and are outlined below. Following on from this, I take these points and show how they may be applied to the performance evaluation of parallel computer systems.

State Goals and Define the System The first step in any performance evaluation review is to state the goals of the review and to define the system within which the review is to be performed. This is an exercise in defining the boundaries of the review.

The goal is usually to review the performance of the system with the idea of improving the performance of that system in some way or just identifying the performance limiting factors.

List Services and Outcome Every computer system provides a set of services. For example a database system responds to queries, a processor performs a sequence of instructions. It is useful to list these services and possible out-

comes as a means of identifying useful performance metrics and workloads.

Select Metrics The metrics are the criteria on which performance is compared. The criteria usually measure the speed, accuracy and availability of services. For example in a database processing system a common metric is the number of transactions performed (queries answered) per second.

List Parameters The next step is to list all those parameters that are believed to affect performance. These parameters can be broken into two sets, system and workload parameters. System parameters include hardware and software parameters. Workload parameters summarize characteristics of users' requests.

Select Factors to Study From the parameters identified above a subset must be identified as *factors*. These are the parameters which will be investigated during the review and they will take values at different *levels*. To have a manageable review the number of factors is usually smaller than the number of parameters.

It may not always be possible to use the desired metrics. This may be because it is not possible to instrument some part of the system under study. In this case a related metric would have to be chosen. Factors should obviously include the parameters which the investigator believes have an impact on performance. The choice of factors is also restrained by reality. For example, it is no good if the investigator chooses as a factor to be studied the number of processors in a system if the number of processors can not be increased either due to financial or technical reasons.

Select Evaluation Technique There are three broad techniques for computer performance evaluation. These are analytical modelling, simulation and measuring a real system. The selection of the right technique is based on the time and resources available to solve the problem and also the degree of accuracy required.

Select Workload The workload consists of a set of service requests to the system under investigation. For example the workload to a database system may consist of a mixed set of queries. The workload takes different formats depending on the evaluation technique chosen. For example with analytical modelling, the workload could be expressed as the probabilities of certain requests. In simulation, it could be a trace from a real system. On a real system, it would most likely be a series of test programs that characterize a typical user.

Design Experiments Once the factors and levels have been decided, a sequence of experiments can be performed to provide the maximum information with minimal effort. This can best be achieved through the use of the *statistical design of experiments* described later in Section 3.3.

Analyze and Interpret Data Interpreting the results of an analysis is a key part of the analyst's art. The results provide the basis on which the analysts can draw their conclusions. It is of course possible to draw more than one conclusion from the same results. It is important that the analysis of the data is considered when the experiment is constructed. The results must be easy to collect in a form that is acceptable to any data analysis tools. The experiment and the analysis should go hand in hand. Statistical techniques used to analyze the results are also presented later in this chapter (§3.3).

Present Results The last step of all performance reviews is to present the results to those people who are involved in making a decision based on the results. At this stage it may become apparent that a more detailed review needs to be performed, and so the analyst may have to go back to an earlier point in this process.

3.2.2 Performance Evaluation of Parallel Systems

The systematic approach to performance evaluation outlined in the previous section can be applied to parallel systems. The points above will now be reviewed in the context of parallel computing systems and will demonstrate the particular problems raised by parallel systems. The points of the systematic approach should be kept in mind again when the methodology proposed by this thesis is presented in Chapter 4.

In single processor performance reviews, the system to be studied may be the computer system as a whole or some component part like the CPU or disk subsystem. This situation is more complicated in parallel systems where there are multiple CPU's and other hardware components such as specialized network interconnect. The system itself may cover a large geographic area and so care must be taken in clearly defining the system boundaries.

From a macroscopic point of view, a parallel system provides the same services as a uniprocessor system. This point is a very salient one. If it were possible to provide exactly the same services to the user without their having to change their programming model then parallel computing would be completely integrated into mainstream computing. Unfortunately, as described previously, the sequential programming model has already hidden the natural parallelism in the problem. So while it is correct to say that the parallel system provides the same services and outcomes, i.e. job execution and results, the way in which this happens is completely different.

Parallel systems provide similar metrics to uniprocessor systems, though some of them are more difficult to obtain. Average job throughput on a multi-user parallel system can easily be compared against the same uniprocessor measurement.

Some metrics specific to parallel systems are also required. For example in a multi-user system it is desirable to maximize the utilization of the processors. This measures how effectively the parallel programs are using the processing resources. The utilization also becomes more and more important as the relative costs of the interconnect to the processor increases. This has been a general trend over the last few years, with an increase in both the speed and bandwidth of the interconnect. The importance of network bandwidth became more obvious in distributed systems when for certain applications the communication costs started to dominate performance [Cla90].

In parallel computing the workload parameters are not as well understood as they are on uniprocessor multitasking systems, where standard benchmarks are commonly known and available. Distributed systems research is still at the stage of working with innovative hardware and software designs and common workloads that are appropriate for all machines are relatively unknown.

One of the main problems in parallel computing is that there are a large number of parameters, arising from the fact that there are several processors and a interconnection network. These two components have many parameters hidden inside such as the speed of individual processors and network.

Parallel processing poses a particular problem in that the resources under investigation are very expensive to use. Sometimes, they lack the instrumentation facilities in either hardware or software which would facilitate any sort of performance evaluation other than execution speed of the program. This has meant that a lot of performance evaluation of parallel systems has been performed either using analytical modelling techniques or simulation (§2.8).

3.3 The Statistical Design of Experiments

One of the major contributions of this thesis is the application of the standard techniques of experimental design to performance analysis of parallel programs. The relevant techniques are described in this section. At the start of this investigation the only relevant work on the application of these techniques was that by Nance and Moose [NMF87]. They had used the techniques to evaluate capacity assignment strategies in computer network design.

Phillips [Phi94] used the techniques and much of the software developed for this thesis to study process migration strategies for time varying parallel programs. Lyon, Snelick and Kacker [LSK94] used statistical techniques to identify bottlenecks in MIMD programs. They did this by adding artificial delays or time perturbations to parts of the program to see if this affected the performance.

In the context of this investigation it is apparent that there are many parameters of a parallel system which may affect performance. A systematic way to construct performance evaluation studies and to rapidly evaluate which parameters have significant effect is necessary. The factorial design of experiments and analysis of variance satisfy these needs.

3.3.1 Experimental Designs

There are other types of experimental design, but the most commonly used are the simple design, the factorial design and the fractional factorial design.

Simple Designs

In simple design, we start with a typical configuration and vary one factor at a time to see how that factor affects performance. Given k factors, with the ith factor having n_i levels, a simple design requires n experiments, where

$$n = 1 + \sum_{i=1}^{k} (n_i - 1) \tag{3.1}$$

However, this design does not make the best use of the effort spent. It is not statistically efficient because only one parameter is investigated at a time. Also, if the factors interact, this design may lead to incorrect conclusions. This design is not recommended by serious practitioners.

3.3.2 Full Factorial Designs

The analysis of factorial experiments was proposed by Fisher in 1935 [Fis58] with the introduction of his Analysis of Variance technique. This permitted the devising of experimental designs which can handle the simultaneous testing of large numbers of different factors efficiently to see what effect they have on the value of one or more output variables, otherwise known as the *response variables.*

The term *factorial designs* is used because they evaluate the effect of different combinations of experimental factors. In any experiment, each component part that can be varied is a *factor.* For example, an experiment to make a cake has many factors such as the amounts of each ingredient, the oven temperature and

the baking time.

Factorial experiments are designed to test two or more factors at the same time, economically in the number of experiments and to evaluate the effects of interactions between factors. This is done by comparing the results obtained with different *levels* of each factor. The experiment for baking the cake, may for example, be repeated at different temperature levels. It is important to note that these levels may in fact be the presence or absence of the particular factor.

The key aspect of these designs is that the factors are varied simultaneously rather than one at a time in order to facilitate estimating the effects of interactions between factors.

A full factorial design utilizes every possible combination at all levels of all factors. A performance study with k factors, with the ith factor taking n_i levels, requires n experiments, where

$$n = \prod_{i=1}^{k} n_i \tag{3.2}$$

3.3.3 Analysis of Variance

In the analysis of experiments, we want to decide whether observed differences are attributable to chance or if there are real differences among the means of the populations of the sampled values. To make this decision, we can not just rely on visual tests, an automatic technique is necessary. This technique is called the *analysis of variance*, usually abbreviated to *ANOVA*.

An ANOVA attempts to partition the variation into an explained and unexplained part and provides a guide to how much of the explained variation is genuine. The variation under investigation is that which is inherent in the data produced from an experiment. We need to know whether this variation is due to real differences in the measured data values, whether it is due to not measuring the same thing twice on different replications or whether it can be attributed to chance.

In general, we have k independent random samples of size n from k populations. The jth value from the ith population is the denoted x_{ij}, that is

$$\begin{aligned} &Population1: \quad x_{11}, x_{12}, \ldots, x_{1n} \\ &Population2: \quad x_{21}, x_{22}, \ldots, x_{2n} \\ &\vdots \\ &Populationk: \quad x_{k1}, x_{k2}, \ldots, x_{kn} \end{aligned} \tag{3.3}$$

and we shall assume that the corresponding random variables $\mathbf{x_{ij}}$, which are all independent have normal distributions with the respective means μ_i and the common variance σ^2.

Stating this differently, we could say that the model is given by

$$x_{ij} = \mu_i + e_{ij} \tag{3.4}$$

for $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, n$ where the e_{ij} values are values of nk independent random variables having normal distributions with zero means and the common variance σ^2. The term e_{ij} can be attributed to experimental error, or to the effects of parameters which have not been included in the model. This

is usually written in the form

$$x_{ij} = \mu + \beta_i + e_{ij} \tag{3.5}$$

for $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, n$, where μ is referred to as the *grand mean* and the β_i are called the *treatment effects* such that $\sum_{i=1}^{k} \beta_i = 0$.

The term *treatment* for referring to the different populations comes about from the origin of the analysis of variance techniques. They were commonly used in agricultural experiments to compare the effectiveness of different fertilisers. In this work, for example, a process placement strategy could be analogous to different fertilisers. The opportunity to repeat experiments on computer systems is obviously easier than in the agricultural "field", but these powerful statistical techniques are still thoroughly appropriate.

We must now test the *Null Hypothesis* that the population means are all equal, namely, that $\mu_1 = \mu_2 = \ldots = \mu_k$ or equivalently that

$$H_0 : \beta_i = 0 \quad \text{for } i = 1, 2, \ldots, k \tag{3.6}$$

Correspondingly, the alternative hypothesis is that the population means are not all equal, namely, that

$$H_1 : \beta_i \neq 0 \quad \text{for at least one value of } i \tag{3.7}$$

The test, itself, is based upon an analysis of the total variability of the combined data (as might have been expected from its name). The total variability of the combined data is given by

$$\sum_{i=1}^{k}\sum_{j=1}^{n}(x_{ij} - \bar{x}_{..})^2 \tag{3.8}$$

where

$$\bar{x}_{..} = \frac{1}{nk}\sum_{i=1}^{k}\sum_{j=1}^{n}x_{ij} \tag{3.9}$$

If the null hypothesis is true all the variability is due to chance, but if it is not true then part of the above sum of squares is due to differences among the population means.

It can be shown (in any statistics text book e.g. [FW80]) that the above sum of squares can be rewritten so that

$$\sum_{i=1}^{k}\sum_{j=1}^{n}(x_{ij} - \bar{x}_{..})^2 = n\left(\sum_{i=1}^{k}(\bar{x}_{i.} - \bar{x}_{..})^2\right) + \sum_{i=1}^{k}\sum_{j=1}^{n}(x_{ij} - \bar{x}_{i.})^2 \tag{3.10}$$

where $\bar{x}_{i.}$ is the mean of the observations from the ith population and $\bar{x}_{..}$ is the mean of all nk observations.

It is usual to refer to the expression on the left hand of the identity in Equation 3.10 as the *Total Sum of Squares*, the first term of the expression on the right hand side as the *Treatment Sum of Squares* and to the second term as the *Error Sum of Squares*. Here "error" denotes the experimental error, or chance. If we

denote these three sums of squares by SST, SS(Tr) and SSE we can then write

$$SST = SS(Tr) + SSE \tag{3.11}$$

This has split the measure of the total variation SST into two components. The second component SSE measures chance variation. The first component SS(Tr) measures chance variation when the null hypothesis is true, but it also reflects the variation among the population means when the null hypothesis is false.

It can be further shown that we can bring the decision for rejecting the null hypothesis down quite simply to the following. We reject the null hypothesis that the population means are all equal if the value we obtain for

$$F = \frac{k(n-1)SS(Tr)}{(k-1)SSE} \tag{3.12}$$

exceeds the value $F_{\alpha,k-1,k(n-1)}$. The ratio of the sums of squares follows an F distribution with $k-1$ and $k(n-1)$ degrees of freedom. α is the level of significance that we wish. α is the probability of rejecting H_0 when it is really true. The assumptions underlying this analysis are described in Section 3.4.1

The procedure described above is called the one-way analysis of variance and the details of this technique are usually summarised in a table of the kind shown in Table 3.2 – known as an analysis of variance table.

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F
Treatments	$k-1$	SS(Tr)	MS(Tr) $= \frac{SS(Tr)}{k-1}$	$\frac{MS(Tr)}{MSE}$
Error	$k(n-1)$	SSE	MSE $= \frac{SSE}{k(n-1)}$	
Total	$kn-1$	SST		

Table 3.2: One Way Analysis of Variance Summary Table

Combination	Response
A	300
B	400
AB	200
Ω	500

Table 3.3: Execution Times (seconds) for our Sample Program

3.3.4 Worked Example

A short example from a simple computer performance experiment will illustrate the techniques described above. Suppose we have a single program and have identified two parts of the program where we believe we can make performance increases. We do not know whether the two supposed performance enhancing factors will work against one another.

We can call these two factors A and B and we want to evaluate their separate and combined effects on the execution time. To do this we run the program for every combination of levels of the two factors. Each factor is taken at one of the two levels, "present" or "not present". This gives us the four combinations A, B, AB and the control program Ω which has both factors absent (i.e. our original unaltered program). This will require four experiments.

Suppose that we get the execution times shown in Table 3.3 for each combination of factors.

The program runs must be repeated at each factor combination to get an average

Combination	Response
A	300
B	400
AB	150
Ω	500

Table 3.4: Execution Times with Interaction

for each value above, and to find the variation due to experimental error. The experimental errors can only be quantified if the measurements are repeated under the same factor-level combination.

The difference in execution times with A compared with the control program, i.e. $A - \Omega$, tells us just what effect A has had. In the present case A has decreased the execution time by $500 - 300 = 200$ seconds. By the same reasoning, B has decreased the execution time by $500 - 400 = 100$ seconds. Now we must also look to see if there has been an interaction between A and B. An *interaction* is the phenomenon where two or more factors interact so that their combined effect is different from the simple addition of their separate effects.

If there were no interaction between A and B, the result of AB should just be a summation of the execution time of the control program plus the effect of A (-200 seconds), plus the effect of B (-100 seconds), which gives a total of 200 seconds. This is, in fact, just what occurred, so we conclude that there was no interaction between A and B. This will not always be the case. For example Table 3.4 shows that the response from the AB combination is not a simple summation of their individual effects. There is an *interaction* between the factors.

In this simple example no replications of the experiment have been shown. Now let us consider a more realistic example where the experiments are replicated at each factor level.

	Samples	Mean
Strategy A	77, 81, 71, 76, 80	77
Strategy B	72, 58, 74, 66, 70	68
Strategy C	76, 85, 82, 80, 77	80

Table 3.5: Sample Program Data for Analysis of Variance (Execution Times)

Assume that we have a parallel program consisting of communicating sequential elements. We have 3 placement strategies that we wish to compare to see which if any has an effect on the performance of the program. The program is run with its processes placed by each of the three strategies 5 times. The execution times for all these runs are shown in Table 3.5. The means of the samples are shown alongside the data.

What we want to know is whether the difference among the mean values is significant or whether it can be attributed to chance.

We will perform an ANOVA on the data in Table 3.5. We will test the data at a 99 percent confidence or a significance level of 0.01 ($\alpha = 0.01$). So our null and alternative hypotheses are:

1. $H_0 : \beta_i = 0$ for $i = 1, 2, 3$
   $H_1 : \beta_i \neq 0$ for at least one value of i

2. Critical Regions: $F \geq F_{.01,2,12} = 6.93$

3. Computations: By performing the necessary calculations (there are easy ways to calculate SST and SS(Tr)) we get the analysis of variance table shown in Table 3.6.

4. Decision: Since $F = 8.48$ exceeds $F_{.01,2,12} = 6.93$, the null hypothesis H_0 must be rejected at the 1 % significance level, and we conclude that the three placement strategies are not all equally effective.

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F
Treatments	2	390	$\frac{390}{2} = 195$	$\frac{195}{23} = 8.48$
Error	12	276	$\frac{276}{12} = 23$	
Total	14	666		

Table 3.6: Analysis of Variance of Three Placement Strategies

In the worked example 3.3.4 shown above it may have seemed reasonable to conclude that the three placement strategies are not all equally effective. There could be many reasons why this was not a reasonable conclusion. What if the programs using strategy A or C were run at a time when another user was on the system and was using some of the available processors? Of course it is possible that the differences amongst the three sets of results are due to the effectiveness of the strategies, but as we have noted above other factors could be held responsible. The point is that a significance test can only show if differences amongst sample means are too large to be attributed to chance, but it cannot say why the differences occurred.

What if there were some bias introduced by the replicates or by the order in which the experiments were performed? This can be analysed by showing the analysis of variance table in a slightly different form which separates the effects into three separate components. Those due to the errors, the treatments and the replications. We can then see easily where the variation is coming from, whether it really is due to the treatments or just variability or bias between the replications. The new structure of the analysis of variance table is shown in Table 3.7. This new structure of the table requires two calculations for the F distribution test. For the treatments

$$F_{\alpha,n-1,(n-1)(k-1)} \tag{3.13}$$

Source of Variation	Degrees of Freedom	Sum of Squares	Mean Square	F
Treatments	$k-1$	SS(Tr)	$\text{MS(Tr)} = \frac{SS(Tr)}{k-1}$	$\frac{MS(Tr)}{MSE}$
Replicates	$n-1$	SSR	$\text{MSR} = \frac{SSR}{n-1}$	$\frac{MSR}{MSE}$
Error	$(k-1)(n-1)$	SSE	$\text{MSE} = \frac{SSE}{(k-1)(n-1)}$	
Total	$kn-1$	SST		

Table 3.7: Analysis of Variance Summary Table Showing Effect of Replicates

and for the replicates:

$$F_{\alpha,k-1,(n-1)(k-1)} \tag{3.14}$$

3.4 Estimation of Modelling Parameters

Once we have performed the ANOVA it is possible to use the values calculated during the process to create a linear mathematical model. This model allows one to estimate or predict the random variable as a function of several variables. The estimated variable is called the *response variable* and the variables used to predict the response are called *predictor variables*, *predictors* or *factors.*

It is useful to know what is meant by a good or bad model. Figure 3.1 shows three examples of measured data and attempted linear models. The measured data is shown by scattered points whilst the model is shown by a straight line. Most people would agree that the first two models are *good* because the straight line is relatively close to all the data points. Linear regression uses this approach to produce a *good* model.

A regression model attempts to reduce the distance measured vertically between an observation point and the model line (or curve). The motivation for this is that

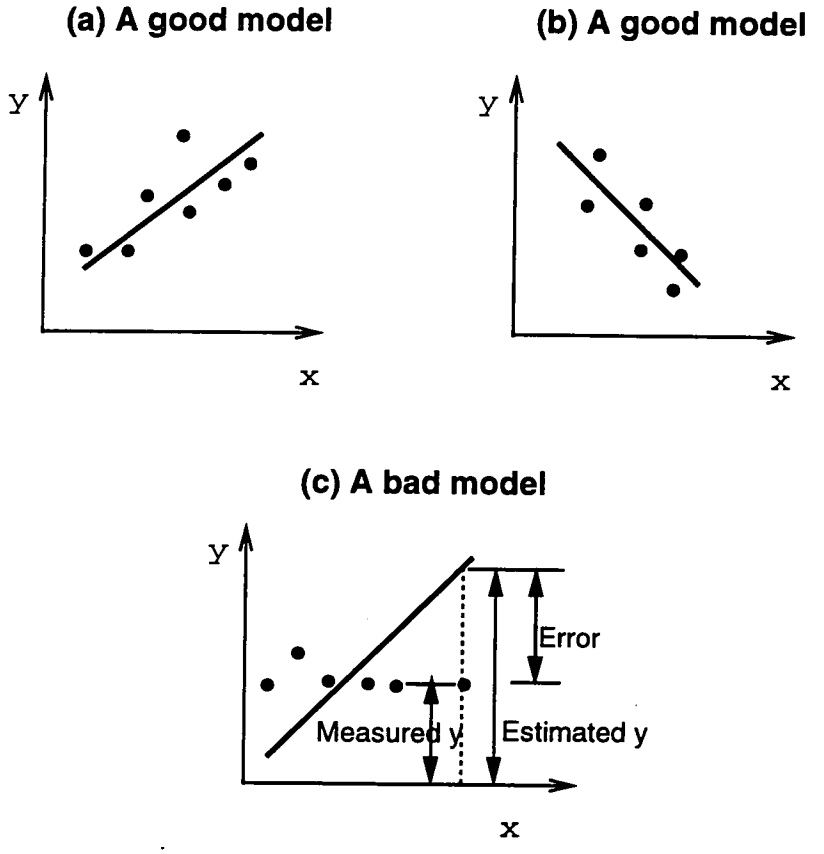


Figure 3.1: Good and bad regression models

given any value of the predictor variable x, we can estimate the corresponding response variable simply reading the y-value on the model line at the given x-value.

The difference in height is called the *residual*, *modelling error*, or simple *error*. The terms residual and error are used interchangeably. Some of the errors are positive and some are negative. One obvious requirement of a good regression model would be to have zero overall error. Unfortunately, there are many lines that will satisfy this criterion. Additional criteria are needed. One such criterion is to choose the line that minimises the sum of squares of the errors. This criterion is called the *least squares* criterion and is the criterion used to define the best linear regression model.

Mathematically, suppose that the linear model is

$$\hat{y} = b_0 + b_1 x \tag{3.15}$$

where $\hat{y}$ is the predicted response when the predictor variable is x. The parameters b_0 and b_1 are fixed regression parameters to be determined from the data (via the steps in the analysis of variance). Given n observation pairs $(x_1, y_1), \ldots, (x_n, y_n)$, the estimated response $\hat{y}_i$ for the ith observation is

$$\hat{y}_i = b_0 + b_1 x_i \tag{3.16}$$

The error is

$$e_i = y_i - \hat{y}_i \tag{3.17}$$

The best linear model is given by the regression parameter values, which minimises the Sum of Squared Errors (SSE):

$$\sum_{i=1}^{n} e_i^2 = \sum_{i=1}^{n} (y_i - b_0 - b_1 x_i)^2 \tag{3.18}$$

subject to the constraint that the mean error is zero:

$$\sum_{i=1}^{n} e_i = \sum_{i=1}^{n} (y_i - b_0 - b_1 x_i) = 0 \tag{3.19}$$

It can be shown that this constrained minimisation problem is equivalent to minimising the variance of the errors. The regression parameters that give minimum error variance are

$$b_1 = \frac{(\sum_{i=1}^{n} x_i y_i) - \bar{x}\bar{y}}{(\sum_{i=1}^{n} x_i^2) - n\bar{x}^2} \tag{3.20}$$

and

$$b_0 = \bar{y} - b_1 \bar{x} \tag{3.21}$$

where $\bar{x}$ is the mean of the values of the predictor variables and $\bar{y}$ is the mean response.

3.4.1 Analysis of Variance – Assumptions

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

- Errors must have constant variance over the entire range of the response.
- Errors must be Independent and Identically Distributed (IID) normal variates with zero mean.
- The effects of factors and errors must be additive. In other words the underlying model must be structurally adequate.

These assumptions should always be tested before proceeding with an analysis. There are a set of visual tests for verifying the above assumptions. The first assumption can be tested by constructing a scatter plot of the residuals versus the predicted response. If a trend is visible in such a plot then one can conclude that the errors do not have constant variance.

The second assumption can be tested by producing a normal quantile-quantile plot of the residuals. 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 the observed data does indeed come from the theoretical distribution. 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 can not be described by a normal distribution with zero mean.

Several situations could lead to the third assumption not being satisfied. Amongst these are if the residuals are of the same order as the response or if the response

covered more than a single order of magnitude.

If any of the above assumptions are discovered to have been violated then a transformation of the data may need to be considered, see for example [Jai91a].

The modelling described here can readily be extended to any number of parameters, though full factorial experiments may become too time-consuming to be feasible as the number of parameters increases, in which case we may use fractional factorial experiments which presuppose that some high order interactions are negligible. The advantage of either approach is that it allows flexibility to explore the relative merits of alternative models as predictors of performance quickly, as a great deal of information can be obtained with a limited input of resources. The approach can be extended to non-linear models by using factorials at three or more levels of some or all parameters.

Due to the amount of effort involved in a full factorial experiment, usually if a series of experiments is to be conducted, the number of factors is decreased from one experiment to the next if it becomes apparent that a factor is having no influence on the outcome.

3.5 Summary

This chapter has described the general technique of computer systems performance analysis and the particular problems presented by parallel systems. It has also described the statistical techniques that are necessary for conducting a proper performance evaluation study.

The performance analysis described here is taken forward in this work by the concept of an experiment and software that allows such experiments to be described succinctly and executed without further user intervention. This "experiment" software is described in the next chapter. Examples of the use of the software to describe experiments and the application of the statistical techniques are given in Chapters 5 and 6.

Chapter 4

A Methodology for Parallel Program Performance Evaluation

4.1 Precis

This chapter describes the main body of work of this thesis. It describes a methodology for conducting performance experiments on arbitrary parallel programs. Many interesting experiments can be conducted but the primary interest is with the identification of factors that affect performance and with their relation to placement strategies.

The problems particular to performance evaluation of parallel systems have already been discussed (§3.2.2). In the first section of this chapter (§4.2) I suggest how these problems can be solved through a systematic approach to parallel performance evaluation. There then follow two specific sections on tools that have been constructed specifically to solve these problems. The *experiment generator* is presented in Section 4.3 and a multiprocessor simulation system, *MIMD* is presented in Section 4.4. A short summary of the details of how an experiment is conducted is given in Section 4.5

4.2 Methodology

The methodology is outlined in terms of the Systematic Performance analysis approach described in Section 3.2.1.

The methodology described in this chapter can be stated very quickly. It is the application of designed experiments to a parallel computer system to identify important performance parameters. A byproduct of this is the ability to predict the performance of parallel programs through the use of statistical models.

Parallel programs on parallel machines are complex systems which have a response based on a large set of factors. The response is usually the execution time - but can be other metrics, such as the processor utilization. The approach taken here is to view the parallel system as a black box, and assume that we have the ability to control the factors (or variables) which enter the black box, read the responses from the black box and construct an empirical relationship from these observations. This is very similar to the approach taken in chemical process control and other physical systems where the systems are too complex to

model in their entirety and so they are modelled empirically as a black box with a set of inputs and outputs.

4.2.1 Goals

The goals of the experiment are to analyse the performance of a parallel system to see which factors are responsible for good or bad performance. The methodology allows for the investigation of an arbitrary number of parameters and an arbitrary system, although in practice there are limits imposed by the time required to carry out a large number of experiments.

4.2.2 Define the System – Synthetic Programs

At the outset of this study it was necessary to define the parallel system under study and identify workloads. It became apparent that there were not sufficient existing suitable parallel programs to permit a systematic performance study. The decision was taken to use artificial workloads, which I have termed synthetic programs. Stated simply such a synthetic program is a representation of a process based program that is intended to capture the essential qualities of a computation.

There were other compelling reasons that made the choice of synthetic programs sensible. Firstly, though some real programs did exist, it is always difficult to understand a program fully if you have not been involved with its construction. But if the program is to be run in an instrumented fashion then an understanding of the program is necessary. At the start of this investigation it was also very

difficult to instrument a program and run it with different parameter settings automatically. This was particularly true on the machine and operating system combination that was available.

The author felt that if program performance parameters were to be seriously investigated then a system was necessary to create programs of specific types on demand. This must have the ability to create programs with factors at specified parameter levels.

If only existing programs were used then there would be a degree of uncertainty about how representative those programs were of the more general class from which they were taken since they were in general tailored to a given machine environment. A synthetic program, and the ability to create programs on demand at particular factor levels, gives the investigator the ability to fully explore classes of programs through designed experiments.

When considering the creation of the artificial workloads or synthetic programs it is necessary to identify what is necessary and sufficient to describe a parallel program. From a programmer's point of view the most important factors may be the number of processes in the program and how they are connected together. This is expressed in the size, shape and regularity of the program graph. For each process in the program the ratio of the amount of computation to communication is important as well as the relative sizes of processes. Considering these points led to the following definition of synthetic programs.

A synthetic program is made up of two parts, a weighted graph similar to that described in Section 2.7.4 and one or more process templates. The process graph gives the shape and connection of the processes in the program along with a set of weights for the processes and edges. The weights of the process graph are

meaningless until a process template is added. The process template describes the behaviour of the processes in the program. Each process template is responsible for *bringing to life* the weights associated with the graph.

Each node must be given a process template, but the same process template need not be given to each node in the graph. The weights associated with the edges radiating from a node are interpreted by the process templates associated with that node in appropriate ways.

This weighted process graph provides a way of describing a program independently of how the program is to be executed. This is an important point in the design of the methodology. The process templates are usually dependent on the experimental domain. For example, a process template could be part of a simulation program or it could describe a process in a real program.

4.2.3 Example Synthetic Program

A complete synthetic program is shown in Figure 4.1. This shows a simple program graph with just two nodes and a process template written in pseudo code. Each process in the program computes for a period specified by the weight of the node and then exchanges data with its neighbour. Note that the send and receive of the messages is performed in parallel and so the program will not *deadlock*. This is signified by the indentation of the statements beneath the `PAR` statement in a notation that is reminiscent of the Occam programming language [Inm89a]. The `SEQ` statement means that all lines with the same indentation below this statement are executed sequentially like a normal programming language.

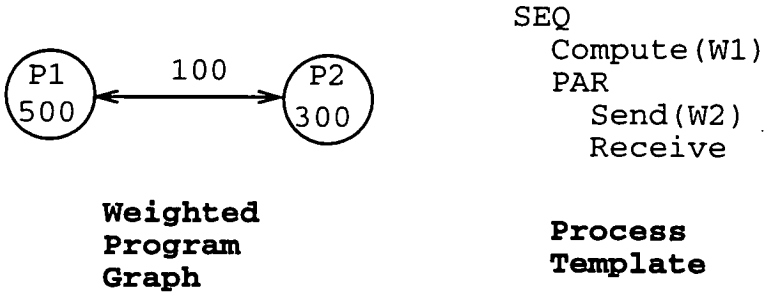


Figure 4.1: A Simple Synthetic Program

This synthetic program describes a program where one process, P2, will spend a large majority of its time waiting on process P1 to finish its computation and communicate. This process graph and the same process template could be executed in a simulation system or as a real program running on a real machine.

4.2.4 Performance Metrics

The choice of metrics for parallel systems has already been discussed in Section 3.2.2. All metrics are suitable as long as they can be measured accurately, and provide useful feedback to the programmer. The most useful metric is usually wall clock time of program execution. This gives the speed of the system. The second most common metric is processor utilization. This gives a measure of how well the parallel processing resource has been used. The two metrics are obviously related, for example, if for a fixed number of processors and a sample program, the execution time can be decreased by better utilizing the processing resource available. This is usually achieved by a better placement of the processes or a reordering of communication events.

Using synthetic programs does lead to a difficult choice of performance metric. The use of synthetic programs (possibly randomly generated) means that the

process templates have to be constructed in such a way that the program will not deadlock. This also produces programs that do not terminate as the process templates have a repeating nature. With programs that do not terminate there is no concept of wall clock time. To get around this problem I will use the total number of clock cycles computed by the program in the fixed simulation time as the metric. Where appropriate I will show this as a percentage of the total amount of time available. The total amount of computation time possible is the product of the simulation time and the number of processors.

4.2.5 Factors and Their Values

The methodology allows for an arbitrary number of parameters to be specified to the experiment generator. These parameters are then combined in a full factorial experiment. If each factor is taken at two levels the number of experiments is given by

$$\text{Number of Experiments} = 2^k r \tag{4.1}$$

where k is the number of factors and r is the number of replicates of each experiment.

The particular factors to be investigated are chosen by the user. Simple obvious factors have been chosen for the initial experiments, e.g. the number of processes in the program, the average connectivity of the processes, the average weight of the processes and the weight of communication.

4.2.6 Evaluation Technique

The evaluation technique for the methodology can be chosen by the user. All that is required is that the evaluation process takes a description of a parallel system, program graph, hardware description and mapping, and executes the program, returning the requested metrics.

As previously discussed, the evaluation technique can be analytical modelling, simulation or measurement on a real system. The methodology will work with any of these evaluation techniques. It has been successfully used with simulation and real systems.

The actual execution of parallel programs on real parallel systems is sometimes a long and tedious process and it is not always possible to measure the desired metrics. To get around this problem most of the evaluation techniques used were simulation based and real programs were only used for validation purposes.

4.2.7 Select the Workload

Finding standard workloads for parallel systems is not an easy task. This is because of the diversity of the systems and the diversity of programming languages used. It is difficult to get a representative sample of programs and for this reason, as well as those previously described I chose to generate synthetic programs. This meant that a large range of programs that were easily available and were simple to understand were accessible to me. Other reasons for this decision have also been outlined in Section 4.2.2.

The actual generation of the synthetic programs graphs is dealt with in more detail in Section 4.3.3. There is one peculiarity with using synthetic parallel programs. It is necessary to ensure that the programs are deadlock free. This is based on the assumption that the programs under investigation from a real system would be deadlock free so any synthetic programs should also be deadlock free. The assumption is made because this work is aimed at performance enhancement of "correct" programs not debugging of developing programs. Fortunately the class of programs described as loosely synchronous in Section 2.6.2 are most easily written in a deadlock free manner. This is achieved by having two main phases, one of computation and one of communication. The communications occur in parallel, much like the simple process template shown in Figure 4.1

4.2.8 Experimental Design

The experimental design is based on a full factorial statistical experiment. To help the user of the methodology an experiment generation tool called **eg** was created (described in Section 4.3). This allows the user to clearly specify the parameters under investigation and what levels they would take. The output from the experiment generator is a set of experiment trials (or experiment instances) which can then be evaluated within an experimental domain. The experimental domain can be a simulation environment or a real parallel processing system.

The second tool that has been developed is a simulation environment called MIMD for simulating arbitrary parallel programs on distributed memory MIMD computers. I will now describe these two tools in more detail.

4.3 The Experiment Generator

Before describing the experiment generator we must first make some definitions clear.

Experiment An experiment is the task of investigating the effects of certain parameters characterizing a parallel program or its hardware environment on the system performance. The experiment designer must have a clear idea exactly what it is they want to investigate before starting the experiment. It should also be noted that nothing can be inferred about any parameter which does not vary during an experiment. One experiment consists of one or more execution instances (or experimental trials).

Experiment Design The experiment design states in a formal manner what parameters are to be varied in the experiment. The experiment design is given in a language which is interpreted by the Experiment Generator Tool (**eg**).

Experiment Design Language This language is used to articulate the experimental design in a way that is intelligible to a human being and is also input to the experiment generator program **eg**. This language is fully defined in the program documentation for **eg** [Ski92] and the syntax for the language is described in Appendix A.

Execution Instance An execution instance is one particular setting of the variable parameters described in the experimental design. An experiment instance is the input for the modelling engine and corresponds to one execution of a parallel program.

Modelling Engine The modelling engine takes one experiment instance and interprets it in a meaningful manner. It will produce results that can be

processed and combined as part of the results of the whole experiment.

An experiment design describes a set of program graphs, a set of processes and communication channels connected with this graph, and a set of mapping strategies that will be used to map the programs to the hardware.

From this description of the experiment a set of individual experiment instances are created. An experiment instance consists of a particular program graph, a particular hardware graph, and a process mapping.

The program is described by a weighted graph as previously discussed. The meaning for these weights is provided by the modelling engine. The weighted graph can be generated by the experiment generator or it can be a user defined graph.

A hardware graph is described by a set of nodes and links. This can be one of the types of hardware configuration provided by **eg** (e.g. hypercube, mesh) or a user defined type. These must of course be supported by the particular modelling engine which is being used.

The process mapping is just a simple onto map from the set of processes to the set of processors. This mapping is generated by one of the mapping strategies provided by the experiment generator. The user can add their own mapping strategies to the system.

An experiment instance is domain independent because it is just a description of a parallel program through the use of a weighted graph and other values - it does not specify how these weights are interpreted, as there is no program code in an execution instance.

The execution instance is designed so that it can be taken by a modelling engine which will execute that particular instance in a particular modelling domain. This gives the user a certain amount of flexibility. Different process models can be used for the nodes of the weighted graph. The weights can be interpreted in different ways by different process models. It also allows for similar experiments to be conducted in different domains. For example it is possible to write a modelling engine in the MIMD simulation language (several have been written) but it would also be possible to write a C or Fortran program using Meiko's CSBuild environment [Mei92] (or similar multiprocessor programming system) to do the same thing.

The steps of this experiment are shown graphically in Figure 4.2. **eg** provides the left hand side of the diagram by generating the set of execution instances. The modelling can be provided by a variety of systems, though mostly it has been provided by MIMD and the statistical analysis is performed either by using standard statistical packages or by use of a tool developed by the author for this specific purpose.

4.3.1 An Example Experiment using the Experiment Generator

Imagine that we have a parallel program and would like to investigate what would happen to the program if we made some changes to it. Our program is a regular grid decomposition program with boundary swapping. Each process computes for a fixed time step and then swaps values with its four neighbours. We want to investigate what will happen if instead of swapping data at the end of every time step we swap data at the end of every other time step. This requires us to

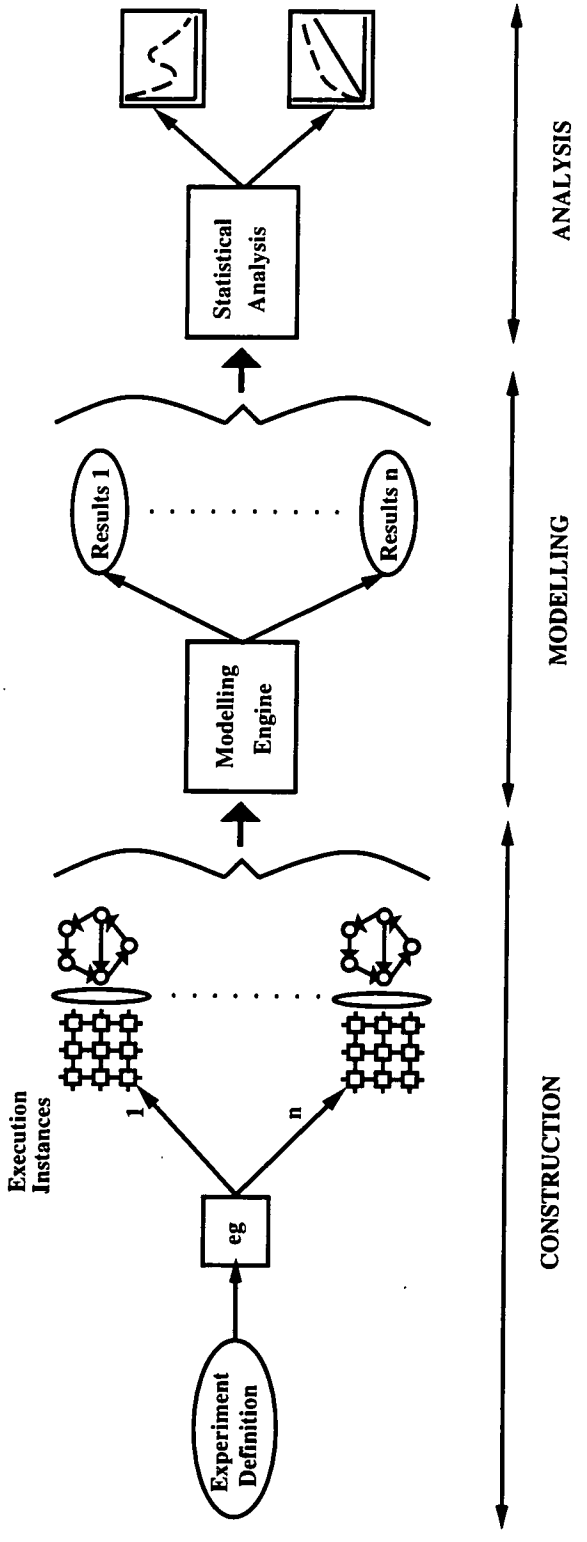


Figure 4.2: The Experimental Framework Provided by **eg**

send larger packets at the end of two time steps. Let us say for simplicity that these packets have to be twice as large.

That is our experiment design. Now we need to write this down in a formal manner that the experiment generator can understand. First of all we need to define a process. This process will have one parameter which will define the amount of computation that the process performs before swapping boundary values. Our experiment design says that this parameter is to take two values, one being twice as large as the other.

This can be written in the experiment design language through the use of a process template, as follows:

```
Begin Define Processes
  Process1 { int compute-time [ 1000, 2000 ] }
End Define Processes
```

This states that we have defined a process called `Process1` which has one integer valued parameter called `compute-time`. This parameter is to take two distinct values over the experiment. These values are 1000 and 2000 units. The units are meaningless at this stage. They only have significance to the modelling engine.

This process definition has shown one of the major features of the experiment generator, namely the varying of parameters over the experiment. If no other parameters were varied in this experiment, then this would create exactly two experiment instances. These would be identical except for the value of the parameter `compute-time`.

Following the process definition we need to give communication channel defini-

tions with which to link together these processes. In a similar manner we can define a channel with one parameter. This parameter will specify the size of the data packet to be sent down this channel. Our experiment design requires this to take two values, one double the value of the other. This could be written as:

```
Begin Define Channels
  Channel1 { int pkt-size [ 50, 100 ]  }
End Define Channels
```

That describes all the process and channel types that are necessary for this example experiment. Obviously, more than one process or channel type could be defined if necessary.

It is now necessary to specify the shape of the program graph. The program is a regular graph of degree four. The experiment generator will randomly generate such a graph. Random graphs can be generated in a variety of ways. See Section 4.3.2. We also need to describe the hardware on which we want our program to be modelled (simulated). We can use one of the topologies provided by the experiment generator for this. We want to run our program on a ring with twenty nodes. This would lead to the following description:

```
Begin Graph Parameters
  Graph Type Redfield
  Degree 4
  Number Nodes 40
  Hardware Ring 20
End Graph Parameters
```

Now all that is left to describe is how to map the program graph onto the hardware

graph. To do this one or more placement strategies provided by the experiment generator is specified. These strategies are given integer identifiers. A full list of these strategies can be found in documentation for **eg** [Ski92]. In this case placement strategy number one, a round robin process allocation is used. This is described as follows:

```
Begin Placement
  Algorithm 1
End Placement
```

Figure 4.3 shows all the separate parts of the design combined together to make the complete experimental design for our example investigation.

Three extra sections have been added to complete the design. The section entitled `Allocate Processes` tells the experiment generator that all the processes in our random generated graph are to be identical and they are to be of type `Process1`. A similar specification is done for the channels. There is also a section entitled `Modelling Parameters`. These are parameters which are important to the modelling engine, such as the total simulation time, and do not affect the generation of the experiment instances.

Now that a full experiment design is complete, it can be presented in a computer file to the experiment generator program. This would then generate all our experiment instances. It does this by creating random experiment instances for a full factorial experiment based on all the parameters which are varying. In our case, only two parameters are varying, at two levels. This gives four experiment instances to be passed to the modelling engine.

```
Begin Experiment
  Begin Define Processes
    Process1 { int compute-time [ 1000, 2000 ] }
  End Define Processes
  Begin Define Channels
    Channel1 { int pkt-size [ 50, 100 ] }
  End Define Channels
  Begin Graph Parameters
    Graph Type Redfield
    Degree 4
    Number Nodes 40
    Hardware Ring 20
  End Graph Parameters
  Begin Allocate Processes
    Map Process1 to 100 percent
  End Allocate Processes
  Begin Allocate Channels
    Map Channel1 to 100 percent
  End Allocate Channels
  Begin Placement
    Algorithm 1
  End Placement
  Begin Modelling Parameters
    .......
  End Modelling Parameters
End Experiment
```

Figure 4.3: Example Experimental Design

4.3.2 Experiment Design Language

Some of the experiment design language has been described in the previous example. This section describes the language and the interpretation of it in more detail. The experiment design can be split into two parts - there is the part concerned with the experiment instances and there is the part concerned with the modelling engine. The modelling parameters tend to be domain dependent and usually do not vary.

Experiment Instance Parameters

Varying Parameters It is through the varying of parameters in the experiment design that the user generates a set of execution instances. If parameters are not varied then just one execution instance is generated. This is useful if you want to test your modelling engine on a single execution instance or if you want to make observations on a specific known program.

Parameters are varied by specifying more than one value for them in the experiment design. A parameter can either be a fixed value, a range of values with a constant difference or a list of disjoint values. A parameter which takes a constant value or a list which contains only one element are essentially the same thing. A parameter which only takes one value is not strictly speaking a varying parameter.

Note that it is easy to generate a lot of experiment instances, but the instances are never generated until you need them. The experiment generator will therefore inform the user of how many experiment instances it intends to create and will ask if it should go ahead.

Program Graph This can either be a user defined graph or it can be generated for the user by a random graph generation technique. After the graph has been generated the weights are added to the graph.

If the graph is entered by the user then that graph size and shape is fixed for the set of experiments. If the graph is to be generated randomly, then the degree (the number of edges per node) and the number of nodes can be one of the parameters for the experiment.

4.3.3 Random Graph Generation Strategies

For the creation of synthetic programs, introduced in Section 4.2.2, it is necessary to have some graph generation algorithms. The experiment generator contains three example graph generation strategies; two regular graph generation strategies and an irregular random graph generation strategy. The graphs produced by these strategies give the shape for the synthetic program into which the program parameters are laid. Two regular graph generation strategies were provided because though one was quicker: it was limited in the degree of the graphs that it could create.

The strategies are now introduced in turn. Some sample graphs generated by the three strategies are shown in Figure 4.4

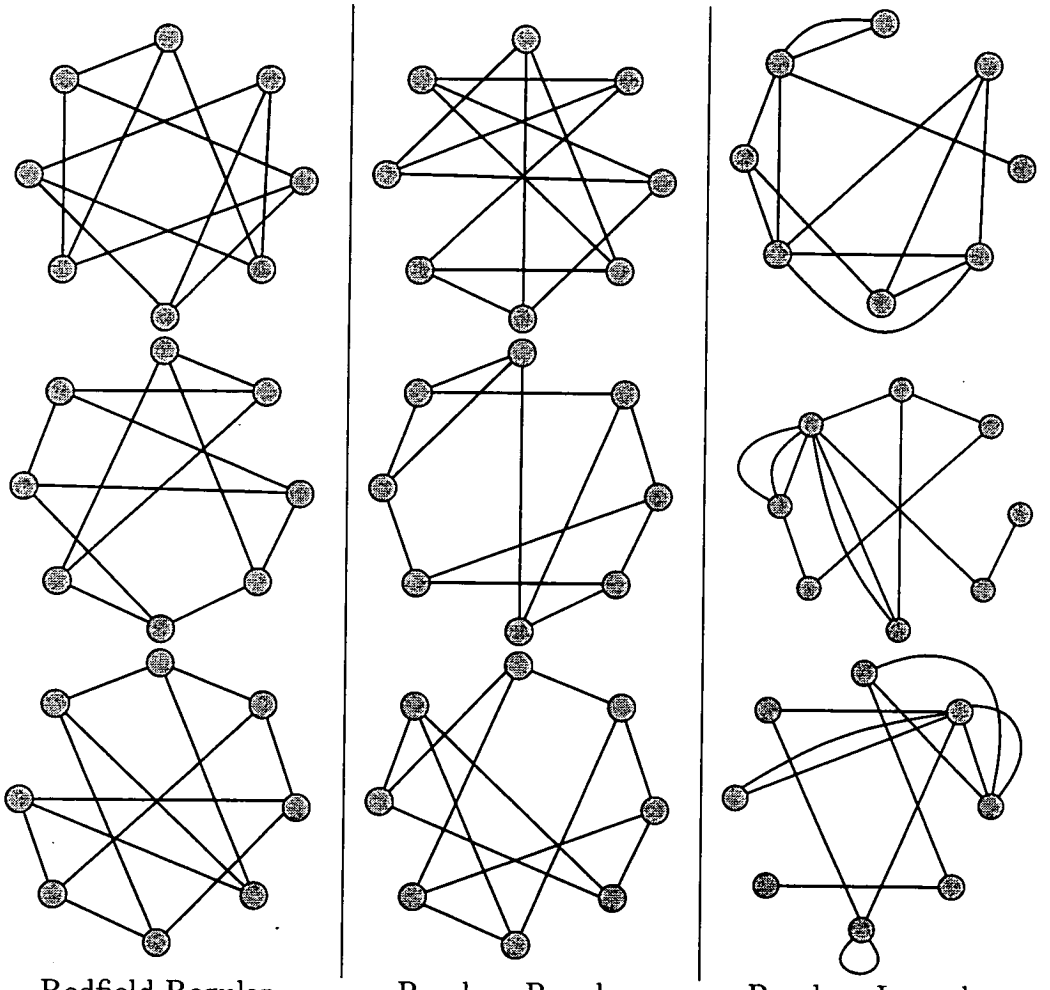


Figure 4.4: Example Regular Graphs, 8 Nodes, Degree 3, implies 12 Edges

Random Regular Graphs

A *k-regular* graph is a graph where each vertex has the same number of edges. There are some simple properties that follow from this definition.

- For a n-vertex k-regular graph the number of edges is $\frac{nk}{2}$.
- For a n-vertex k-regular graph, both n and k can not both be odd.
- The degree k must be less than the number of nodes n.

Random Regular Graph Generation due to Redfield This algorithm due to Redfield [Pal85] can be used to generate a k regular graph for any k, $3 \leq k \leq 5$. This is a straightforward approach which makes connections randomly between the nodes, ensuring that the specified degree of the graph is not exceeded.

A free list of nodes that have a degree less than the desired degree must be maintained to ensure that the graph is generated in a reasonable time. The algorithm is not given here in detail as it is fairly straightforward.

Random Regular Graph Generation due to Jerrum and Sinclair This algorithm due to Jerrum and Sinclair [JS88] creates a k-regular graph of any degree. The algorithm first creates a graph of the required degree deterministically and then manipulates it randomly for a number of steps. To ensure that the graph has reached uniformity the number of steps usually chosen is n^2k^2. This also gives the timing characteristics for the algorithm. The algorithm is given in more detail in Figure 4.5

```
Input n, k and Number of Steps

Generate Deterministic k-regular graph

/* Manipulate Graph Randomly */
steps = 0
finished = FALSE
while not finished
  if graph is k-regular then
    remove edge selected at random
  else if one vertex has degree k-2 then
    find non-adjacent vertex and remove edge
    connect vertex to non-adjacent vertex
  else /* Graph has two vertices of degree k-1 */
    if random(0:1) > 0.5 then
      connect the two vertices to non-adjacent nodes
      disconnect an edge from the two newly connected nodes
    else
      connect the two vertices together
      if ( steps > Number of Steps )
        finished = TRUE
      end if
    end if
  end if
  steps = steps + 1
end while
```

Figure 4.5: Algorithm for Generation of k-regular Graph due to Jerrum and Sinclair

```
Input  n,and  k

Number of Edges = nk/2

 for i=1; i <= Number Edges; i++
   x = random(1:n); y = random(1:n)
   Define edge connecting  x and  y
```

Figure 4.6: Algorithm for Generation of Irregular Graph

Name	Parameter(s)	
Mesh	Length	Width
Hypercube	Dimension	
Pipeline	Length	
Ring	Size	
Star	Size	
Tree	Depth	Arity
Random	Size	Links

Table 4.1: Built-in Hardware Types

Random Irregular Graph Generation This algorithm was devised by the author to create a random irregularly connected graph that has a specified average degree. The algorithm is outlined in Figure 4.6. The number of nodes is specified and a degree is given which is taken to be the average degree of the graph. From this the number of edges is calculated and then these edges are connected to nodes at random. The nodes are selected with replacement so it is possible to have an edge connecting a node to itself. This is not permitted in the previous two algorithms.

Hardware Graph Only one hardware graph is allowed per experiment at the moment. This restriction is based on the author's interests and could be changed. The user can ask for one of the built-in hardware types or can enter a particular wiring configuration by using a user defined graph. The available hardware topologies are shown in Table 4.1.

```
Begin Define Processes
  Fred { int  Parameter1 10
         double Parameter2 20 to 30 step 0.5
       }
  ...
End Define Processes
```

Figure 4.7: Process template in the Experiment Design Language

Placement Algorithm A number of placement algorithms have been implemented. At least one of these placement algorithms must be chosen by the user for each experiment. The current placement algorithms are those used in the experiment described in Section 6.3. These placement algorithms have all been coded in the 'C' programming language and more can be added to by users of the experiment generator. This is described in the documentation [Ski92] where the placement strategies are also described in more detail. A description and investigation of several placement strategies is given in Section 6.2.

Process and Channel Weights The user is allowed to define a number of process templates in the experiment design. The user also controls how these processes are actually allocated to the program graph specified above. A process template has a name and a set of named numeric parameters. The numeric parameters can be of type *integer* or *double*. An example process template in the experiment design language is shown in Figure 4.7. This shows a process which takes two parameters one of which is varied by the use of a lower and upper bound and a step size. It has already been shown how a fixed set of levels can be used instead. The other is constant. This process definition alone would generate twenty experiment instances. The channel weights are described in a similar fashion.

This process template is interpreted by the modelling engine. This is described in Section 4.3.5.

Assigning Templates to the Software Graph Process and channel templates are randomly assigned to the software graph through the use of a percentage specification. Say, for example, that we wanted to assign the process defined in Figure 4.7 to half of the process nodes, then the following would do that:

```
Begin Allocate Processes
  Map fred to 50.0 percent
  Map bert to 50.0 %
End Allocate Processes
```

This also allocates a process called `bert` to the other half of the nodes in the graph. The word "percent" or the symbol '%' are equally valid.

Modelling Parameters

The other parameters are all fixed value parameters. These are used to control the execution of the experiment instances and are usually domain related. For instance, for modelling through the MIMD simulation environment a total simulation time, the name of the modelling engine executable and the amount of swap space to be used are all given.

4.3.4 Experiment Generator – Summary

To summarise the functionality of the experiment generator: it allows the investigator to construct experiments to investigate the effect of parameters on performance. In particular, it allows the investigator to study the performance of randomly generated programs from a given class of parallel programs. The class of programs is defined by the process templates and the random selection of particular instances from that class is performed by the experiment generator. The parameters can be varied to include real programs within that range.

It will be shown that the randomly generated programs behave in a similar way to real programs (§5.3.1) and so the results for this random selection imply that the results will be valid for all programs of this class. This would not necessarily be the case if a fixed set of test programs were used. It would be difficult to justify an inference about any other programs except those in the test set.

The experiment generator allows the investigator to choose parameter levels and conduct experiments and then apply standard statistical tools to the results. This would not be possible with a single fixed existing program in the same way.

4.3.5 The Modelling Engine

A modelling engine must be able to take an experiment instance as input, generate an executable program, execute the program and then report its results.

The modelling engine can be a real parallel program which is able to emulate the parallel computation specified in the execution instance or it can be a simula-

tion engine which takes the instance and simulates the activities of the program executing in the particular environment specified in a statistical sense.

The modelling engine also provides the process definitions for the nodes on the graph. An execution instance just defines the shape of the graph and a set of weights associated with the nodes and edges. The modelling engine describes how each of these weights is to be interpreted. In this way the same execution instances can be used in different domains as well as in the same domain by different modelling engines.

The modelling engine must be constructed when the experiment is designed. The experiment design language allows different types of processes to be defined and this information is passed through to the modelling engine in a format described in Appendix A.

4.4 The MIMD Modeling System

MIMD is a discrete event simulation package tailored specifically towards the modelling of process based computations on distributed processor systems. It is built on top of the DEMOS (Discrete Event Modeling on Simula) package [Bir86] and hence on top of the Simula programming language [Com87]. MIMD runs on workstations for which a Simula compiler is available.

MIMD is not itself a parallel program. This allows the simulation system to have global system knowledge about the simulated parallel system. This is difficult to achieve for a parallel simulation system. Simulation of individual experiment instances can take advantage of multiple workstations as each instance is inde-

pendent.

MIMD is the usual acronym for Multiple Instruction Multiple Data stream parallel computers and hence was the obvious name for a simulator of those type of machines.

The investigation of performance factors affecting parallel programs requires a general purpose simulation environment. When the investigation was started, it was unknown which parameters were to be investigated, or in what detail, and so the flexibility and power of a programming language was needed.

4.4.1 MIMD – A Brief History

At the time this project started, the University of Edinburgh had just installed its first distributed memory multicomputer. This was an Inmos T800 Computing Surface from Meiko which grew to house 400 processors and came to be known as the Edinburgh Concurrent Supercomputer (ECS) [1].

The ECS had no operating system as such. All programming had to be done in Occam 2 [Inm84, Inm88] which was based on the work of Hoare and his Communicating Sequential Processes or CSP [Hoa84]. All programming, compiling, and running of programs was performed in the Transputer Development System.

A CSP based simulator package already existed [Gui88]. The author inherited this package on his arrival at Edinburgh and decided to use it as part of the project. Major changes were made to the message passing, routing and topology

[1] Sadly, this machine was withdrawn from service in July 1994.

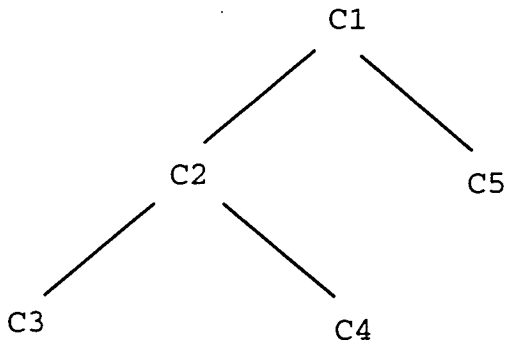


Figure 4.8: Example of Class-Based Inheritance

code for both "real" and "simulated" message passing. The process and subprocess models were completely rewritten. Most of the three and a half thousand lines of Simula code have been rewritten at some point in the last five years.

The Simula programming language

Simula is a general purpose programming language with concurrency implemented by co-routines. It was the original language to introduce the class construct, which made it the first object-oriented language. The class construct is an extension of the abstract data type that allows procedures and variables to be held in a single module.

Classes can be declared in a hierarchy where classes lower in the hierarchy inherit all the procedures and variables of the classes above them. For example in Figure 4.8, C_3 inherits from C_2 which inherits C_1's procedures and variables.

A class object is created by the statement

```
new <class name> ( <parameters> )
```

Pointers are used to access the values and procedures in the class structure. A pointer in Simula is known as a reference. So for example

```
ref(my_class) my_ptr;
```

means that `my_ptr` is a pointer or reference to a class of type `my_class`. Simula also provides the ability for a class to detach or suspend itself and then resume itself later. This ability makes Simula far more applicable than most programming languages for simulation.

Simula also provides the ability to write packages that can be separately compiled, then prefixed in a Simula program. Then the program inherits all the concepts defined in the package. This was the technique used to create the MIMD package.

For a detailed description of the Simula programming language see Pooley [Poo85] or Birtwistle [Bir86].

4.4.2 The DEMOS system

DEMOS is an acronym for Discrete Event Modeling on Simula. It is implemented as a Simula prefix block and can be separately compiled to the user's program. To use DEMOS the user simply prefixes their main program with `DEMOS` and then the program inherits all the procedures and classes defined in DEMOS. The user's program may declare new classes as being sub-classes of a DEMOS class and they will inherit the attributes of that DEMOS class.

Some classes provided by DEMOS are: `entity` which is a basic modelling pro-

cess, `res` or `bin` which are finite resources, and `waitq` which is an entity-entity synchronization device. Together with these, DEMOS provides a set of data collecting classes like histograms and tally tables.

DEMOS provides an internal event list and software to maintain it. Users' programs can schedule entities and have them compete for resources. DEMOS also provides a queuing mechanism for holding entities outside the event list. As well as this, DEMOS provides random number generators for various statistical distributions.

DEMOS keeps a reference table of all the jobs it creates and at the end of the simulation provides an automatic reporting mechanism for all these objects. Since DEMOS is completely written in Simula, all DEMOS programs inherit all Simula properties. For an introduction to DEMOS see Birtwistle [Bir86].

The Structure of MIMD

The implementation of the MIMD model has been directed towards providing an application package that extends the DEMOS modelling system into the specialized area of modelling distributed computation.

The prefix classes `DEMOS` and `MIMD` are separately compiled and the MIMD class is used as a context just as the DEMOS class is used as a context. To use the system, the user's program must be prefixed by the `MIMD` block. A template for a typical user's program is given in Section 4.4.2 where a demonstration program is analysed in detail.

Note that the user's program only requires to prefix the MIMD class since the

MIMD context inherits all the attributes of the DEMOS system. The MIMD class contains all the tools which the user's program may need to simulate a distributed computation.

MIMD provides the performance modeller with a set of high level class definitions with which to model process based computations. These are the process and channel entities, hardware components (processor and link resources) and the means to combine, manipulate and refine these components through the use of the Simula language.

Within the process construct four main primitives are provided for the performance analyst. These are: compute, sleep, send and receive. These are based on the CSP model of computation [Hoa84]. Though MIMD does not follow this model of computation exactly, it simulates the main features i.e. there can only ever be one message outstanding on a communication channel and messages use blocking send calls for message passing. MIMD also provides the modeller with a process to process communications harness. This was provided in software on the ECS by several message passing harnesses, and MIMD can be programmed to simulate their operation. The modeller does not have to explicitly place process channels onto hard wire links. The use of the link resources is performed by the (simulated) communications harness. The communications harness is a straightforward table look up routing strategy. This is possible within MIMD because it is a sequential simulation environment and so global knowledge of the states of all entities within the simulation can be known. This opportunity for global knowledge is not open to real parallel systems and is difficult to obtain for distributed simulation systems.

The performance modeller constructs a Simula program using the MIMD classes. The performance of any real parallel program can be represented within the MIMD

system using the above constructs. The program can be a complete model of a real program - even including branching and looping due to the fact that it is coded in the Simula programming language.

The MIMD system provides a simple hardware model similar to the Inmos Transputer family [Inm89a]. This assumes that there is a hardware scheduler and so usually there is no operating system overhead to be built into the model although it has been extended to support process migration [Phi94].

Performance characteristics are available from within the MIMD system by using the DEMOS data collection devices (histograms, charts) or by using user-defined collection routines. The performance of any part of the simulated program or hardware is open to scrutiny and subsequent analysis.

An Example MIMD Program

The easiest way to get an impression of the MIMD language is to look at an example program in some detail.

Program Description

This section describes an example program which can be used to demonstrate the package. The simple program has two communicating processes. The first process continually sends and receives a message. The second process continually receives a message, computes and then sends back a reply. This is a simple deadlock free program.

The example program is shown in Figure 4.9. The line numbers have been added for descriptive reasons and would not usually be present.

The whole of the program is wrapped in a block of code like this

```
Begin
External Class MIMD;
MIMD
Begin
.......User's Program Here........
End;
End;
```

This wrapper is used for every MIMD program.

Lines 8-11 are standard declarations that are used for most MIMD programs. These declare pointers to objects that are necessary to build a simulation model. These pointers are filled in further on in the program with objects of suitable types. Here we are declaring a pointer to a `Domain`, a `Program`, a `Pattern` and a `Hardware_description` object. Line 12 declares a pointer to one `Channel` object.

Now we need to describe some subprocesses. We have two types of subprocess in our example program. One is called `SendFirst`, the other `RecvFirst`. They continually send a packet of size 512 bytes backwards and forwards along a channel. RecvFirst computes for 100 time units in between receiving the packet and sending it back. SendFirst is declared in lines 14-21. The `While TRUE` construct ensures that this process will never terminate. Note the passing of the channel as a parameter to this subprocess. This shows the flexibility of using a programming language to describe these models.

```
 1  Begin
 2
 3  External class MIMD;
 4
 5  MIMD
 6  Begin
 7
 8  ref(Domain) Seat;
 9  ref(Program) Prog;
10  ref(Pattern) Wiring_structure;
11  ref(Hardware_Description) ECS;
12  ref(Channel) Comm_Chan;
13
14  Model1_Subprocess Class SendFirst(C1);
15  ref(Channel) C1;
16  Begin
17  While TRUE do
18  Begin   Send(C1,512);
19          Receive(C1);
20  End;
21  End;
22
23  Model1_Subprocess Class RecvFirst(C1);
24  ref(Channel) C1;
25  Begin
26  While TRUE do
27  Begin Receive(C1);
28        Compute(100);
29        Send(C1,512);
30  End;
31  End;
32
33  Ref(SendFirst) Send;
34  Ref(RecvFirst) Recv;
35
36  ECS :- New Hardware_Description("ECS",5000,0,3/40,0);
37
38  Wiring_structure :- NewPatternMesh("Meiko",2,2);
39
40  Seat :- New Domain(Wiring_Structure,ECS);
41
42  Prog :- new Program("Message", 2);
43
44  Comm_Chan :-Prog.DeclareChannel(1,2);
45  Send:- New SendFirst("SendFirst",Comm_Chan);
46  Recv:- New RecvFirst("RecvFirst",Comm_Chan);
47
48  Prog.DeclareSubprocess(1,Send);
49  Prog.DeclareSubprocess(2,Recv);
50
51  Seat.Load(1,Prog,1);
52  Seat.Load(2,Prog,2);
53
54  Start(Seat,Prog,30000,FALSE);
55  Reporting(Seat,Prog);
56
57  End;
58  End;
```

Figure 4.9: The Demonstration Program

RecvFirst is described in lines 23-31. Note the difference between the two subprocesses. RecvFirst has a `Compute` statement in it. It should also be noted that this period of computation does not have to be a fixed period. It could be a sample from a distribution. For example the lines

```
Ref(RDist) S;
S :- New Normal("N",5,1);
```

declare a normal distribution of mean 5 and variance 1. To compute for a period selected from this distribution then the code is

```
Compute(S.Sample);
```

It should also be noted that the subprocesses are not limited to infinite repeat loops. Any Simula programming structure can be used to characterize the subprocess. Lines 33 and 34 declare two pointers one to each of the new subprocesses just defined.

The next task is to describe the objects for the simulation. Line 36 describes the hardware on which the program will run. It says that processor type will be *ECS*. These processors will have a timeslice period of 5000 units, a soft channel transfer time of zero, a hard channel transfer ratio of 3/40 and a strategy computation time of zero. These numbers are examples chosen to be close to that for an Inmos T800 Transputer. See The Transputer Data Book for more details [Inm89a].

Line 38 says that the processors will be connected in a two by two mesh. Note that the word `New` is juxtaposed to `PatternMesh`, this is because it is an MIMD

procedure and not just a Simula `New` statement. `MIMD` provides several built in patterns of processor connectivity. The programmer is also able to describe any arbitrary connectivity.

Next the *seat* is declared, this is a set of processors linked together in a certain fashion. The term *seat* is a historical term relating to the fact that you used to have to physically sit at the computer in front of the processors you were using. So to declare this you need say what processor type (ECS) and what pattern is being used (i.e. `Wiring_Structure`). This is done on line 40. This concludes the description of the hardware. Next comes the software.

A modelling entity *program* called `Message` is created on line 42 which is to have two separate processes. There is a communication channel which is to link processes 1 and 2 declared on line 44.

Lines 45 and 46 declare two subprocesses one of type RecvFirst and one of type SendFirst. They both take `Comm_chan` as a parameter. All that now remains is to declare these processes to the program. Lines 48 and 49 do this.

The hardware and software specifications for the simulation are now complete. All that remains to do is to load the software up onto the hardware and run it. The loading is achieved by lines 51 and 52. This says load process 1 which is to be found in `Prog` onto processor 1. The same for process 2 and processor 2. Note we are only using two out of our four processors. To start the simulation the subroutine `Start` is used. This starts `Prog` executing on `Seat` for a simulation time of 30000 time units with tracing turned off.

After the simulation, reports on the hardware and software are printed out. This gives a lot of data and it is not envisaged that the user would use this procedure

all the time, but would be more selective in using report procedures. Most objects in MIMD have reporting procedures.

The reporting is verbose and statistics about all entities and resources in the system are available. Usually some subset of the available statistics are used. The most popular statistics for performance analysis are the processor utilization and the number of computation cycles performed.

4.4.3 MIMD – Summary

The MIMD modelling system provides the performance modeller with a general purpose tool for creating performance models of parallel computations and machines. The performance model is created in the Simula programming language. This gives great flexibility to the system. It is straightforward to extend the system, thanks to the object oriented nature of Simula, to provide a different processor description or a different message routing strategy.

MIMD models a lot of detail. Each message transfer and computation cycle is measured and collated. Unfortunately this level of detail and flexibility leads to long simulation runs. This can be on the order of several hours for a complex simulation (up to 200 processes on a 64 processor domain) of 1 minute of simulation time. This could be overcome by using a faster processor of course, recoding in a faster language without garbage collection (e.g. C++) or even reimplementing MIMD as a parallel simulation system.

4.5 Conducting an Experiment

Once an experiment and the modelling domain are decided upon the experiment can be conducted. I have created a set of shell scripts that help this process. These scripts are combined with the tools already described to allow the experiment to proceed almost automatically. The necessary steps and how they are performed are outlined below. These steps allow one to progress from the left to the right of Figure 4.2.

The experiment generator will create a directory named after the experiment description file. This directory contains a set of numbered files that are the individual experiment instances. It also creates a shell script that can be used to execute the modelling engine for each individual experiment instance. This script can be split into separate portions if the user plans to run the simulations on separate workstations at the same time.

The modelling engine produces a separate result file for each experiment instance and for each replication of that instance. Each result file has the desired metrics in it. Once all the experiments have completed another shell script can be used to create a table of data which lists each combination of factor level against the results for each replication. This file can then be used as input to a standard statistical package or to a tool that the author developed for producing statistical tables and other output in the form of LaTeX or ASCII text. This includes the Analysis of Variance table presented in Section 3.3.3. Examples of tables produced by this program can be seen in Chapters 5 and 6.

4.6 Summary

This chapter described a methodology for conducting performance experiments on arbitrary parallel programs. Two tools that enable a systematic investigation of the performance factors have also been introduced.

Synthetic programs were presented as a way of describing a class of parallel programs. The construction of a synthetic program from a program graph and a set of process templates was described and it was shown how the experiment generator could be used to achieve this. The metric total number of compute cycles executed was selected. This metric is necessary as an alternative to execution time because synthetic programs do not terminate.

A small example which showed how the experiment generator could be used to vary a set of factors at different levels to conduct a factorial experiment was presented. The different graph generation techniques used by the experiment generator to create synthetic programs were also presented.

The MIMD modelling system was presented as a convenient way of conducting parallel program performance evaluation that allows many difficult metrics to be measured. In the next chapter a validation of the MIMD modelling system will be presented as it plays a crucial role in the performance analysis.

Chapter 5

Validation of the MIMD Modelling System and Program Model

5.1 Precis

This chapter describes the validation of two important parts of the work of this thesis. In Section 5.2 the MIMD simulation system is validated against a process systems program. The second section in this chapter gives a complete experiment on a simple program model. This demonstrates the way in which the methodology is applied and demonstrates that the simplified program model behaves as expected.

5.2 Validation of the MIMD Modelling System

This section describes one of the more interesting experiments used to validate the MIMD modelling system. These results were presented at the 11^{th} Occam User Group Technical Meeting [CLS89].

5.2.1 Distillation Simulation

The validation used an early version of a distillation simulation program developed by M^{c}Kinnell as part of his Ph.D. thesis [McK94]. The version of the program that was used to perform the validation was an early version written in Occam [Inm84] for the Meiko Computing Surface at Edinburgh University. Occam provides a simple communicating sequential process (message-passing) system to the programmer. This gives a simple process based approach.

The program modelled a distillation column by providing an Occam process for each physical plate in the column. This is a very natural approach to parallelism, sometimes called functional parallelism, which breaks the problem into parallel processes based on the physical geometry of the problem. The functional parallelism approach has several benefits. The data or *information flow* within the program closely follows the material flow within the real system. It was also found that the separate processes working independently gave a more robust simulation system because errors were localised in individual processes. These themes are expanded on in the paper by Ponton et al. [PFMS91] and by McKinnel in his thesis [McK94].

As well as the processes to represent the physical plates of the distillation column

extra processes were needed for the reboiler and reflux units. Two more processes were needed to control the whole simulation, the master, and another to collect and display results on a graphics screen. The process graph for this program is shown in Figure 5.1.

Each distillation plate or *stage model* has four communication arcs connecting it to the plate above and below. These communication arcs are used to transmit the data representing the real flow of liquid and vapour between the plates. In Occam communication arcs are called *channels* - channels can be thought of as conduits of information. Looking at one plate in detail, Figure 5.2 the naming scheme for the array of communication channels is shown.

The communications capabilities for the first version of Occam for the Meiko Computing Surface were not very flexible – they did not allow true process to process communication. Because of this the distillation simulation program was written using a special communications harness called Tiny [Cla90] which allowed for direct process to process communication. Tiny was also much faster than the communications system provided with the system.

The communications harness was specially instrumented to collect statistics by Candlin and Luo [CL89]. The harness could collect statistics about the size, frequency and timing of message transfers. Because the instrumentation could only collect data concerned with communications a metric had to be chosen to compare the real program with the simulation from the set of statistics available to the instrumented harness. For this reason the average communication delay between sending messages up and down the Occam channels was decided upon as a valid metric. It was felt that this was a valid metric because it would relate to the amount of computation that was being performed on the processes above and below an individual process.

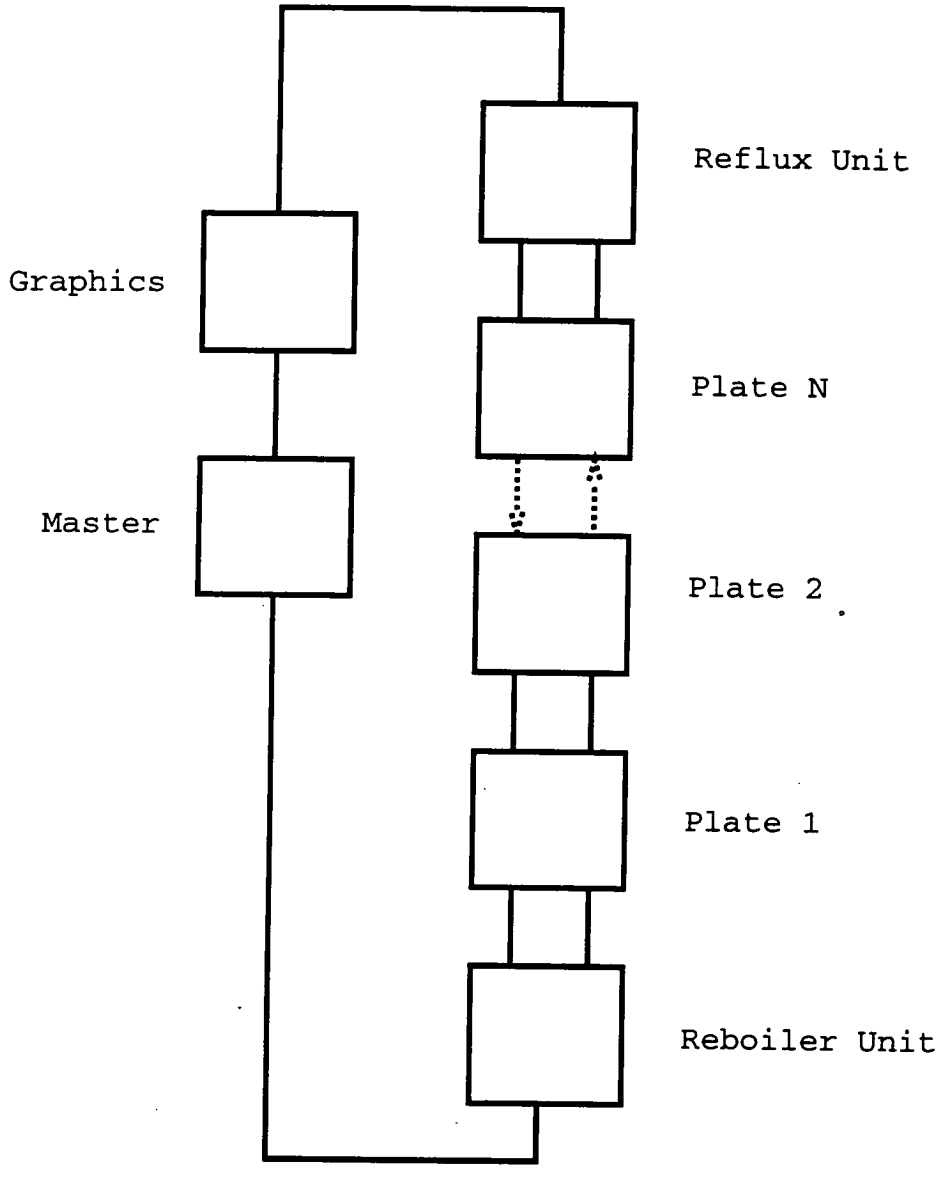


Figure 5.1: Process Graph for the Occam Distillation Simulation Program

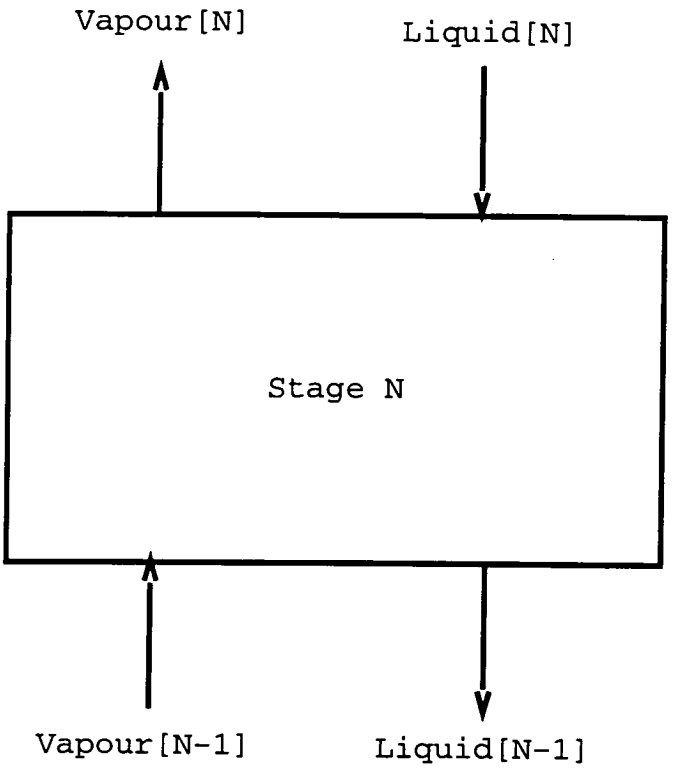


Figure 5.2: Detail of a Stage Model and its Channels

Each stage model had a two phase cycle. The first phase in the cycle was to swap data values with the stages above and below. The second phase was to compute new results for this stage. The stage model would proceed in this fashion until it received a message telling it to stop or to send a packet containing results to the graphics process. This was done by sending a packet through the processes in the program. Each stage would add its own results to the packet.

5.2.2 Creating a Simulation Model

A simulation model of the distillation simulation program was constructed with the aid of a tool developed by the author for this purpose. The tool was called *Occula* and it attempted to convert an Occam 2 source code file into a MIMD

simulation model [Ski89]. It did this by inspecting the source code of the Occam program and replacing all continuous computational statements by a single MIMD compute statement. The number of cycles to compute for was calculated by interpreting the Occam instructions and calculating suitable values based on timings found in the Transputer Compiler Writers Guide and the Transputer Data Book [Inm89a, Inm89b].

This translation was not completely automatic because the control structure of the program presented a problem. A DO loop which depended on the value of a variable could not be translated, nor the path taken at an if statement that also depended on the value for a variable. In these cases the user was asked to give an average number for the number of iterations and a probability for loop branches respectively. This meant that it was possible to make a data independent non-deterministic simulation model quickly. This did of course have major drawbacks. The translation phase could turn a deterministic non-deadlocking program into just the opposite. For this reason alone the work on Occula did not go any further. Occula was most useful for getting the large grained computation and communications structure (i.e. the weighted process graph) quickly from the Occam program into a simulation model. This model could then be hand corrected to create a satisfactory simulation model.

5.2.3 Results and Conclusions

A set of results comparing the average communication time on all the channels in the real and simulated program are shown in Table 5.1. This was for 23 seconds of the running of the distillation program - which simulated 2 minutes of real distillation in the column.

Channel	Real	Simulated	% Difference
Liquid[1]	40491	38937	-4.0
Vapour[1]	321	330	+2.8
Liquid[2]	39984	37672	-6.1
Vapour[2]	40316	38754	-4.0
Liquid[3]	393	362	-8.5
Vapour[3]	40239	37632	-6.9
Liquid[4]	399	362	-10.2
Vapour[4]	40322	38911	-3.6
Liquid[5]	461	426	-8.2
Vapour[5]	330	362	+9.6
Liquid[6]	40217	38405	-4.7
Vapour[6]	338	362	+7.1
Liquid[7]	40241	38405	-4.8
Vapour[7]	323	362	+12.0
Liquid[8]	40229	38405	-4.7
Vapour[8]	347	362	+4.3
Liquid[9]	40216	38405	-4.7
Vapour[9]	39418	36297	-8.5
Liquid[10]	1117	924	-20.8
Vapour[10]	39539	36184	-9.2
Liquid[11]	387	362	-6.9
Vapour[11]	40079	38112	-5.1
Liquid[12]	389	362	-7.4
Vapour[12]	40153	38820	-3.4
Liquid[13]	410	362	-13.2
Vapour[13]	40232	39105	-2.9
Average Absolute % Difference			7.30

Table 5.1: Average Communication Time on Each Occam Channel (MicroSeconds)

In general there is sufficiently good agreement between the simulated and real values. There is a tendency for the simulated results to be too low. This may be due to a scaling error in translating the simulated time into real time or it may be a genuine effect from the extra load placed on the processors due to the instrumentation of the communications harness.

With an average absolute difference of around seven percent it was felt that this was a good validation test for the MIMD modelling system. Other simpler programs such as pipelines and trees of communicating processes were also used for validation purposes. This rather complicated example was presented here as evidence because the same program is used later (§6.6) to test placement strategies. MIMD was also tested with several hand crafted programs written both in Occam and using the CS-Tools package [Mei92]. The tests gave an overall agreement between MIMD and the real programs of around ten percent. This is felt to be an acceptable error for such a system where the timings of individual instructions are replaced by one amalgamated instruction. There is little point aiming for a greater accuracy than this when assumptions such as the instruction amalgamation have already been made.

5.3 Program Model Justification and Validation

This section demonstrates a use of the methodology to investigate program performance parameters for the time-invariant program model. This was one of a set of experiments that were used to validate the program model that is proposed. The validation consisted of seeing whether the synthetic programs behaved in a consistent and reasonable way compared to a real program and that the parame-

terization was at a suitable level of detail. This demonstrates that a certain class of real programs can be described by a simple program model.

5.3.1 Program Model

In this example a simple, but important, class of parallel program will be simulated at a range of parameter values. The results obtained from the simulations will be used to produce a predictive model which will then be applied to other programs in the same class.

In this experiment the machine and connection topology remain fixed. The intention is to explore the effects of changing program parameters only. The hardware selected was a grid based machine with four links on each processor. This represents a simple low connectivity topology machine similar to the Inmos transputer.

As previously discussed (Section 4.2.2) simple program models can be produced which may be representative of a larger class of parallel programs. The particular model used for these experiments and the parameters chosen are described below.

Program Model – Parallel Geometric Decomposition

This class of parallel program is very common, and forms one of the simplest types of parallel computation. It is known by several names the most common being *structured spatial decomposition.* It is suitable to application areas which are data-centric and the data can be fragmented into independent blocks. Often data has to be exchanged with neighbouring processes which contain the relevant

```
While true  Begin
  SEQ
    Compute
    PAR
      Exchange Messages with Program Neighbours
End
```

Figure 5.3: Pseudo Program for Geometric Decomposition

boundary data. A common example of this is an image processing program or a geographical information system.

The program is made up of a set of identical processes, each communicating with a fixed set of neighbours, which, depending on the mapping, may be situated anywhere on the machine.

The dynamic behaviour is iterative, each process computes some new results, and then synchronizes with its neighbours to exchange data, before beginning the next iteration. In programs like this, the graph structure is fixed and regular, and the behaviour of each process can be summarized by the pseudo-program shown in Figure 5.3.

All programs in this class have regular graphs and a dynamic behaviour according to the pseudo code. It is important to note that the communications with the neighbour(s) occurs in parallel. This means that the order of the communication is unimportant but the next compute period will not commence until all the communications for this process have completed. Also, if the communication is in parallel there will be no deadlock in the system.

Programs in this class may differ in the following respects: in the size and connectivity of the program graph, the amount and variability of computation per-

Parameter	Description
N	Number of nodes in the graph
c	Connectivity of the graph
μ_c	Mean number of instructions between synchronizations
σ_c	Standard deviation of the number of instructions
μ_m	Mean message length
σ_m	Standard deviation of the message length

Table 5.2: Simple Program Model Parameters

formed by each process and the size and variability of each message that has to be exchanged with the neighbours.

Therefore within this class a program can be represented by the set of parameters shown in Table 5.2. Two important assumptions have been made in this program model. Firstly it is assumed that the size of each message sent to the neighbours has the same probability distribution. The second assumption is that the probability distributions for the amount of computation and size of messages can be described by the mean and standard deviation of a truncated normal distribution. It is unlikely that this is the correct probability distribution for all programs in this class, but it does represent a reasonable approximation, since preliminary experiments indicated that results were not very sensitive to the distribution used.

The μ and σ values are averages over the graph, so there are many programs which have the same parameter set but which may differ substantially in their patterns of node and edge weights. Also, the parameters represent time-averages over the whole execution time, so many differences in dynamic behaviour are compatible with a given parameter set. Early experiments showed that there was very little difference between instances of programs with the same parameter values. This is encouraging because it shows that it is possible to use statistical distributions to characterize a program.

As has been described earlier, due to the construction of the synthetic programs described by this parameter set, it is not possible to measure the execution time of these programs and so the metric which is used is defined as the total number of computation cycles carried out in a fixed simulation time.

5.3.2 Parameter Settings

The experiment instances are generated from an experiment definition script through the Experiment Generator **eg**. This is a two level, full factorial experiment varying the six parameters identified in the previous section. The values chosen are shown in Table 5.3. These parameters deserve some explanation for the choice of their values.

Apart from the program parameters, all other factors were held constant throughout the experiment. A 4x4 connected mesh of processors was used with a round robin placement strategy (§6.3.2).

The number of nodes N and the connectivity c were chosen to be at the lower and upper bounds of the size of a reasonable process graph for a 16 processor system. A 32 node graph of connectivity 4 fits easily onto the system, but a 64 node graph with a connectivity of 6 is a medium load for such a system.

The amount of computation time at each node was assumed to follow a Normal distribution truncated at zero. The mean at each node was allowed to take one of two values, and so was the standard deviation. The lower value for the computation mean, 500 clock cycles, represents 25 microseconds on a T800 Transputer

Program Parameters	
Parameter	Value(s)
N	{ 32, 64}
c	{ 4, 6 }
μ_c	{500, 500000}
σ_c	{5,50}
μ_m	{1,10000 }
σ_m	{0.01,0.1}
Other Parameters	
Parameter	Value(s)
Hardware	4x4 Mesh
Placement	Round Robin
Trial Length	200,000,000 (10 seconds)
Replications	3

Table 5.3: Parameter Settings for Initial Experiment

at a clock speed of 20 MHz [1]. 500 clock cycles is enough time to perform several floating point operations. The larger value is one thousand times that value and hence represents 25 milliseconds of computation.

In a similar way, the message lengths on each edge were assumed to follow a Normal distribution with one of two values for the mean and standard deviation. The two message lengths chosen were a single byte message and a ten thousand byte message. Hopefully it is clear that these program parameters represent upper and lower bounds of *reasonable* programs for a 16 processor machine.

The values for the parameters are summarised in Table 5.3. Those values shown within curly braces are the parameters that are varied and the different levels are shown.

[1] A clock speed of 20 MHz implies that one clock cycle = 1/20e6 = 5e-8 = 50 nanoseconds.

		$\mu_c = 500$							
		$\sigma_c = 5$				$\sigma_c = 50$			
		$\mu_m = 1$		$\mu_m = 10000$		$\mu_m = 1$		$\mu_m = 10000$	
c	N	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$
4	32	17.45	15.41	7.94	11.71	15.29	13.70	9.32	11.38
4	64	16.72	15.75	19.04	16.63	16.84	17.00	17.47	16.54
6	32	11.25	11.67	9.59	8.68	12.01	12.33	9.54	8.46
6	64	12.23	13.76	12.49	12.92	15.03	13.91	12.80	10.51

		$\mu_c = 500000$							
		$\sigma_c = 5$				$\sigma_c = 50$			
		$\mu_m = 1$		$\mu_m = 10000$		$\mu_m = 1$		$\mu_m = 10000$	
c	N	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$
4	32	29.97	27.97	27.88	25.90	29.97	27.97	27.85	27.89
4	64	31.97	31.97	31.96	31.94	31.97	31.97	31.92	31.92
6	32	31.96	27.96	29.83	25.81	23.97	29.96	29.77	29.78
6	64	31.96	29.96	31.91	31.93	31.95	31.96	29.90	31.92

Table 5.4: Total Computation Achieved (Divided by 1e8)

		$\mu_c = 500$							
		$\sigma_c = 5$				$\sigma_c = 50$			
		$\mu_m = 1$		$\mu_m = 10000$		$\mu_m = 1$		$\mu_m = 10000$	
c	N	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$
4	32	54.24	48.17	24.83	36.60	47.78	42.81	29.13	35.57
4	64	52.24	49.21	59.49	51.98	52.62	53.13	54.58	51.69
6	32	35.15	36.46	29.96	27.13	37.54	38.54	29.83	26.42
6	64	38.22	43.02	39.02	40.37	46.95	43.47	39.99	32.85

		$\mu_c = 500000$							
		$\sigma_c = 5$				$\sigma_c = 50$			
		$\mu_m = 1$		$\mu_m = 10000$		$\mu_m = 1$		$\mu_m = 10000$	
c	N	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$	$\sigma_m = 0.01$	$\sigma_m = 0.1$
4	32	93.66	87.42	87.13	80.94	93.66	87.41	87.02	87.17
4	64	99.89	99.90	99.87	99.80	99.90	99.91	99.76	99.75
6	32	99.86	87.39	93.21	80.95	74.91	93.61	93.03	93.08
6	64	99.87	93.61	99.71	99.80	99.85	99.87	93.43	99.73

Table 5.5: Percentage Utilization of Processing Available

5.3.3 Results

From Section 3.3.2 if six parameters are varied at two levels this gives a total of 64 experiments. Each experiment was replicated three times. The raw averaged values from the simulation are shown in Table 5.4. This table shows the total number of computation cycles achieved by the program. The percentage utilization of the total number of cycles used by all the processes from the maximum number of available cycles is shown in Table 5.5. The maximum number of available cycles is calculated by multiplying the simulation time by the number of processors.

The top and bottom halves of the Table 5.4 are markedly different. The set of experiments where the average computation block was 500 units have much smaller total computation values than those in the lower half of the table.

This can be explained by the fact that correspondingly more messages are being sent by the programs in the top half of the table. The interruptions to the computation happen more frequently, also there are a lot more communications to be made in general. Each communication that travels across the network uses some processor resource to work out where to go next. With the program blocks being smaller, this means that it is more likely that a communication event will be scheduled rather than the next computation cycle. Hence more interruptions to the computation.

Looking at the bottom part of the table the computation achieved is dominated by the factor μ_c and all the processors are nearly saturated, so the results are very similar. There is still some slackness within the system as more computation is achieved by the sixty four process system than the thirty two even for the large computation mean.

This becomes even more obvious if we look at the number of cycles performed plotted against experiment number to see how diverse the results are. This is shown in Figure 5.4 where the three replicates have been overlaid. The actual ordering of the experiments is not important and is just a consequence of the factorial experiment generation software. The top half of Table 5.4 can clearly be seen at the bottom of the graph as the smaller bars.

From Table 5.4 and Figure 5.4 it is clear that the amount of computation achieved is determined to a large extent by the mean computation time μ_c. This is to be expected from the format of the program, with the lower value of μ_c each process

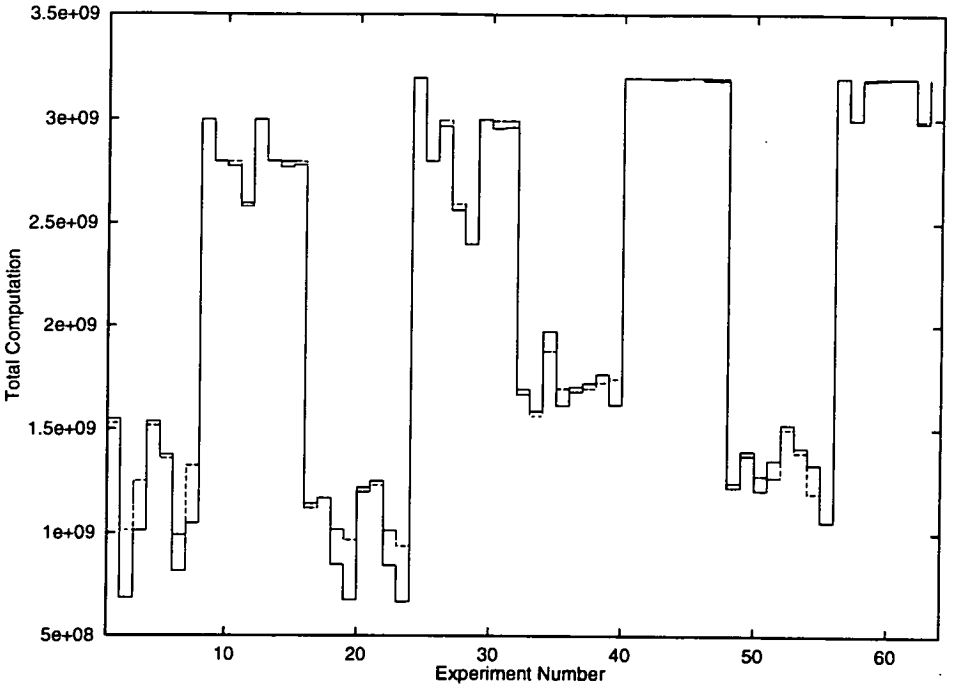


Figure 5.4: Experiment Number vs Computation Achieved

communicates one thousand times more often than the process with the larger value of μ_c. So the amount of communication that is performed is important in this respect.

There is also a secondary effect visible in Table 5.4. Irrespective of the size of the computation mean, more computation is performed where the message size has a mean of one. This effect is small and may not be significant. To see whether this is significant we have to perform the analysis of variance.

Analysis of Variance

The analysis of variance table for this experiment is shown in Table 5.6. Referring to equations 3.13 and 3.14, the null hypothesis is that all the programs are the same, to test this at a 5% level of significance the critical regions $F \geq F_{0.05,2,126} = 3.07$ for the treatments and $F \geq F_{0.05,63,126} = 5.43$ for the replicates.

Source	D o F	Sum of Squares	Mean Squares	F
Between Replicates	2	2.47867e+16	1.23934e+16	4.47667
Between Treatments	63	1.47999e+20	2.34919e+18	848.561
Residual	126	3.48823e+17	2.76844e+15	
Total	191	1.48372e+20		

Table 5.6: Analysis of Variance Table

Hence we can reject the null hypothesis for the treatments and accept it for the replicates. The differences between the replicates are small relative to the estimate of the variability between program instances for the same parameter values (the error mean square). By contrast the differences between parameter settings (the "treatment" mean square) are very marked.

This implies that the model itself is self consistent. Different replications of the experiment at the same parameter setting do not contribute greatly to the variance of the results.

Estimates of the effects of individual parameters and their interactions can be derived from a transformation of the 64 means at each combination of the parameter settings. These values are presented in the second column of each part of Table 5.7 and correspond to the terms present in the model underlying the analysis (see Equation 3.16 in Section 3.4). Where a factor name appears that means that the estimate is for that factor or the interaction of the factors if there is more than one. The first entry in the table labelled *gm* is the overall mean value of the number of computation cycles, referred to as μ in Equation 3.5. The estimates are the β_i values in the same equation.

Each estimate in Table 5.7, apart from that for the constant term *gm*, has the same standard error which is given at the foot of the table. The *t-values* given in this table are the ratio of estimates to standard error. Values of t less than 10 can be taken as a guide to those terms in the model which are of little importance.

Effects	Estimates	t	% Var	Effects	Estimates	t	% Var
gm	2167057601.00	570.69		σ_m	-19603662.33	-5.16	0.087
c	-80530703.67	-21.21	0.839	$c\sigma_m$	5056155.67	1.33	0.050
N	166440889.00	43.83	3.585	$N\sigma_m$	2279117.67	0.60	0.003
cN	-33442934.33	-8.81	0.145	$cN\sigma_m$	3591129.00	0.95	0.001
μ_c	837757036.33	220.62	90.821	$\mu_c\sigma_m$	-5131940.33	-1.35	0.002
$c\mu_c$	78921250.33	20.78	0.806	$c\mu_c\sigma_m$	7415703.00	1.95	0.003
$N\mu_c$	-1953814.33	-0.51	0.000	$N\mu_c\sigma_m$	22629962.33	5.96	0.007
$cN\mu_c$	9216137.00	2.43	0.011	$cN\mu_c\sigma_m$	-15631377.00	-4.12	0.066
σ_c	-2049401.00	-0.54	0.001	$\sigma_c\sigma_m$	24621631.67	6.48	0.032
$c\sigma_c$	1742627.67	0.46	0.000	$c\sigma_c\sigma_m$	13998324.33	3.69	0.078
$N\sigma_c$	3535869.67	0.93	0.002	$N\sigma_c\sigma_m$	-20678181.67	-5.45	0.025
$cN\sigma_c$	1861131.67	0.49	0.000	$cN\sigma_c\sigma_m$	-26503397.00	-6.98	0.055
$\mu_c\sigma_c$	1424310.33	0.38	0.000	$\mu_c\sigma_c\sigma_m$	38052712.33	10.02	0.091
$c\mu_c\sigma_c$	-14277345.00	-3.76	0.026	$c\mu_c\sigma_c\sigma_m$	35855097.00	9.44	0.187
$N\mu_c\sigma_c$	-3413675.00	-0.90	0.002	$N\mu_c\sigma_c\sigma_m$	-16887666.33	-4.45	0.166
$cN\mu_c\sigma_c$	10794284.33	2.84	0.015	$cN\mu_c\sigma_c\sigma_m$	1480519.67	0.39	0.037
μ_m	-63514260.33	-16.73	0.522	$\mu_m\sigma_m$	3126980.33	0.82	0.000
$c\mu_m$	13414800.33	3.53	0.023	$c\mu_m\sigma_m$	-24881607.00	-6.55	0.001
$N\mu_m$	53691662.33	14.14	0.373	$N\mu_m\sigma_m$	-5598437.00	-1.47	0.080
$cN\mu_m$	-43496886.33	-11.45	0.245	$cN\mu_m\sigma_m$	38399110.33	10.11	0.004
$\mu_c\mu_m$	46880972.33	12.35	0.284	$\mu_c\mu_m\sigma_m$	-2874689.67	-0.76	0.191
$c\mu_c\mu_m$	10550007.67	2.78	0.014	$c\mu_c\mu_m\sigma_m$	12436719.00	3.28	0.001
$N\mu_c\mu_m$	-38953089.00	-10.26	0.196	$N\mu_c\mu_m\sigma_m$	30416039.67	8.01	0.020
$cN\mu_c\mu_m$	19302707.67	5.08	0.048	$cN\mu_c\mu_m\sigma_m$	-482046.33	-0.13	0.120
$\sigma_c\mu_m$	4586263.00	1.21	0.003	$\sigma_c\mu_m\sigma_m$	-9168616.33	-2.41	0.000
$c\sigma_c\mu_m$	-7254588.33	-1.91	0.007	$c\sigma_c\mu_m\sigma_m$	-9938745.00	-2.62	0.011
$N\sigma_c\mu_m$	-42566642.33	-11.21	0.234	$N\sigma_c\mu_m\sigma_m$	10062498.33	2.65	0.013
$cN\sigma_c\mu_m$	-11386004.33	-3.00	0.017	$cN\sigma_c\mu_m\sigma_m$	8516221.67	2.24	0.013
$\mu_c\sigma_c\mu_m$	19761745.00	5.20	0.051	$\mu_c\sigma_c\mu_m\sigma_m$	-3059738.33	-0.81	0.009
$c\mu_c\sigma_c\mu_m$	19691817.67	5.19	0.050	$c\mu_c\sigma_c\mu_m\sigma_m$	-15103812.33	-3.98	0.001
$N\mu_c\sigma_c\mu_m$	-7267461.67	-1.91	0.007	$N\mu_c\sigma_c\mu_m\sigma_m$	2201552.33	0.58	0.030
$cN\mu_c\sigma_c\mu_m$	-25869417.00	-6.81	0.087	$cN\mu_c\sigma_c\mu_m\sigma_m$	16264773.00	4.28	0.001
Standard Error = 3.79723e+06							

Table 5.7: Parameter Contributions

The fourth column of each part of the table gives the percentage of variation that can be attributed to the factor or combination of factors.

Of the six parameters, the t-values for the terms involving the standard deviations for both the computation mean (σ_c) and message length (σ_m) distributions are all markedly small relative to the rest. Only two t-values greater than 10 include either of these parameters. This would lead the investigator to consider leaving these factors out of future experiments as they appear to have little effect at the chosen levels. Alternatively it could be that the levels chosen were too small to have an effect and large values may be necessary for differences to show up.

Table 5.7 attributes 90 percent of the variation in the model due to μ_c the size of the computation mean. We had suspected this by looking at the raw results (Table 5.4) and the plot of computation cycles versus the experiment number (Figure 5.4). There is also a significant effect from N the number of processes as would be expected and one from μ_m. There is a second level interaction between c and μ_c that looks signifcant. All of these interactions could be studied further in a separate experiment.

It is necessary to test the assumptions outlined in Section 3.4.1 before we proceed any further with the analysis. A scatter plot of the residuals versus the predicted response for each replicate are shown in Figures 5.5 through 5.7. From the scatter plots it is clear that there is no visible trend apart from some of the higher values which are very far from the rest. There is clearly a non-constant error variance over the response range. The residuals are an order of magnitude smaller than the predicted response but vary widely.

A normal quantile-quantile plot of the residuals from the first replicate is shown in Figure 5.8. This tests whether the errors are independently and identically

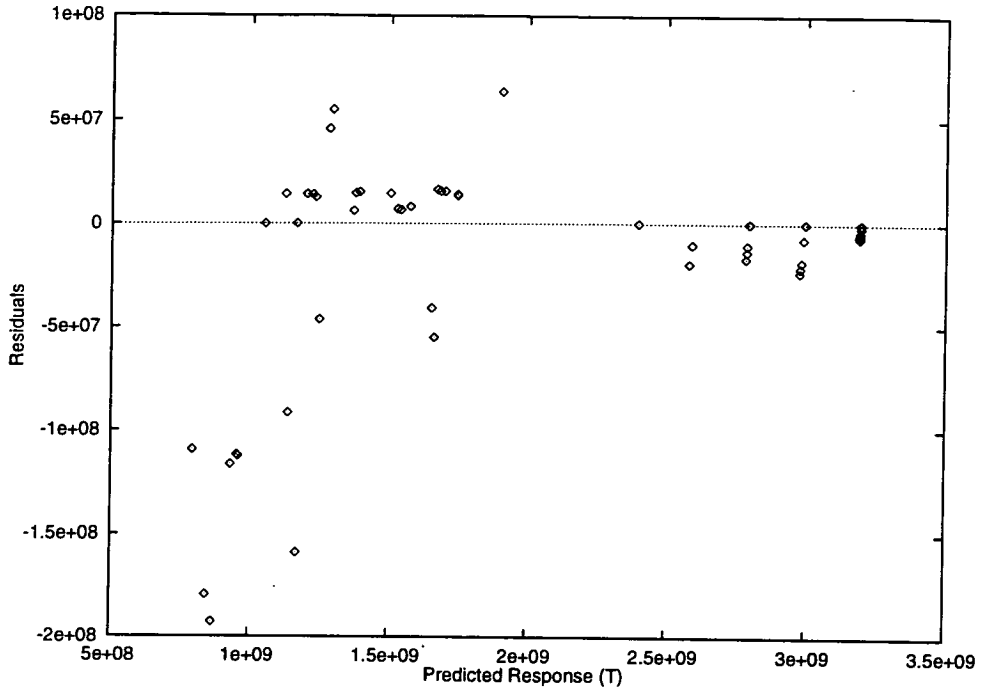


Figure 5.5: Residuals versus Predicted Response – Replicate 1

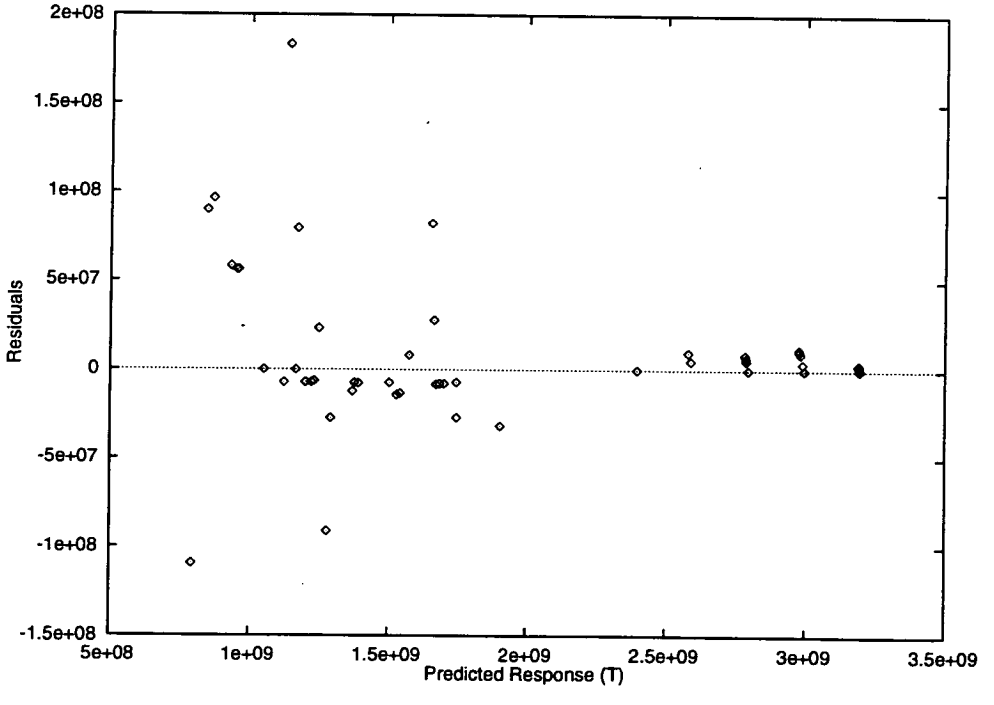


Figure 5.6: Residuals versus Predicted Response – Replicate 2

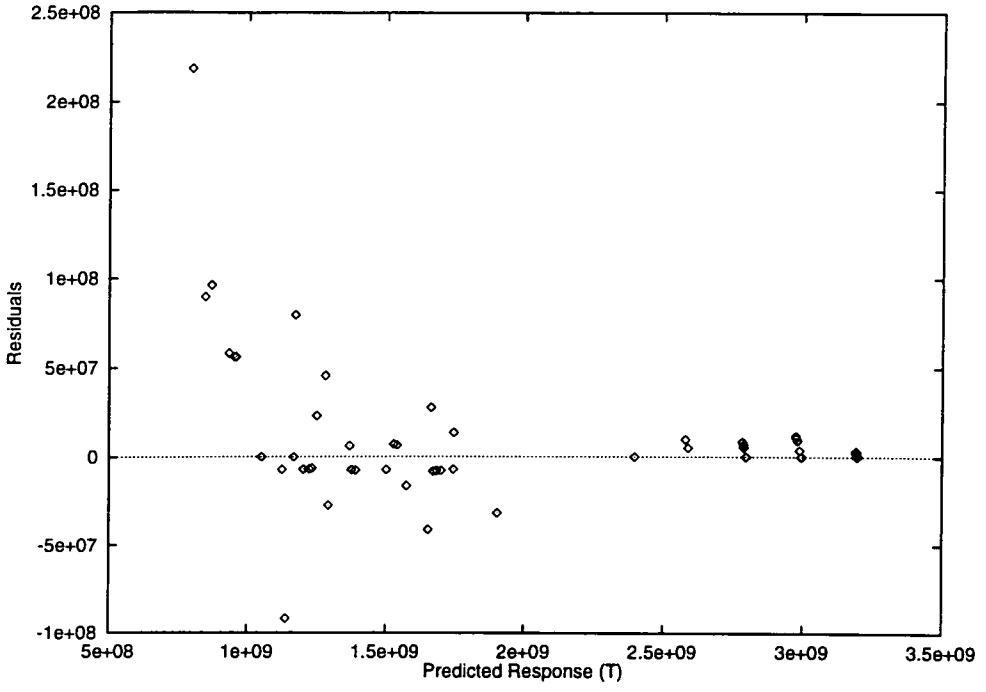


Figure 5.7: Residuals versus Predicted Response – Replicate 3

distributed. If they are then a straight line passing through the origin would be present. A least squares straight line fit has also been plotted. There is a significant deviation from a straight line passing through the origin, this is mostly due to the large residual values which are created by having one dominant factor within the experiment. It is important that the plot does not show the degenerate case of a distinct 'U' shaped curve.

5.3.4 Constructing a Model

These visual tests lead us to believe that the model underlying the system is not linear. This doubt was investigated by Phillips [Phi94] who analysed the model in terms of a transformed response variable using a Box-Cox transformation. Phillips was using processor utilisation U as his metric. He found it necessary to use a Box-Cox transformation U' defined in Equation 5.1 in order for his results to

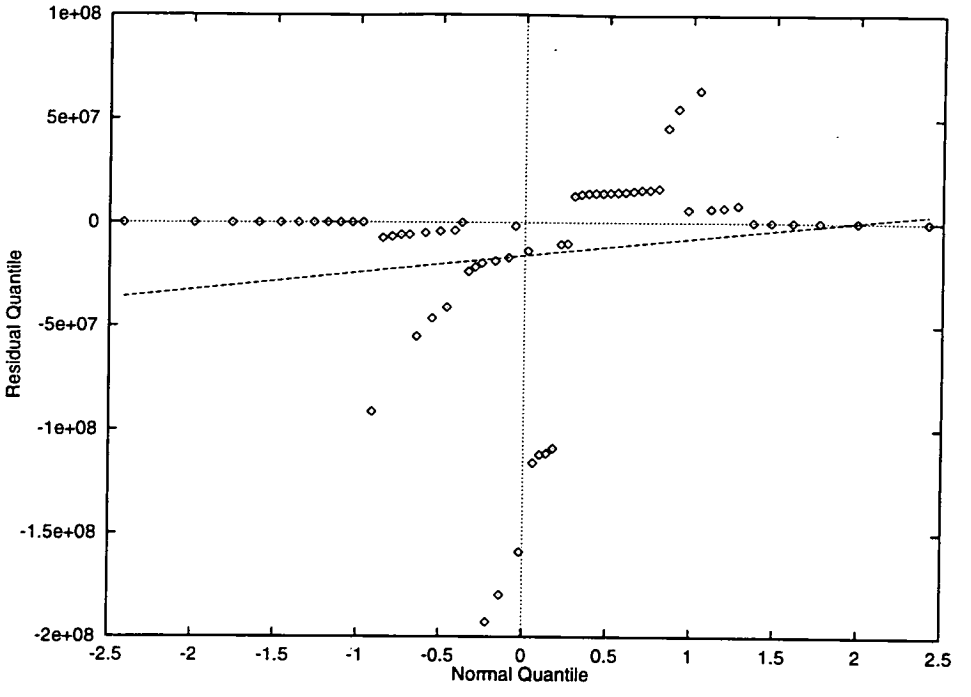


Figure 5.8: Normal Quantile-Quantile Plot of Residuals – Replicate 1

pass all the visual tests. U is the unmodified utilization value. He also found that this reduced the dependence on the higher order interactions, which is desirable for reducing the number of factors to be investigated. The transformation did not however change the importance of the factors.

$$U' = \left\{ \sinh\left(\frac{U - 7000000}{1000000}\right) + 150 \right\}^{2.41} \tag{5.1}$$

Unfortunately using the transformed response did not provide a significant improvement in the screening and selecting of factors. Lyon et al. have also found this to be the case [LSK94]. So it is still useful to look at the linear model.

With reference to the response, it is expected that as more load is added to the processors in our system the processing speed will slow faster than linearly. This is due to the fact that we have a round robin scheduler in each processor and more

context switches and other process overheads must be taken into account. So our model is behaving as expected and in some joint work with Candlin, Fisk and Phillips [CFPS92] some other models are explored and found to be useful. This example is designed to show the system as a whole fitting a model to transformed data is the same operation as using the linear model. Notwithstanding this a linear model can be used to predict performance of other programs in this class. We have found the linear model sufficient for predicting relative performance and it is easier to understand. This is important because simple models can be easily derived by a programmer even at early stages of program development.

5.3.5 Conclusions

This section has shown that a simple synthetic program model can be used to represent a real program in a statistical experiment. The synthetic program behaves in an expected way. This section has also demonstrated that the program model is self consistent, that it does not vary widely between individual instances.

From the statistical analysis it is easy to draw conclusions about which factors under investigation have had a significant effect on the performance. The general approach of applying statistical techniques to synthetic programs seems correct.

5.4 Summary

This chapter has shown the validation of two important parts of the proposed performance investigation environment. The MIMD system was described and

shown to be a useful tool for modelling arbitrary communicating sequential process programs.

The use of a synthetic program to represent a program was described and an experiment was performed on an important subclass of such programs. This demonstrated the correctness of the program model within the limitations of the simplifications made and that it behaved as expected. This also demonstrated the use of the whole experimental framework and showed the results and conclusions that can be drawn.

Chapter 6

Results

6.1 Precis

This chapter displays the use of the methodology on two examples. The first is an experiment, Section 6.4, to study a range of process placement strategies and is performed on a similar type of program as that used in the experiment described in Chapter 5.

The second experiment in Section 6.6 shows the application of the placement strategies on a process engineering program similar to that used for validation of the MIMD system (§5.2).

In each experiment the analysis of the results requires use of the statistical tech-

niques described in Chapter 3. The experiments described here were performed using the tools developed specifically for this purpose, namely the experiment generator (**eg**) and the multiprocessor simulation package MIMD. Each uses synthetic programs which are representative of the class of parallel algorithms previously discussed.

6.2 Placement Strategy Evaluation using Synthetic Program Graphs

This section describes an experiment to investigate the effectiveness of four different placement strategies. Previous work in this area has usually been performed in a non-systematic manner. This often involved the use of one placement strategy on several programs or several strategies on the same program. This experiment will take a set of process placement strategies and will apply them to a range of related programs. This will then allow the following questions to be answered:

1. Did any of the strategies have an effect on performance?
2. Which is the best strategy overall?
3. Is one strategy better for certain types of programs?

The actual outcome from the experiment can be an ordered ranking of strategies in terms of their effectiveness.

In devising this experiment a diverse set of placement strategies has been used. The strategies have been chosen so as to be representative of different approaches

presented in the literature, not necessarily because of their claimed effectiveness.

The process placement problem is essentially an optimization problem. In general terms the problem can be posed as follows:

$$\text{Minimize} f = f(G, H) \tag{6.1}$$

where G is the program graph H is the hardware and f is the execution of that program graph on that hardware. Each placement strategy takes a different approach to this optimization problem.

6.2.1 Cost Function

For various of these strategies a cost function has to be used to give a quick approximation of how good the mapping is. Ideally of course this cost function would give the exact cost of the particular mapping chosen. However, we have no way of deriving this accurately. So a short cut method is used to approximate the cost of the mapping. This is based on an intuitive basis that a good mapping is one that distributes the load of the processes but does not make the communications paths too long. These two opposing forces drive the random search strategies until they have found suitable solutions. The higher the cost of the mapping, the worse the mapping is considered to be. This is given in arbitrary cost units.

This cost function was used as the function to be minimised by the genetic algorithm and also as a way of comparing results from other strategies. For placements produced by any strategy a cost can be calculated. The cost function is a

modified version of that proposed by Sinclair [Sin87] and is described below.

Problem Representation for Cost Function Evaluation

A program consists of m modules. A module i can in general be executed on any of the p processors in the system. An *assignment* A of the program is a complete specification of the processors on which the modules are executed. A is an onto mapping from the set of modules to the set of processors; $A(i) = s$ if and only if module i is assigned to processor s by A.

If module i is executed on processor s, it incurs an execution cost $e_i(s)$. When any module i must be assigned to one particular processor t, $e_i(s) = \infty$ for all s, $1 \leq s \leq p$ and $s \neq t$. If two modules i and j are executed on the same processor, the cost of communication between them is assumed to be zero. If they are on different processors s and t, the cost of communication is $c_{i,j}(s,t)$.

The cost $C(A)$ of an assignment is the sum of all the module execution costs and intermodule communication costs given the assignment A.

$$C(A) = \sum_{i=1}^{m} e_i(A(i)) + \sum_{i=1}^{m} \sum_{\substack{j=i+1 \\ A(i)\neq A(j)}}^{m} c_{i,j}(A(i), A(j)) \tag{6.2}$$

The cost function for execution cost is as follows:

$$e_i = mw_i \sum_{j=1}^{m} \delta_{ij} \tag{6.3}$$

Where mw_i is the weight of the module i and

$$\delta_{ij} = \begin{cases} 1 & A(i) = A(j) \\ 0 & A(i) \neq A(j) \end{cases} \tag{6.4}$$

This effectively means that if a more than one module is assigned to the same processor then the execution cost of that module is multiplied by the number of modules assigned to that processor.

And for communication costs

$$c_{ij} = cw_{ij} dist(A(i), A(j)) \tag{6.5}$$

Where $cw_{i,j}$ is the weight of the communication channel linking modules i and j, and $dist(A(i), A(j))$ is the number of hops between processors. This distance can be zero and hence the cost for an on processor communication is zero.

This cost function assumes that the weight of processes affects the cost in a multiplicative manner with the addition of modules to a processor. Evidence from our earlier experiment (§5.3.3) suggests that this is the case. Also there must be a balance between the size of the weights of the computation and the cost of the communication. This is merely a problem of the units involved in specifying the weight of the process.

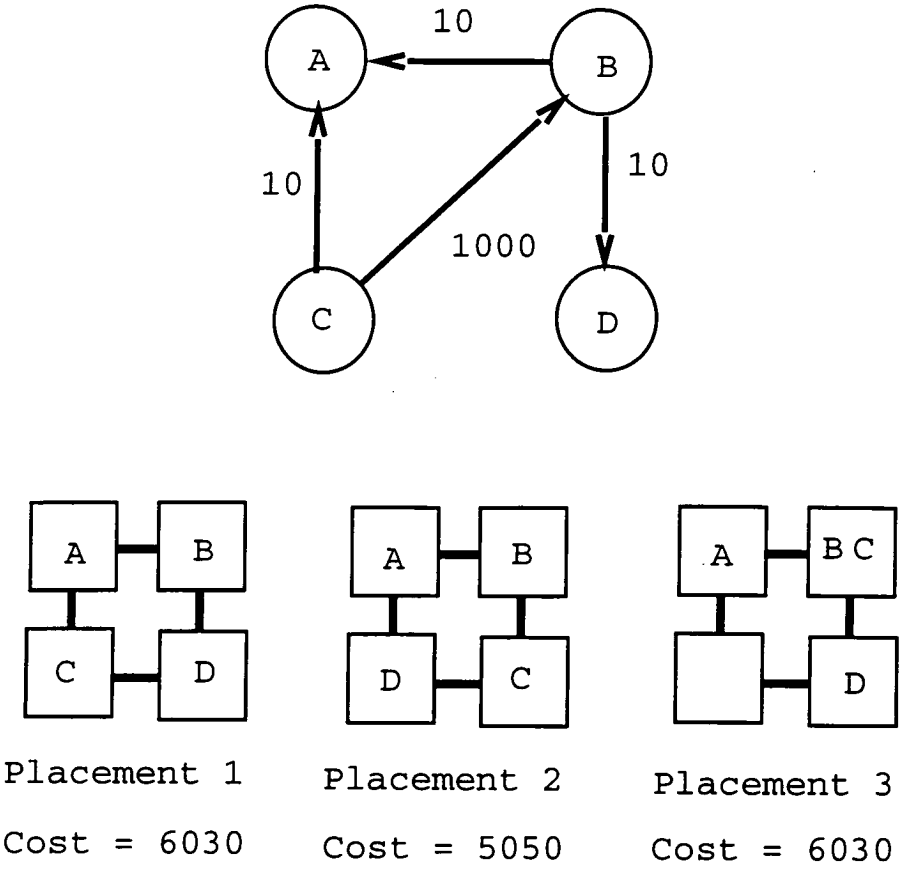


Figure 6.1: Simple Process Graph, Placements and Costs

6.2.2 Example use of Cost Function

To demonstrate the use of the cost function it will be applied to the process and the three placements shown in Figure 6.1. Imagine that the four nodes in the program have a computational weight of 1000 units each. Then the weights for each placement are shown under the placement in the diagram. The two components are the computation and the communication weights as calculated by Equation 6.2.

Cost of Placement 1

If we let the left part of Equation 6.2 be called Compute and the right part Communication then we have the following calculations:

$$\begin{aligned}
Compute &= (1000 + 1000 + 1000 + 1000) \\
&= 4000 \\
Communication &= (10 \times dist(A(B), A(A)) + 10 \times dist(A(B), A(D)) \\
&\quad +10 \times dist(A(C), A(A)) + 1000 \times dist(A(C), A(B))) \\
&= (10 \times 1 + 10 \times 1 + 10 \times 1 + 1000 \times 2) \\
&= 2030 \\
Total &= 6030
\end{aligned} \tag{6.6}$$

Cost of Placement 2

$$\begin{aligned}
Compute &= (1000 + 1000 + 1000 + 1000) \\
&= 4000 \\
Communication &= (10 \times dist(A(B), A(A)) + 10 \times dist(A(B), A(D)) \\
&\quad +10 \times dist(A(C), A(A)) + 1000 \times dist(A(C), A(B))) \\
&= (10 \times 1 + 10 \times 2 + 10 \times 2 + 1000 \times 1) \\
&= 1050 \\
Total &= 5050
\end{aligned} \tag{6.7}$$

Cost of Placement 3

$$\begin{aligned}
Compute &= (1000 + 1000 + 2000 + 2000) \\
&= 6000 \\
Communicate &= (10 \times dist(A(B), A(A)) + 10 \times dist(A(B), A(D)) \\
&\quad +10 \times dist(A(C), A(A)) + 1000 \times dist(A(C), A(B))) \\
&= (10 \times 1 + 10 \times 1 + 10 \times 1 + 0 \times 1) \\
&= 30 \\
Total &= 6030
\end{aligned} \tag{6.8}$$

Placement 2 reduces the communication costs by swapping processes C and D. Placement 3 tries to reduce the costs further by placing B and C on the same processor, unfortunately this has given an extra cost as B and C will now run more slowly. In the examples above Placement 1 and Placement 3 have the same cost. In practice they are unlikely to have the same execution time as this will depend on whether how costly communication is compared to computation and how quick the processor can context switch between several processes. This a limitation of the current cost function which can only be remedied by having more realistic models of the communication and computation.

If the values of the communication and computation weights are not of the same order then we require an additional constant to describe their relative weightings. This can be used as a multiplicative factor on either part of Equation 6.2.

6.3 Strategies

Each strategy is introduced and given a short description, followed by an outline of the algorithm and then a timing analysis.

6.3.1 Random Process Allocation

Each process is allocated to a processor selected at random. The selection of the processors is performed with replacement so highly unbalanced placements can result. This strategy takes a time that is linear in the number of nodes.

This is intended to be the baseline strategy against which all other algorithms can be measured. It can be expected that other strategies will perform better than this strategy.

6.3.2 Round Robin Process Allocation

Each process is allocated to a processor in turn until all the processes have been allocated. This strategy shares the processes as evenly as possible over the processors, not accounting for the relative weights of the processes. This strategy takes a time that is linear in the number of nodes.

1. Initialise a population of chromosomes.
2. Evaluate each chromosome in the population
3. Create new chromosomes by mating current chromosomes; Apply mutation and recombination as the parent chromosomes mate.
4. Delete members of the population to make room for the new chromosomes.
5. Evaluate the new chromosomes and insert them into the population.
6. If time is up, stop and return the best chromosome; if not go to 3.

Figure 6.2: The Genetic Algorithm

6.3.3 Genetic Algorithm

Genetic algorithms provide a framework for optimization based on genetic search techniques. Genetic algorithms are task independent optimizers. They have no inherent knowledge of the task that they are optimizing, and so represent a very different approach than the other strategies employed here. The task independence is such that the user of a genetic algorithm only has to supply an evaluation function which returns a value when given a particular point in the search space. The evaluation (or in this case the cost function described in Section 6.2.1) gives the fitness value for any member of the population.

Genetic algorithms were invented to mimic some of the processes observed in natural evolution by John Holland in the early 1970's [Hol75]. The main things that make genetic algorithms different from other optimization routines are the mutation and crossover operators.

The steps involved in a genetic algorithm, due to Davis, are shown in Figure 6.2 [Dav91]. A generation in genetic algorithm terms consists of the steps: selection, mutation, crossover and evaluation.

Parameter	Value
Crossover Rate	0.6
Mutation Rate	0.001
Population Size	100
Total Trials	10000

Table 6.1: Parameters for the Genetic Algorithm

In order to use a genetic algorithm you have to have a set of chromosomes which are the population. Each chromosome is made up of a set of genes. It is these genes that encode possible problem solutions. I chose to have each process as a gene where the value of the gene represents the processor it is to be placed onto. The genetic algorithm then had to find a set of values for the genes such that the cost function was a minimum. The cost function employed was that described in Section 6.2.1.

I employed the Genesis genetic algorithm system which was freely available and easy to use [Gre90]. In terms of the genetic algorithm the important characteristics are shown in Table 6.1. Most of these are built into the system though I did have to modify the code to perform placements automatically.

This genetic algorithm uses the roulette wheel parent selection algorithm due to James Baker [Bak87]. The idea is to allocate to each chromosome a portion of a spinning wheel proportional to the chromosome's relative fitness. A single spin of the wheel determines the number of offspring assigned to every chromosome. The mutation and crossover rate were set experimentally as these produced a decent set of strategies within the time span. The execution of the genetic algorithm takes around thirty seconds of CPU time on a SparcStation ELC.

6.3.4 Node Swapping

Given any random configuration, it is usually possible to perturb it so as to reduce the total cost. This algorithm swaps processes between processors if doing so results in a reduced total sum of the edge weights radiating from a processor. It proceeds in this way for all the processes in the system. The steps of the algorithm can be outlined as follows

1. Allocate processes to processors randomly as an initial configuration.
2. For all possible pairs of processes swap them and evaluate cost.
3. If cost is lower then keep change otherwise reverse change.

All possible pairs are tried, so this represents a large number of combinations. A different approach to this algorithm would have been to evaluate the effect that each swap would have had and then only perform the swap that reduces the cost by the largest amount.

6.4 Evaluation of Four Placement Strategies

This experiment demonstrates how, using the methodology proposed in this thesis we can evaluate the four different process placement strategies. The strategies that are to be used were outlined in Section 6.3.

The parameters in this experiment are different from the previous experiment, obviously, because we wish to look at placement strategies. The program graph

Program Parameters	
Parameter	Value(s)
N	70
c	{4,7}
μ_c	{5000, 500000}
σ_c	0
σ_m	0
μ_m	{1,1000}
Other Parameters	
Parameter	Value(s)
Hardware	8x8 Mesh
Placement	Random (1)
	Round Robin (2)
	Genetic Algorithm (3)
	Node Swapping (4)
Trial Length	20,000,000 (1 second)
Replications	2

Table 6.2: Parameter Settings for Placement Strategy Experiment

will have a fixed number of nodes (70) and the program will run on a 64 processor machine. This was felt to be a reasonably large processor system with a medium program load. The process model described in Section 5.3.1 is used again, though this time only the process calculation mean and message length mean are varied. The parameters for the experiment are summarised in Table 6.2.

It is instructive to look at the placements produced by the different strategies. The distribution of processor load (the number of processes allocated to a processor) generated by each strategy for each different experiment is shown in Figures 6.3 through 6.6. If we consider the top left graph in Figure 6.4, this shows that the strategy decided to allocate one process to each of 58 processors and to allocate two processes to each of six processors. This makes the respective totals of seventy processes and sixty four processors. The cost of each placement is also shown in arbitrary cost units. The higher the cost, the worse the placement.

The raw results from this experiment are shown in Table 6.3. The corresponding

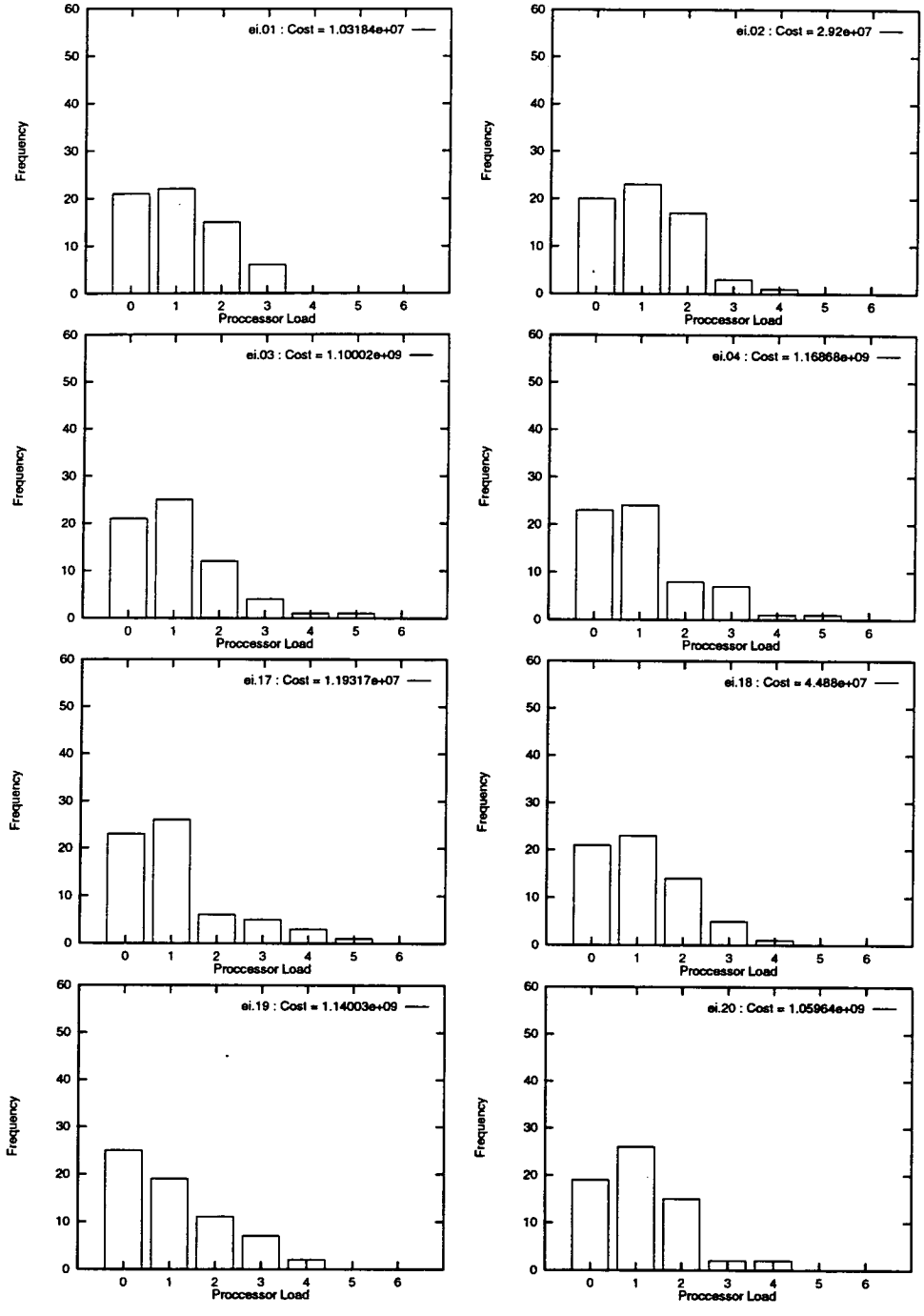


Figure 6.3: Processor Loading Frequencies Strategy 1 – Random

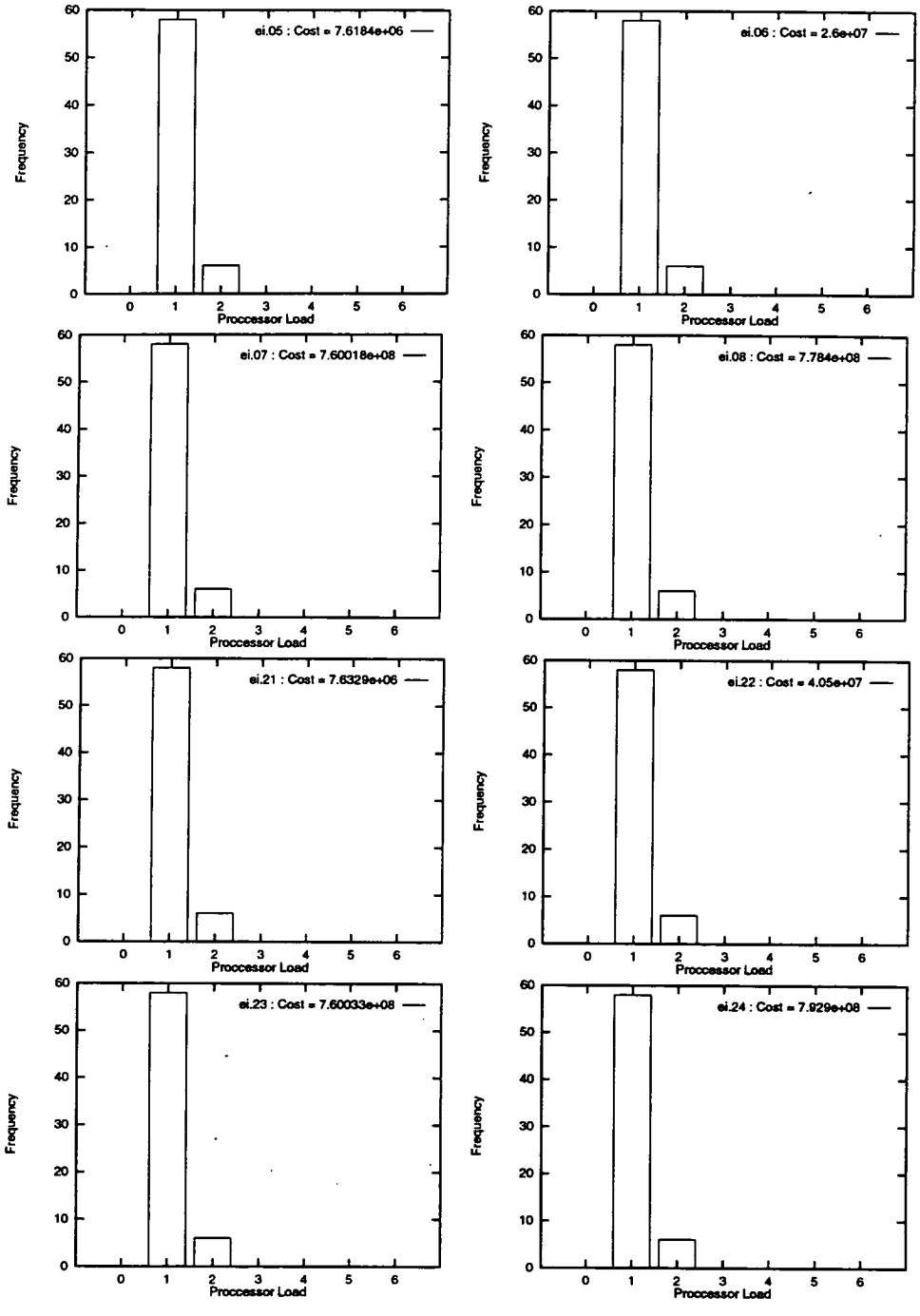


Figure 6.4: Processor Loading Frequencies Strategy 2 – Round Robin

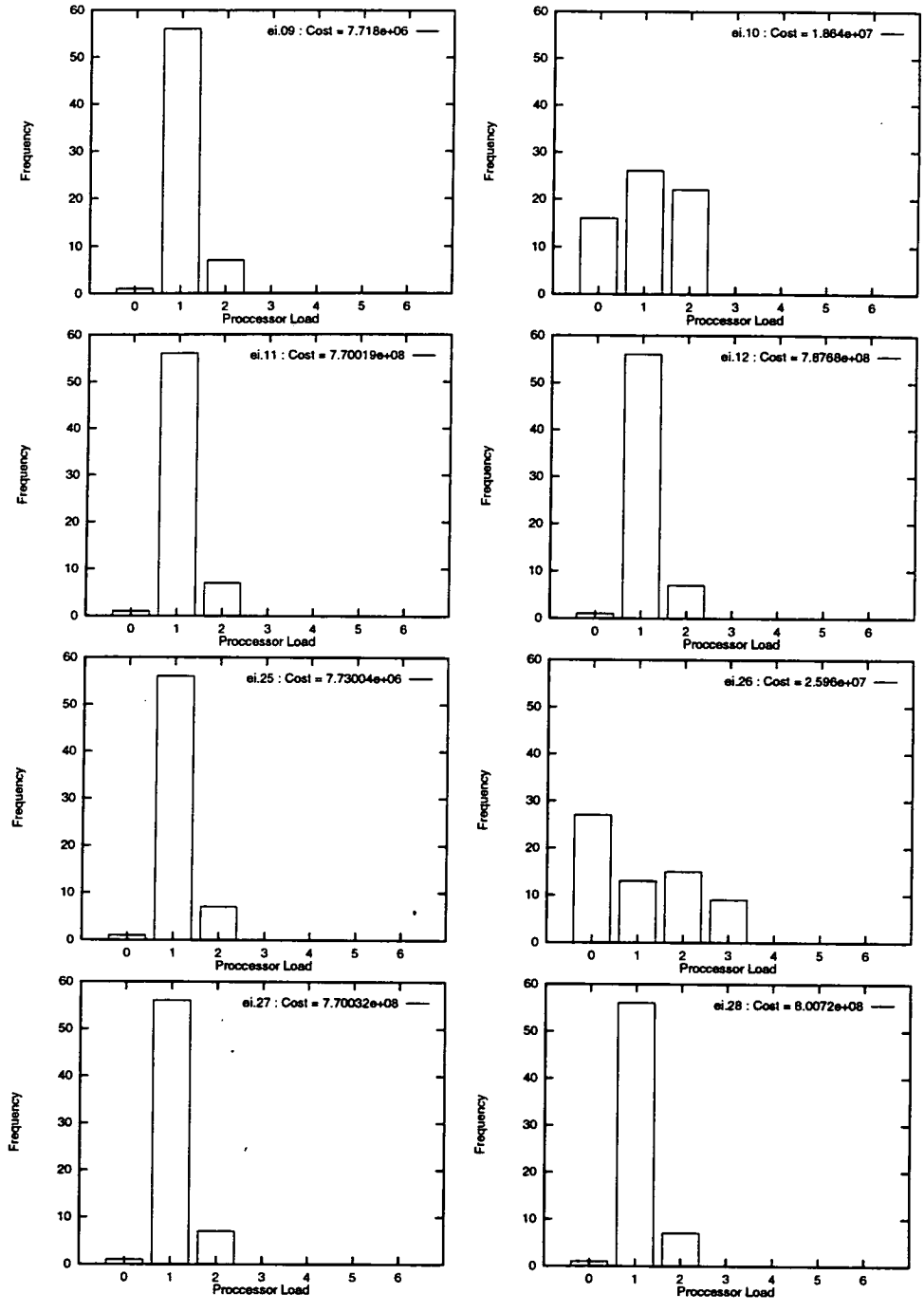


Figure 6.5: Processor Loading Frequencies Strategy 3 – Genetic Algorithm

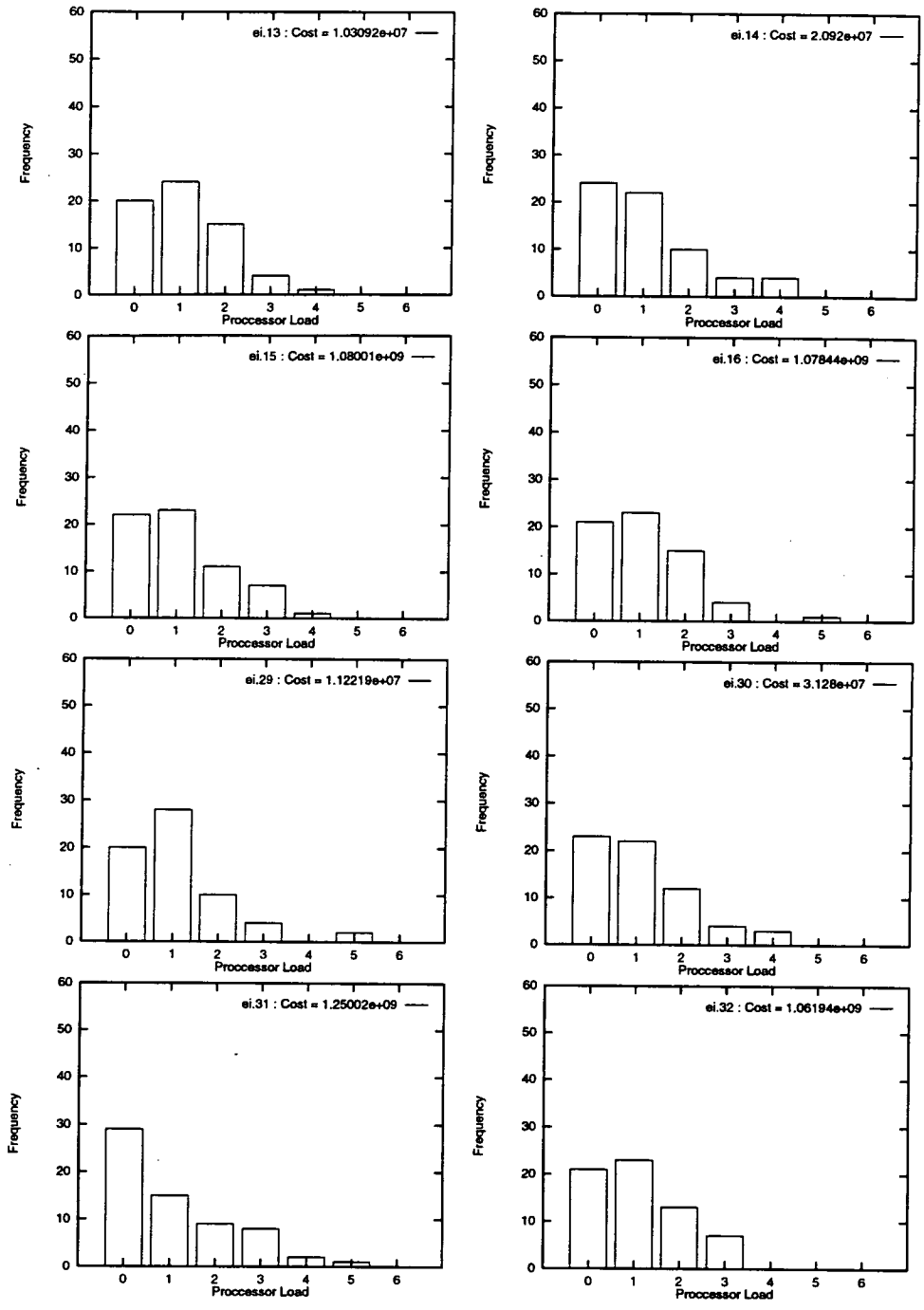


Figure 6.6: Processor Loading Frequencies Strategy 4 – Node Swapping

utilization figures are given in Table 6.4. From these tables we can see that the round robin and the genetic algorithm perform best, but that the round robin algorithm is more consistent.

The round robin algorithm performs well in this instance due to the nature of the program graph. The connections of the process graph are fairly random and so the random placement strategy and the round robin strategy have both benefited from this - using very little knowledge.

The genetic algorithm does not perform as well as the round robin algorithm and in one instance performs worse than the random algorithm. This is the case where we have a graph of high degree, a low computation mean and a large communication mean. This could suggest that the genetic algorithm was not sensitive enough to the communication needs of the program. The fault with this would be in the cost function. The cost function assumes that the cost of sending a message is linear with the size of the message. It has been shown by Clarke [Cla90] amongst others that this is not in fact the case and the communication time has a large setup period followed by a linear relationship on the size of the message. The MIMD modelling system follows this relationship accurately, but the cost function does not.

The node swapping algorithm performs worse than the random placement strategy in more than half of the cases. This is not a very good performance but the algorithm is not very complicated.

It should be noted that the genetic and the node swapping algorithms use a starting position generated by the random method. Unfortunately this is not the same initial placement that is used for the random placement method in that same experimental batch. This means that it is not possible to compare the performance

	$c = 4$				$c = 7$			
	$\mu_c = 5000$		$\mu_c = 500000$		$\mu_c = 5000$		$\mu_c = 500000$	
	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$
Random	8.46	8.54	8.60	8.20	7.99	8.27	7.80	9.00
Round Robin	12.54	12.26	12.79	12.79	12.38	12.07	12.79	12.79
Genetic	12.34	9.32	12.60	12.59	12.20	7.16	12.59	12.59
Node Swop	8.69	7.80	8.40	8.60	8.59	7.90	7.00	8.60

Table 6.3: Computation Cycles Achieved (Divided by 1e8)

	$c = 4$				$c = 7$			
	$\mu_c = 5000$		$\mu_c = 500000$		$\mu_c = 5000$		$\mu_c = 500000$	
	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$
Random	66.10	66.68	67.18	64.04	62.41	64.57	60.92	70.28
Round Robin	97.96	95.79	99.94	99.91	96.73	94.31	99.89	99.89
Genetic	96.44	72.85	98.40	98.36	95.33	55.95	98.33	98.33
Node Swap	67.86	60.97	65.62	67.16	67.10	61.75	54.67	67.15

Table 6.4: Percentage Utilization of Computation Cycles

of, say for example, the random and node swapping within the same experiment as they are starting from different random positions. A further investigation of these strategies would use the random output as the starting point for both of these strategies. This would then allow a true comparison of the strategies and you could see whether the node swapping and genetic algorithm were actually improving the placement or making matters worse. The strategies themselves were not the main part of the investigation so the individual performance is not crucial, it was instead the ability to compare arbitrary strategies.

Using the raw data we can already rank our strategies for the class of programs that we are investigating. The most consistent strategy is the round robin strategy in this case. This may not be the case if the process weights were highly skewed because then the round robin strategy would produce many bad placements. The genetic algorithm may perform better on some of these unbalanced programs.

It is interesting to compare the costs as calculated by the cost function for all the placements against the responses observed. The calculated costs are shown on each individual processor loading frequency graph and in Table 6.5. These

	$c = 4$				$c = 7$			
	$\mu_c = 5000$		$\mu_c = 500000$		$\mu_c = 5000$		$\mu_c = 500000$	
	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$
Random	1.19e+07	4.48e+07	1.03e+07	2.92e+07	1.14e+09	1.06e+09	1.10e+09	1.17e+09
Round Robin	7.63e+06	4.05e+07	7.61e+06	2.60e+07	7.60e+08	7.93e+08	7.60e+08	7.78e+08
Geentic	7.73e+06	2.60e+07	7.72e+06	1.86e+07	7.70e+08	8.01e+08	7.70e+08	7.88e+08
Node Swap	1.12e+07	3.13e+07	1.03e+07	2.10e+07	1.25e+09	1.06e+09	1.08e+09	1.08e+09

Table 6.5: Costs of Placements

	$c = 4$				$c = 7$			
	$\mu_c = 5000$		$\mu_c = 500000$		$\mu_c = 5000$		$\mu_c = 500000$	
	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$	$\mu_m = 1$	$\mu_m = 1000$
Random	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Round Robin	2.530	1.108	1.353	1.124	1.499	1.337	1.447	1.501
Genetic	2.500	1.730	1.336	1.567	1.481	1.322	1.428	1.484
Node Swap	1.063	1.435	1.001	1.397	0.912	0.998	1.018	1.083

Table 6.6: Inverse Normalised Placement Costs

figures show a higher value for a worse placement and so the normalised inverse costs are shown in Table 6.6. These figures show a high degree of correlation with the figures in Table 6.4. This suggests that the cost function is a reasonable measure of program performance. Though it appears that the cost function is over estimating the relative performance improvement from the different placements. This is particularly true for the first two costs given for the Round Robin and the Genetic algorithm.

We can now perform an ANOVA on the data in Table 6.3. The data will be tested at a 5 percent level of significance ($\alpha = 0.05$). The null hypothesis is that the strategies are all equally effective. The analysis of variance table is shown in Table 6.7.

Referring to equations 3.13 and 3.14 the critical regions are $F \geq F_{0.05,1,15} = 245.9$

Source	D o F	Sum of Squares	Mean Squares	F
Between Replicates	1	1.36224e+10	1.36224e+10	10.3515
Between Treatments	15	1.55815e+18	1.03877e+17	7.89347e+07
Residual	15	1.97398e+10	1.31599e+09	
Total	31	1.55815e+18		

Table 6.7: Analysis of Variance Table - Placement Strategy Experiment

for the treatments and $F \geq F_{0.05,15,15} = 12.3$ for the replicates. Hence we can reject the null hypothesis for the treatments and accept it for the replicates. So all the placement strategies are not equally effective. Once this conclusion has been reached further investigation can be made into the effectiveness of the individual strategies.

6.5 Placement Strategy Experiment Summary

This section has shown that the experimental framework can be used to investigate process placement strategies and allows the user of the system to answer the questions posed in Section 6.2. Namely, that the placement strategies do indeed have an effect, that we can rank the strategies and that some strategies are only effective for some programs.

It appears that the round robin algorithm performs just as well for this range of programs as does the more complicated genetic algorithm. This may not be true for more unbalanced programs.

It is important to note that from our point of view, it is less interesting that a strategy be, or not be effective. What is important is the fact that the experimentation system combined with the statistical techniques can be used evaluate these strategies and other performance parameters. It has been shown that the placement strategies did have an effect on the outcome of the experiment.

6.6 Evaluation of Placement Strategies on Process Systems Example

This experiment takes a real process systems program and applies the four placement strategies described in the previous section. It uses the experiment generation software to describe the program model so that the placements can be constructed automatically.

The program under investigation is a newer version of the dynamic distillation simulation program described in Section 5.2. This program is essentially a pipeline of processes, where each process represents a tray in the distillation column. Each process computes for a certain period of time until it converges to a new solution this data is then passed to the trays/processes above and below in the pipeline. This is essentially the same program model that was used in the previous experiment, though in this instance the actual program model is not necessary.

The program was written in C and Fortran using the CS-Tools library by R C M^{c}Kinnel [McK94]. It is designed to run on a network of workstations or on a Meiko computing surface.

In order for the placement strategies to operate they need a weighted process graph. Luckily, the simulation program was already instrumented and so the time that each process spent calculating, the size of messages and the delay in sending and receiving those messages was already available. This data was used to specify some weights that could be passed to the experiment generator. The experiment generator can produce a weighted pipeline graph and other standard topologies for the placement strategies to operate on. Once the placements were

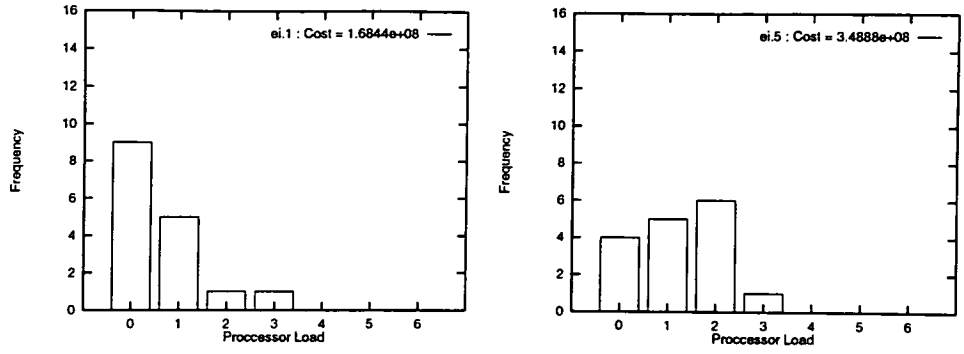


Figure 6.7: Processor Loading Frequencies Strategy 1 – Random

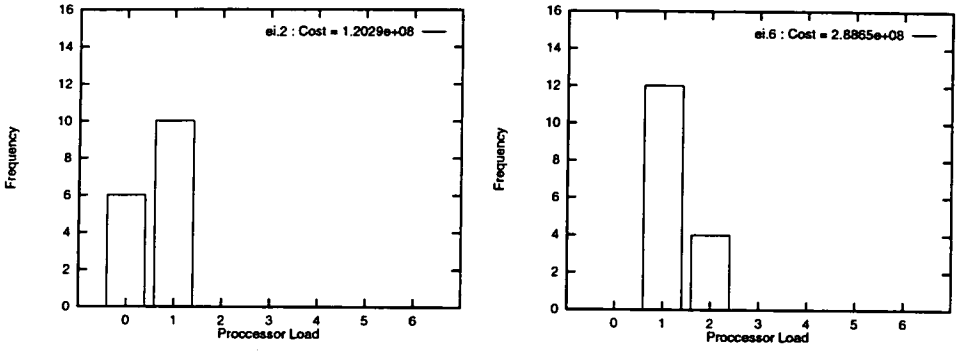


Figure 6.8: Processor Loading Frequencies Strategy 2 – Round Robin

generated they were converted into the necessary CS-Tools configuration file so that the processes could be loaded onto the correct processor.

The actual machine used was a 16 processor Meiko computing surface. Two versions of the program were used, one modelling ten distillation plates and the other modelling twenty. The simulation program was to execute for one thousand seconds of simulated time. The four placement strategies introduced in Section 6.2 were used. This gave eight different experiment instances. The loading frequencies created by each of the strategies are shown in Figures 6.7 through Figure 6.10. The ten process system is shown on the left of each figure.

Several runs were made and the average execution times are shown in Table 6.8. As these are real execution times in seconds the smaller the value the better the result. These values can be normalised by dividing the numbers in each column

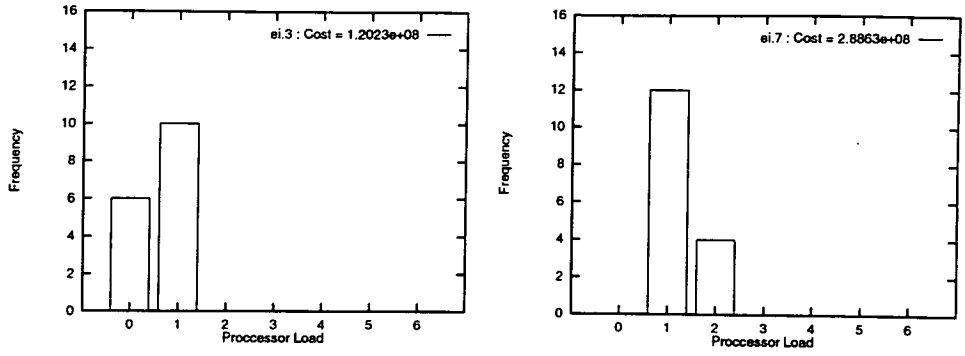


Figure 6.9: Processor Loading Frequencies Strategy 3 – Genetic

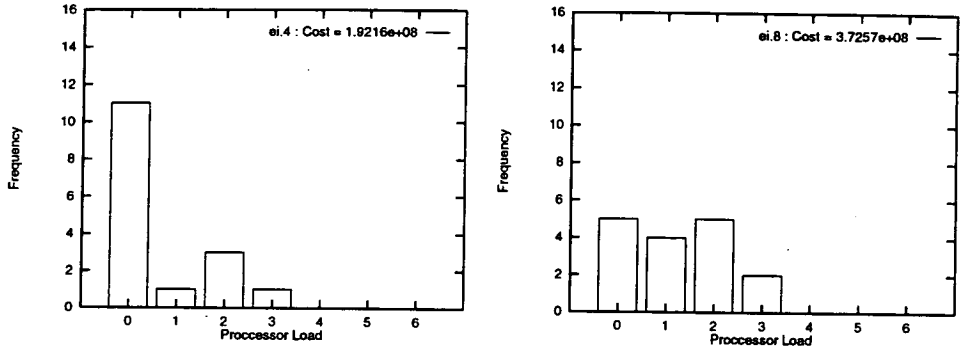


Figure 6.10: Processor Loading Frequencies Strategy 4 – Node Swapping

by the head number to produce Table 6.9. Again the smaller the number the better. The Round Robin and the Genetic algorithm both produce improvements between forty seven and twenty seven percent.

Tables 6.10 and 6.11 show the costs produced by the placement strategies and the normalised costs respectively. In this case because a high cost implies a poor placement, hence a high execution time the costs can be normalised in the same way as the execution times.

Strategy	10 Nodes	20 Nodes
Random	190.68	308.82
Round Robin	100.98	225.09
Genetic	100.98	225.11
Node Swapping	250.48	329.75

Table 6.8: Execution Times for Programs under Strategies (seconds).

Strategy	10 Nodes	20 Nodes
Random	1.00	1.00
Round Robin	0.53	0.73
Genetic	0.53	0.72
Node Swapping	1.31	1.07

Table 6.9: Normalised Execution Times.

Strategy	10 Nodes	20 Nodes
Random	1.6844e8	3.4888e8
Round Robin	1.2029e8	2.8865e8
Genetic	1.2023e8	2.8863e8
Node Swapping	1.9216e8	3.7257e8

Table 6.10: Costs for Placements

6.7 Process Systems Placement Strategy Experiment Summary

This section has shown that the placement strategies are effective on a real program. It has also shown that the experiment generation software can be used to generate an experiment for a real program provided some timing data is available for that program.

It has also confirmed our suspicions about the effectiveness of the placement strategies under investigation. The same ranking is achieved as in the simulation experiment. It is obvious by now that the Node Swapping strategy is very poor. This is also reconfirmation that the program model is realistic as we have similar behaviour under the strategies.

Strategy	10 Nodes	20 Nodes
Random	1.00	1.00
Round Robin	0.71	0.83
Genetic	0.71	0.83
Node Swapping	1.14	1.07

Table 6.11: Normalised Costs for Placements

The cost function has proved to be a good guide to the effectiveness of a placement throughout these experiments.

6.8 Summary

This chapter has shown how the experimental framework can be used to investigate process placement strategies both for real and simulated programs. There are advantages gained by having relatively automatic experimentation directly from the experimental design through to the statistical analysis.

Through the use of the experimental framework the systematic investigation of the interaction between different placement strategies and program types becomes possible.

Chapter 7

Conclusions and Future Work

7.1 Precis

This chapter begins with a restatement of the contribution of the thesis and then these points are discussed in more detail and conclusions drawn. The last section describes possible future work that could be performed.

7.2 Contributions of Thesis and Discussion

- The application of the methodology of statistical design of experiments to study the performance of arbitrary parallel programs.

- The use of synthetic programs to study a wide range of parallel programs.
- The employment of these techniques in a systematic investigation of static placement strategies.
- A demonstration of the utility of these techniques in producing performance models for both synthetic and real programs.

When this work began the application of statistical techniques to parallel program performance evaluation had not previously been applied in a systematic fashion. This work has shown that serious performance analysis can be conducted on an arbitrary parallel program where it is possible to specify values for parameters or even where the parameters can only be measured. The identification of performance limiting factors is possible using these techniques on any parallel program. Using a synthetic program model this can be done largely automatically with the experimentation software demonstrated here.

This work has demonstrated that synthetic programs can be used to adequately represent a communicating sequential process program. The behaviour of the synthetic program was demonstrated to be similar to that of a real program. This implies that the general structure and time averaged communications and computation behaviour is more important than the individual instructions, control flow and synchronization for performance analysis at this level. Simplified program models are adequate for performance evaluation and early large scale tuning of applications. Simplified program models also allow rapid performance evaluation as opposed to the real program.

It has also been demonstrated that the application of statistical techniques combined with synthetic programs and the rest of the framework developed here is a sufficient system for investigating the performance of process placement strate-

gies. This systematic evaluation of placement strategies had not been performed before. It is also proposed that by using synthetic programs more information is gained about the placement strategies effect on a range of programs rather than just individual programs. The use of an artificially generated program allows the parameter levels to be specified, whereas an off the shelf program may not be very representative.

The original motivation for this work was to study different placement strategies, but this was soon extended into a study of the ways of performing quantitative evaluations of the effects of relevant factors on parallel program performance. The system that has been constructed is capable of performing a wide range of performance studies.

The system can also be used to predict the performance of parallel programs. I have shown that for the simple program models described here linear regression models can be used with acceptable results. More complicated models are possible and necessary for more complicated programs. More complicated models are not necessarily needed as useful predictions can be obtained from the regression equations derived from the synthetic programs for programs whose control structure does not conform exactly to the loosely synchronized model assumed.

7.3 Future Work

There are three main areas in which this work can continue: the modelling system itself, the software engineering tools and extension of the experimental designs. The following sections deal with each of these areas, followed by a section with some subsidiary points.

7.3.1 Modelling System

Modelling and simulation will always be part of performance analysis. The concept of Schrödinger's cat tells us that we can not know if we are interfering with the system by monitoring it. My proposal was that the best approach was to simulate rather than monitor the system. As parallel machines become faster and faster the simulation of those machines can also speed up. One obvious area of future work would be in the modification of a general purpose parallel simulation system to be able to model parallel computations as MIMD does.

The models of parallel programs can be expanded to take in different and more complex program behaviour. It would be useful to extend the modelling system to be aware of new hardware and software such as networks of workstations or the Transputer T9000 chip.

7.3.2 Software Engineering

The rapid development of proper operating systems and software engineering tools has caught up with the fast development of parallel hardware. It is now common practice for supercomputers to be hosted by a Unix workstation or run Unix on its processors rather than a proprietary operating system. This means that standard and familiar software engineering tools can be used to develop programs and conduct performance studies. A performance study now could probably use a meta-modeller or modelling engine that was a real program that modified its behaviour based on the experiment instance. This would give results which could be measured in the terms of execution time rather than computation achieved. An early version of this type of meta-modeller was produced for use

with the CS-Tools environment but it was cumbersome. A meta-modeller of this nature could now be built using the PVM software.

Message passing systems seem to be very much in vogue at the moment. This is largely due to the popularity and robustness of the Parallel Virtual Machine (PVM) software and also the proposal for the Message Passing Initiative (MPI) [GBD+93, Mes94]. PVM is now provided as standard on several parallel machines. This type of message passing software has also encouraged cluster computing and the use of networks of workstations as loosely coupled parallel machines. A program written using PVM can, in theory, be taken from a development environment of a single workstation through to a network of workstations and on to a high performance machine such as the Cray T3D with little effort. This means that is likely that more productive use can be made of the most expensive machines as bugs can be ironed out on standard workstations.

Perhaps the development of a single placement module that could be built into the operating system of our new distributed memory multicomputers is still some way away. It would take the set of modules that needed to be run and execute that monitoring it could then run the program again in a better configuration using the most appropriate placement strategy. A set of rules for choosing between placement strategies based on the calculated parameters for the program and the applicability of different strategies to different programs. The parameter levels for a program could be estimated with normal sequential profiling tools and by building a database of program performance during the development of an application. Then the placement of the program could be developed alongside the program code. This cooperative development should lead to a much better placement.

7.3.3 Experimental Design

Phillips has used the framework developed in this thesis to investigate dynamic placement strategies [Phi94]. Dynamic placement strategies are necessary if the programs have a time variant nature. This means that the initial static placement strategy is invalidated by the behaviour of the program and a change needs to be made. Phillips also developed a more complicated program model for describing time variant programs. There is work here that is necessary to see how important a good initial placement is when a migration strategy is to be used. The combination of a static strategy, which has time to analyse a program in detail and then pass on these details to the migration strategy may be productive.

It may also be suggested that as well as the placement strategy being developed alongside the program the use of statistical performance analysis becomes part of the standard software engineering tools for parallel programming. This is an increasing interest in systematic experimentation for programmers, as can be seen through the work of Lyon and coworkers on time perturbation tuning [LSK94].

7.3.4 Subsidiary Points

Other placement strategies that were not covered here may have better performance. The technique of simulated annealing could be investigated in more detail. Some initial work was undertaken for this study but the results were fairly inconclusive. The analogy of a physical system cooling to a stable state seems quite similar to the arrangement of processes across processors with their links being equivalent to chemical bonds or nearest neighbour interactions. I do not think that I gave Genetic Algorithms enough study and I may not have used

them to their best potential. I am convinced, based on evidence of other uses of genetic algorithms that they could produce better results. They were used as an example strategy of a different approach and in that way they fulfilled their role. There are a lot of parameters which can be manipulated with the genetic algorithm itself and I did not want to start a whole new parameter study on that algorithm. Once the system was producing a good stable population the levels of the factors affecting the genetic algorithm were left intact.

It may also be feasible to do a full state search of all possible placement combinations using a branch and bound technique. This would need a cost function that was quick to evaluate so that decisions could be made between different placements. The difficulty in this approach would be in ordering the possible combinations in such a way that pruning of the search tree is possible.

The most important future work is to carry on with the application of the statistical techniques to the performance evaluation of parallel systems. There is much to do in the study of placement strategies and their interaction with the programs.

And Finally ...

Large parallel machines are very expensive things to use and anything that aids the productive use of these machines is a benefit.

Glossary

Asynchronous/Synchronous Programs Shu and Wu suggest that application programs can be classified into three categories: synchronous, loosely synchronous and asynchronous [SW95].

Synchronous programs have a uniform structure. In each time step every processor executes the same operation over different data, resulting in a naturally balanced computational load.

Loosely Synchronous programs can be structured iteratively with two phases: the computation phase and the synchronisation phase. In the synchronisation phase, processors exchange information and synchronise with each other. In a system that supports dynamic migration some process migration may occur at this phase boundary. In the computation phase, different processors can operate independently.

Asynchronous programs have no synchronous structure. Processes may communicate with each other at any time. The computation structure can be very irregular and the load imbalanced.

Domain Used in the context of the "experimental or modelling domain" - this means the system or systems that constitute the parallel program and machine under study. This could be a program written in the MIMD modelling system or it could be a distributed memory computer and software.

Fractional Factorial Design For the description of a Full Factorial Design see Section 3.3.2. Sometimes the number of experiments required for a full factorial design is too large. This may happen if the number of factors or their levels is large. It may not be possible to use a full factorial design due to the expense or the time required. In such cases a fraction of the full factorial design can be used. A fractional factorial design chooses some of the combinations of the factors. These designs save time and expense but the information gained from such an experiment is less than that from a full experiment. It may not be possible to get interactions from all factors due to confounding. On the other hand, if some of the interactions are known to be negligible, this may not be considered a problem.

Granularity The size of a process chosen for parallelism is called the granularity or grain size. Krishnamurthy defines granularity with the following three levels [Kri89]:

Data Level Here the parallelism results in effecting the same operation on multiple data items simultaneously, for example vector addition. This is called "fine granularity".

Operation Level Here the parallelism arises by the execution of independent instructions simultaneously. For example consider $(a + b) \times (c/d)$ here we can execute the addition and division simultaneously. This is called medium granularity.

Task Level Here the parallelism is at a specific task level or program level, for example simultaneous reading from a database. This is called coarse granularity.

For task based programs the grain size problem occurs. This problem is how to determine the best clustering of tasks in the task graph such that the task graph execution is minimized. For more details see Kruatrachue [Kru87].

Hot spot A hot spot is a place where a resource is placed under a large demand relative to other resources. This can be a physical resource such as a network link or a disk, or a software resource such as a process or even a memory location. The Gecko software system showed hot spots on transputer systems through the use of colours, red being hot [SB89, Har89a].

Process Template This is a parameterised piece of program code. The parameters to the program define the functional and temporal behaviour of the process. A simple process template which loops continually computing and sending messages may take two parameters; the number of times to loop and the size of messages to send. More complex process templates can be constructed whose behaviour varies with time. The code for the process is written in the domain of the meta-modeller.

Scatter and gather This is a term describing a master/slave parallel computation where the master scatters data amongst many slaves and then waits to gather the results. This operation is usually repeated. It is also common for all processes to perform scatter/gather operations with their nearest neighbours.

Simulation Time This is the logical time within the simulation. This is specified in logical time units.

Bibliography

[AJ88] Rakesh Agrawal and H V Jagadish. Partitioning techniques for large–grained parallelism. *IEEE Transactions on Computers*, C–37(12):1627–1634, December 1988.

[Ame92] American National Standards Institute (ANSI), 1430 Broadway, New York, NY 10018. *Fortran 90*, ANSI X3.198-1992 edition, September 1992.

[AO85] S J Allan and R R Oldehoeft. HEP SISAL: Parallel functional programming. In *Parallel* MIMD *Computation:* HEP *Supercomputer and its applications*, pages 123–150. The MIT Press, Cambridge, MA, 1985.

[Bak87] James E Baker. Reducing bias and inefficiency in the selection algorithm. In J J Grefenstette, editor, *Proceedings of the 2nd International Conference*, Genetic Algorithms and their Applications, pages 14–21, LEA, Cambridge, MA, July 1987.

[BBC86] G Balbo, S Bruell, and S Chanta. Combining queueing networks and generalized stochastic petri net models for the analysis of some software blocking problems. *IEEE Transactions on Software Engineering*, 12:561–576, April 1986.

[BBK68] G Barnes, R Brown, and M Kato. The ILLIAC IV computer. *IEEE Transactions on Computers*, 17(8):746–757, 1968.

[BDQ86] J C Bermond, C Delorme, and J J Quisquater. Strategies for interconnection networks : Some methods from graph theory. *Journal of Parallel and Distributed Computing*, 3:433–449, 1986.

[Bir86] G B Birtwistle. *Discrete Event Modelling on Simula.* Macmillan, London, 1986.

[Bok81a] Shahid H Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30:207–214, March 1981.

[Bok81b] Shahid H Bokhari. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Transactions on Software Engineering*, SE–7 No. 6:583–589, November 1981.

[Brä93] T Bräunl. *Parallel Programming: an Introduction*, chapter 3, pages 17–29. Prentice Hall International, 1993.

[Cam87] C W Campbell. POOPS : A prolog object oriented programming system. *Department of Computer Science University of Aberdeen Internal Report*, AUCS/TR8703, 1987.

[Can88] R Candlin. Process migration in parallel computations, computer science report csr–284–88. Technical report, Department of Computer Science, Edinburgh University, October 1988.

[CC87] B Ciciani and G Cioffi. A proposal for autonomous and dynamic cooperating processes. *1987 IFIP Distributed Processing Conference*, pages 339–353, 1987.

[CCL88] Marina Chen, Young–il Choo, and Jingke Li. Crystal : From functional description to efficient parallel code. *The Third Conference on Hypercube Concurrent Computers and Applications*, Volume 1 – Architecture, Software, Computer Systems and General Issues:417–433, January 1988.

[CFPS92] R Candlin, P Fisk, J Phillips, and N Skilling. Studying the performance properties of concurrent programs by simulation experiments on synthetic programs. *Performance Evaluation Review, Special Issue, 1992 ACM Sigmetrics and Performance 92 International Conference on Measurement and Modelling of Computer Systems*, 20(1), June 1992.

[CG89a] N Carriero and D Gelertner. *How to write parallel programs: A first course.* The MIT Press, Cambridge, MA, 1989.

[CG89b] N Carriero and D Gelertner. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[CG89c] N Carriero and D Gelertner. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[CK88] T L Casavant and Jon G Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[CL89] R Candlin and Q Luo. Communications performance in occam programs on the meiiko computing surface. *Proceedings of Parallel Computing 1989*, 1989.

[Cla90] Lyndon J Clarke. *Achieving Parallel Performance in Scientific Computations.* PhD thesis, University of Edinburgh, September 1990.

[CLS89] R Candlin, Q Luo, and N Skilling. The investigation of communication patterns in occam programs. In John Wexler, editor, *Developing Transputer Applications*, Proceedings of the 11th Occam User Group Technical Meeting, pages 99–108. IOS Press, ISBN: 90 5199 020 0, September 1989.

[Com87] Simula Standard Committee. *Simula Programming Language.* Simula Standard Committee, 63 61 14 SIS, Box 3295, Stockholm, Sweden, 1987.

[Cor87] Thinking Machines Corporation. **Lisp Programming Language*, 1987.

[Cor91] Thinking Machines Corporation. *Connection Machine CM-200 Series Technical Summary.* Thinking Machines Corporation, Cambridge, Massachusetts, June 1991.

[Dav91] Lawrence Davis. *Handbook of Genetic Algorithms.* Van Nostrand Reinhold, 1991.

[DS88] J G Donnet and D B Skillicorn. Simulated annealing and code partitioning for distributed multimicroprocessors. Technical report, Queens University, Department of Computing and Information Science Technical Report, 1988.

[ERL90] Hesham El-Rewini and T G Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, June 1990.

[FB89] David Fernandez-Baca. Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, 15 No. 11:1427–1436, November 1989.

[FFI85] G A Frank, D I Franke, and W F Ingogly. An architectural design and assessment system. *VLSI Design*, 6:30–50, August 1985.

[Fis58] Ronald A Fisher. *Statistical Methods for Research Workers.* Number 5 in Biological monographs and manuals. Edinburgh London : Oliver & Boyd, 13th ed. edition, 1958.

[Fly66] Michael J Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12), December 1966.

[FO88] Edward W Felton and Steve W Otto. Coherent parallel C. *The Third Conference on Hypercube Concurrent Computers and Applications*, Volume 1 - Architecture, Software, Computer Systems and General Issues:440–450, January 1988.

[FT88] I T Foster and S Taylor. Flat parlog: A basis for comparison. *International Journal of Parallel Programming*, 16(2), 1988.

[FT90] I T Foster and S Taylor. *Strand: New concepts in parallel programming.* Prentice Hall, 1990.

[FW80] John E Freund and Ronald E Walpole. *Mathematical Statistics*, chapter 15. Prentice Hall, 1980.

[Gai90] Jason Gait. Scheduling and process migration in partitioned multiprocessors. *Journal of Parallel and Distributed Computing*, 8:274–279, 1990.

[GBD+93] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3.0 Users Guide and Reference Manual.* Oak Ridge National Laboratory, Tenessee, Technical Report ORNL/TM-12187 edition, February 1993.

[Gre90] John J Grefenstette. *A User's Guide to GENESIS - Version 5.0.* University of Colorado, 1990.

[GS87] K Gates and D Socha. Programming NCUBE's with a graphical parallel programming environment versus an extended sequential language. In M T Heath, editor, *Hypercube Multiprocessors 1987*, pages 17–27, Philadelphia, PA, 1987. SIAM.

[Gui88] T Guilfoy. The simulation of parallel programs. Master's thesis, Department of Computer Science, University of Edinburgh, September 1988.

[Han77] W Handler. The impact of classification schemes on computer architecture. In *Proceedings of the 1977 International Conference on Parallel Processing (New York)*, pages 7–15. IEEE, 1977.

[Har89a] E Hart. *Gecko User Manual Version 1.* Polytechnic of Central London, March 1989.

[Har89b] E Hart. *Transim User Manual Version 2.* Polytechnic of Central London, March 1989.

[HG94] Fred Hemery and Jean-Marc Geib. A platform to study dynamic scheduling strategies. In Monique Becker, Luc Litzler, and Michel Trehel, editors, *Transputers '94 Advanced Reserach and Industrial Applications*, pages 179–190. IOS Press, 1994.

[HK72] Maurice Hanan and Jerome M Kurtzberg. A review of the placement and quadratic assignment problems. *SIAM Review*, 14(2):324–342, April 1972.

[Hl84] P Heidelberger and S S lavenberg. Computer performance evaluation methodology. *IEE Transactions on Computers*, 33(12):1195–1220, 1984.

[Hoa84] A Hoare. *Communicating Sequential Processes.* Ellis Horwood, 1984.

[Hol75] John H Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor : The University of Michigan Press, 1975.

[HP92] Hans-Ulrich Heiss and Achim Payer. PASTE: A tool for evaluation of processor allocation strategies. *Computer Performance Evaluation '92: Modelling Techniques and Tools*, 1992.

[HQ91] Philip J Hatcher and Michael J Quinn. *Data Parallel Programming on* MIMD *Computers.* Scientific and Engineering Computation Series. The MIT Press, 1991.

[Inm84] Inmos. *The Occam Programming Manual.* Prentice Hall International Series in Computer Science. Inmos Limited, 1984.

[Inm88] Inmos. *Occam 2 Reference Manual.* Prentice Hall International Series in Computer Science. Inmos Limited ISBN 0-13-629312-3, 1988.

[Inm89a] Inmos. *The Transputer DataBook.* Inmos Limited, 1000 Aztec West, Almondsbury Bristol, BS12 4SQ, 1989.

[Inm89b] Inmos. *The transputer instruction set – a compiler writers' guide.* 100 Aztec West, Almondbury, Bristol, BS12 4SQ, 1989.

[Jai91a] Raj Jain. *The Art of Computer Systems Performance Analysis.* Wiley Professional Computing. John Wiley & Sons Inc., 1991.

[Jai91b] Raj Jain. *The Art of Computer Systems Performance Analysis*, chapter 2, pages 22–25. Wiley Professional Computing. John Wiley & Sons Inc., 1991.

[JS88] M R Jerrum and A J Sinclair. Fast uniform generation of regular graphs. Technical report, Department of Computer Science, Edinburgh University, 1988.

[KM88] O Kramer and H Muhlenbeim. Mapping strategies in message-based multiprocessor systems. *Parallel Computing*, 9:213–225, 1988.

[KME89] Alex Kapelnikov, Richard R Muntz, and Milos D Ercegovac. A modeling methodology for the analysis of concurrent systems and computations. *Journal of Parallel and Distributed Computing*, 6:568–597, 1989.

[KP94] J P Kitajima and B Plateau. Modelling parallel program behaviour in ALPES. *Journal of Information and Software Technology*, 36(7):457–464, 1994.

[Kri89] E V Krishnamurthy. *Parallel processing principles and practice.* International Computer Science Series. Sydney Wokingham Addison-Wesley, 1989.

[Kru87] B Kruatrachue. *Static Task Scheduling and Grain Packing in Parallel Systems.* PhD thesis, Department of Computer Science, Oregon State University, 1987.

[Kun82] H T Kung. Why systolic architectures? *Computer*, 15:37–45, 1982.

[Lo88] Virginia Mary Lo. Heuristic algorithms for task assignments in distributed systems. *IEEE Transactions on Computers*, 37:1384–1397, November 1988.

[LSF88] C C Lee, S Skedzielewski, and J Feo. On the implementation of applicative languages on shared memory, MIMD multiprocessors. *SIGPLAN Notices*, 23(9):161–172, September 1988.

[LSK94] Gordon Lyon, Robert Snelick, and Raghu Kacker. Synthetic-perturbation tuning of MIMD programs. *The Journal of Supercomputing*, 8(1):5–27, 1994.

[McK94] R C McKinnel. *An Investigation Into the Application of Parallel Computers for the Dynamic Simulation of Chemical Processes.* PhD thesis, Edinburgh University, 1994.

[Mei92] Meiko Limited, 1000 Aztec West, Bristol. *Meiko CS–Tools Reference Manual*, 1992.

[Mes94] The Message Passing Interface Forum (MPI), Argonne National Laborotory. *Message Passing Interface Forum, Version 1.0*, 1994.

[MGSK87] H Muhlenbeim, M Gorges-Schleuter, and O Kramer. New solutions to the mapping problem of parallel systems: The evolution approach. *Parallel Computing*, 4:269–279, 1987.

[ML82] Perng-Yi Richard Ma and Edward Y S Lee. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C–31 No 1:246–252, January 1982.

[MS89] R Miller and Q Stout. An introduction to the portable parallel programming language SEYMOUR. In *Thirteenth Annual International Computer Software and Applications Conference.* IEEE Computer Society Press, 1989.

[MTM88] Richard P Ma, Fu-Sheng Tsung, and Mae-Hwa Ma. A dynamic load balancer for a parallel branch and bound algorithm. *The Third Conference on Hypercube Computers and Applications*, 2 - Applications:1505–1513, January 1988.

[MY87] B P Miller and C Q Yang. An interactive and automatic performance measurement tool for parallel and distributed systems. In *Proceedings of the 7th International Conference on Distributed Computing Systems.* IEEE Press, New York, 1987.

[NCTI92] M G Norman, G Chochia, P Thanisch, and E Issman. Predicting the performance of the Diamond DAG computation. Technical report, Department of Computer Science, University of Edinburgh, 1992.

[NHM+87] J Nehmer, D Haban, F Mattern, D Wybranietz, and H D Rombach. Key concepts of the INCAS project. *IEEE Transactions on Software Engineering*, 13:913–923, August 1987.

[NMF87] Richard E Nance, Robert L Moose, and Robert V 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, May 1987.

[Ost89] A Osterhaug. *Guide to Parallel Programming on Sequent Computer Systems.* Prentice Hall, Englewood Cliffs, New Jersey, Second edition, 1989.

[Pal85] Edgar M Palmer. *Graphical Evolution : An Introduction to the Theory of Random Graphs.* Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons ISBN 0-471-81577-2, New York, 1985.

[PD89] Kee-Hyun Park and Lawrence W Dowdy. Dynamic partitioning of multiprocessor systems. *International Journal of Parallel Programming*, 18 No 2:91–120, 1989.

[Pet62] C A Petri. *Kommunikation mit Automation.* PhD thesis, University Bonn, Germany, 1962.

[Pet81] J Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[PFMS91] J W Ponton, E S Fraga, R C McKinnel, and N Skilling. Parallelisation strategies for process modelling on MIMD computers. *The 1991*

IChemE Research and Technology Symposium and Recruitment Affair, pages 143–146, February 1991.

[Phi94] J Phillips. *A Statistical Investigation of the Factors Influencing the Performance of Parallel Programs, with Application to a Study of Process Migration Strategies.* PhD thesis, Department of Computer Science, University of Edinburgh, 1994.

[PL88] Chrisila C Pettey and Michael R Leuze. Parallel placement of parallel processes. *The Third Conference on Hypercube Concurrent Computers and Applications*, Volume 1 - Architecture, Software, Computer Systems and General Issues:232–238, January 1988.

[Poo85] R J Pooley. *Introduction to Programming in Simula.* Blackwell, 1985.

[RB84] Anthony P Reeves and D Bergmark. Parallel Pascal: An extended Pascal for parallel computing. *Journal of Parallel and Distributed Computing*, 1:64–80, January 1984.

[RB87] Anthony P Reeves and D Bergmark. Parallel Pascal and the FPS hypercube supercomputer. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 385–388, 1987.

[RJ87] J R Rose and G L Steel Jr. C*: An extended C language for data parallel programming. Technical Report PL 87-5, Thinking Machines Corporation, 1987.

[Sar87] Vivek Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. *Stanford University Technical Report*, CSL–TR–87–328, 1987.

[SB89] Marc Stephenson and Oliver Boudillet. Gecko : A graphical tool for the modeling and manipulation of Occam software and transputer hardware topologies. *Occam User Group Technical Meeting*, 9, 1989.

[Sch80] Robert Schreiber. Systolic arrays for eigenvalue computation. Technical report, Department of Computer Science, Stanford University, California, 1980.

[Seq87] Sequent Computer Systems, Inc. *Symmetry Technical Summary*, 1003-44447 Revision A edition, 1987.

[Sin87] J B Sinclair. Efficient computation of optimal assignments for distributed tasks. *Journal of Parallel and Distributed Computing*, 4:342–362, 1987.

[SK86] R G Scarborough and H G Kolsky. A vectorizing FORTRAN compiler. *IBM Journal of Research and Development*, 30(2), March 1986.

[Ski89] N Skilling. *Occula - An Occam 2 to MIMD translator*, 1989.

[Ski92] N Skilling. **eg** – *The Experiment Generator*, 1992.

[Smi92] Justin R Smith. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1992.

[Sny84] L Snyder. Parallel programming and the Poker programming environment. *Computer*, 17(7):27–36, July 1984.

[Sto88] L Storc. Sequent balance series. In R G Babb, editor, *Programming Parallel Processors*, pages 143–154. Addison-Wesley, MA, 1988.

[SW95] Wei Shu and Min-You Wu. Asynchronous problems on SIMD parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):704–713, July 1995.

[Tha90] P Thanisch. A review of mapping strategies for parallel programs. Technical report, Department of Computer Science, University of Edinburgh, July 1990.

[YM89] C Q Yang and B P Miller. Performance measurement for parallel and distributed programs: a structured and automatic approach. *IEEE Transactions on Software Engineering*, 15:1615–1629, December 1989.

[ZBG88] Hans P Zima, Heinz-J Bast, and Michael Gerndt. SUPERB : A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.

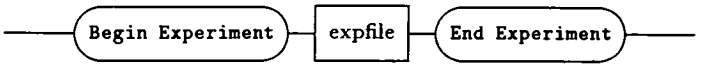
Appendix A

Experiment Generator Reference

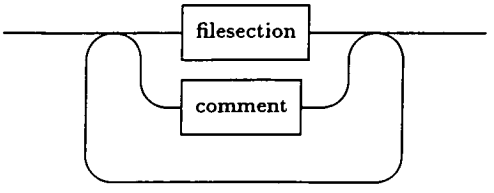
This appendix includes the complete syntax diagrams for the Experiment Definition language used by the Experiment Generator **eg**.

A.1 Syntax Diagrams for the Experiment Design Language

experiment



expfile



filesection

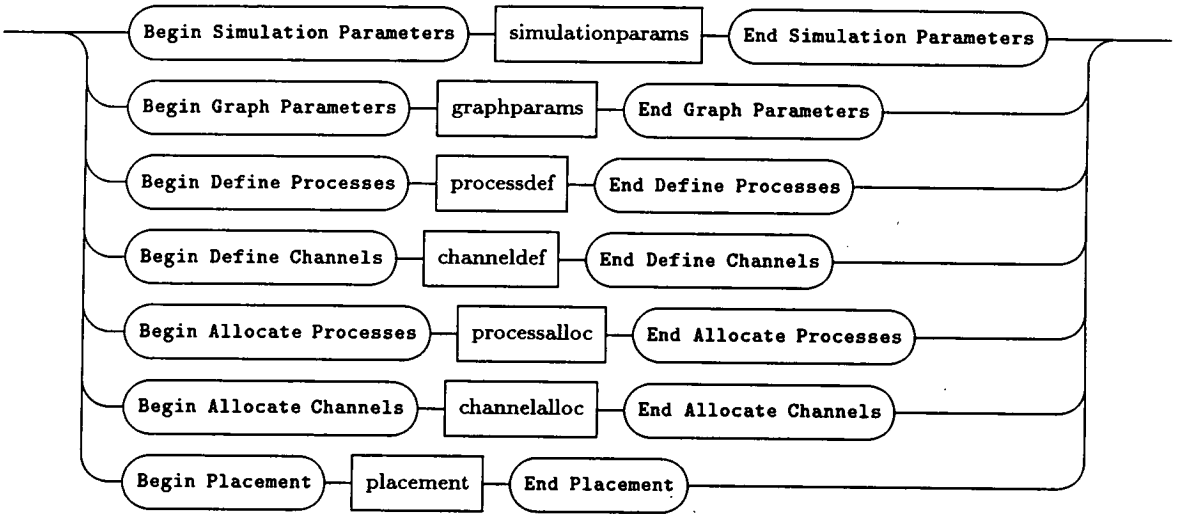


Figure A.1: Top level components of an experiment description

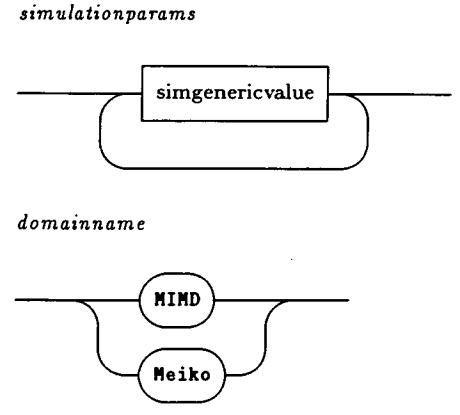


Figure A.2: Simulation Parameters

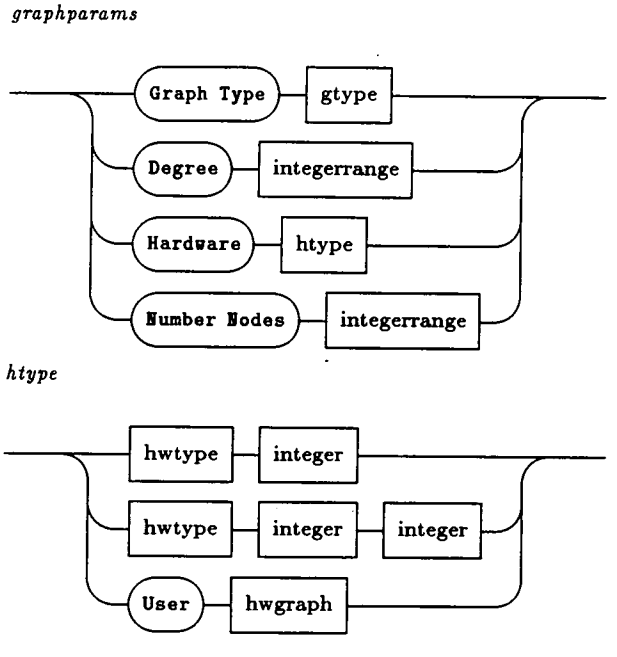
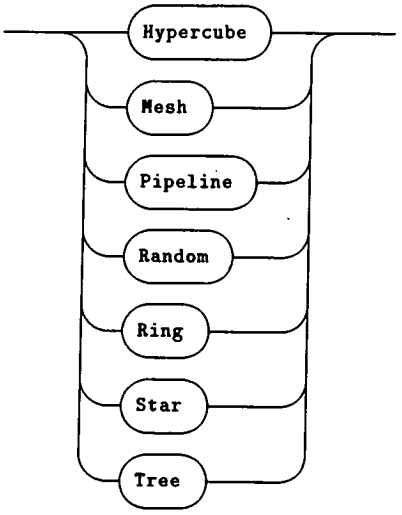
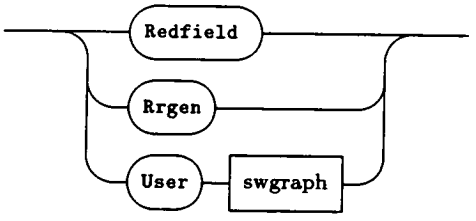


Figure A.3: Graph Parameters

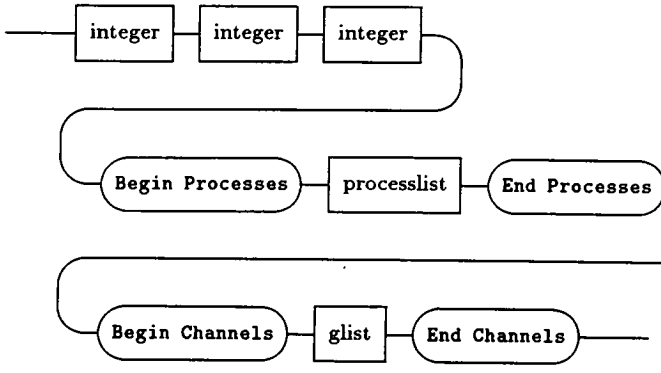
hwtype



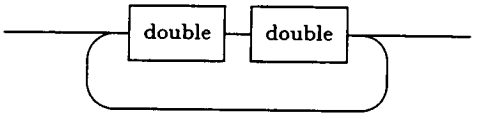
gtype



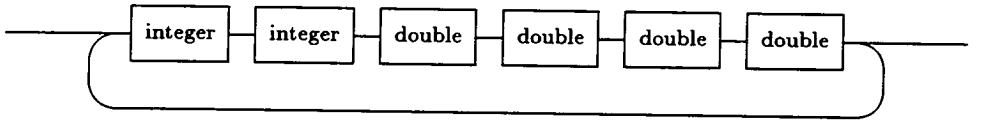
swgraph



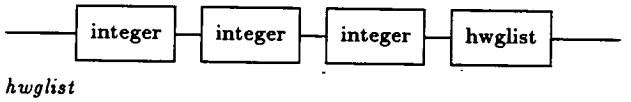
processlist



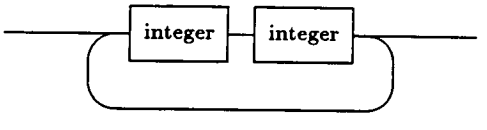
glist



hwgraph



hwglist



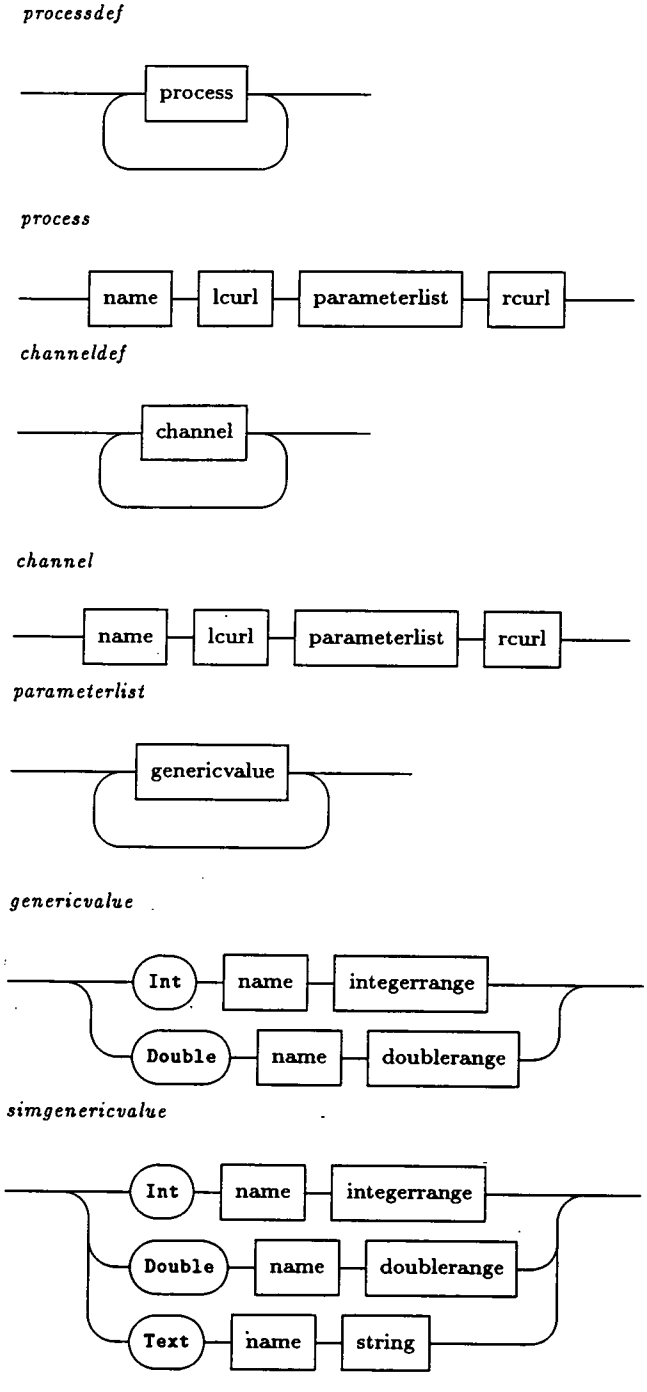


Figure A.4: Process and Channel Definition Syntax

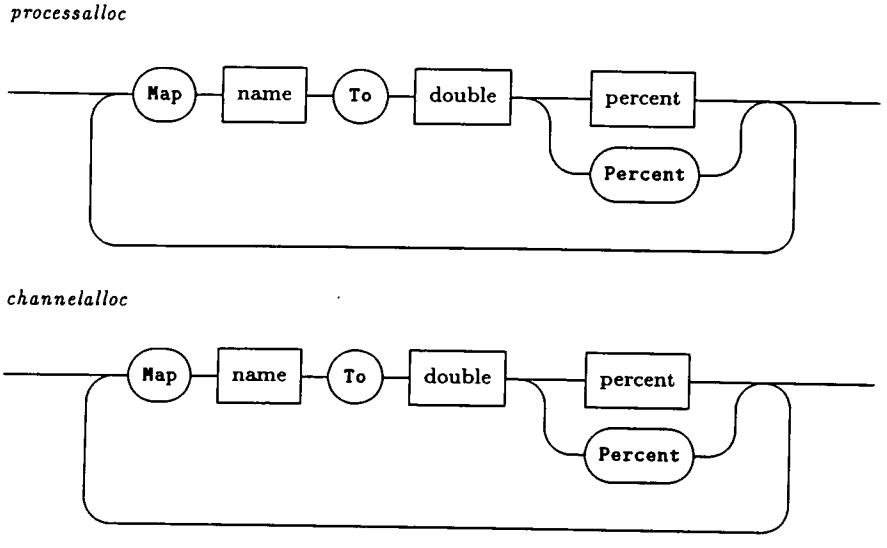


Figure A.5: Process and Channel Allocation Syntax

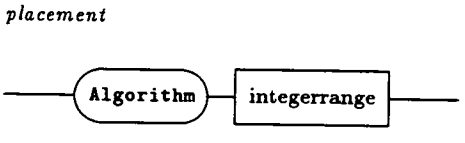
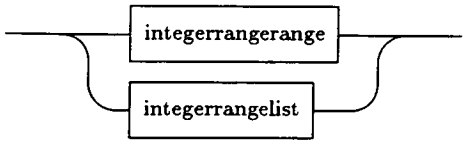
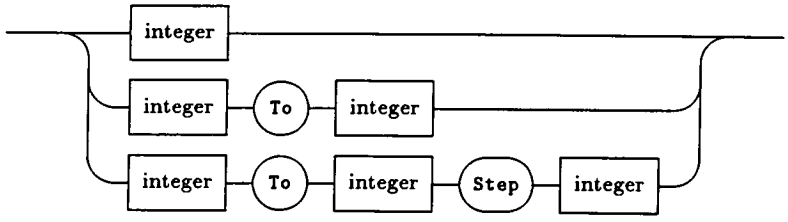


Figure A.6: Placement Syntax

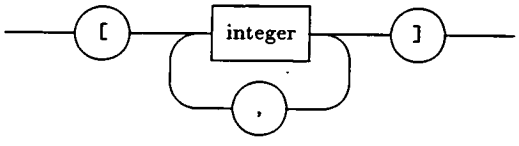
integerrange



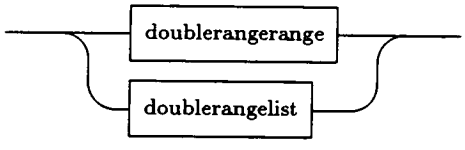
integerrangerange



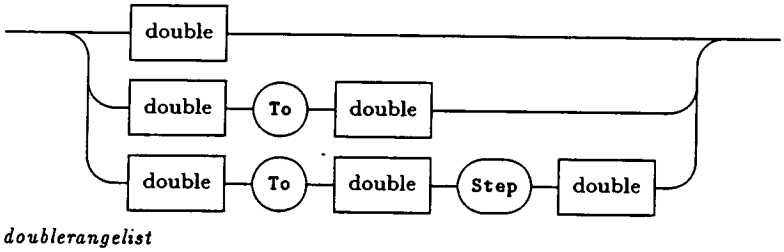
integerrangelist



doublerange



doublerangerange



doublerangelist

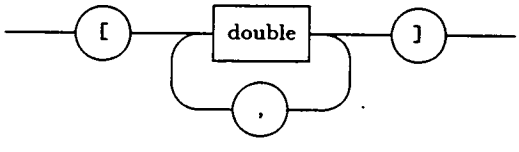
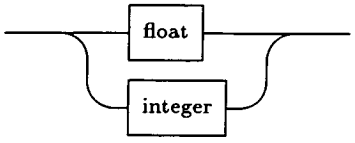
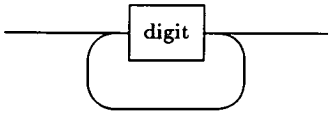


Figure A.7: Range Syntax

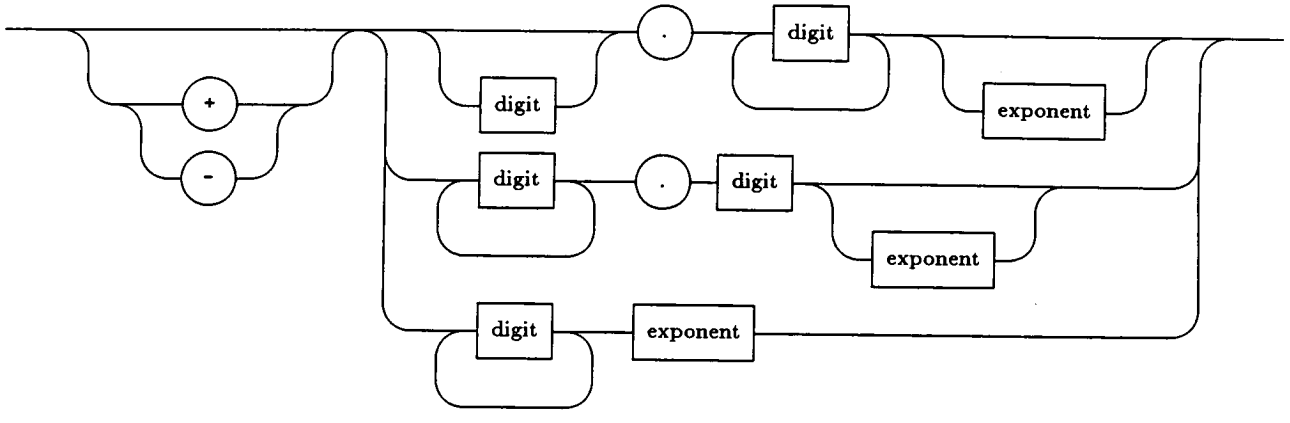
double



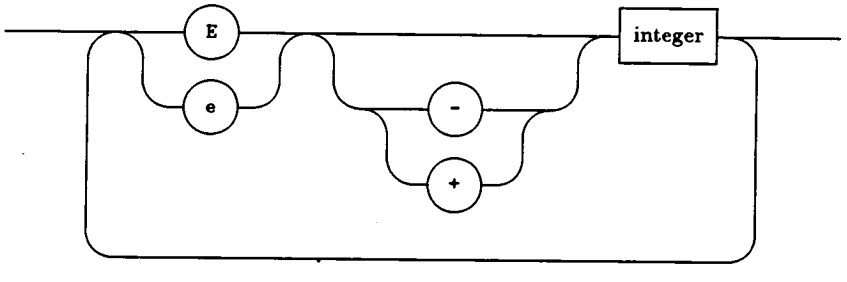
integer



float



exponent



string

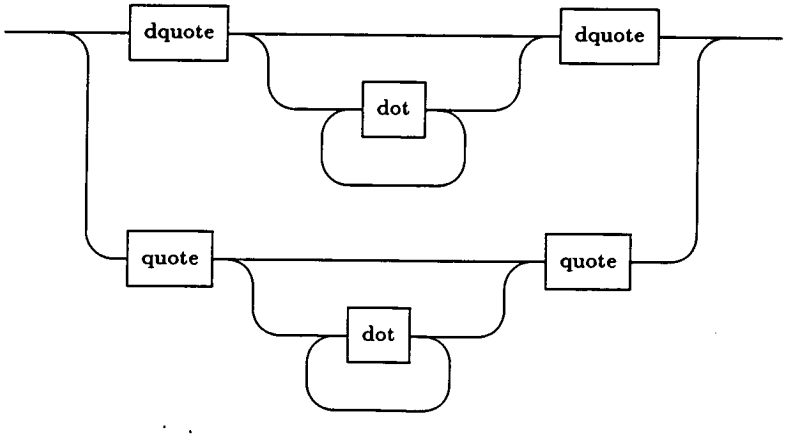
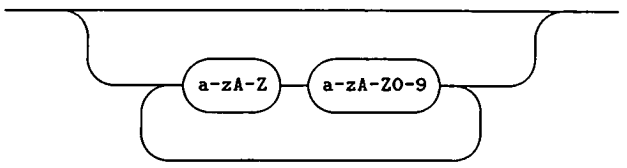


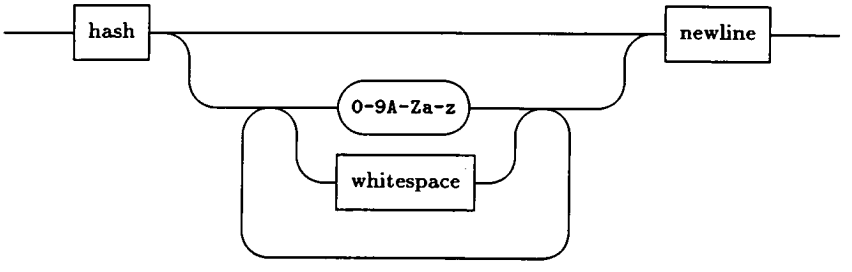
Figure A.8: Basic Token Syntax

name

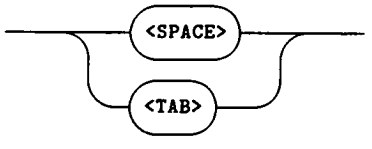


digit

comment



whitespace



newline

Figure A.9: Basic Token Syntax Continued

A.2 Syntax Diagrams for the Experiment Instance File

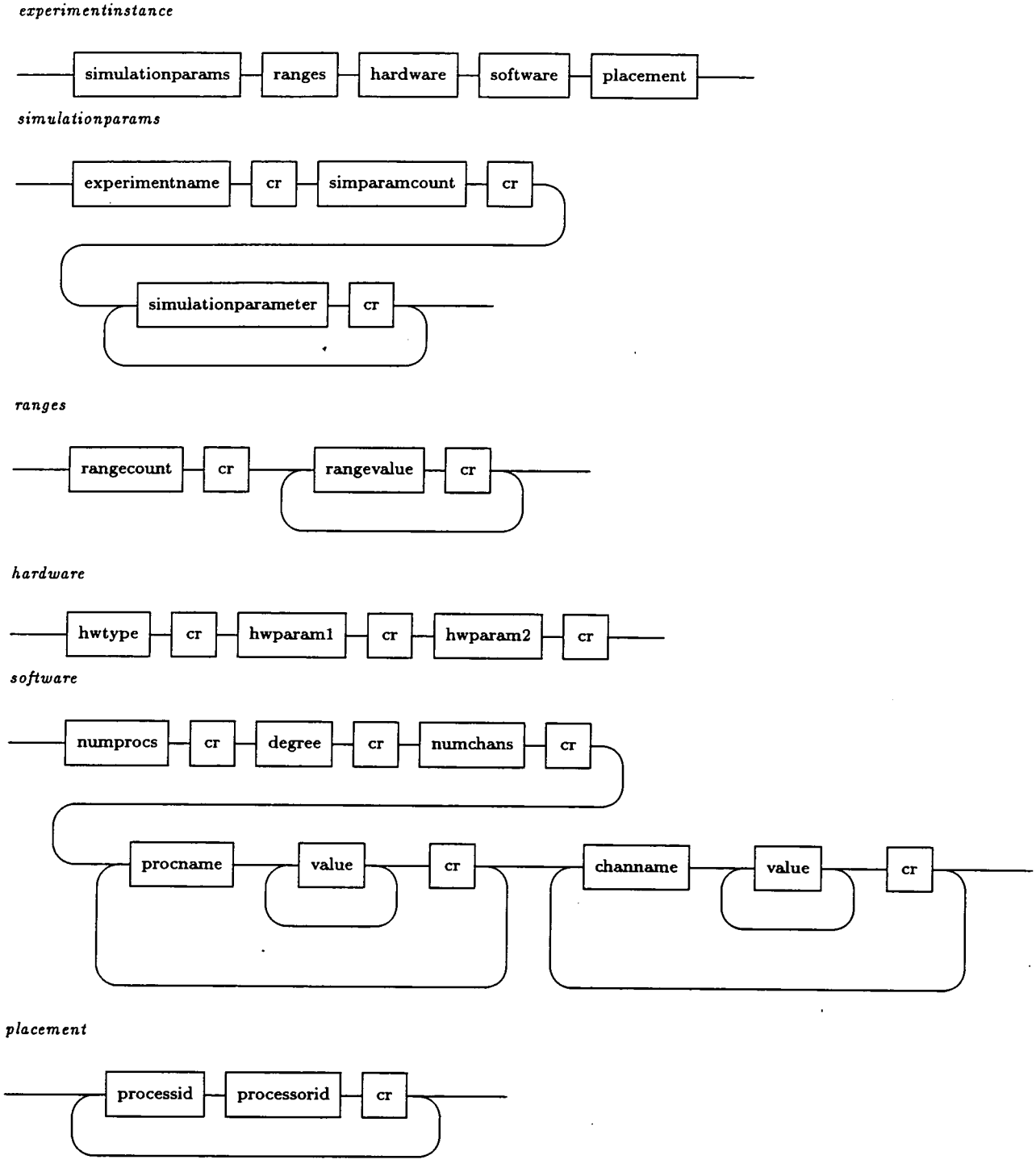


Figure A.10: Output Format for Experiment Instance