PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1016a ANSI/ISO #2 EQUIVALENT

1.0

1.1

1.25

PRECISION™ RESOLUTION TARGETS
PROCESSOR SCHEDULING IN A TRANSPUTER

SYSTEM

by

Yuet Ning Chan, B.C.S.

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario
April 23, 1996
© Copyright
1996, Yuet Ning Chan
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-13829-1
The undersigned recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis.

"Processor Scheduling in a Transputer System"
submitted by
Yuet Ning Chan, B.C.S.

in partial fulfillment of the requirements
for the degree of Master of Computer Science

Thesis Co-Supervisor

Thesis Co-Supervisor

Director, School of Computer Science

Carleton University
May 13, 1996
ABSTRACT

Multicomputers are distributed memory systems in which each processor has its own private memory. Processor communication is facilitated by a inter-connection network. Performance of such system is affected by several factors. These factor include processor scheduling policy, inter-connection network characteristics (topology, bandwidth, etc.), and application software architecture. This thesis studies the impact of processor scheduling on the performance of a Transputer system.

Two processor scheduler, namely time-sharing scheduler and static scheduler, have been implemented on a distributed memory transputer system with 16 processors. Three types of applications are used to evaluate the performance of the schedulers. The first application is matrix multiplication that characterizes applications with no communication among the worker processes. The second application is parallel sorting, which represents applications with high communication among the worker processes. The third application is a generic workload model that is used to provide general communication behaviour in order to emulate most other real world applications.

The thesis focuses on the performance of these two processor scheduling policies. Results are also reported that indicate the performance sensitivity of these two policies to transputer system and workload parameters.
ACKNOWLEDGMENTS

First, I would like to thank my thesis co-supervisor, Shikharesh Majumdar, for introducing me to this topic and his incisive comments on an earlier draft of this thesis. I would also like to thank my co-supervisor, Sivarama Dandamudi, for his advice and technical guidance. Special thanks go to them for their encouragement and support.

I also wish to thank Tony Vanon for his comments and feedback on this thesis.

I would like to thank my parents for their loves and supports. Without their encouragement and supports, I would not be able to complete this research.

Finally, I want to thank Oi Yi Chum for her love, patience and support.
# Table of Contents

<table>
<thead>
<tr>
<th>Chapter/Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptance Sheet</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>xii</td>
</tr>
<tr>
<td>Chapter I Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Parallel Processing System</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Job Allocation</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Processor Scheduling</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Goal and Motivation</td>
<td>7</td>
</tr>
<tr>
<td>1.5 Objective</td>
<td>8</td>
</tr>
<tr>
<td>1.6 Overview of the Thesis</td>
<td>9</td>
</tr>
<tr>
<td>Chapter II Processor Scheduler</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Space Sharing</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1 Static Partitioning</td>
<td>12</td>
</tr>
<tr>
<td>2.1.2 Semi-Static Partitioning</td>
<td>13</td>
</tr>
<tr>
<td>2.1.2.1 ASP</td>
<td>14</td>
</tr>
</tbody>
</table>
3.2.1.2 Partition Schedulers 39
3.2.1.3 Local Schedulers 40
3.2.1.4 Coordinator and Worker Processes 40
3.2.2 Inter-Process Communication 42
3.2.2.1 Communication Switches 43
3.2.2.2 Transmitters 44
3.2.2.3 Receivers 45
3.2.3 Memory Management 45
3.2.4 Preemption Control 47
3.2.5 Collection of Statistics 50

CHAPTER IV EXPERIMENTS AND RESULTS 52
4.1 Matrix Multiplication 53
4.1.1 Description of Software 53
4.1.2 Description of Experiments 55
4.1.3 Results 57
4.2 Parallel Sorting 63
4.2.1 Description of Software 64
4.2.2 Description of Experiments 66
4.2.3 Results 67
4.3 Generic Workload 71
4.3.1 Description of Software 72
4.3.2 Description of Experiments 75
4.3.3 Result 76

4.4 Discussion 88

4.4.1 Effect of Memory Allocation, Context Switching and Synchronization 89

4.4.2 Effect of Communication Factor on Applications 90

4.4.3 Variable vs. Fixed Architecture 91

4.4.4 Effect of Network Topology 92

4.4.5 Effect of Task Variance 93

4.4.6 Effect of Scheduling Policy 93

CHAPTER V CONCLUSIONS 95

5.1 Summary 95

5.2 Contributions of the Thesis 96

5.3 Future Research Directions 97

REFERENCES 99
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Architecture of UMA</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>Architecture of NORMA</td>
<td>3</td>
</tr>
<tr>
<td>1.3</td>
<td>Architecture of NUMA</td>
<td>4</td>
</tr>
<tr>
<td>1.4</td>
<td>Centralized Ready Queue</td>
<td>5</td>
</tr>
<tr>
<td>1.5</td>
<td>Distributed Ready Queues</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Typical speedup curve</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>Layout of transputer in the system</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>INMOS C004 transputer switch</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Layout of Linear Topology</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>Layout of Ring Topology</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Layout of Mesh Topology</td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Layout of Hypercube Topology</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>Structure of Schedulers</td>
<td>37</td>
</tr>
<tr>
<td>3.8</td>
<td>Structure of Communication Network</td>
<td>42</td>
</tr>
<tr>
<td>3.9</td>
<td>Transputer's Communication Links</td>
<td>45</td>
</tr>
<tr>
<td>3.10</td>
<td>Pseudo-code of preempt() function</td>
<td>49</td>
</tr>
<tr>
<td>3.11</td>
<td>Time Sharing Support</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>Structure of matrix multiplication</td>
<td>53</td>
</tr>
</tbody>
</table>
Figure 4.2: Pseudo-code of coordinator process for matrix multiplication

Figure 4.3: Pseudo-code of worker process for matrix multiplication

Figure 4.4: Speedup of matrix multiplication (Fixed Software Architecture)

Figure 4.5: Speedup of matrix multiplication (Variable Software Architecture)

Figure 4.6: Response time of matrix multiplication (Fixed Software Architecture)

Figure 4.7: Response time of matrix multiplication (Variable Software Architecture)

Figure 4.8: Structure of parallel sorting

Figure 4.9: Pseudo-code of coordinator process for parallel sorting

Figure 4.10: Pseudo-code of worker process for parallel sorting

Figure 4.11: Speedup of parallel sorting (Fixed Software Architecture)

Figure 4.12: Speedup of parallel sorting (Variable Software Architecture)

Figure 4.13: Response time of parallel sorting (Variable Software Architecture)

Figure 4.14: Response time of parallel sorting (Fixed Software Architecture)

Figure 4.15: Structure of generic workload

Figure 4.16: Pseudo-code of coordinator process for generic workload

Figure 4.17: Pseudo-code of worker process for generic workload

Figure 4.18: Generic workload mimics matrix multiplication (Fixed Software Architecture)

Figure 4.19: Generic workload mimics matrix multiplication (Variable Software Architecture)

Figure 4.20: Generic workload mimics parallel sorting (Fixed Software Architecture)

Figure 4.21: Generic workload mimic parallel sorting (Variable Software Architecture)
Figure 4.22: Generic workload for CF=0 (Fixed Software Architecture) 80
Figure 4.23: Generic workload for CF=0 (Variable Software Architecture) 81
Figure 4.24: Generic workload for CF=1 (Fixed Software Architecture) 83
Figure 4.25: Generic workload for CF=1 configured as linear array (Fixed Software Architecture) 84
Figure 4.26: Generic workload for CF=1 (Variable Software Architecture) 85
Figure 4.27: Generic workload for CF=∞ (Fixed Software Architecture) 86
Figure 4.28: Generic workload for CF=∞ configured as linear array (Fixed Software Architecture) 86
Figure 4.29: Generic workload for CF=∞ (Variable Software Architecture) 87
Figure 4.30: Generic workload with high process variance 88
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A+</td>
<td>Allocation policy based on AVG and system workload</td>
</tr>
<tr>
<td>A+&amp;mM</td>
<td>Allocation policy based on AVG, minimum and maximum parallelism and system workload</td>
</tr>
<tr>
<td>Adaptive-AVG</td>
<td>Adaptive average parallelism allocation policy</td>
</tr>
<tr>
<td>ASP</td>
<td>Adaptive Static Partitioning</td>
</tr>
<tr>
<td>AVG</td>
<td>Average parallelism allocation policy. Other allocation policies based on average parallelism in making scheduling decision are A+, A+&amp;mM, SDF-AVG and Adaptive-AVG</td>
</tr>
<tr>
<td>CV</td>
<td>Coefficient of Variation</td>
</tr>
<tr>
<td>CF</td>
<td>Communication Factor</td>
</tr>
<tr>
<td>FCFS</td>
<td>First Come First Serve</td>
</tr>
<tr>
<td>FF</td>
<td>First Fit allocation policy. Other deviations of FF are FF+FIFO, FF+LA and FF+FIFO+LA</td>
</tr>
<tr>
<td>LS</td>
<td>Local Scheduler</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>PS</td>
<td>Partition Scheduler</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>PWS</td>
<td>Processor Working Set</td>
</tr>
<tr>
<td>RRjob</td>
<td>Round Robin Job allocation policy</td>
</tr>
<tr>
<td>RRprocess</td>
<td>Round Robin Process allocation policy</td>
</tr>
<tr>
<td>Rx</td>
<td>Receiver</td>
</tr>
<tr>
<td>SA</td>
<td>Smallest average parallelism first allocation policy</td>
</tr>
<tr>
<td>SDF</td>
<td>Smallest Demand First allocation policy</td>
</tr>
<tr>
<td>SDF-AVG</td>
<td>Hybrid of SDF and AVG allocation policies</td>
</tr>
<tr>
<td>SS</td>
<td>Super Scheduler</td>
</tr>
<tr>
<td>SW</td>
<td>Communication Switch</td>
</tr>
<tr>
<td>TS</td>
<td>Time Sharing</td>
</tr>
<tr>
<td>Tx</td>
<td>Transmitter</td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

A parallel processing system is a type of computer system in which multiple processors are inter-connected by a high speed interconnection network. Programs that execute on parallel systems are called parallel programs, which consists of several processes. Processes are run on multiple processors concurrently and communicate with one another to exchange intermediate results. In order to achieve good performance, scheduling policies are used to determine how many processors are assigned to each program, the order of execution of programs and how much time is spent on each execution. The purpose of this research is to identify parameters that are significant determinants of a policy’s performance.

The extensive computing power and large memory resources of the parallel system can be used to solve complex problems which cannot be solved with a single processor computer. Although parallel systems are becoming more and more popular, further research needs to be completed before parallel systems can be used extensively. One of the challenges confronting parallel systems involves the complex nature of the parallel programming. As this process is still immature, it is important to build better compilers and run-time environments so that the development of parallel programs can be easier and the system resources can be more effectively utilized. To make a multiprocessor system more cost effective, the system should be shared with multiple applications. This makes
developing system software that can manage system resources effectively equally important.

One set of the most important resources in a parallel system is the processors. Therefore, finding an efficient way to manage the use of processors is essential. Since the objective of a parallel system is to provide high performance computation and most of the jobs submitted to the system are expected to have high performance, the system scheduler is responsible to ensure each individual application achieves a high level of performance. In a system which shares resources with applications, the scheduler must also maximize the system throughput. However, it is difficult to fulfill both requirements.

1.1 Parallel Processing System

Parallel processing systems can be divided into two types: uniform-access-shared-memory systems (UMA) (also known as multiprocessors or tightly-coupled systems) and distributed-memory systems (also known as multicomputers or loosely-coupled systems). Depending on the type of system, communication can be achieved by passing messages and/or shared memory.

Memory in a shared-memory system can be accessed by all processors in the system (Figure 1.1). Therefore, processors can use the shared memory as a communication medium. Message passing can also be used to establish communications among processors; however, using this medium is much slower than using shared-memory. Two advantages of this type of system are that: 1) the cost of code and data
sharing is low and 2) it allows finer grain parallelism than the distributed-memory system. However, the architecture of a shared-memory system is complex and expensive to implement. Therefore its scalability and extensibility is limited. Some examples of a shared-memory system are the Encore Multimax, the Sequent [Lovett & Thakkar 1988] and the DEC Firefly [Thacker et al. 1988].

![Figure 1.1: Architecture of UMA](image)

Distributed-memory system involve a group of processors, each having its own memory module connected by a high speed network. This type of system is called no remote memory access (NORMA). In a NORMA system, the architecture of the system does not allow for accessing memory from other processors (Figure 1.2). The transputer system used in this thesis is a NORMA system.

![Figure 1.2: Architecture of NORMA](image)

There is a class of systems called non-uniform-access-shared-memory (NUMA). This is a hybrid of shared and distributed memory system. In a NUMA system, each processor can have its own memory and it can also access the memory contents of the
other processors directly; however it is much more expensive than accessing local memory (Figure 1.3). Some of the NUMA systems are BBN Butterfly system, Standford's Dash [Lenoski et al. 1989] and Toronto's Hector [Vranesic et al. 1991].

![Figure 1.3: Architecture of NUMA](image)

The only known way of providing scaleable access to distributed memory is by exchanging messages. As a result, the speed of communication depends on the throughput of the communication links and the amount of data transmitted. The extensibility of this type of system is much higher than the shared-memory systems. For this reason many national laboratories and supercomputer centers are using distributed-memory systems such as the CM5 and Intel Paragon. One disadvantage of multicomputers is that communications among processors are slower than multiprocessors.

### 1.2 Job Allocation

A job in a multiproccessing system is divided into processes (grains or tasks) and are executed concurrently in processors. Grain size describes the average execution time of a process. It is one of the characteristics that can affect the performance of a parallel
system. At one extreme, each process may consist of a single instruction. For efficient utilization of processors in the system, the grain size should be as fine as possible, such that processes can achieve maximum parallelism. However, this kind of parallelism may increase overhead; for example, increasing context switching and synchronization between processes.

In a typical parallel system, a job with processes is submitted to the system and is inserted into the system's ready queue where it waits to be scheduled. There are two ways to organize the system's ready queue: the centralized approach and the distributed approach. A centralized system queue (Figure 1.4) has a single global ready queue, and all processes are inserted in that queue [Majumdar et al. 1988].

All processors are able to access the global queue, and a processor can retrieve a ready process when the processor is idle. This approach will provide higher performance because it can provide a better load-balanced environment. However, this approach is only suitable for systems with a small number of processors.

![Diagram](image)

**Figure 1.4: Centralized Ready Queue**
By using the distributed approach [Zahorjan & MacCann 1990], a local ready queue is allocated to each processor (Figure 1.5). When a job is submitted to the system, the system will distribute the job to some processors. The processes are then inserted into the local ready queue. Then an idle processor will retrieve a process from its own local ready queue. Although this approach is less efficient than the centralized approach, it can be applied to systems with large numbers of processors. Therefore the distributed approach is used here to ensure the generality of the research.

![Diagram of Distributed Ready Queues](image)

**Figure 1.5: Distributed Ready Queues**

1.3 Processor Scheduling

While the processes are waiting in the ready queue, a processor scheduling policy is used to determine the distribution of processes to a set of processors and the execution sequence of each process on a processor.

A processor scheduling policy is a scheme that decides how jobs/processes are allocated on a multiprogram parallel system. Scheduling policies for a multiprocessor system can be categorized into two general classes: space sharing and time sharing. Space
sharing policy is also known as partitioning policy, which divides the system processors into disjoint sets. Jobs submitted to the system are routed to one of these partitions by the system scheduler and executed on the processors in that partition until they are terminated. Partitioning policies are further classified as static, semi-static and dynamic policies.

Static partitioning is a non-preemptive scheduling policy. Each job runs to completion without interruption on the set of processors initially allocated to it and the partition size is fixed. Under semi-static partitioning, the number of processors allocated to a job is not fixed and is determined by the system workload and the resource requirement of the job. Once a job is allocated with a number of processors, the partition is fixed until the job is terminated. Dynamic partitioning will adjust the size of the partition allocated to a job during its execution.

Time sharing allows more than one job to be executed on a set of processors. Each job takes alternate turns and is executed for a time quantum, yielding the processors to another job at the end of that quantum. The cycle is repeated until all the jobs are completed.

1.4 Goal and Motivation

It is important to develop an efficient processor scheduling policy to obtain high performance-to-cost ratio. However, there is no known general processor scheduling policy that is best fit for either a shared memory system or a distributed memory system. Since it is a NP-complete problem, instead of finding a general solution to the problem,
researchers have conducted studies [Eager et al. 1989, Sevcik 1989, Gupta et al. 1991, Majumdar et al. 1991, Vaswani & Zahorjan 1991, Peris et al. 1994] to find characteristics to describe a parallel system such that a specific policy can be applied to systems with certain attributes. Much research has been conducted on shared memory systems recently. However, analysis of distributed memory system using message passing is still a relatively unknown area, compared to shared memory systems. Therefore, our goal is to find some characteristics to help describe a distributed memory system.

1.5 Objective

From the literature, it is known that the effectiveness of a scheduling policy depends on the characteristics of both the system and the jobs submitted. Some of the known characteristics are: the job's average parallelism [Eager et al. 1989], the job's parallelism profile [Sevcik 1989], the variance of job demand, workload, the grain size and speedup and efficiency of a job. Our objective is to compare the performance of different approaches to multiprogramming, with the aim of investigating how the topology of an inter-connection network of processors and the communication pattern of job affects the processor scheduling policies on distributed-memory system. In addition, this thesis will concentrate on static partitioning and time sharing policies. We have performed a set of experiments to evaluate the effectiveness of the inter-connection network of processors and the communication pattern of job to scheduling policies.

Two sets of experiments were performed on a 16 processor transputer system which has a distributed memory architecture. One set of experiments uses space
partitioning policies; the other set uses time sharing policies. We have used a matrix 
multiplication application, a sorting application and a generic model in our experiments to 
demonstrate different communication patterns of a job. Various inter-connection 
networks were applied to each model with different partition sizes, numbers of partitions, 
and workloads to measure the effect of these characteristics to scheduling policies.

1.6 Overview of the Thesis

A more detailed study of the problem will be given in the following chapters.

In chapter two we will survey some of the previous studies on processor 
scheduling policies for multiprocessor parallel computers. The survey includes both space 
sharing and time sharing policies. For space sharing scheduling policy, three categories 
will be discussed, which are static partitioning, semi-static partitioning and dynamic 
partitioning.

In chapter three we present the architecture of the transputer system environment 
used to perform the experiments in this thesis. Chapter three also describes the underlying 
software architecture of the system and explains how the inter-process communication can 
be done. In addition, the memory management and the preemption mechanism used to 
implement the time sharing policy will be discussed in the chapter.

Chapter four analyses the results obtained by the experiments. Furthermore, the 
impact of static and time sharing scheduling policies under different inter-connection
networks is described. This section also analyzes the effect of software architecture, workload and partition size on the scheduling policies.

Chapter five summarizes the results obtained in this thesis. It also provides pointers to further research.
CHAPTER II

PROCESSOR SCHEDULING

This chapter presents a survey of related work on processor scheduling in a parallel processing system. As stated in the introduction, scheduling policies for multicomputers can be classified into two general classes: space sharing and time sharing.

2.1 Space Sharing

An important element with regards to spacing sharing is that the speedup of a multiprogram is not linear, that is doubling the number of processors allocated to a job will not double the performance of the job. The speedup curve of a typical parallel program is illustrated in Figure 2.1.

![Typical Speedup Curve](image)

Figure 2.1: Typical speedup curve
The diminishing return in reduced execution time is mainly caused by algorithmic overhead, parallelization overhead, synchronization overhead and communication overhead [Fox et al. 1988]. Allowing more applications, running on partitions with small sizes, will better utilize system resources, which could improve the overall performance of a system. There are three principal categories of space sparing policies: static partitioning, semi-static partitioning and dynamic partitioning.

2.1.1 Static Partitioning

The partition size of static partitioning policies is fixed at the time the system is started. The allocation of jobs to partitions are either specified by the user or scheduled by the system scheduler. These allocations are based on the characteristics of job, such as priority and requirement of resources and whether the allocated job in a partition will run to completion.

Normally, in a system that uses the static partitioning scheduling policy, a job is inserted into the system ready queue at its arrival and is allocated to a partition when there is suitable partition available for the job. Then the job is executed until it is terminated. The completed job is removed from the partition and the free partition is allocated to the next job.

The advantage of this kind of policy is that implementation is simple. If the workload of jobs in a system is stable and predictable, the system administrator can choose a configuration which best suits the behavior of the workload of the system to provide
reasonably good performance. However, the disadvantages are that this kind of scheduling policy does not adapt to changes in the system load condition and resources requirement of an individual job. An example would be a system with 100 processors and partitioned into partitions with 50 processors each. A job will occupy one partition even if the job requires only 10 processors. The other 40 processors would be left idle.

A very simple example of this kind of policy involves processors that are divided into equal partitions. The job allocations are based on first-come-first-serve basis (FCFS). That is, a job that arrives earliest to the system is inserted at the beginning of the queue and is the first be allocated to a free partition. There are also policies based on priority and demand in making a job allocation decision.

2.1.2 Semi-Static Partitioning

The partition size of semi-static partitioning can be changed at the time of a job allocation. Similar to static partitioning policy, once a job is allocated to a partition, the partition size is fixed until the job is finished, and at that time, the resources of the partition are reallocated. Semi-static scheduling is motivated by resolving limitations of static scheduling. A static scheduler cannot react to the global changes in the parallel system or changes to the behavior of parallel programs executing in the system. On the other hand, semi-static partitioning policies schedule processors based on the programs' behavior and are adaptive to system changes. There have been a lot of policies studies that are based on semi-static partitioning. Here are a few of them:
2.1.2.1 ASP

Adaptive Static Partitioning (ASP) was first defined in [Setia & Tripathi 1991]. The policy allocates the lesser of the number of idle processors in the system and the parallelism of a job (P). Processors are allocated to jobs in queue, one at a time, in round robin fashion until all idle processors are allocated. For example an idle processor is allocated to the job in the head of the queue and then another processor is allocated to the second job at the queue, and so on. After all the jobs in the queue are allocated with one processor each, the cycle repeats again at the head of the queue, allocating a second processor to each job. A job is removed from the queue and starts executing when it gets a number of processors that is equal to its maximum parallelism. The process is repeated until all the idle processors are allocated or all the jobs have been allocated with enough processors to achieve their maximum parallelism.

2.1.2.2 PWS

Processor Working Set (PWS) or the knee of the execution time efficiency profile involves the number of processors that provide maximum marginal benefit in allocating one more processor to a job. Majumdar et al. [1991] have shown that the knee of the execution time efficiency profile is significant in determining the number of processes to be allocated to an application. Execution time efficiency profile is a graph each point in which represents the system performance obtained with a specific number of processors allocated the job. A similar approach is taken by Ghosal et al. [1991] who defined
FF+FIFO and the policy is renamed to PWS in [Setia & Tripathi 1993]. Ghosal et al. proposed several semi-static scheduling policies that rely on the PWS of parallel programs.

The first policy described is the First Fit (FF) allocation policy. The FF scheduler searches for the first job in queue which has a PWS less than the number of available processors. Then the PWS number of processors are allocated to the job and the process is repeated until there is no job in the process ready queue or each job's PWS is larger than the number available processors. Since exactly PWS number of processors be allocated to each job, unallocated processors may remain idle while there are jobs waiting in the queue.

Another scheduling policy presented uses the FF+FIFO allocation strategy. Similar to FF policy, the scheduler searches for the first job with PWS that can fit in the available processors and PWS would be allocated to that job. The allocation continues until no more jobs can be assigned to idle processors. The remaining processors, if any, would be assigned to the first job in the queue. This policy can utilize all processors and thus can provide better performance than FF policy.

The third policy proposed is the FF+LA policy. This is also a work conserving policy, similar to FF+FIFO. It allocates processors in the same way as in FF policy. Instead of letting the remaining processors remain idle, processors are assigned to the last job allocated if there is no more job waits behind in the queue. Hence, it makes the partition size of the last job exceed PWS of the last job. This policy allocates processors based on the fact that allocating more processors to a job may decrease the execution time.
of that job. However, when the system load is high, the decrease in execution time may not be enough to compensate for the increase in waiting time if a job arrives later.

Another policy combining FF+FIFO and FF+LA is called FF+FIFO+LA. It also allocates processors in the way FF does. If there are jobs in the queue, remaining processors are allocated to the first job in the queue. Otherwise, remaining processors are assigned to the last allocated job. When the system load is low, the LA part of the policy can increase the performance and the FIFO part of the policy can increase utilization when the load is high.

The literature surveyed above did not consider I/O in application. The work in [Majumdar et al. 1994] demonstrates an intimate relationship between scheduling and disk I/O performed by application.

2.1.2.3 Average Parallelism (AVG)

Average Parallelism (\(A\)) is defined as the average number of busy processors during an execution of the application, when an unlimited number of processors are available [Sevcik 89]. It is known to be related to the speedup of an application, such that

\[
\frac{NA}{N + A - 1} \leq S(N) \leq \min(N, A)
\]

where the speedup, \(S(N)\), is defined as the ratio of execution time of a job executing on one processor to the execution time of the same job running on \(N\) number of processors. Average parallelism of an application \(j\), \(A_j\), can be obtained from a parallelism profile, which is determined by an experiment, such that an application is considered to be executed on an unlimited number of processors and it can be used in
helping to make decisions on processor scheduling. In this scheduling policy, a scheduler allocates $A_j$ numbers of processors to job $j$ on FCFS basis. A policy using average parallelism can perform well when the amount of variation in parallelism within the application is small. There is also a deviation of this policy called SA. SA scheduling policy allocates jobs with the smallest average parallelism first. Each job is assigned with a partition, with partition size equal to its average parallelism. This policy allows more jobs to be served simultaneously. In a system which has jobs' mean service demand is linearly dependent on the number of processes, SA can provide higher performance.

2.1.2.4 A+ and A+&mM

Two more scheduling policies based on the average parallelism are proposed in [Sevcik 1989]. The last policy considers only one characteristic, $A$, in making a scheduling decision. If one more parameter, system load ($\rho$), is considered in the policy, the number of processors allocated to a job $j$, denoted by $\alpha_j$, is assigned to be $A_j + d_j \times f_j$, where $A_j$ is the average parallelism of an application $j$:

$$d_j \text{ is the deviation of processor allocation to } A_j; \text{ and}$$

$$f_j \text{ is the fraction of the maximum possible variation of an application.}$$

The deviation of processor allocation to $A_j$ is defined as
\[ d_j = \begin{cases} \ y - \rho \times \left[ \frac{y - A_j}{\pi} \right] - A_j & \text{if } \rho < \pi \\ x + (1 - \rho) \times \left[ \frac{A_j - x}{1 - \pi} \right] - A_j & \text{if } \rho \geq \pi \end{cases} \]

where \( x \) is the minimum parallelism;

\( y \) is the maximum parallelism;

\( \pi \) is the workload of the system which the optimal allocation is \( A_i \) and \( \pi \) is chosen some what arbitrarily to be 0.25 in [Sevcik 1989].

The fraction of the maximum possible variation of an application is defined as

\[ \frac{V_j}{(y - A_j)(A_j - x)} \]

where \( V_j \) is the variation of parallelism.

In a system which has high variability in execution time, the total number of processors, \( P \), would be allocated at a very light load and 1 processor would be allocated at a very heavy load. Hence \( \alpha_j = A_j + \frac{V_j}{(P - A_j)(A_j - 1)} [h_j(1, P) - A_j] \) number of processors are allocated to each job in the queue when there are idle processors. This policy is denoted by A+ policy.

When the minimum and maximum parallelism, \( m \) and \( M \) respectively, are known,

\[ \alpha_j = A_j + \frac{V_j}{(M_j - A_j)(A_j - m_j)} [h_j(m_j, M_j) - A_j] \]

would be the number of processors to be allocated to job \( j \).
A+&mM, A+ and AVG scheduling policies are not work conserving; therefore they may not fully utilize all the processors in the system.

2.1.2.5 Adaptive-AVG

[Chiang et al. 1994] reported an adaptive-AVG policy which also uses average parallelism to make scheduling decisions. Unlike the AVG, A+ and A+&mM scheduling policies, adaptive-AVG is a work conserving policy. Adaptive-AVG allocates the minimum of its A and the available processors in the system upon a job arrival. Jobs that cannot be allocated processors wait in the queue. When a job is completed, the released processors are allocated to waiting jobs in the queue. The allocation is based on the average parallelism of the jobs. Jobs are allocated with a number of processors in proportion to their average parallelism, such that each job is allocated with at least one, but no more than average parallelism.

Adaptive-AVG performs better than AVG policy because it considers the length of the process ready queue and the application execution rate characteristics. Adaptive-AVG is less sensitive to the distribution of parallelism and performs best when jobs have a constant parallelism.

2.1.2.6 SDF & SDF-AVG

In Smallest Demand First (SDF) scheduling policy, the demand of a job is used when deciding processor allocation. Demand of a job is defined to be the total processing
requirements of a job on a single processor. In a parallel processing system, demand of a job, \( D \), can be interpreted as the execution time requirement of the component processes in a job. It is obtained by adding the execution time of each individual component process. It is different from the total processor time of a job and the demand of a job does not include the system overhead, such as process synchronization and scheduling costs.

In this policy, a job with the smallest demand in the queue is allocated with the minimum of its parallelism and the number of idle processors. Although the policy is work conserving, the performance is not as good as the policies using average parallelism. Moreover, the demand of a job cannot be estimated easily without actually executing the code.

[Chiang et al. 1994] also defined a better alternative to SDF, called SDF-AVG. Instead of allocating the parallelism number of processors to a job, the average parallelism of a job is allocated when there are enough processors available. This approach improves performance by taking the advantages of both demand and average parallelism in making allocation decision.

2.1.2.7 Equipartition

Equipartition [McCann & Zahorjan 1994] is a policy which divides processors into partitions of equal size. The processors are divided into two sections, the regular section and the remainder section. Again, a regular section is divided into \( 2^X \) rows. Each contains \( Y \) partitions and the remainder section contains \( J - 2^X Y \) partitions, where \( X \) is
chosen to be $\left\lceil \frac{\log J}{2} \right\rceil$ and such that the regular section is made as square as possible.

When considering a system with a mesh having $2^{M+N}$ processors and a system with $J$ jobs, the regular section is assigned with $2^M \times \chi$ rows of processors and columns of processors is assigned to the remainder section. Then the remaining $2^N - \chi$ columns of processors are allocated among the partitions of the regular section as evenly as possible.

The $2^{M+N}$ processors are contracted onto a $R \times C$ grid and a process is allocated to processor $(i,j)$ by a mapping function. The theoretical possible maximum number of processes assigned to a single processor under any process mapping function is $\left\lceil \frac{2^M \times 2^N}{R \times C} \right\rceil$.

The idea of Equipartition Policy is to reduce the maximum loading of processes to a processor from $\left\lceil \frac{2^M}{R} \times \frac{2^N}{C} \right\rceil$, which is the maximum number of processes assigned to a processor of the mapping function $map(i,j)$.

2.1.3 Dynamic Partitioning

Under both static and semi-static partitioning, a partition allocated to a job is fixed and the job executes on that partition until it is completed. In contrast, dynamic partitioning can alter the size of a partition according to the current system conditions during a job is executing on that partition. Therefore, dynamic scheduling policy can adapt to changes in the system workload better than static and semi-static scheduling policies.
The tradeoff of better adaptation to system changes is the cost of reconfiguration of partitions and data migration between processors or address spaces, when a process is preempted and allocated to a different location. Both reconfiguration and migration are expensive operations in the current generation of multiprocessing systems. Moreover, the implementation of dynamic scheduling policy is much more difficult than the static and semi-static policies.

2.1.3.1 Dynamic

The procedures of Dynamic scheduling policy [McCann et al. 1990] can be separated into two levels. The first level handles the allocation of new job arrivals. The second level maintains the number of processors held by executing jobs.

Level I:

When a job arrives into the system, if there are idle processors, the job is executed using the idle processors. Otherwise, a processor is taken away from a job holding two or more processors. If either case cannot satisfy the request of a newly arrived job, the job will wait in the queue until a processor becomes available. Upon release of one or more processors from other processes, one processor is assigned to each job in the queue. If there are still idle processors available, processors are allocated to jobs in queue in a FCFS basis.
Upon arrival each job with request $q$ processors and the number of processes waiting for a processor in job $J$ is denoted by $q_j$. When a process in a job $J$ is finished, if the job holds two or more processors and there is no idle processor available to the system, the job will release one processor to a newly arrived job which has not been assigned with any processor. If there is no unallocated job waiting in the queue and $q_j = I$, the process will be dequeued and start executing. On the other hand, if $q_j > I$, a process from job $J$ is dequeued and executed, then job $J$ will announce that $q_j = q_j - I$ number of processors are requested. If $q_j = 0$, and there are jobs waiting for processors, the processor held by the finishing process will be released. Otherwise, the processor will wait to be allocated until the job $J$ requests for an additional processor or another job makes a request on acquiring processors. Preference is given to the processes in the same job when processes from another job request the processor, to reduce the context switching overhead.

This policy gives highest priority to a newly arriving job, such that the policy ensures that a new arrival receives a processor to start executing promptly, and it is shown to be important in affecting performance in [Majumdar et al. 1988].
2.1.3.2 Folding

Folding is a scheduling policy proposed by [McCann & Zahorjan 1994]. It assigns all the processors to a newly arrived job if the system is idle. Afterwards, half of the processors in the largest partition in the system are assigned to a newly arriving job. To ensure fairness of processor usage among jobs, the rotation of partition is applied to jobs. Rotation is proceeded by assigning a letter ‘b’ to a larger partition and a letter ‘s’ to a smaller partition. All the partitions will form a string with combinations of ‘s’ and ‘b’. Each rotation is to replace each occurrence of ‘sb’ in the string with ‘bs’. Upon a job departure, processors in the partition allocated to the departing job cannot be freed by only local operations. One way to handle job departure is to re-compute the partitions and to perform a global assignment of jobs to those partitions. Another way is to mark the freed partition as available and perform the rotation as fast as possible, until the idle partition has been allocated.

The motivation for this policy is to maintain high efficiency preservation. Efficiency presentation of a system with \( P \) processors and \( J \) copies of application \( app \) running under policy \( policy \) is defined as

\[
EP_{policy, app}(J, P) = \frac{\sum_{j=1}^{J} AE_{policy, app}(A_j)}{AE_{uniprogramming, app}(P)}
\]
where $A_j$ is the number of processors allocated to $j^{th}$ copy of application; and

$AE_{policy, app}(p)$ is the ratio of the total computation time of the application when running on $p$ processors.

Policies using efficiency preservation only do not guarantee high performance, especially in a system with a high arrival rate. Efficiency preservation may also provide misleading information, such as a FCFS policy allocating all processors to each gives $EP = 1$. However, $EP$ can be useful in comparing alternative processor allocation policies.

### 2.2 Time Sharing

In contrast to space sharing policies, time sharing scheduling policies allow more than one job to be executed in a partition. A time sharing policy can be separated into two parts: system scheduling and node scheduling. There are three decisions to be made by system scheduling: partition size, degree of multiprogramming and specific partition allocated to a program. Node scheduling decides how the processes share the resources. Processes share the same processor by dividing CPU time into time-slices and each process is executed during its time-slices. The duration of a time-slice can also affect the efficiency of time sharing policy. Making a time-slice too short will increase the context switch overhead; making a time slice too long will increase the synchronization cost between processes, as the time required for a process to resume execution is longer. Typically, a process in a parallel program gets data from other processes and performs some computation, then sends the resulting data to some other processes. Ideally, the
length of a time-slice should be long enough to finish the computation part of a process and begin the communication part at the end of the time-slice. There are two classes of preemptive policies: coordinated and uncoordinated. Coordinated preemptive policy, also called coscheduling policy [Crovella 1991], restricts all the processes in the same job are preempted from execution at the same time. Unlike coordinated preemption, uncoordinated preemption allows preemption of a process from one job while another process from the same job is executing in another processor. The following sections provide some examples of coordinated and uncoordinated preemption policies.

2.2.1 Gang Scheduling

Gang scheduling is one of the coordinated preemption policies proposed by [Ousterhout 1982]. In a system with communication primitives for receiving messages that are blocking in nature, a receiving process must be synchronized with a sending process before they can be communicated. Although time sharing allows a blocked process to be preempted from execution and replaced with another process, it cannot fully utilize a processor. Granularity of synchronization [Feitelson & Rudolph 1992, Leutenegger & Vernon 1990] refers to the number of instructions executed by processes between communications. In a system with processes which granularity of synchronization is small, gang scheduling can provide better performance. Gang scheduling can reduce the communication and synchronization overhead by ensuring that all processes of a job are scheduled to be executed at the same time. This condition is achieved by context switching processes in the same job synchronously. However, this
policy will perform poorly when the granularity of synchronization is low to moderate. Context switching is done synchronously causing processors not to be fully utilized. Therefore processes cannot be equally distributed on the processors, such that processes will spend a lot of time waiting for messages sent from some other processes.

2.2.2 RRprocess and RRjob

Round robin process (RRprocess) and round robin job (RRjob) [Leutenegger & Vernon 1990] are two uncoordinated preemption policies. The RRprocess places all processes of a newly arrived job into a global ready queue. Processes in the queue are executed in round robin fashion. Round robin scheduling means that each process is allowed to be executed for a fixed time quantum. After a process has finished its quanta, it will be preempted and put into the end of the ready queue and next process in the ready queue is allowed to be executed. This process is repeated until all the processes in the ready queue are finished.

Similar to RRprocess, scheduling is done round robin on the jobs. Instead of placing all processes in each job into a global ready queue, a job entry is appended to the queue and each job entry has its own process ready queue. A round robin scheduling policy is invoked on the process queue. Each job gets $P$ quanta of size $q$ where $P$ is the number of processors in the system. If a job has $N$ processes and $N < P$, each process is given a quantum of size $P/N \times q$. If $N > P$, there are two choices. The first choice is that only the first $P$ processes are executed and each process receives one quanta of size $q$. Another choice is that each process is given with a quantum of size $P/N \times q$. 

27
2.3 Summary

Two general classes of scheduling policies, space sharing and time sharing, were reviewed. Space sharing may be further classified as static, semi-static and dynamic partitioning and time sharing may be divided into two classes: coordinated and uncoordinated preemption policy. Several scheduling policies have been reviewed in each category. FCFS is the simplest static scheduling policy and the scheduling scheme is fixed at the beginning of system. In order to achieve higher system performance, semi-static scheduling policies can be used to adapt changes in workload to a system. Semi-static scheduling policy changes it's scheduling scheme according to the system workload. Once processors are assigned to a job, the job runs to completion and the assignment cannot be changed. Several semi-static scheduling policies are discussed. Dynamic scheduling is the most flexible scheduling policy and it changes scheduling according to the current system workload. Dynamic and folding are two examples of this kind of scheduling policy. Time sharing policies reviewed in this chapter are gang scheduling, round robin job and round robin process scheduling policies. Gang scheduling is a coordinated preemption policy. Under this policy, all processes in a job are preempted at the same time. Opposite to gang scheduling, round robin job and round robin process are uncoordinated preemption policies which allows some processes in a job are executing while some other processes in the same job are waiting in the ready queue.
CHAPTER III

THE TRANSPUTER ENVIRONMENT

In this chapter, we present an overview of the transputer environment used in performing the experiments. First, we describe the hardware architecture of the transputer system. A description of the software architecture will then follow.

3.1 Hardware Architecture

The hardware used in this thesis is a TransTech PARAStation transputer system [CSA 1990a]. The system contains sixteen T805 transputers and is expandable by connecting to other PARAStations. Each transputer is equipped with 4MB of local memory and the memory is not directly accessible from remote processors. Each processor has 4 links and is connected in the structure as shown in Figure 3.1.

The sixteen processors are hardwired in four pipelines of four processors each of which is called a nap. Processors of a nap are connected to the processors of its adjacent naps. Each processor can connect to any other processors in the same nap by using an INMOS C004 programmable switch [INMOS 1990] which is not shown in Figure 3.1. The thickened lines shown in Figure 3.1 are hardwired. Two ends of a nap are called the main up and main down respectively. Because the transputer does not have the circuitry to support I/O, any I/O must be redirected to one of the processors which connects to a host and the host will be used to manipulate the I/O. The main up connector is used to
attach to the host and the main down connector is used to connect to one or more PARASTations in a daisy chain configuration to provide high extensibility.

![Diagram](image)

Figure 3.1: Layout of transputer in the system

3.1.1 System Support

The system provides two important classes of services: management of process and inter-processor communication. Each of these is briefly described in the following sections.
3.1.1.1 Multiprogramming Support

The T805 transputer has hardware support for multiprogramming. Each transputer has a built-in runtime scheduler for processes running concurrently on the same transputer. Processes waiting for input or output, or waiting on a timer, do not consume any CPU resources and the context switching time can be as little as one microsecond. Two levels of priority are supported by transputer: high priority and low priority [CSA 1990b]. There are two ready queues in each transputer, such that processes with high priority are inserted into the high priority ready queue, and the low priority processes are inserted into the low priority ready queue. High priority processes run to completion or until blocked for I/O, IPC or timer. Low priority processes are time shared and each process has a fixed quantum of 2ms. The transputer scheduler always fetches processes from the high priority queue first, until all the processes in the high priority queue are terminated or blocked. Then processes in low priority queue are fetched and time shared. A low priority process must yield control to a high priority process whenever a process in the high priority queue is ready and its unfinished time slice is lost. The transputer contains high and low priority clocks which can be used to implement the delayed execution of processes. The high priority clock has high resolution which generates one clock tick per 1 ms and is accessible to high priority processes only. The low priority clock can be accessed by low priority processes and it has lower a resolution of one clock tick per 64 ms [CSA 1990c].
3.1.1.2 Communications Support

The communications between processors are provided by 4 communication links. Each communication link is handled by 2 DMA controllers, one for input and one for output, and it is capable of transmitting or receiving at 20Mbit/s. The DMA controllers can be configured to transfer data in, out or both simultaneously and independently without the intervention of the CPU. Therefore processes can continue execution, as messages are transferred to and from adjacent processors. The message passing scheme of transputer is known as a point-to-point scheme [CSA 1990b] and has the advantage of not needing arbitration; however the disadvantage of potentially limited connectivity can be problematic. Although a point-to-point communication scheme can increase the communication bandwidth as the number of transputer are added to the system, the diameter of the network (the number of intermediate processors that a message must go through) increases as well.

Since the communications between processors do not support shared memory, the communication media used to transfer data between processors is message passing. The transputer model defines inter-process communication in term of channels. A channel is an unidirectional communication path between two processes and the communication in a channel is synchronous and is directly supported by the transputer instruction set. Hence, a process that transmits a message to another process, whether it is in the same processor or not, will be blocked until the other process receives the message. A maximum of 2 channels can be placed on each link, one for receiving and one for transmitting.
3.1.2 Possible Topologies

Several topologies can be configured from the TransTech PARAStation. The configuration is set up by connecting or disconnecting various links between a pair of naps or between processors in the same nap. The hardwired links between processors are used to form naps and these are high speed links with a throughput of 20Mbit/s [CSA 1990b]. They are shown as thickened lines in Figure 3.1. The other links can be connected by using INMOS C004 transputer switches. Each switch is responsible for connecting processors in the same nap or in its adjacent nap as shown in Figure 3.2, and there are 4 switches associated with the 4 naps respectively. A link from a processor connected to the transputer switch can be rerouted to any other processors connected to the switch. The rerouted links are slower than the hardwired links, which provide a throughput of 10Mbit/s.

![Diagram](image)

Figure 3.2: INMOS C004 transputer switch
(The thick lines are hardwired lines and the thin lines are re-configurable lines)
By using transputer switches, almost all commonly used topologies can be configured in the transputer system. The following is a list of topologies that can be formed in our 16 processors transputer system.

1. Linear

Processors are configured as a pipeline (Figure 3.3). This is the simplest topology that can be made. The diameter of the network increases by one as the number of processors \( p \) in the network increases by one.

![Figure 3.3: Layout of Linear Topology](image)

2. Ring

Ring topology is similar to linear topology. Instead of leaving two ends of the network disconnected, ring topology connects the head and the tail of a pipeline of processors (Figure 3.4). Ring topology can improve the rate of increment of the diameter to half of that of the linear topology.
3. Mesh

The layout of a mesh is shown in Figure 3.5. It is best suited to problems that have an inherent two-dimensional nature, such as graph theory or vision. The diameter of a 2-D mesh is $2(\sqrt{P} - 1)$ where $P$ is the number of processors in the network.

Figure 3.4: Layout of Ring Topology

Figure 3.5: Layout of Mesh Topology
4. Hypercube

Hypercube is an n-dimensional cube and its layout is shown in Figure 3.6. An n-dimensional hypercube directly connects each of $2^n$ processors to n other processors. A hypercube can be defined recursively as follows. A zero-dimensional hypercube is a single processor and a one-dimensional hypercube connects two zero-dimensional hypercubes. Generally, a hypercube of dimension $n+1$ is constructed by connecting corresponding processors in two hypercubes of dimension $n$. The diameter of hypercube grows logarithmically with the size of the network. A 4-dimensional hypercube requires 4 links to connect to 4 other processors, however since main up is used to connect to host, a hypercube of 16 processors cannot be implemented.

![Hypercube Diagram](image)

**Figure 3.6: Layout of Hypercube Topology**

3.2 Software Architecture

We have developed a system which is compatible to space shared and time shared system resources. Partitions used by a space sharing scheduling policy are described in a
configuration file which defines how the processes are distributed on the network and it also defines the priority of processes. Because the transputer system supports multiprogramming, the order of execution of processes is controlled by the transputer system. We do not have enough control to set the execution sequence and the time for each execution using only the multiprogramming support provided by the transputer system. We have developed a second layer of software using the basic services of the transputer system to provide our own preemption control. The second layer also provides communication services among processes located on any processor in the network.

The following sections will present how jobs are distributed to processors. Then enhanced inter-process communication, shared memory and time sharing support will be discussed. Finally the method of collecting statistics will be presented.

3.2.1 Job Allocation

The structure of the schedulers is presented in Figure 3.7.

![Figure 3.7: Structure of Schedulers](image)
In order to implement a distributed ready queue approach described in section 1.2, three levels of scheduling are implemented. A super scheduler (SScheduler) manages the system ready queue. A job submitted to the system is handled by the super scheduler and it will be dispatched to one of the partition scheduler. Then the partition scheduler notifies local schedulers that manage the local ready queue of each processor.

Processes of jobs are pre-loaded into partitions and are specified in the configuration file. The processes are halted until a signal is given from local schedulers. Only one process in a processor is allowed to be executed at once and it is controlled by local scheduler. Another process is executed either when the executing process is finished or it has completed its time slice.

Job scheduling in the multiprogrammed environment is achieved with a number of software modules developed as a part of this research. Each of these is briefly described.

3.2.1.1 Super Scheduler

The super scheduler is responsible for distributing jobs to different partitions according to the resources requirement of jobs. It is always located on the processor connected to the host, so that it can get inputs from user and display results received from jobs. A job is created in here to simulate a job that is being submitted to the system. The job is then queued up and waits for a spare partition to accept it. The super scheduler will be notified by partition schedulers when partitions are able to accept new jobs. Then the
super scheduler will send the job's information to the partition scheduler and wait while
the job is finished.

### 3.2.1.2 Partition Schedulers

Partition schedulers are used to manage the usage of a partition. A partition may
contain 1 to 16 processors and each partition is equipped with one partition scheduler.
Depending on the scheduling policy used a partition scheduler can accept one or more
jobs executing in one partition. A signal is sent to the super scheduler from the partition
scheduler if the partition can accept a new job. After the information of a job is received
from the super scheduler, the partition scheduler will notify the local schedulers in its
partition and processes of that job will be executing in the processors managed by them.

If we configure the number of processors of a partition to 16, then space sharing is
disabled. The partition either executes jobs submitted to the system in a FCFS basis or
they are time shared. In this case, a job can achieve the highest parallelism; however either
the waiting time of a job will be increased if FCFS is used to schedule jobs, or with time
sharing the execution time of a job increases because of a longer processes queue and an
increment of context switching overhead. Also note that the speedup of parallel system is
not linearly proportional to the number of processors in the system. For example, the
marginal performance gain of a parallel program will decrease as the number of processors
used increase, until a certain point, when adding one more processor will decrease the
performance of the system. In this case, space sharing is preferred to further utilize the processors in the system by allowing more jobs to be executed in smaller partitions.

At the other extreme, one processor is a partition. Each job is executed in a partition as if it is running on a single processor machine. Opposite to the previous case, the waiting time of a job is minimized; nevertheless the parallelism is minimized as well.

3.2.1.3 Local Schedulers

One local scheduler is associated with one processor and it manages processes allocated to that processor. Each local scheduler has a local ready queue and all the processes assigned by the partition scheduler are inserted in the queue. Local schedulers behave differently with FCFS and time sharing scheduling policies. If the FCFS policy is used, local schedulers allow a process to run to completion and fetch another process after the last process is finished. Local schedulers support time sharing by using their own preemption control, which will be discussed in 3.2.4.

3.2.1.4 Coordinator and Worker Processes

A job consists of processes and these processes consist of a coordinator process and a number of worker processes. A coordinator process is the coordinator of a job, and is responsible for organizing the worker processes of that job. Coordinator processes also determine the number of processors to be used and how the data of an application will be distributed to worker processes. Worker processes are where the procedures actually

40
compute and manipulate the data received from coordinator processes. Three applications are used in this thesis and three kinds of coordinator process and their corresponding worker processes are implemented: matrix multiplication (MultiMain and MultiSub), parallel sorting (SortMain and SortSub) and generic workload (GenMain and GenSub). More than one instance of the same application can be loaded into the system and each application has unique group identification number (GID). All the processes in an application share the same GID and each process has unique process identification number (PID). A process can be identified by combining the GID and PID to form a GUID, which is used in the inter-process communication that will be described in section 3.2.2. Basically, a coordinator process receives the job information from the local scheduler and then it generates data regarding the information received. According to the job information, it then divides the data it has generated and distributes the divided data to the corresponding worker processes. Following this, worker processes compute the data received and exchange intermediate results with other worker processes. Ultimately, the results are sent back to the coordinator process and the computation part of the job is finished. At this point in time all processes of a job notify the local scheduler that a job has been completed and that they can be removed from the local ready queue. The coordinator process also notifies the partition scheduler that a partition is ready to allow the execution of a new job.
3.2.2 Inter-Process Communication

The transputer system uses a point-to-point communication scheme. In this scheme messages from one processor to another processor are required to pass through intermediate processors. However, the channel package provided by the transputer's software library only supports communications between adjacent processors. The transputer system does not support communications between non-adjacent nodes. This limitation complicates the implementation of some parallel algorithms. Consequently, an enhanced version of the inter-process communication package was developed. Instead of using a channel which is a synchronous communication media, we have implemented a mailbox to provide asynchronous communication to avoid potential deadlock of processes. The package consists of a number of processes: a communication switch (SW), four pairs of transmitters (Tx), and receivers (Rx). The structure of the communication package is given in Figure 3.8.

![Diagram of communication network]

**Figure 3.8: Structure of Communication Network**

The heart of the communication software is the communication switch. A communication switch is connected to four pairs of transmitters and receivers by channels.
Each transmitter-receivers pair is associated with one of the four communication links of a transputer. A high priority mailbox is built into the communication network to provide minimum delayed communication which is used by schedulers to exchange job distribution information. Mailbox is a facility to provide communication among processes in the same processor. Generally, mailbox is a piece of shared memory associated with a set of functions which are responsible to put mails into the mailbox and get mails from the mailbox.

3.2.2.1 Communication Switches

A communication switch has two responsibilities:

1. To receive messages:

   Messages can be received from one of the receivers, schedulers, main processes or sub processes. At the beginning of the execution, a communication switch creates a mailbox for each process located in the same processor as the communication switches. Then each process receives a mailbox of its own and a mailbox of the communication switch. A process can send a message to another process by putting a message header followed by the actual contents of the message into the mailbox. In the message header, it records the destination processors that the message will be sent to and the GPID of the receiving process. The communication switch can retrieve a message from its mailbox, and send it to the destination process.
2. To route messages to one of the transmitters or to direct messages to appropriate mailboxes

The processor where a communication switch is located may not be the final destination of a message received by the communication switch. The communication switch is required to route the message to a link that leads to the destination processor of the message. A lookup table is applied to locate the link to be used. Once the destination of a message is reached, the communication switch will put the message into the mailbox of the recipient process according to the G PID recorded in the message header.

3.2.2.2 Transmitters

Transmitters are attached to the four communication links and they are connected to the receivers of the adjacent processors by channels. Since the sending function of a channel will duplicate the contents of a message, the receiver of a channel will get a copy of the message. Therefore, after the transmission of a message to the receiver, a transmitter is required to free the memory of the message to keep the usage of memory low. For example, only one copy of a message exists in the system at a time. Note that the speed of the communication link 0 and link 3 are half of the link 1 and 2 (Figure 3.9). However, in many other distributed memory system used in scientific computing all the links are of the same speed. In order to simulate links of same speed, transmitters
attached to link 1 and link 2 will send the header and the body of a message twice to compensate the difference.

![Diagram of Transputer's Communication Links](image)

**Figure 3.9: Transputer's Communication Links**

### 3.2.2.3 Receivers

Similar to transmitters, receivers are attached to the communication links and receive twice on link 1 and link 2. To avoid the potential of deadlock in using a synchronous channel, receivers receive messages from channels and put them into the mailboxes of communication switches immediately.

### 3.2.3 Memory Management

Shared memory support between processes in the same processor has been implemented. A memory management unit (MMU) is added to every processor. MMU is used to allocate and release shared memory. A large piece of memory is reserved by the MMU and processes can send a request to the MMU asking for a piece of shared memory of certain size. Then MMU will mark a free memory space as used and return the address
of that piece of memory. Similarly, freeing memory can be done by sending a memory
releasing request together with the address of the memory to be freed. The MMU will
mark that address space as unused that allows for later reallocation.

Shared memory is used to reduce the usage of memory by allowing data sharing
between processes in the same processor. A typical usage of shared memory in our
system is the implementation of mailbox. Mailboxes use shared memory to provide
communications between processes. Using channels provided by the transputer system
will result in having two copies of messages, which is not desirable in a system with a
limited amount of memory.

One example showing the advantage of using shared memory is the transition of
messages from a receiver to a transmitter. A sufficient amount of memory must be
allocated before a receiver can retrieve messages from a channel. These messages will be
sent to a communication switch and then to a transmitter if the destinations of the
messages are not reached. By using the channel's to receive operations provided by the
transputer's library [INMOS 1990], the communication switch and the transmitter both
have to allocate memory for messages before retrieving the messages. In contrast, by
using shared memory, only one copy of messages are kept in a processor and only pointers
are transmitted between processes, which reduces the communication overhead.
Moreover, the memory allocated by receivers can be released by transmitters or any
process after it has finished using the memory.
However, the communication between processors has to be attained by passing messages through channels. Since we do not have control on the execution sequences of communication units, allocation of memory to the receiving buffers may be a problem when a lot of messages for other processors are passing through a intermediate processor. This problem will be discussed in detail in the next section. These transition messages will use up all the shared memory, which prevents processes from allocating any memory and new messages to be received from other processors. A process will be blocked until the required memory can be allocated causing a deterioration in performance hit.

3.2.4 Preemption Control

The multi-tasking support of the transputer system provides only two levels of priority. Processes are either high priority, which run to completion, or low priority. Low priority processes are run by a round robin scheduler provided by the transputer that uses a fixed time slice. We can neither stop a running process nor schedule a process to be executed next. Therefore, we have implemented our preemption control mechanism such that executing processes can be stopped whenever we want and still allow processes to be executed. The implementation uses the fact that a high priority process can interrupt a lower priority process. All the schedulers, memory management unit, communication switches, transmitters and receivers are high priority processes; processes of applications are low priority processes. The processes are configured this way to ensure a low priority
process can allocate or free memory from MMU and the communications between processes can be done immediately.

Preemption control is provided by the cooperation between local schedulers and coordinator and worker processes. Before the execution of a coordinator or worker process, a process control structure is sent from the local scheduler to each coordinator and worker process. The process control structure contains an execution instruction of a process and a semaphore to control the flow of a process, and it can be accessed by both the local scheduler and the process. Preemption is controlled by local scheduler which keeps a list of active processes and changes the execution instruction of processes according to the scheduling policy used. A process is allowed to be executed only if the execution instruction of the process control structure is true. Otherwise, a process would be stopped by letting the process wait on a semaphore. The process is allowed to continue execution only if the local scheduler signals the semaphore again and changing the execution instruction to true. Preempt() functions are inserted in coordinator and worker processes where preemption are expected to occurred. Good choices of placing the preempt() function are before and after receiving messages and somewhere inside a loop. The pseudo-code of the preempt() function is presented in Figure 3.10. The preempt() checks for the execution instruction and it allows continue execution only if the execution instruction is true. Otherwise, a process is stopped and waits for the semaphore in the process control structure.
Preempt(process_control) {
    if (process_control.execution_instruction == TRUE){
        return;
    } else {
        semaphore_wait(process_control.semaphore);
    }
}

Figure 3.10: Pseudo-code of preempt() function

Time sharing is supported by using the preemption control. A flow chart of the time sharing support provided by local scheduler is given in Figure 3.11.

Figure 3.11: Time Sharing Support

At the very beginning, a local scheduler waits for a signal from the partition scheduler (PS) and then a process is inserted into the local ready queue. The process
added to the queue is then allowed to execute. Following this the local scheduler will
sleep for a time quantum to allow the process to run. Afterwards, the local scheduler
wakes up and checks if the running process is finished or not. If the process is finished, it
will be removed from the local ready queue. Otherwise, the local scheduler will check for
new processes from the partition scheduler and add them into local ready queue if any. If
there are more than one process in the queue, the running process will be preempted by
changing the execution instruction in its process control structure to false. The change
will be captured by the preempt() function in the running process and the running process
will be stopped waiting for the semaphore in its process control structure. Then next
process will be fetched from the queue. Thereafter, the local scheduler changes the
execution instruction of the next process to true and signals the semaphore in the process
control structure to allow the next process to continue execution.

3.2.5 Collection of Statistics

Traditional measures such as average execution time and average response time of
jobs are used to compare the performance of scheduling policies. The statistics are
collected in the super scheduler. The time of a job creation is recorded and the time at
which a job starts execution is also recorded. These are used to calculate response times
and execution times respectively. When a job is finished, the response time of the job is
computed as the finishing time minus the creation time, while the execution time of the job
is computed as the finishing time minus the starting time.
Messages sent from one process to another process are queued in the mailbox of the communication switch, which may increase the transfer time of a message. For time critical messages like job distribution messages and job finishing messages, delays in transition may result in collecting inaccurate data. Therefore, a special high priority mailbox is integrated into the inter-communication support that is used to transfer time critical messages. A message in the high priority mailbox is processed before messages residing in the communication switch’s normal mailbox are processed.
CHAPTER IV

EXPERIMENTS AND RESULTS

A typical parallel program consists of a coordinator process and several worker processes. The coordinator process divides a problem into smaller problems and distributes the smaller problems to the worker processes. A worker process solves its problem by itself or by exchanging intermediate results with other processes. Solutions of the smaller problems are sent to the coordinator process and the coordinator process processes the results to form the final solution to the original problem. Every parallel program requires some form of communication among coordinator and worker processes.

In order to explore the effect of communication patterns and topologies on the scheduling policies, several sets of experiments have been performed. The applications used to perform the experiments are matrix multiplication, parallel sorting and generic workload applications. Matrix multiplication is used to model applications that do not exchange intermediate results with other processes. Parallel sorting is used to model divide-and-conquer types of applications. To provide a more general structure of different applications, a generic workload model is used to provide more control on how processes in an application communicate. The following sections provide the description of the experiments.
4.1 Matrix Multiplication

The matrix multiplication application performs multiplication of two matrices. This application can be speeded up by distributing the workload of the multiplication to different processors. For example, in a system with 4 processors, each processor is responsible for calculating \( \frac{1}{4} \) of the resulting matrix.

4.1.1 Description of Software

Matrix multiplication has a simple fork-join structure (Figure 4.1).

![Diagram of matrix multiplication structure]

Figure 4.1: Structure of matrix multiplication

The coordinator process performs multiplication of two matrices by sending parts of the matrices to the worker processes, which then perform the multiplication. After this, the results are sent back to the coordinator process to form the resulting matrix.

Figure 4.2 and Figure 4.3 are the pseudo-codes of the coordinator and the worker processes, respectively.
Coordinator Process:
A[0..ROWS_A] = matrix A with ROWS_A number of rows
B[0..ROWS_B] = matrix B with ROWS_B number of rows
/* A[0] represents the first row of matrix A and
   A[0..10] represents from row 0 to row 10 of matrix A */
N = number of worker processes
rows = ⌊ROWS_A / N⌋
/* If ROWS_A is not divisible by N, the remaining rows are evenly distributed
   to the first remainder worker processes where remainder = ROWS_A - rows * N */
FOR I = 0 to N-2
    start = I * rows
    end = start + rows
    Send A[start..end] to worker process I+1
    Send B[0..ROWS_B] to worker process I+1
ENDFOR
start = (N-1) * rows
end = ROWS_A
Result[start..end] = Multiply A[start..end] with B[0..ROWS_B]
FOR I = 0 to N-2
    start = I * rows
    end = start + rows
    Result[start..end] = Receive rows rows from worker process I+1
ENDFOR

Figure 4.2: Pseudo-code of coordinator process for matrix multiplication

Matrix A and matrix B are the matrices to be multiplied and they have ROWS_A and
ROWS_B number of rows respectively. For an application with N worker processes,
matrix A is divided into N parts such that each part consists of ROWS_A / N rows. Each
part is sent to a worker process. Also, the whole matrix B will be distributed to each
worker process. While the worker processes are working, the coordinator multiplies the
remaining of matrix A with matrix B and puts the result into matrix Result. Then, the
coordinator process waits for the results from all worker processes and puts the received
results into proper place in the matrix Result. After receiving results from all worker
processes, the matrix Result will contain the product of the multiplication.
**Worker Process:**
A[0..ROWS] = Receive ROWS rows from Coordinator Process
B[0..ROWS_B] = Receive ROWS_B rows from Coordinator Process
Result[0..ROWS] = Multiply A[0..ROWS] with B[0..ROWS_B]
Send Result[0..ROWS] to Coordinator Process

Figure 4.3: Pseudo-code of worker process for matrix multiplication

A worker process receives two matrices from coordinator process and multiplies the matrices. Then the product of the matrices is sent back to the coordinator process to form the final result.

### 4.1.2 Description of Experiments

The software architecture of an application depends on the application types and system software, etc. For applications using variable software architecture, the number of worker processes forked from a coordinator process changes according to the number of processors allocated to the application; for applications using fixed software architecture, the number of worker processes forked from the coordinator process are fixed independent of the number of processors allocated. For example, under fixed software architecture, the number of worker processes in a partition with 8 processors are the same as the number of worker processes in a partition with 16 processors. In our experiments, both software architectures are considered.

The first experiment is performed to measure the speedup of the application. Speedup(N) is the ratio of the execution time of running an application using one processor to the execution time of the same application running on N processors.
Figures 4.4 and 4.5 show the speedup curve of matrix multiplication using fixed and variable software architecture respectively. Note that logarithmic scale is used in the x-axis of the figures. Therefore a linear speedup is not shown as a straight line in the figures. The curve is obtained by performing matrix multiplication of two 60x60 matrices using 1, 2, 4, 8 and 16 processors configurations. For each configuration, all possible topologies are constructed to evaluate any impact of the inter-connection network on the performance. Since the transputer system we are using is a single user machine such that the execution time of each experiment are very close. Each experiment is repeated for only five times and results are within 10% of the mean values.

Another set of experiments involves measurements of mean response times of a batch with 16 matrix multiplication applications using time sharing and static scheduling policies. The response time of a job is the sum of the waiting time and the execution time of that job. The response time of each job is measured and is used to calculate the mean response time. The batch consists of 16 jobs in which 12 jobs are small jobs and 4 jobs are relatively large jobs. A small job multiplies two 50x50 matrices and a large job multiplies two 100x100 matrices. These two job sizes are used because we want to introduce some variance in job size.

The second set of experiments is performed on a 16 processor transputer system and processors are equally partitioned. The number of partitions in the system depends on the partition size. For a partition with size $p$, the number of partitions in the system is $16/p$. For example, in a system with 1 partition, each partition has 16 processors; with 2

56
partitions, each partition has 8 processors; with 4 partitions, each partition has 4 processors; with 2 partitions, each partition has 8 processors; and with only 1 partition, the partition has all 16 processors.

Using static scheduling policy, one job is assigned to each free partition and the job is run-to-completion. Other jobs wait in the queue until a partition is available. In time sharing, all 16 jobs are distributed equally among the partitions. Since the response time measured includes the waiting time of a job, the order of a job's execution affects the performance of the static scheduling policy. The reason is that if a large job is executed first, the waiting time of other jobs will increase, which increases the mean response time.

To ensure fairness in comparing static and time sharing scheduling policies, the response time of a job in static scheduling is taken as the average of the best and the worst response time. The best response time is obtained by executing jobs with the smallest demand first, such that the waiting time of jobs is minimal. The worst response time is obtained by executing jobs with largest demand first, such that jobs with smaller demand are kept waiting in the queue.

Both fixed and variable software architectures are studied and each topology with different partition size are compared.

4.1.3 Results

The speedup curves of matrix multiplication in fixed and variable software architecture are similar in shape. This is because matrix multiplication does not reduce problem size. The difference between fixed and variable software architecture is that the
number of processes in fixed software architecture is fixed and the number of processes in
variable software architecture depends on the partition size. Assume that the number of
processes used is $N$, that the number of rows and columns of matrix $A$ is $ROWS_A$ and
$COLS_A$ and the number of rows and columns of matrix $B$ is $ROWS_B$ and $COLS_B$. A
worker process receives $ROWS_A / N$ number of rows and the whole matrix $B$. The
worker process requires $OP = COLS_A \times ROWS_B \times (ROWS_A / N) \times ROWS_B$
operations to obtain the partial result. Therefore, $OP \times N = ROWS_B^2 \times ROWS_A^2$
operations are required to get the product of matrix $A$ and matrix $B$, which is independent
to the number processes used. Accordingly, adding more processes into the application
does not substantially increase performance. The performance gain is mainly due to
better utilization of the CPU in fixed software architecture. The hypercube has the largest
speedup followed by the mesh, then the ring, with the linear architecture producing the
least speedup. This is because increasing in partition size has the least effect on hypercube
and the most effect on linear array. The speedup starts decreasing with partition size = 8,
since the communication overhead increases at a faster rate than the performance gain in
parallelism with large partition. Hence, the linear topology performs inadequately
compared to other topologies on large partitions, because it suffers from increased
overhead much more than the other topologies.
Figure 4.4: Speedup of matrix multiplication (Fixed Software Architecture)

Figure 4.5: Speedup of matrix multiplication (Variable Software Architecture)

Figure 4.6 and Error! Reference source not found. show the results of the second experiment. The x-axis represents the partition size. The corresponding number of partitions is 16 / partition size. The letter following the partition size indicates the
topology used in the partition. Letters ‘l’, ‘r’, ‘m’ and ‘h’ represent linear array, ring, mesh and hypercube respectively. For example, label ‘8l’ means there are $16/8=2$ partitions, each partition has 8 processors and the partition is configured as a linear array.

![Matrix Multiplication - Fixed Software Architecture](image)

**Figure 4.6: Response time of matrix multiplication (Fixed Software Architecture)**
Figure 4.7: Response time of matrix multiplication

(Variable Software Architecture)

The result of these experiments show that time sharing always performs worse than static scheduling for this application. This is due to the problem of memory allocation and the message congestion. Memory allocation problem occurs when time sharing scheduling policy is used. The problem is that time sharing loads all the jobs in the queue into the system and all processes of the jobs start execution at the same time. For a system with limited memory, the system will run out of memory very soon because processes allocate memory to store data for processing and the system allocates memory for passing messages between processors. Once all the memory are occupied, any more allocation will be blocked until some memory is released from other processes. Therefore it introduces a queuing delay in memory allocation.

61
Another problem is the congestion problem. The problem occurs when many processes try to send messages to other processes. Time sharing loads a lot more processes into the system than static scheduling such that time sharing is more likely to be suffered in the congestion problem. Messages transit from one processor to the others by passing through communication switches, transmitters and receivers. The communication switch fetches a message from its mailbox and passes the message through an appropriate link to other processors. Since the communication between processors are blocking, a message sending through link 0, for example, will block other messages trying to use link 0. Therefore messages sent through the same link wait in the mailbox until the link is free. The memory allocated to the messages in the mailbox cannot be released until the messages are sent to other processor. Therefore, as the number of messages in the mailbox increases, the more memory is used. Hence a smaller amount of memory can be allocated to receive messages from other processors. Consequently, a chain reaction will be started and cause a processor to wait for messages to be received from other processors to release memory for receiving messages, however, other processors may fall into the same situation so that they are waiting for messages to be cleared as well. Eventually memory will be cleared but a queuing delay will be introduced.

Note that for partitions with size of 4 and 8 for all topologies, response time of time sharing is much worse than static scheduling. The response time of time sharing increases significantly when the partition size increases and the different between time sharing and static scheduling increase substantially. This is caused by the memory problem we have just discussed. As the partition size increases, the number of jobs per partition
increases in fixed software architecture. Therefore, more processes are loaded in each processor causing the memory and congestion problem. However, static scheduling does not suffer in this situation since at most one job is executing per partition in static scheduling. Accordingly, the performance of static scheduling is much higher than time sharing policy.

In variable software architecture, both time sharing and static scheduling perform the best with partition size, \( p = 2 \) because the speedup increases most extensively from \( p = 1 \) to 2. Since matrix multiplication does not reduce problem size, increases in number of processes do not improve performance. Therefore, the system will obtain best performance if partition is set with size with the highest marginal increase in speedup. Another reason for time sharing working best at \( p = 2 \) is that there are only two processes in each application and the inter-process communication occurs only between two processors and the number of messages between processors are minimal. It is less likely that the problem of message congestion will occur.

4.2 Parallel Sorting

A parallel sorting application sorts an array of numbers in ascending order. Parallel sorting accelerates the sorting process by sharing the workload and reducing problem size. This sorting is done by assigning half of the array to one process and the other half to another process. The process sorts the array by splitting it and assigning the split array to other processes, until the array is small enough to be sorted by itself.
Afterwards, all sorted arrays are merged to form a large sorted array. The detailed implementation of the program is presented in the next section.

4.2.1 Description of Software

The structure of parallel sorting described in this section is represented by a binary tree (Figure 4.8). Nodes in the circle represent the same process and its function depends on its level at the tree.

![Diagram of parallel sorting structure](image)

**Figure 4.8: Structure of parallel sorting**

The parallel sorting program implemented here allows either 1, 2, 4, 8 or 16 processes in each application and the number of processes used is the same as the number of leaf nodes in the binary tree. The behavior of a coordinator process and a worker process are the same. The only difference between them is that the coordinator process initiates the data distribution, while the worker process waits for data input before it starts...
and returns a sorted array upon completion. The pseudo-code of a coordinator process and a worker process are shown in Figure 4.9 and Figure 4.10 respectively.

The coordinator process and worker process both sort and merge data. The behavior of a process depends on its level in the tree. The parallel sorting application is divided into $n$ levels and the array is split and distributed at each level, except at the last level $n$. For a process at level $s$ where $1 \leq s < n$, the process splits the array into half, maintains one half of the array and passes the second half to a new process. A new process is created at level $s+1$. This new process splits the array fragment that received from the previous level in half. This splitting is repeated until the process reaches level $n$. A process starting from level $n$ will perform the sorting and the sorted array is returned to its parent process. A process proceeding to stage $n$ sorts the undistributed elements in the array and returns them to the previous level. Here it waits for its child to return a sorted array. This is the end of the sorting phase and the algorithm enters its merging phase.

In the merging phase, a process receives an array from its child and it will merge the received array with the array sorted by itself. If the process at level $s$ is equal to the level in which it was created, the merged array is returned to its parent process. Otherwise, the process maintains the merged array and returns to level $s-1$. Eventually, the coordinator process will obtain a sorted array.
**Coordinator Process:**

\[ n = \text{number of stages} \]
\[ A[0..N] = \text{array } A \text{ with } N \text{ number of elements} \]
\[ \text{remaining} = N \]

\[
\text{FOR stage} = 1 \text{ to } n-1 \\
\quad \text{split} = A[\text{remaining}/2..\text{remaining}] \\
\quad \text{remaining} = \text{remaining}/2 \\
\text{Send } \text{split}[0..\text{remaining}] \text{ to worker process } 2^{\text{stage}-1} \\
\quad \text{Send } \text{stage}+1 \text{ to worker process } 2^{\text{stage}-1} \\
\text{ENDFOR} \\
\text{sorted} = \text{sort}(A[0..\text{remaining}]) \\
\text{FOR stage} = n-1 \text{ to } 1 \\
\quad \text{Receive } \text{temp} \text{ from worker process } 2^{\text{stage}-1} \\
\quad \text{sorted} = \text{merge}(\text{sorted}, \text{temp}) \\
\text{ENDFOR}
\]

**Figure 4.9: Pseudo-code of coordinator process for parallel sorting**

**Worker Process:**

\[ n = \text{number of stages} \]

Receive \[ A[0..N] \] from parent process
Receive \[ \text{starting\_stage} \] from parent process

\[ \text{remaining} = N \]

\[
\text{FOR stage} = \text{starting\_stage} \text{ to } n-1 \\
\quad \text{split} = A[\text{remaining}/2..\text{remaining}] \\
\quad \text{remaining} = \text{remaining}/2 \\
\quad \text{Send } \text{split}[0..\text{remaining}] \text{ to worker process } 2^{\text{stage}-1} \\
\quad \text{Send } \text{stage}+1 \text{ to worker process } 2^{\text{stage}-1} \\
\text{ENDFOR} \\
\text{sorted} = \text{sort}(A[0..\text{remaining}]) \\
\text{FOR stage} = n-1 \text{ to } \text{starting\_stage} \\
\quad \text{Receive } \text{temp} \text{ from worker process } 2^{\text{stage}-1} \\
\quad \text{sorted} = \text{merge}(\text{sorted}, \text{temp}) \\
\text{ENDFOR} \\
\text{Send } \text{sorted}[0..N] \text{ to parent process}
\]

**Figure 4.10: Pseudo-code of worker process for parallel sorting**

### 4.2.2 Description of Experiments

The first set of experiments performed on parallel sorting are conducted to measure the speedup of the application. The experiments sort an array of 6500 elements
using 1, 2, 4, 8 and 16 processors with different topologies. Each experiment is repeated five times to obtain mean execution time and to make sure all data obtained are within 10% of the mean value.

The second set of experiments uses the same configuration described in the second experiment of matrix multiplication. Similarly, 16 jobs are executed in each experiment; there are 12 small jobs and 4 large jobs. A small job in these experiments sorts 6000 elements while a large job sorts 14000 elements. Experiments are performed for all possible topologies with partition sizes set to 1, 2, 4, 8 and 16 processors. Both fixed and variable software architecture are also considered as well.

4.2.3 Results

The speedup curves of parallel sorting for fixed and variable software architecture is presented in Figure 4.11 and Figure 4.12. The speedup depends significantly on the performance of the inter-connection network. Hence, hypercube performs the best; mesh is the second; ring is the third; and linear is the worst. The speedup in fixed software architecture is almost linear when the number of processor used ranges from 1 to 4. The incremental benefit of using more parallelism decreases when more processors are used. This is due to the fact that communication overhead increases. In addition synchronization overheads, which involves the time used to synchronize processes, also increase.

For example, in our case the waiting time for a message from one to the other process increases when the number of processes in the system increases. This is because
the more processes are in the system, the more likely that more communications have occurred in the network. Consequently, messages queue up in the communication switch and the time required to receive a message increases. Therefore, the speedup slows down when the number of processor used is in the range of 8 to 16.

![Sorting - Speedup (Fixed Software Architecture)](image)

**Figure 4.11: Speedup of parallel sorting (Fixed Software Architecture)**

The speedup in the variable software architecture increases dramatically with the number of processors, as the parallel sorting algorithm used reduces a problem's size by half when the number of processors used is doubled. The speedup increases in order n^2 and slows down in 8 and 16 processors per partition for the same reason as in the fixed software architecture.
Figure 4.12: Speedup of parallel sorting (Variable Software Architecture)

Results from the second experiment demonstrate that time sharing suffers from congestion and memory allocation problems, as in matrix multiplication. The difference in performance between time sharing and static scheduling increases as the number of processors per partition increases. A symptom of the congestion problem is that, as the amount of communication between processors increases, it is likely more messages are passing through the same link, and the response time of a job increases significantly. In Figure 4.14, the performance of the system with partition size = 2 under time sharing is better than static scheduling because most of the communications between processes are local and there is less congestion problem. Under the same configuration with variable software architecture in Figure 4.13, static scheduling performs better than time sharing because time sharing is best for utilizing CPU time among processes. In variable software architecture, the number of processes used depends on the partition size. Since the
Algorithm of parallel sorting cuts the problem size in half when the number of processors is doubled, the mean response time of jobs in the system decreases significantly when the partition size increases.

Figure 4.13: Response time of parallel sorting (Variable Software Architecture)
Figure 4.14: Response time of parallel sorting (Fixed Software Architecture)

4.3 Generic Workload

Generic workload is proposed by [Nanda et al. 1991] is used to emulate the behavior of different programs in order to represent various workload patterns. The model reads a set of parameters to identify the characteristics of a specific workload and emulates the execution of the workload. The workload parameters that can be used to characterize a workload are:

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>comp</td>
<td>mean computation time</td>
</tr>
<tr>
<td>comm</td>
<td>mean communication time</td>
</tr>
<tr>
<td>proc</td>
<td>number of processes to be used</td>
</tr>
<tr>
<td>B</td>
<td>branching factor</td>
</tr>
</tbody>
</table>
Computation time specifies the time to be spent in computation in each process, and the time spent in each process can be different to provide variance among processes in the workload. The parameter $comp$ represents the average computation time of each process. The amount of communication specifies the amount of data to be sent by a process, and each process can have different amount of communication. The parameter $comm$ represents the average amount of communication of all processes. In a typical parallel program, processes have application cycle of where they exchange intermediate results and then process the data. Therefore $comm$ can be used to simulate the intermediate results to be sent and $comp$ can be used to simulate the time spent on processing the data. The parameter $proc$ controls the number of processes to be used in the workload, such that emulation of the fixed and variable software architecture can be done. The branching factor controls the number of processes that can be forked from one process. For example, in parallel sorting the branching factor is 2, such that a process can split the array into two parts and assign them to two processes.

4.3.1 Description of Software

Generic workload has a similar structure to parallel sorting (Figure 4.15). Instead of doubling the number of processes in each level, the increase in number of processes is according to the branching factor, $B$, such that the number of processes at level $n$ has $B^n$ processes. The number of level, $n$, in the model depends on the number of processes specified in the workload parameter, which can be calculated as $\log_b(proc)$. 
The pseudo-code of generic workload is presented in Figure 4.16 and Figure 4.17.

A process spawns $B-1$ processes in each stage until level $n$ is reached. Typically, parallel programs receive intermediate results from other processes. Then they evaluate the data and return the results to their parents. In order to emulate general parallel programs, each process is given a similar software structure, such that a process receives some messages and then it performs some computations before returning some messages to the parent. The number of messages received and returned to the parent and the duration of the computations are specified with the help of workload parameters.
Figure 4.16: Pseudo-code of coordinator process for generic workload

The coordinator process sends messages to all its children and then performs some computations and waits for messages to return for further manipulation. The length of a message to be sent can be specified by variable `comm` and the value changes according to the current stage of the process. The computation time is specified by variable `comp1` and `comp2` and the computation is performed by `compute()` and `compute2()` functions. The `compute()` and `compute2()` functions are dummy functions using `comp1` and `comp2` as inputs respectively. The two functions are designed primarily to spend time specified by their input parameters. The computation behavior can be changed by changing the complexity of the `compute()` and `compute2()` functions. For example, `compute()` could be a function with complexity of $O(n^2)$ to simulate the insertion sort and `compute2()` could be a function with complexity of $O(n)$ to simulate a merging function. The variables `comp1`
and \( \text{comp2} \) represent the time to be spent on computing, before and after synchronization respectively. The two variables together represents the workload parameter \( \text{comp} \).

```
Worker Process:
ID = worker process number
n = \log_2(\text{proc})
Receive messages from parent process
Receive \text{starting level} from parent process
FOR level = \text{starting level} to n-1
    FOR b = 0 to B-2
        /* worker process is not at level n */
        Send message[\text{commID}] to process with ID */
        Send message[\text{commID}+\text{level}] to worker process \( ID+B^{\text{level}+1}+b \)
        /* tell the starting level of worker process */
        Send level1 to worker process \( ID+B^{\text{level}+1}+b \)
    ENDFOR
ENDFOR
compute(\text{compID}+\text{level})
FOR level = n-1 to \text{starting level}
    FOR b = 0 to B-2
        Receive message's from worker process \( ID+B_{n-1}^{\text{level}+1}+b \)
        compute2(\text{comp2ID}+\text{level})
    ENDFOR
ENDFOR
```

**Figure 4.17: Pseudo-code of worker process for generic workload**

Similarly, a worker process receives messages from a parent process and spawns more processes as necessary. Then it computes an amount of time determined by the workload for some times specified in the workload parameter and, finally, sends a message to the parent process.

### 4.3.2 Description of Experiments

In order to validate the generic workload, experiments using generic workload to emulate the behavior of matrix multiplication and parallel sorting applications are performed. Matrix multiplication has the same structure as generic workload with the
number of levels = 1. That is, the number of processes is equal to the branching factor. Parallel sorting can be emulated by setting the branching factor to 2.

The generic workload has been used to find the effect of different communication patterns and different topologies on performance under time sharing and static scheduling policies. Three communication patterns are used and a communication factor is used to characterize the workload. Communication factor (CF) of an application is the ratio of the total communication time of the application to the total computation time of the application. Communication time includes all the time required to send a message from a source process to a destination process and computation time is the time used to perform data processing locally. The first experiment has minimal communications among processes, such that the time spent in communication is negligible to yield $CF = 0$. Another experiment has the same communication and computation time such that $CF = 1$. The last experiment is the opposite of the first experiment such that the time spent in communication is dominant, to provide $CF = \infty$.

4.3.3 Result

Experiments using generic workload to model the matrix multiplication and parallel sorting applications using fixed and variable software architecture are performed. Results are given in Figure 4.18, Figure 4.19, Figure 4.20 and Figure 4.21. The results of the experiments show that the data obtained from the original applications (see Figure 4.6, Figure 4.7, Figure 4.13 and Figure 4.14,) are similar to those obtained from the generic workload. Since the experiments emulate all the behaviors of the original applications
including memory usage, size of messages and point of synchronization, they also suffer in the same memory and congestion problems. After validation, we can use the generic model to find the relation among topologies, workload and scheduling policies.

![Graph](image)

**Figure 4.18:** Generic workload mimics matrix multiplication

*(Fixed Software Architecture)*
Figure 4.19: Generic workload mimics matrix multiplication

(Variable Software Architecture)

Figure 4.20: Generic workload mimics parallel sorting

(Fixed Software Architecture)
Figure 4.21: Generic workload mimic parallel sorting
(Variable Software Architecture)

To eliminate the congestion problem, experiments with negligible communication
time were conducted. In fixed software architecture, both time sharing and static
scheduling perform best at partition size, $p = 4$ because applications can enjoy the speedup
provided by adequate number of processors while providing sufficient number of partitions
to reduce the waiting time of jobs. Time sharing performs better than static scheduling at
partition size ranges from 2 to 4 processors. The reasons behind such behavior are briefly
presented. First, time sharing produces higher CPU utilization by allowing more jobs to
execute simultaneously. This is because the execution cycle of a job is such that the leaf
nodes are busy and then they are idle and then their parents are busy and then they are
idle. Instead of letting the processor sit idle, using time sharing, the idle processors can be
shared by other jobs to increase system utilization. Secondly, a larger number of partitions keep the number of jobs per partition small enough such that the overhead introduced by timesharing is less than the benefit gained from multiprogramming.

![Generic Workload - Fixed Software Architecture (CF=0)](image)

**Figure 4.22: Generic workload for CF=0 (Fixed Software Architecture)**

Similarly, the performance of time sharing is better than static scheduling when partition size ranges from 2 to 4 in variable software architecture. Since the compute() function used here has the same complexity as in parallel sorting, the computation time is reduced when the number of processors per partition increases.

The optimal number of processors per partition is the size of a partition which provides the best performance. The optimal number of processors per partition increases from 4 in fixed software architecture to 16 in variable software architecture using both time sharing and static scheduling for all topologies. The reason is that the performance of a job suffers significantly when the number of processes used is reduced. Since the
number of processes used in variable software architecture depends on the partition size. In order to achieve same level of parallelism as fixed software architecture, the partition must be enlarged. Therefore, for optimal performance in variable software architecture, the partition size must be increased to increase the parallelism to compensate the performance lost in software architecture.

![Generic Workload - Variable Software Architecture (CF=0)](image)

Figure 4.23: Generic workload for CF=0 (Variable Software Architecture)

If we increases communication time to the same amount as computation time (CF = 1), we obtain a graph with a shape similar to the one obtained from CF=0. The optimal number of processors per partition in fixed software architecture drops from 4 to 2 for all topologies. This is because since computation is not dominant the increase in number of partition which reduces job's waiting time outweigh the decrease in performance by using smaller partition. Another reason is that the increase in communication can be offset by
reducing the partition size, since many processes are located on same processor so that the cost of inter-process communication is reduced.

Note that the experiments show extraordinary results when the system is configured as linear array as shown in Figure 4.25. Time sharing demonstrates an inferior performance compared to static scheduling when the inter-connection networks are ring, mesh and hypercube. However, time sharing is superior than static scheduling when the topology is linear for partition sizes of 1, 2, 4 and 8 processors. This is because time sharing provides some kind of pipe-lining which is not achieved with static scheduling. A message sent from one processor to other processors is required to pass through some intermediate processors. In static scheduling only one job is allowed to be executed at a time such that the intermediate processors sit idle after the message is sent to other processors. Time sharing allows more than one message in transition at the same time.

This is illustrated in the following example. Consider a message passing through a linear array of 8 processors. This requires 7 time units (0.14 message/time unit) in static scheduling. In time sharing 7 time units are required as well to send the message. However, in time sharing another message can be sent after the first message is transmitted from the first processor to the second processor. Therefore, in addition to the original message, there are six additional unfinished messages being sent to its destination. The first message has completed 6/7 of its way; the second message has completed 5/7 of its way and so on. An average of $\frac{6}{7} + \frac{5}{7} + \frac{4}{7} + \frac{3}{7} + \frac{2}{7} + \frac{1}{7} = 3$ additional (unfinished) messages are also in transition to their destination. Counting the message in transition yields an average transfer rate of $\frac{4}{7} = 0.57$ messages/time unit. This effect is most
PM-1 3½" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

<table>
<thead>
<tr>
<th>1.0</th>
<th>2.8</th>
<th>2.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>1.25</td>
<td>2.0</td>
<td>1.8</td>
</tr>
<tr>
<td>1.4</td>
<td>1.6</td>
<td></td>
</tr>
</tbody>
</table>

PRECISION™ RESOLUTION TARGETS
noticeable when all messages are sent in the same path as in the case of linear array. The effect becomes more and more significant when the communication factor increases.

The effect does not show up in partition with 16 processors because the overhead of time sharing such as context switching, communication and synchronization overhead, offset the benefit of pipe-lining in this experiment.

![Generic Workload - Fixed Software Architecture (CF=1)](image)

**Figure 4.24**: Generic workload for CF=1 (Fixed Software Architecture)
Figure 4.25: Generic workload for CF=1 configured as linear array

(Fixed Software Architecture)

Variable software architecture with $CF=1$ has results similar to $CF=0$ because at $CF=1$, the computation time changes according to the number of processes used in the same way as in $CF=0$. However, the amount of communication is fixed such that the computation time is still controlling the shape of the graph. For the same reason as in $CF=0$, the optimal partition size of variable software architecture increases comparing to fixed software architecture to improve performance in parallelism.
Figure 4.26: Generic workload for CF=1 (Variable Software Architecture)

Applications with $CF=\infty$ that perform pure communication operations can display the pipe-lining effect more clearly. As shown in Figure 4.28, without the interference of computation time, time sharing performance proves better than static whenever the topology is a linear array. In other inter-connection network, the effect of pipe-lining is less effective such that the benefit is offset by the cost of context switching and synchronization among processes. Without any computation, the optimal partition size is always equal to 1. It is because all processes are located in the same processor such that the communications among processes can be done by shared memory. Therefore the cost of communication is minimal.
Figure 4.27: Generic workload for $CF=\infty$ (Fixed Software Architecture)

Figure 4.28: Generic workload for $CF=\infty$ configured as linear array

(Fixed Software Architecture)
Without any computation, the effect of software architecture is less dramatic. The cost of communication is even lower in variable software architecture because there are less processes in a job and the amount of communication is less than in fixed software architecture.

![Diagram of Generic Workload - Variable Software Architecture (CF=∞)](image)

**Figure 4.29: Generic workload for CF=∞ (Variable Software Architecture)**

Processes in a job from previous experiments are having the same size which is beneficial in static scheduling policy. This is because processes with same size will finish execution almost at the same time and they will communicate with other processes. Therefore, time sharing will suffer in context switching and synchronization overhead. In experiment which CF=1, static scheduling performs better than time sharing when partition size larger than 4 using fixed software architecture. In order to show the effectiveness of process variance, the same workload parameters are used. The only
difference between the two experiments is that the process variance is changed to high
($CV=15$). Figure 4.30 shows that time sharing outperforms static scheduling in very high
process variance. Comparing to Figure 4.24, performance of system with jobs having high
process variance is worse than jobs with low variance. This is because the distribution of
workload is uneven and the complexity of the compute() function used in this experiment
is $O(n^2)$. Therefore the computation time increases significantly.

![Generic Workload - High Process Variance](image)

**Figure 4.30: Generic workload with high process variance**

### 4.4 Discussion

This section will summarize some of the major observations obtained from the
experiments.
4.4.1 Effect of Memory Allocation, Context Switching and Synchronization

Memory allocation, context switching and synchronization are some of the factors affecting the performance of time sharing scheduling policy. The memory allocation problem was discussed in section 4.1.3. It is caused by processes in a processor competing for memory and causing delay in execution when the system is out of memory. Time sharing allows more than one job executing in a processor and therefore there are a lot more processes competing for memory in comparison to static scheduling. Therefore, in many situations the performance of time sharing is poor compared to static scheduling (see Figure 4.6, Figure 4.7, Figure 4.13 and Figure 4.14). In our configuration, a fixed number of jobs are run, thus the larger the partition size, the more jobs are allocated to each partition. Hence, there are more processes executing in each processor and processes are competing harder for memory. Therefore, time sharing performs worse when the partition is getting larger in this case.

A context switch occurs when a process finishes executing for a time quantum and the execution switches to another process. There are costs associated with the switch such as saving and loading registers and updating variable tables and lists. The cost is called context switching overhead. Time sharing provides better CPU utilization, however, it also introduces the context switching overhead. The overhead increases when there are more processes executing in a processor concurrently. Therefore, a system with large partitions (i.e., more jobs per partition) using time sharing suffers a high context
switching overhead. This can lead to an inferior performance at large partition sizes as shown in Figure 4.22 and Figure 4.24.

Synchronization cost is the time spent on synchronizing processes in a job. For example, in the parallel sorting application, a process splits an array and sends data to another process and waits for the other process returns some results. The time spent on waiting is the synchronization cost. The synchronization cost increases if there are more processes are executing in the system because more times are spent on waiting for result among processes. The additional time spent includes the delay of memory allocation and the queuing delay of messages sending between processors. Time sharing increases the number of processes executing in a partition, therefore it is not desirable to be used in systems with large partitions.

4.4.2 Effect of Communication Factor on Applications

A generic workload model is used to explore the effect of communication factor on applications.

Communication factor \((CF)\) of an application is the ratio of the total communication time of the application to the total computation time of the application. In one extreme, an application with \(CF = 0\) means there is no communications among processes in that application. At the other extreme, an application with \(CF = \infty\) means there is no computation in that application. In our experiments, a generic workload model with \(CF=n\), represents all the jobs in the model have \(CF=n\) when they are executing using one processor.
According to the results from the generic workload model with $CF=0$, time sharing performs better than static scheduling with small partitions size and static scheduling outperforms time sharing with larger partitions. Time sharing performs poor than static scheduling in large partition is mainly because there is a tradeoff between speedup and number of jobs per partition using time sharing policy. By increasing the partition size, the speedup will increase performance. At the same time, the number of jobs per partition will increase context switching and synchronization cost. In our case, the increase in speedup is not sufficient to overcome the increased overhead when the partition size is large.

The result of both $CF=0$ and $CF=1$ has the same shape showing similar observations. It is because the amounts of communication in both cases are small so that the computation part will control the shape of the result.

In a system with $CF=\infty$, the partition size should be as small as possible for both static scheduling and time sharing since the communication cost is dominant when $CF=\infty$ so that a small partition can reduce the communication cost to provide better performance.

4.4.3 Variable vs. Fixed Architecture

Matrix multiplication application has two main characteristics: 1) the algorithm used does not require exchanging intermediate results, hence there is no communication between worker processes; 2) speedup depends on the number of processors used but not the number of processes spawned.
A parallel sorting application can be classified as an application with a lot of communication among processes. In contrast to matrix multiplication application, parallel sorting application reduces problem size as the number of processes used increases.

When comparing these two types of applications, matrix multiplication is desirable in a variable software architecture and parallel sorting is desirable in fixed software architecture. This is because a fixed software architecture has a fixed number of processes per job and a variable software architecture has a number of processes depending on the partition size. Since the performance of matrix multiplication does not increase when more processes are spawned, a lot of time will be wasted in distributing data to processes when the number of processes per job is fixed.

In contrast to the matrix multiplication application, the parallel sorting application gains performance substantially by increasing the degree of parallelism. However, the communications among processes follows a tree structure that limits the increase in communication cost as the partition size increases. Therefore, the performance gain can offset the loss due to communication easily. Consequently, this type of application can perform better by using a fixed software architecture.

4.4.4 Effect of Network Topology

Network topology affects system performance when there are a lot of messages being exchanged among processes. Figure 4.28 shows that the network topology used in a system can affect the decision making of the scheduling policy. When a system consists of a lot of communications and the network topology is configured as a linear array, time
sharing scheduling policy can provide better performance than static scheduling policy. This is caused by some kind of pipe-lining effect described in section 4.3.3. A brief explanation of the effect follows. Time sharing allows more than one message in transition at the same time while in static scheduling, a processor sits idle until the transition of the message is completed.

4.4.5 Effect of Task Variance

Processes with same size are beneficial in static scheduling. Time sharing performance does not provide any additional advantage when all the processes have same size. However, processes with same task size will finish execution almost at the same time and communicate to other processes at the same time. Therefore, time sharing experiences an increase in the context switching and synchronization overheads. Experiment with jobs having larger task variance shows that by increasing the task variance, performance of static scheduling gets worse and eventually time sharing outperforms static scheduling. Therefore, in a system with most of the processes having similar sizes should use static scheduling policy in allocating system resources.

4.4.6 Effect of Scheduling Policy

Results of experiments using matrix multiplication and parallel sorting show that the static scheduling policy performs better than time sharing most of the time. This is mainly because of the influence of the memory allocation problem described in section 4.1.3. The memory allocation problem is not an artifact of our implementation but can
also exist in other systems. It is unavoidable that system memory will be used up when a lot of processes are allocating memory. Consider a system with limited memory, if applications using a lot of memory are time shared, they will spend most of their time competing for memory. Under this situation, one should consider using static scheduling.

However, the generic workload model shows that if the memory problem does not exist (this can be interpreted as the system has enough memory for all processes), time sharing can perform better than static when the partition size is small and the processes of the jobs have high variance.
CHAPTER V

CONCLUSIONS

5.1 Summary

The objective of this thesis is to study the performance of time-sharing and space-sharing policies in distributed memory systems. Two processor scheduling policies - round-robin time-sharing policies and static space-sharing policies - have been implemented on a 16-processor Transputer system. We have studied how network topology, software architecture, variance of process demand and communication factor of application affect the performance of the two scheduling policies. Experiments have been conducted on a 16 processors, point-to-point, distributed memory Transputer system.

This study involves implementing the communication facilities for the Transputer system, the time sharing and static processor scheduler, and three applications. The applications implemented are matrix multiplication, parallel sorting and a generic workload model.

Section 4.1 and 4.2 show that memory allocation and link congestion can affect the performance of a scheduling policy significantly. These two factors are significant for systems with limited memory and that use point-to-point communication links. Most previous studies, which are primarily based on simulation, did not consider these two
important aspects of a distributed memory system. In general, the results show that for a system with jobs requiring large amount of memory, static scheduling is preferable.

Section 4.3 shows that without the interference of the memory allocation problem, time sharing performs better than static scheduling when the partition size is small. We also found that network topology has no significant impact on the relative performance between time sharing and static scheduling policies. Network topology affects the choice of scheduling policy only if the communication factor is high. Under this condition, time sharing has better performance than static scheduling when the inter-connection network is configured as a linear array. The result also shows that time sharing has superior performance to static scheduling if there is high variance in process service times.

Two different software architectures, fixed and variable, have been used for implementing the applications. The variable software architecture is better for applications having the following characteristics:

1) Speedup improves with the number of processors but no additional benefit is achieved by spawning processes more than the number of processors.

2) There is no communication among worker processes.

The fixed software architecture is preferable if applications speedup depends on the number of processes spawned.

5.2 Contributions of the Thesis

This section summarizes the contributions of this thesis. Most of the previous studies in processor scheduling area is based on simulations and analytic models. Many of
the studies have focused on shared memory systems. A lesser amount of work exists in the area of distributed memory systems. We have implemented two processor schedulers on a 16 processor Transputer system in order to evaluate the performance of time sharing and static scheduling policies using real applications. The transputer system is a distributed memory system and communication among processors is achieved by a point-to-point communication networks.

This study has identified that memory allocation and link congestion problem can seriously affect the performance of a system with limited memory that uses point-to-point communication networks.

The results have shown that network topology does not affect the relative performance of time sharing and static processor scheduling policies unless the communication factor is high. In contrast, variance of process service demands can significantly affect the relative performance of these two policies.

5.3 Future Research Directions

Several improvements can be made to further study the effect of process communications on processor scheduling.

Our experiments have been performed on a Transputer system. Each processor is connected to 4 other processors and processors communicate using point-to-point communication network. Experiments using more processors should be performed. The effect of changing network topology may be substantial when the partition size is large on a system with a larger number of processors.
We have studied one time-sharing policy and one space-sharing policy. Other scheduling policies can be studied. Dynamic scheduling policies have not been studied in detail in the context of distributed memory systems. Dynamic scheduling policies require more communication among the schedulers. It is interesting to see how network topology affects the processor allocation decision and the performance of scheduling policies, in a system with a large number of processors.
REFERENCES

[Chiang et al. 1994]


[Crovella 1991]


[CSA 1990a]


[CSA 1990b]


[CSA 1990c]


[Eager et al. 1989]

[Feitelson & Rudolph 1992]

[Fox et al. 1988]

[Ghosal et al. 1991]

[Gupta et al. 1991]

[INMOS 1990]

[Lenoski et al. 1989]

[Leutenegger & Vernon 1990]
[Lovett & Thakkar 1988]


[Majumdar et al. 1988]


[Majumdar et al. 1991]


[Majumdar & Leung 1994]


[McCann et al. 1993]


[McCann & Zahorjan 1994]

[Nanda et al. 1991]

[Ousterhout 1982]

[Peris et al. 1994]

[Setia & Tripathi 1991]

[Setia & Tripathi 1993]

[Sevcik 1989]

[Thacker et al. 1988]
[Vaswani & Zahorjan 1991]


[Vranesic et al. 1991]


[Zahorjan & MaCann 1990]

END
2 1-1 1-9 6
FIN