INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

A Hierarchical Scheduling Policy for Large Scale Distributed Memory Multicomputer Systems

by

Thanalapati Thyagaraj
B.Sc.(Bangalore), M.Sc.(Mysore)

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
September 17, 1997

© 1997, Thanalapati Thyagaraj

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

0-612-26976-0

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
**A Hierarchical Scheduling Policy for Large Scale Distributed Memory
Multicomputer Systems**
submitted by
Thanalapati Thyagaraj, B.Sc.(Bangalore), M.Sc.(Mysore)
in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University
September 17, 1997

Abstract

Since recent times, we are witnessing tremendous increases in the computational power of large scale distributed memory multicomputers. This is allowing an increasing number of researchers and developers to exploit the processing power. Also, attempts are being made to solve more complex problems using these systems. Due to factors such as cost and efficiency, it makes it imperative to share these systems amongst various users and applications. Traditionally, space sharing and time sharing have been used for scheduling in multiprocessor systems. Space sharing performs well at low loads but leads to premature queuing of jobs and wasting of resources at medium to heavy loads. At these loads, however, time sharing policies tend to perform well.

In this thesis, we propose and investigate a novel hybrid approach to scheduling in large scale distributed memory multicomputers, called Hierarchical scheduling, that reflects the potential benefits of space sharing and time sharing while overcoming their drawbacks at the same time. We illustrate the advantages of such a combination through detailed simulation experiments wherein we evaluate the relative performance of the hierarchical policy together with some of the best space- and time-sharing policies under two workload models. In the first model, we consider the popular fork-and-join type of workload and a variant consisting of multiple phases of fork-and-join. The second model is representative of the characteristics of applications at high performance computing centres. Under both these models, we illustrate the superior performance of hierarchical scheduling.

Acknowledgements

First and foremost, I would like to thank Professor Sivarama Dandamudi. He has been my supervisor, advisor, critic, mentor, friend, and much more over the past two years. I am grateful for his excellent technical guidance, generous monetary support, impassionate advice on career, and numerous favors. I am sure that I can always look towards him even in the future.

Over the same period of time, I have had the good fortune to have Professor John Oommen as my friend. His time, help, and encouragement at all times have helped me to a tremendous extent. I know that our relationship exceeds far beyond Carleton and 1997.

Sivarama and John are primarily responsible for me choosing to study at Carleton in the first place.

Special thanks to Professor Evangelos Kranakis, and Professor Danny Krizanc for numerous discussions and conversations.

But for the sacrifices and encouragement of my parents, I would not have been here in the first place. My eternal gratitude to them.

Finally, thanks are due to all my friends, too numerous to name here, for showing me that there is a 'lighter' side to life, after all!

Contents

Abstract	iii
Acknowledgements	iv
List Of Tables	viii
List Of Figures	ix
1 Introduction	1
1.1 Distributed Memory Multicomputers	2
1.2 The Problem of Task Scheduling	3
1.3 Scheduling Road-map	4
1.4 Scheduler or Task Queue Organization	6
1.5 Hierarchical Scheduling Policy (HSP)	8
1.6 Research Considerations	8
1.6.1 The System Model	9
1.6.2 Workload Models	9
1.7 Contributions and Organization of the Thesis	10
List Of Abbreviations	1

2	Previous Work	12
2.1	Relevant Definitions	12
2.2	Coscheduling	14
2.3	Semi-Static Policies	16
2.4	Adaptive Multiprogrammed Sharing Policies	19
2.5	Multiprogrammed Policies for Distributed Memory Multicomputers .	20
2.6	Gang Scheduling	23
2.7	Process Control	25
2.8	Dynamic Partitioning	27
3	The Hierarchical Model for Task Scheduling	30
3.1	Description of HTQ	31
3.1.1	Job and Task Transfer Process	32
3.1.2	Granularity of Transfers	35
3.1.3	Minimization of Message Passing	37
3.1.4	Implementation Methodology	38
3.2	The System Model	39
3.3	Workload Parameters	41
3.3.1	Ideal Workload Considerations	41
3.3.2	Realistic Workload Considerations	42
3.3.3	The Maximum Parallelism – p	43
3.3.4	The Work Imbalance – ϕ	43
3.3.5	The Essential Work – W	43
3.3.6	The Communication and Congestion Delays – β	44
3.3.7	The Parallelism Overhead – α	44
3.4	Performance Evaluation	46
3.5	Simulation Model for Existing Policies	47
3.5.1	Modified RR_{job} – Time Sharing	48

3.5.2	Modified Adaptive Partitioning – Space Sharing	49
3.6	Validation of the Simulation Process	51
4	Performance Evaluations Under Ideal Workloads	52
4.1	Characteristics of MAP2, RRTS, and HSP	53
4.2	Relative Performance under Workload Imbalance – 50-50	55
4.3	Relative Performance under Workload Imbalance – 50-25	57
4.4	Relative Performance under Workload Imbalance – 50-75	58
4.5	Relative Performance under Varying Job Service Demand	60
4.6	A Summary of the Relative Performance	63
5	Performance Evaluation for Realistic Workload Models	65
5.1	Relative Performance under Workload WK1	66
5.2	Relative Performance under Workload WK2	67
5.3	Relative Performance under Workload WK3	69
5.4	Relative Performance under Workload Combinations	69
5.5	Considerations for High Variance in Service Demands	72
5.6	Considerations for Parallelism Variation	73
5.7	Summary of the Relative Performance under Real Loads	76
6	Conclusions	78
6.1	Epilogue	78
6.2	Contributions of this Thesis	79
6.3	What Lies Ahead?	82
	References	84

List of Tables

- 2.1 Partition sizes and priorities of the types of jobs to be assigned the partitions. 22
- 2.2 Job types and the type of partitions that can be allotted. 22
- 3.1 Quantum lengths and context switch times of some commercial systems 40
- 3.2 Load and system parameters as used in our experiments. 48
- 4.1 Example of service time divisions and corresponding utilization 60

List of Figures

1.1	A typical distributed memory multicomputer system. P stands for processor and M stands for memory	2
3.1	A logical view of hierarchical task queue organization with $P = 64$, $B = 4$, and $L = 3$	31
3.2	Job and task transfer mechanisms under HSP. Job indices in italics indicate them being transfered.	36
3.3	Application structure as represented by the three workloads WK1, WK2, and WK3	46
4.1	Performance of the three policies under ideal workload	54
4.2	Performance of the three policies under 50-50 distribution (service time $CV = 10$)	56
4.3	Performance of the three policies under 50-25 distribution (service demand $CV = 10$)	57
4.4	Performance of the three policies under 50-75 distribution (service demand $CV = 10$)	59
4.5	Performance of the three policies under 50-50 distribution at low service time $CV (= 1)$	61
4.6	Performance of the three policies under 50-50 distribution at high service time $CV (= 15)$	62

5.1	Relative performance of the three schemes under WK1 (Matrix Multiplication) with overheads	66
5.2	Relative performance of the three schemes under WK2 (Gravitation Dynamics) with overheads	68
5.3	Relative performance of the three schemes under WK3 (Mean Value Analysis) with overheads	70
5.4	Relative performance of the three schemes under WK4 (workload combination, $CV = 10$)	71
5.5	Response time versus utilization for high variance in service demand under WK4 ($CV = 15$)	72
5.6	Effect of variation of maximum parallelism at low loads under WK4 .	74
5.7	Effect of variation of maximum parallelism at high loads under WK4	76

Chapter 1

Introduction

In the last few years tremendous improvements have taken place along two dimensions in the realm of large scale message passing multiprocessors such as Cray T3E, Intel-TFLOPS, NCUBE 3, etc. First, the systems are becoming increasingly accessible to a large and growing number of users. For instance, many of the sites housing these systems such as Ames Laboratory, Argonne National Laboratory, The Maui High Performance Computing Center, etc. are providing remote accesses to researchers and developers. Second, the nature of applications being addressed is becoming wide and varied. For example, numerous researchers and developers are actively involved in such compute intensive scientific applications as data mining/warehousing, multidimensional turbulence, crystal structure prediction, molecular dynamics visualization, parallel performance measurements, parallel application debugging, high performance graphics, etc. In the face of these developments, it becomes imperative to manage the relevant resources effectively and efficiently so as to optimize the performance of these systems. The method of managing resources is called *scheduling*. Appropriate management of one of the resources, namely, the available processing capacity in multicomputer systems is the focus of interest of this thesis. More specifically, we explore the problem of job and task scheduling in distributed memory multicomputers.

Figure 1.1: A typical distributed memory multicomputer system. P stands for processor and M stands for memory

1.1 Distributed Memory Multicomputers

As the name implies, *distributed memory multicomputers* (DMMs) or *massively parallel processors* (MPPs) constitute a family of very large scale multiprocessor systems composed of physically separate *nodes*. Each node is a computer by itself in the traditional sense.¹ Figure 1.1 is a schematic illustration of such a general purpose system.

Typically these systems are composed of hundreds or thousands of nodes. A number of commercial DMMs exist including CM5, Cray T3E, Fujitsu AP3000, IBM SP2, Intel Paragon, nCUBE 3S, etc. All these systems are architecturally characterized by distribution of memory and control. This allows different tasks of a single application to be running on different processors or even multifarious applications to be running on the system at the same time. Hence, these systems tend

¹We use the terms multiprocessor, multicomputer, or parallel system interchangeably to mean a system having a number of these computers connected by an interconnection network.

to be classified as Multiple Instruction Multiple Data (MIMD) according to Flynn's taxonomy [Fly66]. Each task or application uses local data within a node. Data is otherwise shared through explicit message passing through an interconnection network. Although applications running on MPPs synchronize through messages, they are asynchronous computers and represent the most general form of parallel processing.

1.2 The Problem of Task Scheduling

Each application being executed on MPPs could be viewed as being composed of a single task or a collection of tasks which may run serially or in parallel. In the most general sense, the scheduling problem assumes a set of *resources* and a set of *consumers* serviced by these resources according to certain policy. Depending on the nature of and the constraints on the consumers and the resources, the problem is to find an efficient method for managing the access to and the use of the resources by various customers to optimize some desired performance measure [RLA94]. General solutions to this problem are NP-Complete as well as several restricted cases [GJ79]. A number of recent theoretical studies have tried to solve this problem in a restricted sense [HL90]. In addition several complexity studies have been undertaken to study this problem under a restricted sense [HvdVV92, HLV92]. In an attempt to solve the problem in the general sense, several researchers have proposed heuristics as well as approximations which do not, however, guarantee an optimal solution. However they try to find near optimal solutions most of the times [BKN95].

The *task scheduling problem*² can be loosely defined as a methodology to devise an efficient mechanism to map the tasks of an application onto the processors in such a fashion as to optimize some performance measures. Typical performance criteria are minimization of the expected response time of each application, maximization of the average system utilization, robustness of the scheduling algorithm under various

²We would like to point out that all references to tasks are equally applicable to jobs (or applications or processes) unless otherwise specifically stated.

constraints, etc. These also constitute the *performance metrics*, on the basis of which one could compare various scheduling policies. It is often the case that numerous parameters affect the performance of a scheduling discipline. These could be due to one or more of the following: the underlying architecture, the inherent characteristics of the policies themselves, certain special requirements of the users, characteristics of the nature of applications being executed on the system, and so on.

1.3 Scheduling Road-map

All scheduling schemes for multiprocessors have been heavily influenced by the techniques employed in uni-processor systems. However, presence of multiple processors adds complexity to the process of scheduling in the case of parallel processing systems. Traditionally scheduling schemes for MPPs have belonged to one of either *space sharing* or *time sharing*, reflecting the spatial and temporal properties during execution. Based on the extent of these properties, space sharing policies are further divided into static, semi-static, and dynamic schemes.

Space sharing policies are those in which the executing tasks of a job tend to be scheduled on the same processors until completion. This approach comes in a natural manner as the multicomputers are divided into “partitions” consisting of different number of processors. Typically each partition caters to one job at a time. This implies that a partition, once allocated, is retained by the job till its completion. The primary advantages with this scheme are the ease in implementation, minimal management of resources, cost effectiveness through fewer cache corruptions, etc. These schemes essentially perform very well when the number of tasks closely matches the partition size, defined as the number of processors in a partition. In addition, task synchronization tends to be simple and cost efficient. The disadvantage is that if some tasks of the application finish processing before others, then there is scope for *processor fragmentation*, that is, groups of processors idle within a partition while some jobs may be queued up due to lack of free processors. The severity is directly related to the number of idling processors. In extreme cases,

if an application is characterized by long sequential phases, then all but one of the processors will idle during these periods. As an example, consider a job J being allocated a partition of 256 processors. If its total execution time is 1000 units and it had 500 units of serial phase, then 255 processors would be idle for 500 units irrespective of other jobs being queued up.

Timesharing schemes, on the other hand, tend to allocate sets of processors to jobs with the passage of time. Quantitatively, anywhere between one to all processors in the system could be allocated to a job during its execution. In the absence of priorities, the processors are uniformly shared by multiple jobs over time. The allocation is completely arbitrary, and thus tasks could be scheduled on different processors at different times. This approach overcomes the fragmentation problem naturally. Even if one processor is allocated for more than one task then the available processing power is shared uniformly amongst the tasks. In addition, it tends to provide matching response times for all jobs. For example, let us assume a system for which the quantum size is one time unit. If a task t_a with 1 unit of cumulative demand is concurrently scheduled with another task t_b with demand 100 units, then assuming t_b was scheduled first, the response time of t_a would be 2 time units plus the overhead in context switching and house keeping operations. Under space sharing with fragmentation, t_a would have accumulated a response time of about 101 units. The superiority of time sharing is attributed to "fairness" in scheduling. Apart from this, timesharing tends to cater to varying parallelism of different applications, defined as the number of tasks constituting an application. However there are some disadvantages associated with these policies. They are known to be very sensitive to the quantum length during multiprogramming. This is the length of time a task is executed before being context switched. In addition, if tasks are accessed and put back from a centralized queue, then at high loads, a non-negligible amount of time could be spent in this process in terms of reloading the corrupted caches and translation look aside buffers.

Based on the lessons learnt thus far through various analyses and implementations it has been concluded that space sharing with ideal division of processor

power amongst various jobs is known to perform the best [SST93, Sev94]. Given the dynamics of large scale systems and corresponding applications, this ideal can be reached through a combination of space and time sharing, thus giving rise to a new class of scheduling schemes which are “hybrid” in nature. This approach has been explored in a number of recent studies in different fashions. Instances of this are provided in [FR90, SST93, DC95]. Generally, in such systems, a task once scheduled onto a processor is allowed to run solely on it, amounting to space sharing. However, when more than one task is assigned to the same processor, it is timeshared equally amongst them, akin to timesharing. Variations to this scheme are introduced in the manner in which the tasks are dispersed as well as scheduled. For instance, lightly loaded processors could be assigned more tasks than heavily loaded counterparts, and so on.

Static, semi-static, and dynamic cases occur depending on the manner in which the processors are utilized. The mechanism is also directly influenced by the system configurations. Static policies are the ones in which the processor set assigned to a job remains fixed until its completion. This number is decided, and dependent, as a characteristic of the underlying hardware system structure. In semi-static policies, the number of processors assigned to a job is decided at schedule time usually taking the number of queued jobs and/or the number of scheduled jobs into account. In certain schemes, the number of processors assigned during the lifetime of a job varies. Such schemes are called dynamic. The advantages and disadvantages of these schemes are closely related to the previous classification and hence applicable here as well.

1.4 Scheduler or Task Queue Organization

In almost all computer systems, new jobs tend to be queued in a single (job) queue in some manner such as first-in-first-out (FIFO), or shortest-job-first (SJF), etc. Whenever resources are free the scheduler removes jobs from the front of the queue for scheduling. In some cases partially executed jobs are put back into the queue

while in other cases they tend to be enqueued in the processors' local queue for future processing. In this case, presence of a single job queue deems the task queue organization to be called as *centralized*. Employment of a single queue gives rise to various advantages. For example, as the scheduler has a global view of the system, processors could be allocated to tasks only when they become idle thus leading to perfect load sharing. In addition, the management of resources becomes simple and so would programming such a system. However, there are also severe disadvantages. Whenever a large number of jobs reside in the system to be scheduled, the single queue becomes a bottleneck. This could lead to idling of resources. A second disadvantage arises in relation to the scale. Whenever systems are scaled up with the introduction of additional processors, the bottleneck problem worsens considerably – it has to be noted that such systems are part of one of the foci of this thesis. In addition, they are a single source of fault.

At this point one is lead to assume that employment of multi-task queues would alleviate the above problems. In fact, the twin problems of access contention and scalability are overcome when the *distributed* task queue organization is considered. Here one or a small set of processors are served by a single task queue. Arriving jobs could be put at the end of any queue based upon some distribution strategy. By this scheme all the problems of the centralized scheme are overcome. At the same time, due to the inadequacy of having a global view of the system by any task queue this would eventually lead to load imbalance in the system. In this case, some task queues may be heavily used while others would idle at the same time.

In the face of advantages and disadvantages of the above two policies, it seems to lead to a no-win situation. The ideal is to wish for having the advantages of the above two policies while at the same time overcoming their individual disadvantages. The *hierarchical task queue* (HTQ) organization as proposed initially by Dandamudi and Cheng [DC95] provides a means to achieve this. Here the task queues are organized in the form of a tree – balanced tree for the sake of simplicity – with the processors forming the leaf nodes.

Unlike centralized schemes, and similar to the distributed organization, presence

of multiple task queues overcomes the disadvantages of the former. At the same time, due to multiple levels of control, it offers more flexibility than the distributed organization. This is because load is adaptively shared amongst the task queues based on “need”. That is, tasks are moved from higher levels of the tree to the lower levels depending on the requests from the latter. This means that load can be shared “almost” evenly amongst the task queues. In retrospective, the hierarchical task queue organization overcomes the problems of both the centralized and distributed scheduler organizations while at the same time exhibiting the advantages of the same. Use of HTQ to design novel scheduling schemes based on, both, space- and time sharing is the focus of this thesis.

1.5 Hierarchical Scheduling Policy (HSP)

As mentioned before, a combination of space partitioning and time sharing is beneficial in making scheduling decisions in large scale systems [SST93, AD96]. This is our motivation to propose a novel scheduling scheme called *hierarchical scheduling policy* (HSP) for MPPs. This scheme is based on HTQ as discussed briefly in the previous section. In this scheme, jobs and tasks are adaptively transferred from the root queue to the processors based on demand. We define the number of processors reachable from a task queue as its *partition-reach*. During the process of scheduling, tasks of a job present in a task queue are constrained to be executed only in its partition-reach. This is similar to space sharing. As more than one task could be scheduled on a processor, it employs multiprogramming to allocate equal power to the tasks. This is exactly like time sharing. As multiprogramming is employed within a partition, it is possible to schedule tasks in a near optimal manner.

1.6 Research Considerations

In this section we describe our considerations such as the system model and workload characteristics in regards to HTQ and HSP.

1.6.1 The System Model

Performance comparisons amongst the scheduling policies could be obtained in one or more of the following means: analyses, simulation, and experimentation. In this work, we chose to compare the HSP with the adaptive space sharing policy as propounded by Rosti *et. al.* [RSS⁺94], and a modified version of job based round-robin policy as proposed by Leutenegger and Vernon in [LV90]. The main reason for adopting a simulation based experiment was that it is extremely hard to properly analyze the policies theoretically. Also the cost for implementing them is prohibitively high. Following is a brief description of some of the important system parameters we considered for our experiments.

For all the experiments, we modeled a 64-node homogeneous message passing system. Each node in the system is composed of a computational as well as a communication element. All data sharing amongst the nodes is by explicit message passing. As per the existing system characteristics, the system allows computation and communication to overlap. Task queues are connected in the form of a balanced tree. Each task queue is uniquely mapped onto a processor. All queues are assumed to be finite. Many other relevant parameters such as quantum length, context switch time, etc. are based on the values found in actual commercial systems.

1.6.2 Workload Models

It has been proved conclusively that the performance of any scheduling policy is interlinked to the type of jobs that have to be scheduled [Sev94, FN95, JPF⁺97]. Fortunately a few models of parallel applications comprehensively represent the characteristics of applications usually encountered in the supercomputing environments. For this thesis, we have chosen to model the following three models representative of almost all parallel applications as detailed in [Wu93].

1. The highly parallel matrix multiplication algorithm that exhibits a basic fork and join structure. Here a job is assumed to be broken down into N identical

tasks which could be executed independently. Upon processing, all the tasks synchronize at a common point.

2. The parallel version of the mean value analysis solution for queuing networks with two classes composed of N customers each. In this case, a strict precedence order is maintained amongst the tasks, that is, any task could execute only upon the processing of all its preceding tasks.
3. The Barnes and Hut clustering algorithm characterized by five distinct phases of variable fork and join structure in the number of tasks spawned during each forking structure.

1.7 Contributions and Organization of the Thesis

This thesis proposes a novel scheduling discipline for distributed memory multi-computers based on HTQ. So far, almost all commercial and research based systems employed either centralized or distributed task schedulers. Only in recent times the hierarchical task queue organization has been researched and implemented [DC95, AD96, FR96]. However, it must be noted that HTQ has been researched only in the realm of shared memory systems. Through the process of a detailed simulation study, it is shown that this organization performs appreciably better than pure space partitioning and time sharing especially at medium to heavy system loads.

On a more general note, most of the previous simulation studies have failed to take into account the fact that speed-ups actually degrade rather than just level off when an excessive number of processors is applied to a parallel application. This work considers the effect of the degradation in a more realistic and explicit manner based on a number of recent studies on workload characterization [Sev89, Sev94, FN95, PS95, Sev96].

The rest of the thesis is structured as follows.

Chapter 2 provides a detailed overview of the work carried out in job/task scheduling for distributed memory systems. It describes prior results in terms of

simulation as well as experimental evaluations.

An exhaustive description of the hierarchical task queue model is described in Chapter 3. Detailed descriptions of the job and task transfer process as part of the hierarchical scheduling policy is brought out in this chapter. Also, the simulation experiment setup together with the system and workload models are explained in this chapter.

Chapter 4 presents the simulation results in reference to the performance of the HSP against pure space and time sharing policies for an ideal workload model wherein parallelism overheads and communication details are not considered explicitly. The motivation to present this has been to evaluate HSP and understand its behavior distinctly, without any unusual influences of the workloads.

Results for the more realistic workload models are presented in Chapter 5. Herein, we take into account explicit overheads for the various workloads.

Finally, Chapter 6 draws out the conclusions from this study. Also, the contributions of this thesis to the area of multiprocessor operating systems is brought into focus. The chapter ends with a summary of future research directions.

Chapter 2

Previous Work

In this chapter, we review some of the literature that is relevant to this thesis. Our choice of the research studies has been motivated by principles and techniques that can have strong influences on the performance of distributed memory multicomputer systems.

As with any nascent research field, some entities have been defined in numerous ways. This can lead to a great deal of misinterpretation. We, therefore, would like to define the appropriate terms in the following section so as to alleviate this confusion in the context of this thesis.

2.1 Relevant Definitions

All references to jobs are applicable to tasks unless specifically stated.

Adaptive/Semi-Static Scheduling – Scheduling schemes wherein the decision to allocate processors to a job is made at schedule time.

Asymptotic Bandwidth[XH96] – The maximum bandwidth achievable when the message length approaches infinity.

Average Response Time – The sum of all job response times divided by the number of completed jobs.

Distributed Memory Multicomputer/Message Passing System/Massively Parallel Processors – A parallel system consisting several *nodes*, each of which is

composed of a CPU and memory, connected together by a dedicated interconnection network.

Dynamic Scheduling – Scheduling Schemes in which number of processors allocated to a job varies over its course of execution.

Execution Time – The time a job was being actually executed on a processor.

Multiprocessor Scheduling – A method to devise an efficient mechanism to map the threads of control of a job onto the processors in such a fashion as to optimize some performance measures.

Partition Reach – The number of processors reachable from a node in the hierarchy.

Partition Size – The number of processors in a partition.

Processing Time – Length of time since a job is first scheduled onto a processor till it completed execution.

Response Time – The time elapsed from the moment a job was submitted to the system till its execution.

Scheduling[Fei94] – The activity which decides as to which thread of control gets to run on the CPU.

Single Level Scheduling – A scheduling scheme which combines the issues of processor allocation to a job and scheduling threads of control onto these processors.

Space Sharing – A scheduling methodology in which processors in the system are exclusively partitioned. Jobs are scheduled on specific partitions till completion with, at most, one job per partition (usually).

Static Scheduling – Scheduling schemes wherein processors, once assigned, are held by a job until completion. The allocation is done at the time the job enters the system.

System Utilization – The percentage of time any processor in the system was busy.

Two Level Scheduling – A scheduling scheme wherein the operating system allocates processors to a job and the application itself (or a language runtime library) schedules threads of control onto these processors.

Time Sharing – A scheduling methodology in which several jobs share the set of processors in the system without any exclusion.

2.2 Coscheduling

Ousterhout proposed the fundamental method to schedule processes on multiprocessor systems [Ous82]. The motivation in developing the technique known as *coscheduling* was to design scheduling schemes in cases wherein tasks of an application communicate extensively. For this, two methods were proposed that were applicable to short term and long term aspects of scheduling. As is usually the case in multiprocessor systems, any task that is blocked due to some event is context switched in order to activate some other task. This task is subsequently activated whenever the event occurs. The cost of context switch and cache reloads are very high. In order to overcome the inefficiency due to these high costs, the waiting time of a task before the swap is divided into two phases, namely, *pause* and *block*. During the pause phase, the task idles for a predetermined time. If during this time, the event occurs, then it is reactivated. This operation involves less expense than a context switch. If, on the other hand, the event does not occur, then the task enters the block phase whereupon it is context switched. During this time, the process would idle. That is, each task spins for some time (idle state) before being blocked (swapped). Note that this kind of a short term scheduling rule is applicable when communication occurs on a fine grain. That is, there is a high probability of the event occurring in a short time that the wait has begun.

With regards to scheduling on a long term basis, it was felt that the communication patterns of the tasks has to be taken into consideration. A *process working set* is recognized as a collection of processes (or tasks, as in our terminology) that has to be scheduled simultaneously (hence the name coscheduling) in order for the control flow of the parallel program. If this is not accomplished, then swapping of any one process in this set causes other peer processes to be context switched. This is because, the latter processes depend on the former for further execution. Ousterhout

suggested two steps to eliminate this kind of process thrashing. First, to determine the process working sets; second, allocating processors and scheduling the process working sets on them. This ensures that all interacting process are available when needed.

Due to the inherent difficulty in recognizing the process working sets, it was assumed to be statically defined by the programmer. Three algorithms are proposed to execute coscheduling largely from a process space viewpoint and study the resulting benefits. The first of the these, known as the Matrix Method is the simplest of all and performs well. In this case the process slots are organized as a matrix with Q rows and P columns. Each column contains the process slots of one processor while each row contains a process from each processor. The allocation scheme is carried out as follows. Each row is checked in succession till one is found that has enough slots so as to schedule all the process of a single application. Based on this allotment, a round-robin mechanism is used to multiplex the system between the different rows of the matrix. For instance, during the first quantum all processes in row 0 are allotted highest execution priority, thereby coscheduling all processes in that row. In the next time slice processes in row 1 are scheduled and so on until all applications have been scheduled. The process is then repeated over again.

During the scheduling of any row it may be the case that not all processors will be busy because of the application's parallelism profile at that instance. In this case, the idle processors scan their columns to schedule the next runnable process. This is referred to as an *alternate*. This mechanism is applicable independently amongst the processors and no attempt is made whatsoever to maximize scheduling. Simulation results showed that selection of alternates did not degrade the extent of coscheduling. In fact, due to this mechanism the access contention for centralized schedulers in systems employing them was reduced as global scheduling occurs only at time slice boundaries while local scheduling could be carried out at the kernel level, in two level scheduling systems.

Drawbacks of this method include possibility of many processors idling during each quantum. At the same time, no special considerations was given to those

applications for which there were more tasks than the number of allotted processors. In this case, multiple tasks are scheduled on some processors within a quantum. Finally, it is possible for coscheduling to be completely missed due to alternates.

The second scheme known as the Continuous algorithm overcame these deficiencies by using a windowing technique.¹ In this case the allocation is done as follows. A window of width P is initially placed at the left end of the process sequence if the left most process slot is empty or moved right till it coincides with the first free slot. During each quantum, all tasks within a window are scheduled. The window is appropriately moved after each quantum thereby providing a 'circular' way of sharing the processors. If during scheduling there are empty slots, then the alternate method is used to fill up the fragments. This method however is fraught with one major problem in that it could lead to situations wherein processes of an application may be executed in so many fragments that it may lead to unfair processor sharing amongst the applications. In order to overcome this a third method termed as Undivided was proposed that was exactly like the previous one except that it allocated processors such that no application parts were executed over two or more quanta. That is, all tasks of an application had to be scheduled within at most two time slices.

Amongst all the three methods, simulation results indicated that the Matrix method surprisingly achieved the best performance, sometimes outperforming the others by 10 to 20%.

2.3 Semi-Static Policies

In an attempt to remove some of the confusion in the terminology, we prefer not to use the term *adaptive* in this case, as expounded by some of the researchers, and instead call them *semi-static* policies in tune with Majumdar et. al. [MEB88]. It is

¹For simplicity, processes slot 0 on processor 0 was assumed to be in the leftmost position of the window.

our view that the term adaptive is in harmony with the *dynamic* policies which are described in Section 2.4.

Rosti et. al. [RSS⁺94] evaluate a set of semi-static policies in order to study their behavior under various workloads. Their performance metric is *power*, which is defined as the ratio of system throughput to response time. These policies are based on partitioning the available set of processors in space sharing fashion. The “adaptive” nature of the policies adjusts to various workloads by computing the partition size of each job at schedule time. On each such partition a program exclusively runs until completion.

Due to hardware constraints such as system architecture etc., and software constraints such as a job’s maximal parallelism and so on, there is a potential problem of fragmentation in static space sharing policies. In their paper, the authors try to avoid the software generated fragmentation problem by attempting to generate equal sized partitions while adapting to transient workload behavior. For this, they propose five policies named AP1 , AP2, AP3, AP4, and AP5. Since the ideal of assigning equal sized partitions to all the jobs under consideration cannot be achieved in practice, these policies attempt to reach this goal by splitting or merging free partitions as and when jobs arrive or depart. In fact, the policies are characterized by this split/merge mechanism. The algorithm for the basic semi-static policy is given below:

```

    TargetSize ← compute_target_size() /* Policy specific */
    while (QueueLength > 0) ∧ (FreeProcessors > TargetSize) do
        FreeProcessors = FreeProcessors - TargetSize
        QueueLength = QueueLength - 1
        schedule_job(TargetSize)
    od

```

Here, *TargetSize* denotes the number of processors allocated to a job, *QueueLength* refers to the number of jobs waiting in the ready queue, and *FreeProcessors* are the total free processors in the system. *compute_target_size()* and *schedule_job()* are functions that compute the target size as per some mechanism and schedule a

job on processor allocation respectively. One mechanism of target size calculation is by AP2 and it is the policy of interest to us as its performance is comparable to the best policies while, at the same time, being less complex in implementing the split and merge operations. In this case the partition size is computed as follows:

$$\text{available_target_size} \leftarrow \max \left(1, \left\lfloor \frac{\text{TotalProcessors}}{\text{QueueLength}+1} + 0.5 \right\rfloor \right)$$

From the basic algorithm, we notice that AP2 tends toward premature queuing. For example, when a single job arrives at an idle system, it is allocated one half of the processors and then scheduled. On the arrival of another job, the policy enqueues it even though only half the processors are assigned to the first job due to the ($\text{FreeProcessors} > \text{TargetSize}$) condition. Also, any scheduling decision is made by considering only the queued jobs. In order to obviate this problem, we modified the basic policy by taking into account the total number of jobs in the system, which is defined as the sum of queued jobs and the scheduled jobs. This modification is shown below.

$$\text{target_size} \leftarrow \max \left(1, \left\lfloor \frac{\text{TotalProcessors}}{\text{QueueLength} + \text{Fraction} * (\text{ScheduledJobs}) + 1} + 0.5 \right\rfloor \right)$$

The resulting adaptive policy, which is a modification of the basic one, shall henceforth be referred to as MAP2, denoting Modified AP2, is given below. It is to be noted here that we also provide the potential to schedule the jobs if at least one processor is free.

```

TargetSize ← compute_target_size() /* As above */
while (QueueLength > 0) ∧ (FreeProcessors ≥ TargetSize) do
    FreeProcessors = FreeProcessors - TargetSize
    QueueLength = QueueLength - 1
    schedule_job(TargetSize)
od

```

The motivations behind the modification are two fold. One, the goal of equal partition for each job is better realized by taking all the jobs into account. Two, it

provides a good heuristic during processor allocations.² This modification leads to better performance than AP2, and thus, amounts to optimizing AP2.

2.4 Adaptive Multiprogrammed Sharing Policies

In Section 2.3, we briefly discussed the relative advantages and disadvantages of the space sharing policies. As part of this, we noted that using space sharing policies under some conditions could lead to extensive fragmentation and performance degradation. Setia *et. al.* [SST93, SST94] perceived that the degradation in performance could be reduced by time sharing multiple jobs within fixed partitions. They adopt the means of analysis (mostly) and some simulation to support their claim. They examine different scheduling methods for multiple parallel programs in the context of distributed memory message passing environments. For this, they proposed a class of two-level *multiprogrammed partitioning* (MP) policies which can be thought to occur at both, the kernel and the kernel/application levels. They call these the *even processor allocation* policies.

A parameter Z is associated with each policy in the class corresponding to the number of processors allocated to it. Due to this, tasks of a new job are classified into Z distinct groups with $\frac{N}{Z}$ tasks in each group, where N denotes the number of tasks of the job. The scheduler then schedules each group onto a processor which has the shortest queue length. Once assigned in this manner, each processor executes the tasks in its local queue in an independent manner. For optimal utilization of processors as well as catering to applications workload changes, the scheduler allocates more processors to a job when the load is light and continuously restricts the number with increasing load. On further analysis of this mechanism, they found that the allocation policy with $Z = P$, P being the number of processors in the system, achieved the best performance at light system loads. At higher load levels, it was noticed that smaller values of Z provided the best performance.

²We have noticed appreciable changes in the performance of the class of MAP2 policies obtained for various values of *fraction*. The results of this modification will be provided after due finalizations.

The MP class of policies tend to perform better than static space sharing policies in situations wherein the scheduled jobs are inefficient. That is, the tasks of the job running on different processors tend to spend significant amounts of time waiting for messages or data from peer tasks. This arises because of uneven distribution of load within the partitions, which in turn leads to large processor idle times.

The MP policies are *adaptive* in that the partition size, and the degree of multi-programming can change in response to changes in the workload. Also, they *always* attempt to provide equal sized partitions so as to perform better than all other adaptive and semi static policies for workloads consisting of jobs with the same maximum parallelism [RSS⁺94]. As a variation of the basic scheme, they consider a slight modification to the basic scheme wherein tasks are context switched only at instances when they attempt to communicate with peers.³

From the analysis and simulation experiments they conclude that MP policies outperform static space sharing policies at medium to heavy loads. This is especially true for programs with widely varying maximum parallelism. Also, they claim that these are more robust than the semi-static policies of Rosti et. al [RSS⁺94] for a wide range of workloads. At the same time they note that these policies tend to perform favorably well for workloads with almost same maximal parallelism. Undesirable effects due to MP policies were observed for varying quantum sizes.

2.5 Multiprogrammed Policies for Distributed Memory Multicomputers

Naik *et. al* [NSS95] analyzed three general class of space sharing policies under a workload representative of large-scale scientific computing. At the same time they proposed certain objectives for adaptive and dynamic schemes which make use of user supplied information about the resource requirements of submitted jobs. This is a distinctive feature of proposal and analyses of recent scheduling schemes. The

³For this they assume the blocking mode of communication – ironically their system model considers processors together with specialized communication co-processors.

analyzed policies were Multi-class Fixed Partitioning (MFP), Multi-class Adaptive Partitioning (MAP), and Dynamic Partitioning (DP).

MFP is motivated by reduced response times in uniprocessor systems wherein smallest-demand-first principle is followed and similar advantages on multiprocessor systems performance as well. Under this, jobs tend to be divided into different classes based on their processor requirement. Thus every job is classified to be in one of *small*, *medium*, or *large* pools. A fixed amount of system resources is allocated to each pool statically (or until further reconfiguration). If multiple jobs are allotted the same partition, then they are time shared with other tasks. It is possible for multiple partitions of a class to exist. In addition, there is a flexibility of processors belonging to one partition executing jobs of another partition under special circumstances. However, jobs belonging to the same class assume greater priority than those which are foreign to the partition. Moreover this priority is non-preemptive. In addition, a definite sequence of priority is defined wherein small jobs have non-preemptive priority over medium jobs for large partitions in the absence of large jobs, while at the same time, larger jobs cannot acquire the smaller partitions.

Each of the job classes is served by a FCFS queue. For the case of the large job class, an arriving job is allocated a large partition if available, otherwise it is simply enqueued for subsequent allocation. If in the case there are no large jobs in the system and there were small and medium jobs, then an attempt is made to allocate all small jobs before any medium job.

In the case of a medium job arriving with no partitions of the medium class, it is enqueued for subsequent scheduling. In the meantime, if a partition of the large class becomes free, and there are no waiting large and small jobs, then it is scheduled on the allocated partition. If large and small jobs were enqueued and no medium jobs then only the small jobs are allocated the medium class partitions.

Small jobs are allocated small partitions initially. If unavailable, and in the absence of, both, medium and large jobs an attempt is made to allocate the medium and large partitions in that order. Neither the medium nor the large jobs are allocated small partitions. The two dimensions of prioritized allotment of partitions and

Partition	Preference Priority of Jobs
large	large $\gg$ small $\gg$ medium
medium	medium $\gg$ small
small	small

Table 2.1: Partition sizes and priorities of the types of jobs to be assigned the partitions.

Jobs	Alloted Partitions
large	large
medium	medium; large (if no small jobs)
small	small; medium (if no medium jobs); large (if no large jobs)

Table 2.2: Job types and the type of partitions that can be alloted.

job preferences for partitions are tabulated in Table 2.1 and Table 2.2 respectively.

The working behavior of MAP is analogous to MFP. However as the name suggests, the processors comprising a pool are adaptively reconfigured based on system load. This knowledge is gauged from the lengths of the job queues maintained by the scheduler. This scheme is further explained in detail in Section 2.3.

Unlike MFP and MAP, there is no concept of preallocated partitions under DP. This does not however preclude some per-class control parameters that could be present and used during scheduling decisions such as minimum sizes of the partitions for the small, medium and large job classification. Another difference pertains to *dynamically* changing the medium and large partition sizes during execution. Jobs are scheduled from a single global FCFS queue. The system scheduler maintains an additional list of running jobs eligible for reconfiguration. The reconfiguration rate is determined by a time interval in order to prevent a form of reconfiguration trashing. When under no constraints DP tries to allocate a partition to a new job whose size is close to the average size of currently alloted partitions including itself.

Under this an attempt is made to allocate $\frac{P}{N}$ processors to each partition where N denotes total number of jobs in the system. This is so attempted when no job is allocated more processors than it has requested and no less processors than the minimum partition size for the class it belongs to.

As mentioned earlier, these policies were tested under a workload representative of a large computing center. The workload was generated by analyzing various observed parameters such as computation, communication, data dependencies, memory usage, data partitioning, load balancing and scalability during execution of parallel applications [NSS95]. The execution times based on the analyses were validated against measurements and simulations as well. A sample of applications that consisted of a mix of varying application characteristics was chosen so as to illuminate on the workloads large variance in requirements.

Analyses based on observations of the behavior of the policies indicate that it is beneficial to dynamically reduce processor allocations with increasing load up to a minimum number of processors especially under workload mixes. A knowledge of the differences in job execution times is very useful so as to balance the benefits of performance of a particular job class and the loss for the other classes. In addition, capability of a scheduling policy to adapt to different arrival characteristics is crucial to obtaining the advantages mentioned thus far. Finally, the authors note that amongst the policies, DP is capable to realizing the said features as it brings out the best results from the system and user perspectives.

2.6 Gang Scheduling

Gang scheduling was proposed by Feitelson and Rudolph [FR90]. This strikes a middle ground between traditional coscheduling [Ous82], and space sharing [RSS⁺94, SST94] in the following manner. Unlike coscheduling, gang scheduling provides a mechanism for coordinated scheduling of alternates. That is, suppose N_{free} nodes are free during a quantum since the tasks scheduled on them have blocked, then the placement algorithm will attempt to find a job with N_{free} or fewer tasks to run on

these N_{free} nodes. This implies scheduling various jobs within a partition at the same time.

Specifically under this scheme, a set of related threads (or tasks as in our context) are scheduled on a parallel machine so that all the threads in a gang execute simultaneously on a subset of processors. It is to be emphasized that even though a process may contain more threads than the number of processors, during scheduling there is a strict one-to-one mapping of the threads to the processor. That is, no gang is permitted to have more threads than the number of allocated processors. However, more than one job could be assigned the same set of processors and during these times, the jobs are time sliced if they cannot be 'packed'⁴ within a partition. Thus, gang scheduling combines the features of space sharing and time slicing. As the schedulable entities are gangs of threads, there is no concept of a job during scheduling. The input to the system is a stream of requests to schedule new gangs. Each request has a size (the number of threads) and is characterized by the duration of its execution on a dedicated machine (this is not known in advance). It is assumed that all the threads within a same gang has the same execution demand.

The control during scheduling is done by the means of *distributed hierarchical control* (DHC). This is based on a hierarchy of controllers that organize the processors in a buddy fashion. A distinct controller is associated with each block of processors. The size of each block is a power of 2 and is a union of two blocks half the size. The controller at level i coordinates the activities of the 2^i processors spanned by its subtree. Controllers in lower level of the tree provide for local control while those in higher levels take care of global coordination. For optimal load balancing, lateral connections connect controllers at the same level as well. It is to be noted that the hierarchy is only logical and does not in any way require the underlying hardware to be structured in the same manner. Two basic driving forces govern the management scheme – efficient utilization of resources and fairness in resource allocation.

The hierarchy of controllers is used to map threads to processors in the following

⁴That is, the different jobs are scheduled simultaneously on the same partition.

manner. A request to map a new gang of size s originates from a thread executing on some processor. The request ascends the tree of controllers until it reaches the appropriate level for its size and then moves across to some controller that will balance the overall load. Care is taken so that jobs are handled by a controller at the level of the smallest partition greater than the processor requirement of the job. Once threads are mapped onto processors, they do not migrate to other processors and control of a gang is not passed between controllers. Upon termination of a job or scheduling of a new job, the knowledge about load changes is shared within the controllers to aid during load balancing. At all times, a controller could be in one of the three states: idle, active, or disabled. The scheduling proceeds in waves that propagate down the tree of controllers. A wave propagates from a level to the next lower due to the completion of the scheduling operations at that level. Whenever a controller is active, it is involved in scheduling all the gangs it controls. The controller transmits a command to execute the threads of the scheduled gang on the processors by passing the execute command down to its subordinate controllers and this is continued to be passed down until it reaches the appropriate processors. It is to be noted that whenever a controller is active all its subordinate controllers are disabled if the scheduled gang utilizes all the processors. In the event it does not, then selective disabling is permitted. When the lowest level of controllers complete scheduling, then a new wave is initiated from the root.

2.7 Process Control

If the number of physical processors are less than the number of tasks that have to be scheduled on them, then the performance significantly degrades. *Process control* was proposed by Tucker and Gupta in order to minimize this degradation in the context of shared memory multiprocessors [TG89]. The degradation could be due to one or more of preempting tasks that are in a spin-lock controlled critical section, unnecessary context switching, or due to processor cache corruption. The motivation of process control is based on the hypothesis that applications perform best when

the number of runnable threads is the same as the number of processors. For this, each application structure is considered to be composed of a set of threads, that is, they are considered as multi-threaded jobs. Under this, applications are made to dynamically control the number of runnable threads to match the number of processors available to them. If this is achieved, then each runnable thread can be scheduled on a distinct processor, there would be no reason for preemption, context switching would be less expensive, and finally, the cache corruption will be minimized.

As regards to limiting the number of runnable threads, the authors note that some threads could be suspended when some other threads of the same application have finished execution or are enqueued for further processing after the current time quantum. A centralized server determines the number of runnable threads for each application. To control this number in each application, the control is provided as part of the application, and at the same time making it completely transparent to the user. Thus the overall structure of the scheme is composed of a central server, and some number of processors partitioned in various ways. In order to determine the number of runnable threads present per controllable application, the server periodically calls the kernel and gets a list of all runnable processes. It then determines the number of runnable threads not belonging to controllable applications. This number is subtracted from the number of processors to determine the number of available processors each application should be using. In addition, it also makes sure that no application is provided more processors than it can possibly use. Also, each application is assigned at least one processor in order to prevent starvation.

As mentioned before, the number of threads of an application to be run is directly controlled by a process of the same application. For instance, once an application begins execution, then the parent process sends a message to the central server notifying it of the process' existence and passing its ID at the same time. The application then begins execution and periodically a thread polls the server to determine the ideal number of runnable threads. In the face of information received

from the server, if the ideal number is greater than the number of actual threads, then the thread wakes up a previously suspended thread. Threads are blocked if the ideal number was less than the actual number and placed on a ready queue. This constitutes the basic method of process control.

Process control was tested on a 16-processor shared memory system for four parallel applications namely, Fast Fourier Transform, Merge Sort, Gaussian Elimination, and Matrix Multiplication. Results were compared between the controlled and uncontrolled versions of the applications. Speedup was found to increase for both versions till the number of threads became more than the number of processors whereupon the speedup drastically decreased under the uncontrolled case. The effect was more pronounced with increasing the number of threads per application. By using process control, speedup in some cases increased by more than a factor of two with reference to those cases where it was not used.

2.8 Dynamic Partitioning

Dynamic partitioning refers to a processor allocation scheme wherein processors are assigned to applications depending upon the current workload on the system. Dussa *et. al.* [DCDP90] cite one of the early experiments in implementing dynamic partitioning policies. This is different from the DP scheme of Naik *et. al.* in that, there is no classification of jobs and processor partitions in this case. Also, the potential benefits of priorities with respect to the workload is not exploited here. In fact, these two features characterize the policies in terms of using knowledge about applications and their behavior during scheduling.

Their main motivation of Dussa *et. al.* was to gain insight into the efficacies of dynamic scheduling schemes for multiprogrammed multicomputers. They implemented and analyzed the results of a dynamic scheduling scheme on a network of eight INMOS transputers, a type of distributed memory multicomputer. Their workload consisted of two applications in which one was computation intensive while the other was characterized by extensive communication. Specifically, the applications

they considered were the N-body problem and a classic genetic algorithm search technique. Jobs consisting of many instances of the above two applications were broken into constituent tasks. Tasks in the N-body program formed a ring while those of the search could fit onto any architecture due to its highly parallel structure. In dynamic partitioning, when the multiprocessor is idle, an arriving application is given all the eight processors. When, a mix of jobs belonging to the two applications arrived, then some n processors were allocated to the search program while $8 - n$ processors were given to the other. Each node of the multiprocessor is dedicated to a specific program. This means that at no point in time would two jobs belonging to both applications be scheduled on the same processor.

In order to dynamically allocate processors to arriving jobs, it is imperative to recognize the interplay between the compiler and the operating system. Compilers are required to recognize and provide breakpoints in the programs so that processor (de)allocation decisions can be performed by the operating system at these junctures for each program. The overheads experienced due to this is referred to as *partitioning overheads*. A scheduled program can be paused at one of these points, its share of processors altered, and subsequently made to run. The performance of such a scheme is completely dependent on the optimizations incorporated at these pause-alter-run points. One important point that has to be noted here is that any partitioning activity is performed if and only if all scheduled tasks have reached the next breakpoint in their execution profile. Thus, this immediately induces varying wait times for the programs. In fact this can prove to be the Achilles heel in terms of any performance gains. Thus depending on the placement of breakpoints, any reduction in the partitioning overhead is complemented by longer waiting times.

Effectiveness of dynamic partitioning was measured through a number of sensitivity analyses with respect to various parameters such as mean arrival rate of jobs, an application's computing and communication activity, partitioning overheads, and the number of breakpoints in the applications. Based on the observed results they conclude that by employing suitable configurations when allocating processors, it is possible to enhance the performance of applications in comparison to other policies

such as first-come-first-serve. Specifically, for the configurations wherein the N-body problem was assigned preemptive priority (that is, all 8 nodes were allocated to it whenever it desired multiprocessor service) in the first case, and at least 6 processors to the genetic search in the other instance. In the first case, dynamic partition does not perform well since the preemptive priority of N-body causes extensive degradation in the performance of the search. However, in other cases, dynamic partition was shown to be useful despite high communication and repartitioning overheads. In addition, they present methods to develop suitable analytical models. By this they showed that it is possible to develop accurate analytical models for studying parallel scheduling schemes.

Chapter 3

The Hierarchical Model for Task Scheduling

Several studies have indicated that in the absence of any knowledge regarding workloads a combination of space and time sharing is beneficial for task scheduling in MPPs [TG89, SST93, SST94, DC95]. Setia et. al. [SST93] showed this in principle only in that they multiprogrammed multiple tasks within a processor partition. In addition, they have not considered various important issues such as task placement onto the processors, arrival and service characteristics of jobs, workload mixes etc. Ayachi and Dandamudi [AD96] used a similar scheduling policy as ours. However, this was applied to NUMA type of machines, whose architecture is profoundly different from message passing systems. Again, their work did not consider realistic workloads nor job mixes. In contrast to these two, we feel that ours is the first attempt at deriving a hybrid resource management structure for multiprocessor operating systems. This hybrid policy combines space partitioning and time sharing while testing this under multifarious workload characteristics which are representative of the properties of job mixes as noticed in actual high performance computing centers. This policy, referred to as hierarchical scheduling policy (HSP), is based on the hierarchical task queue organization. In the following description, we highlight HTQ, and HSP in detail together with the implementation methodology. Following this, we explain in detail the simulation environment under which HSP was tested and relative performance measurements were conducted. As part of this, we provide

Figure 3.1: A logical view of hierarchical task queue organization with $P = 64$, $B = 4$, and $L = 3$

a detailed description of the system model and the application characteristics. It is to be noted that these issues are central to any scheduling discipline.

3.1 Description of HTQ

The HTQ is logically structured in the form of a tree of schedulers or task queues as shown in Figure 3.1. Leaf nodes in the tree are the processors and the non-leaf nodes are represented by the task queues. For the sake of discussion, we assume that the tree is L levels deep with the processors residing at level L and the root perched at level zero. The number of children of any node is referred to as its *branching factor* B . Without loss of generality, we assume that all nodes have the same value of B . With reference to Figure 3.1, B of the tree is 4. The number

of processors, P , is 64. Each task queue or scheduler is mapped onto a distinct processor, and thus causes only minimal contention during operation. The nature of this mapping is general and applicable to all MPPs. In order to see that it is indeed possible, consider a MPP such as Cray T3E with $N \gg 1$ processors. The maximum number of task queues occurs when B is 2, and this equals $N - 1$. Each of these queues can be mapped onto a distinct processor with one more to spare. For higher values of B with the same number of leaf nodes, the number of schedulers are reduced even further. Each scheduler is responsible for only its children, that is, a scheduler communicates only with its parent and all children. At the same time, no parent mediates between the workings of any of its children. Also, higher level queues do not keep track of any details of its children. We define all processors reachable from a task queue to constitute its *partition-reach*. Thus higher level nodes tend to have larger partition-reach than their lower level counterparts. It is not hard to envisage that this concept induces the space partitioning mechanism. For instance, jobs residing at a task queue can only be scheduled on the processors determined by its partition-reach since there are no lateral connections between the nodes. Additionally, the goal of uniform load sharing is also possible by viewing each task queue as a central queue serving the processors in its partition-reach. One might argue about the efficacy of the aforementioned advantages due to space partitioning as HTQ can also be viewed as, and is, a distributed organization. One counter argument to this is the presence of multiple levels of control in HTQ (more importantly, the control is highly coordinated). This means that, it can be viewed as a conglomeration of centralized schedulers working in a democratic fashion. Another reason is the inherent nature of HSP as exhibited when transferring jobs and tasks, which is detailed below.

3.1.1 Job and Task Transfer Process

We now describe the scheduling procedure per-se as carried out on the basis of HSP. As mentioned in the previous section, HSP allows for combined space- and time-sharing. For this, all incoming jobs are enqueued at the root of the hierarchy. We

make two assumptions with reference to these jobs:

1. For the sake of simplicity, by a job being enqueued, we implicitly mean that the job is broken down into constituent tasks and the tasks are enqueued. It is indeed possible to enqueue the job as is and this does not have any bearing on the policy or performance whatsoever.
2. The number of processors required for each job is known *a priori*. This number is a measure of the job's maximum parallelism p_{max} . This is definitely plausible as p_{max} could be estimated approximately by either analytical means, or by repeated observations of application executions as suggested by Sevcik [Sev94].

HSP employs *self scheduling* while transferring work, that is, jobs and tasks. This implies the absence of any centralized scheduler for decision making. On the other hand, all task queues along a branch are involved in making decisions in succession. Furthermore, this is a quasi-demand driven scheme under which work is transferred based on need. At the same time, this differs very largely from any other scheme in terms of the quantity during transfers. Under this, the maximum number of tasks that can be transferred to a processor is limited by the *multiprogramming level* (MPL), while the *transfer factor* determines the number of tasks that can be moved from a parent to any one of its child task queues [DC95]. We denote the transfer factor by Tr . The quantity of work transferred is given by

$$\text{Number of tasks moved one level down the tree} = Tr * \text{Partition-reach of child task queue} \quad (3.1)$$

It is our desire to point out that in the hierarchy, processors and task queues (except the root) could be requesters of work. In addition, only the processors are the final recipients (obviously!). Due to the demand driven nature of HSP, self scheduling is explained with reference to the viewpoints of processors and schedulers. All observations are in consideration to the "system load time scale" that is indicative of the total system load at all moments during execution.

At the first instance when the number of tasks in a processor's local queue becomes less than MPL, it requests a positive number R_t of tasks from its parent task queue that is at level $L - 1$, where

$$R_t = (\text{MPL} - \text{Number of remaining tasks in the processor ready queue}) \quad (3.2)$$

If the parent has N_t tasks, it satisfies the request by sending N_p tasks to the processor where

$$N_p = \min(R_t, \lceil \frac{1}{B} * N_t \rceil) \quad (3.3)$$

If the parent did not have any tasks, it forwards the first and only request from any of its children to its parent which is at level $L - 2$. If tasks are available, it satisfies the request. Otherwise, it forwards the first request to its parent. This procedure is recursively followed by all task queues until one of the following two instances occur; either some queue is able to satisfy the request or the request reaches the root node which just saves it for future processing. If tasks were found at some level in the hierarchy, then the number that will be passed down from a node (say TQ_n) is given by

$$\min (Tr * \text{Partition-reach of the task queue}, \lceil \frac{1}{B} * \text{Number of tasks with } TQ_n \rceil) \quad (3.4)$$

It has to be emphasized that the above expression is applicable to all task queues present until level $L - 2$. Queues at level $L - 1$ transfer N_p tasks as noted earlier. Now, we illustrate this method with an example.

Let us suppose that all ready queues are empty and the root node has 16 jobs J1 through J16 waiting to be executed on a 64 node multiprocessor as illustrated schematically in Figure 3.1. Request message from an idle processor (assume P0) reaches the root queue as all the task queues along the branch $P0 \rightarrow TQ5 \rightarrow TQ1$ are empty. Since the branching factor B is 4 and there are more jobs than B at the root, the policy operates in a *job transfer mode*. In this mode, the entities transferred between successive levels of the tree are the entire jobs. Here we are

making an implicit assumption that the task queue's partition-reach is greater than or equal to the jobs average parallelism. In the example, TQ0 transfers $\frac{1}{B}$ times the enqueued jobs, that is 4, one level down the tree to TQ1 without disturbing the rest of the jobs at TQ0. By this move, J1 through J4 are restricted to the partition consisting of the processors P0 through P15, that is, partition-reach of TQ1. This is analogous to space sharing. At TQ1 the policy still applies job sharing component as the number of jobs is equal to B . Therefore, P0 moves J1 to TQ5. At this point, J1 is allocated a partition defined by the partition-reach of TQ5. This consists of 4 processors P0 through P3. Simultaneously, HSP switches to a *task transfer mode* to implement timesharing amongst the tasks of J1 and assigns at most MPL number of tasks to the processors P0 through P3 based on requests from the same. Notice that this process leaves behind varying number of tasks at different schedulers along the path $TQ0 \rightarrow TQ1 \rightarrow TQ5$. Due to this, certain requests could be satisfied by intermediate task queues thereby reducing contention for the root queue. This behavior is illustrated in Figure 3.2.

3.1.2 Granularity of Transfers

Depending on the load, it would be beneficial if the system adaptively transfers varying chunks of work as seen in the previous example as well. However, it is required that this mechanism does not induce large overheads during scheduling. Below we show how this can be achieved in HSP. Previously, it was noted that a job may or may not be broken down into tasks. If we are dealing with only jobs then it is, but imperative, that we transfer them. However, the transfer process becomes interesting if the jobs were divided into tasks. Depending upon the average parallelism of a job, A_{par} , a scheduler attempts to transfer it down to a child with the least partition-reach greater than or equal to A_{par} . For instance, according to Figure 3.1, if a job residing at the root has as its $A_{par} = 3$, then an attempt will be made to transfer the whole job till level 2. If, on the other hand, another job with $A_{par} = 10$ was present at the root, an attempt will be made to transfer it only till level 1. Note that any such job transfer is initiated at a task queue only if there

Figure 3.2: Job and task transfer mechanisms under HSP. Job indices in *italics* indicate them being transferred.

is an abundance of tasks with it so that it can satisfy future tasks requests. We provide the meaning to this with a counter example. If, for instance, a job with $A_{par} = 5$ was enqueued at the root in Figure 3.1 during system startup time. In this case, it does not try to satisfy a request by transferring all five tasks. Instead, it transfers two tasks for the first request, and one each for succeeding ones as given by equation 3.4.

There are at least two benefits offered due to this adaptive transfer mechanism. One, the number of messages is reduced by an appreciable amount if there is a large number of tasks in any task queue. This is achieved because each time a request reaches a task queue, a number of tasks more than that requested is attempted to

pass back. Therefore at medium to heavy system loads, a request in all probability will be satisfied due to tasks brought down because of another request prior to it. Two, the problem of load imbalance as perceived in distributed schedulers is overcome since tasks are always transferred from a parent to child in response to a request message.

3.1.3 Minimization of Message Passing

The greatest overhead in MPPs is due to the communication. Hence there is always a need to reduce message passing in these systems. Below, we shall see as to how this is possible with HSP.

The branching factor determines the number of message transfers between a parent and its children. In our case at most B messages could be passed onto a parent. It is to be noted that this number is constant over the entire tree irrespective of its depth. Therefore, any queue including the root will get at most B messages (4 in our case) irrespective of its partition-reach. This definitely represents an enormous improvement over the centralized scheme wherein all P processors will pass messages to the single queue. This is made possible because any queue will forward only the first message out of the possible B that can be received by it. Another request is forwarded only after the earlier one is satisfied. All other requests that arrive in the meantime are stored by the corresponding scheduler for future processing. This mechanism is entirely due to the inherent characteristics of HSP. The resources required for this control is optimal since each task queue can be mapped onto a distinct processor. A task queue thus stores and controls the message flow using the resources local to it. This mechanism is advantageous in the sense that no specific locking mechanism nor synchronization management is required as used in Ayachi and Dandamudi [AD96].

As compared to the distributed organization, processors under HSP require more messages to get tasks. However, this does not degrade the performance of HSP dramatically as noted in the study by Dandamudi and Cheng [DC95].

The HTQ description and HSP mechanism have been described independent of any implementation procedures. Choice of a implementation method can be made only after evaluating a number of issues which are illustrated in the following section.

3.1.4 Implementation Methodology

In terms of implementing HTQ there is a need to evaluate both the hardware and software approaches. One hardware based approach is that of the distributed hierarchical control as proposed by Feitelson and Rudolph [FR90]. They argue that the hardware approach is advantageous because it allows various optimizations to be performed by the architect with different processors. This is driven by limited hardware requirements and control. In contrast, ours is a completely software driven approach. Reasons for this decision are cited below.

- Our fundamental consideration is a loosely coupled system consisting of thousands of *homogeneous* processors. Therefore all pertinent optimizations are equally applicable over all processors.
- The main advantage is the flexibility that is provided during modifications of the tree such as its branching factor, etc. This is available to a very small extent in a hardware based hierarchy.
- The task queues do not even need any extra memory as each of them could be mapped uniquely onto a distinct processor. It is worth noting that the event of having the maximum number of task queues in the P processor system is $P - 1$ arising when $B = 2$. This is the worst case and thus even this does not lead to contention of any processor's memory.
- A software based hierarchy allows a programming interface for scheduling that can be provided at either the kernel or application level.
- The tremendous advances in hardware-software interfaces for high speed communication deems our arguments plausible as observed in various studies [vECGS92, MTR95, XH96].

3.2 The System Model

Since we are particularly interested in scheduling policies for DMMs, it is our intention to evaluate HSP against other schemes on a common platform. This means the underlying system architecture should be the same. We modeled such a message passing system and its characteristics are described below.

For all comparisons, we assume a system consisting of $P = 64$ homogeneous processors each having a local CPU and memory. These are connected using a multistage interconnection network. Jobs arrive into the system for service in a Poisson stream with rate λ and are queued at the root task queue. For each job, we assume its average parallelism is known in advance. This value could range between 1 and 64. At this point, we would like to emphasize that we *do not* model the front end of the system. That is, we do not account for the cost of dividing a job into constituent tasks, etc. This is because, the amount of work involved just adds a constant factor quantitatively to the overheads of all the scheduling disciplines and hence does not have any bearing whatsoever on the relative performance of the policies. In other words, from an implementation viewpoint, the process of breaking down a job into its constituent tasks and enqueuing the same into the arrival queue is completely independent of the nature of the scheduling discipline present in the system even though it does act upon these tasks. Therefore we focus only on the activities related to the multicomputer system. Within the system, we do not consider the effects of memory requirements and synchronization delays. For each job, these are appropriately added when generating its execution time based on Sevcik's execution time function for one set of workload conditions [Sev94].

During multiprogramming, it is imperative to account for the time quantum and context switch delays. For our experiments these quantities are based on the overheads in certain real systems such as Fujitsu AP3000, NCUBE 3, IBM SP2, etc [BCG⁺96]. A sampling of the quantum length and context switch times are provided in Table 3.1.

For one set of experiments in particular, the quantum length was 50 ms and

System	Quantum Length	Context Switch Time
DEC based NOW	5 – 10ms	5 μ s
Fujitsu AP3000	1 – 10 ms	5 μ s
Intel iPSC	50 ms	10 μ s
NCUBE 3	10 ms	10 μ s
Stanford-FLASH	30 ms	30 μ s

Table 3.1: Quantum lengths and context switch times of some commercial systems

context switch time 10 μ s.

Since our focus is on distributed memory multicomputers, we use the communication model presented by Xu and Hwang for modeling the communication cost explicitly [XH96]. In this model, each message is assumed to incur a cost given by a “linear” C_{comm} which is defined as:

$$C_{comm} = t_0 n + \frac{m}{r_\infty n} \quad (3.5)$$

where t_0 is the start up time or latency,

n is the number of nodes involved in the communication,

r_∞ is the asymptotic bandwidth which is defined as the maximal bandwidth achievable when the message length approaches infinity, and

m denotes the number of bytes of data to be transmitted.

Note that m/r_∞ represents the transmission delay in passing an m -byte message through a network with an asymptotic bandwidth of r_∞ . The expressions were obtained in two steps:

1. Communication time measurements were performed for various combinations of message lengths and number of nodes.
2. Obtained data was curve fit into the above equation. Based on trying out various forms for t_0 , and r_∞ , simple forms for communication patterns such as broadcast, scatter, gather, etc. were obtained.

3.3 Workload Parameters

As in the characterization of any workload, it is imperative to allow for overheads in a parallel computation when characterizing the jobs. We took a two-pronged approach towards realizing the workloads for our experiments. We refer to these as the *ideal case workload* and *real case workload*.

3.3.1 Ideal Workload Considerations

It is known that workloads and their associated characteristics have definite effects on the scheduling policies. In some instances, scheduling schemes could switch between various transfer modes depending on the arrival distribution. In others, based on the structure of jobs, the schemes may resort to a different partition calculating mechanism, and so on. Recent studies have quantized the type and extent of the effect workloads have on scheduling policies [CMV94, FN95, PS95]. In order to completely understand the characteristic behavior of each of the schemes, it would be nice to observe their performance under a known and analyzed workload model. This was the rationale for choosing to conduct a series of experiments using the popular fork-and-join type of workload model and its variant consisting of multiple phases of fork-and-join. It is obvious that this model represents characteristic workloads only to some extent. Therefore we prefer to call it the ideal workload model. Results obtained while using this model could be analyzed as to whether the observed behavior is due to the workload or some parameters of the scheduling scheme. Due to this, the effect of various parameters involved in scheduling can be studied in detail. The method by which we generated the workload is explained below.

Each job's parallelism structure was generated through a normal distribution as per the studies of Feitelson and Nitzberg [FN95]. This defined the maximum parallelism of the job and was constrained to be between 2 and 64. The service demand of each job was generated through a 2-stage hyper-exponential distribution as in various simulation studies [MEB88, SST93, CMV94, AD96]. In tune with

typical simulation experiments, the job arrival into the system was distributed in a Poisson manner. These characteristics are representative of many popular fork-and-join job structures used in various performance based experiments [TRS90].

3.3.2 Realistic Workload Considerations

In an effort to study the qualitative and quantitative effects of workloads and characteristics on the scheduling schemes, we took into account the overheads due to parallelization and communication and congestion delays. These constitute the second part of our experiments. For this we used Sevcik's execution time function which encompasses a wide range of representative applications [Sev94]:

$$T(p) = \phi \frac{W}{p} + \alpha + \beta p \quad (3.6)$$

where

- ϕ is an indication of the load imbalance in the computation
- W is the cumulative service demand for the job
- p is the number of processors allocated to the job
- α represents the overhead per processor to achieve parallelism
- β represents the amount of interprocessor communication and congestion overhead that increases with the number of processors

$\frac{W}{p}$ represents the ideal division of an application's basic work across p processors. From the equation, it is evident that jobs with different characteristics and inherent structures can be obtained by choosing the values for ϕ , W , α , and β appropriately. In addition, it is possible to generate jobs with the requisite predefined properties and control. Below, we show how the aforementioned relationship is possible through a description of the parameters involved in Equation 3.6.

3.3.3 The Maximum Parallelism – p

This determines the maximum number of processors required by the job. Alternately, this determines the number of tasks that comprise a given job. For our experiments, we decided to generate the parallelism by a normal distribution simply because the jobs themselves were being done so. In addition, we constrained the parallelism to be between 2 and 64. 2 was chosen as this is the minimum processor requirement in order to calculate the overheads [Sev94]. 64 signifies that the tasks of any job would not exceed the number of processors in the system. This was driven by the desire for simplicity without loss of generality.

3.3.4 The Work Imbalance – ϕ

For our experiments, we choose ϕ to be 1, as measurements performed by Wu indicated that in most cases it can be treated as a constant never exceeding 1.14 [Wu93].

3.3.5 The Essential Work – W

W determines the essential minimum service demand of each job. As noted before, in our attempt to model realistic workloads, we simulated the values for W based on observations by Stergios [Ana96] and statistics gathered by a number of researchers at high performance computing centres [Hot96, JPF⁺97]. Continuing in the same spirit, we assumed hyper-exponential distribution for the run times of these jobs as per the observations of Feitelson and Nitzberg [FN95]. For this, we generated it by a 2-stage hyper-exponential distribution as in typical simulation experiments [MEB88, CMV94, AD96, Ana96]. One set of the values used for our experiments consisted of jobs with mean service demand 13.76 and coefficient of variation 3.5. These values were generated by the following cumulative distribution function

$$F(W) = 0.125 * (1 - e^{-W/101}) + 0.875 * (1 - e^{-W/1.3}) \quad (3.7)$$

Other representative values for mean service demand were 16.76, 25, etc. and coefficient of variation were 5, 10, 15, and so on.

3.3.6 The Communication and Congestion Delays – β

The communication overhead β is a function of the maximum parallelism of the job. In addition, the maximum parallelism p_{max} of a job is given by the following equation

$$p_{max} = \begin{cases} \infty & \text{if } \beta = 0 \\ \sqrt{\frac{W}{\beta}} & \text{otherwise} \end{cases} \quad (3.8)$$

Since Feitelson and Nitzberg observed uniform distribution of jobs with different sizes, we uniformly generated jobs with parallelism, p_{max} , ranging between 1 and 64. Given W , it is straight forward to control β . That is, if we require

$$p_{max} = \sqrt{\frac{W}{\beta}} \quad (3.9)$$

then β satisfies the following equality.

$$\beta = \frac{W}{p_{max}^2} \quad (3.10)$$

3.3.7 The Parallelism Overhead – α

Job speedup $S(p)$ is fundamental in determining the parallelism overhead α . This together with the execution time of a job on a single processor $T(1)$ completely defines the execution time of the job for any number of processors p . That is,

$$S(p) = \frac{T(1)}{T(p)} = \frac{W + \beta + \alpha}{\frac{W}{p} + \beta p + \alpha} \quad (3.11)$$

Dividing the numerator and denominator by W and taking into account that $\frac{\beta}{W}$ is equivalent to $\frac{1}{p_{max}^2}$ from 3.10, we get

$$S(p) = \frac{1 + \frac{1}{p_{max}^2} + \frac{\alpha}{W}}{\frac{1}{p} + \frac{1}{p_{max}^2} p + \frac{\alpha}{W}} \quad (3.12)$$

The above powerful optimization now expresses the speedup curve as a function of the maximum parallelism p_{max} and the ratio $\frac{\alpha}{W}$. The implication of this is that jobs tend to experience similar speedups when their maximum parallelism is fixed and the ratio $\frac{\alpha}{W}$ does not vary by much for any value of the service demand. Now, if we make the assumption that $p_{max} > 1$, then the parallelism overhead α is given by:

$$\alpha = W \left(\frac{1}{p_{max}^2} \right)^\mu \quad (3.13)$$

for some

$$\mu \in R \cup \{+\infty\} \quad (3.14)$$

This implies that,

$$S(p) = \frac{1 + \frac{1}{p_{max}^2} + \left(\frac{1}{p_{max}^2}\right)^\mu}{1 + \frac{1}{p_{max}^2}p + \left(\frac{1}{p_{max}^2}\right)^\mu} \quad \mu \in R \cup \{+\infty\} \quad (3.15)$$

From Equation 3.15, it is apparent that the maximum parallelism and the associated overheads due to parallelization determine the speedup experienced by the jobs.

The parameters involved in Sevcik's execution time function thus provide control over the distributions of the speedup of the applications. This together with the distribution of the maximum parallelism of jobs would lead to generating jobs with the desired characteristics and thus provide a basis to perform reliable and clear analyses with respect to their effects.

On the basis of these, we used four workloads in our experiments as defined below [Ana96]. The characteristic structure of applications represented by the workloads are illustrated in Figure 3.3.

1. Workload WK1 consists of curves with relatively good speedup as indicated by p_{max} . They correspond to $\mu \rightarrow +\infty$.
2. Workload WK2 was derived with $\mu = 0.4$ and indicated speedup not as good as WK1.
3. Workload WK3 represented jobs with very poor speedup obtained with $\mu = 0.2$.

Figure 3.3: Application structure as represented by the three workloads WK1, WK2, and WK3

4. Workload WK4 was derived out of an equi-probability combination of the job mix consisting of WK1, WK2, and WK3.

3.4 Performance Evaluation

The system and workload models as described previously suffice to test space partitioning, time sharing, as well as HSP. For relative performance evaluations we used the *mean response time* as the metric. For a job, this is defined as the time elapsed since its arrival into the system till all of its tasks have completed execution. The other metric is the *system utilization*. This indicates the duration the processors were busy during the simulation. This quantity is determined in two stages. First,

the sum of the processors busy time is averaged over the number of processors in the system. This gives the *average busy time* of a processor. This quantity is then divided by the entire simulation duration to obtain the utilization.

Due to the complete control offered by simulation experiments, it is imperative that the results be obtained with a high degree of confidence. This is provided by the *confidence intervals* which indicate the range within which the results could be found. That is, it denoted the relative error associated with each value. For this each experiment was executed 31 times or *batches*. Each batch consisted of executing 50,000 jobs. As is the usual practice, we decided to completely ignore the results from the first batch as the system is 'warmed up' at this stage. Results for the rest of 30 batches were then taken into consideration. The confidence intervals were obtained by using the *student's t-distribution*. All the results we obtained had a 95% level of confidence with errors most often less than 1%, thus ensuring statistical reliability.

In retrospective, it is worth noting that for the simulation experiments, the system is modeled as a typical distributed memory multicomputer. Also, our workload characterization is based on realistic, significant observations at a number of high performance computing centres. For the sake of completeness, Table 3.2 summarizes all the parameters together with their values as used in our simulation experiments.

Relative performance measurements were carried out on the basis of these parameters on a network of Pentium 200 systems running the Linux operating system. This completes the description of the system model in regards to HSP and the workload characterization which was used to evaluate it. In the following sections we present briefly the simulation models for the other two policies.

3.5 Simulation Model for Existing Policies

As was mentioned before, HSP is a hybrid scheme combining space partitioning and time sharing. For this reason, we evaluated HSP relative to one space- and time sharing policies. We modified the Adaptive Partitioning 2 (AP2) policy of Rosti

Parameter	Description of values
Processors P	64 homogeneous processors
Execution time T(p)	$\phi(p)\frac{W}{p} + \alpha + \beta p$
Essential work W	$F(W) = 0.125*(1 - e^{-\frac{W}{16}}) + 0.875*(1 - e^{-\frac{W}{13}})$ $E(W) = 16.36$ and $CoV = 10.0$
Maximum Parallelism p_{max}	Uniform between [1, 64]
Parameter β	$\frac{W}{E_{max}}$
Parameter α	$W(\frac{1}{p_{max}})^\mu$
Workload 1	$\mu = +\infty$; $E(\alpha) = 0.0$; $E(T(1)) = 14.06$
Workload 2	$\mu = 0.4$; $E(\alpha) = 2.0$; $E(T(1)) = 16.06$
Workload 3	$\mu = 0.2$; $E(\alpha) = 5.0$; $E(T(1)) = 19.06$
Workload 4	$\mu \in \{0.2, 0.4, +\infty\}$; $P(0.2) = P(0.4) = P(\infty) = \frac{1}{3}$
Inter arrival Time	$f(t) = \lambda e^{-\lambda t}$ with $\frac{1}{\lambda} = \frac{E(T(1))}{Processors * Load}$
Pt.-to-Pt. communication delay	$t = t_0 + \frac{m}{r_{com}}$
Scheduling overhead	1% of the execution time
Preemption overhead	10 μ secs
Transfer factor Tr	1
Branching Factor	4
MPL	2
Total # of jobs/batch	10000

Table 3.2: Load and system parameters as used in our experiments.

et. al [RSS⁺94] for the former. For the latter, we used the modified version of the round-robin job policy (RR_{job}) of Leutenegger and Vernon [LV90]. Below we give a brief description of the simulation models for these two.

3.5.1 Modified RR_{job} – Time Sharing

The modified version is characterized by the use of a central queue for tasks. At the same time, unlike the original scheme, we do not simulate the scheduler that makes allocation decisions. The motivation for this is to induce the similarity of the demand driven scheduling of HSP. The offshoot of this is; one, absence of centralized scheduler removes a potential contention point. Two, the demand driven nature

makes it more suitable for comparison purposes. An additional modification is the introduction of MPL in each processor which was absent in the original scheme. Because of these modifications, we prefer to call the resulting schemes RRTS, denoting Round Robin Time Sharing.

The system model is similar to HSP, that is, a 64 node message passing distributed memory multicomputer. In addition, all associated overheads during scheduling, communication, and context switches are also considered.

In regards to the jobs, characteristics such as arrival, job characteristics generation, associated overheads are similar to the one considered for HSP. Whenever a job is generated it is divided into constituent tasks representing its maximum parallelism. Depending on this value, the quantum of service given to each job varies so as to account for fairness. Therefore, in the attempt to allocate equal processing power to each job, smaller jobs are given larger quantum than the larger ones. By this, for a job j with maximum parallelism p_{max} in a system with P processors with a default quantum length of q , the duration for each task is:

$$Q_j = \frac{P}{p_{max}} * q \quad (3.16)$$

These tasks are then added at the back of the central queue behind all other waiting jobs. Since we constrain the maximum parallelism of the job to be between 2 and 64, in the case p_{max} becomes equal to 64, then the quantum size defaults to the system quantum length. Now, each processor picks up a number of tasks t given by

$$t = (\text{MPL} - \text{Number of tasks in local queue}) \quad (3.17)$$

In the event multiple processors request tasks from the queue, the messages are serialized based on a first-come-first-served basis. Similar to the experiments for HSP, we ran the experiment for a total of 31 batches with 50,000 jobs in each batch.

3.5.2 Modified Adaptive Partitioning – Space Sharing

Analogous to time sharing, the system was simulated as a 64 node message passing multicomputer. Job characteristics such as arrival rate, service rate, etc were again

similar to timesharing and HSP. Each job that was generated had a maximum parallelism associated with it. This was used to determine the parallelism overhead and communication and congestion delays as per Equation 3.6. An additional overhead was also accrued in an attempt to “fit” the job into a partition. For instance, if the maximum parallelism of a job was, say, 22. Now, if this job was assigned a partition of size 32. Then, an attempt was made to divide the job into 32 tasks. This induces additional overhead for the job. Again, Equation 3.6 was used to calculate the same. Since space partitioning policies also employed a global job queue, the overhead of scheduling a job from this onto the partition was considered as well. The description so far is general enough so as to be applicable to all space sharing schemes. In contrast, the modifications that are explained below pertain to only the Adaptive Partitioning 2 policy. These are a result of the rather poor semantics of the policy as can be found in [RSS⁺94]. For the sake of completeness and uniformity, we had to modify this. First, AP2 creates the *premature* fragmentation problem because of its tendency to allocate partitions if at least two jobs are present in the job queue. One implication of this is that a single job tends to be enqueued even when the whole system is idling. A second modification was made with respect to partition allocations. As the original policy does not take into account the number of scheduled jobs when allocating partitions, we felt that this would be unrealistic. So for an entire set of experiments, we have taken into account 50% of the scheduled jobs. The motivation for choosing 50% is simply that it represents the middle ground between taking all jobs into consideration (or 100%) or none of the jobs into consideration (or 0%). Quantitatively, 50% is the average between these two extremes. Introduction of these features prompts us to call the resulting scheme Modified Adaptive Partitioning 2 (MAP2). Incorporating this modification results in the following expression for partition size calculation.

$$\text{PartitionSize} = \frac{P}{(\text{QueuedJobs} + 0.5 * \text{ScheduledJobs} + 1)} + 0.5 \quad (3.18)$$

Here, P represents the total number of processors in the system. Other parameters are self explanatory on the basis of their names.

Considerations of varying percentages of jobs is the focus of another research study that has already been initiated here.

Results for this scheme were obtained at about 95% to 99% confidence level after running 31 batches of 50,000 jobs each.

3.6 Validation of the Simulation Process

In order to satisfy the need to test the simulation models, we decided to compare the performance results with those of analytical means so as to ensure the correctness and consistence of the models. This approach was taken because, the prospect of using existing results were implausible after the changes we had introduced into the models. However, we did use existing results to compare the behavior of the policies. These trends were a reflection of the inherent nature of the policies themselves. By this we mean that, the effect of overheads due to workload characterization was not considered during these testing phases. Based on these trends, we are able to vouch for the completeness and correctness of the scheduling schemes.

Chapter 4

Performance Evaluations Under Ideal Workloads

In this chapter, details of our simulation experiments carried out to evaluate the performance of HSP are presented.¹ Performance of HSP could be affected by a number of parameters such as MPL associated with each processor, branching factor of the underlying hierarchy, application structures comprising the workload, etc. Here we make a concerted effort to evaluate its performance for all pertinent factors. For this, we tested and compared it with Modified Adaptive Partition 2 (MAP2) as well as Job Based Round Robin Time Sharing (RRTS). The metric under which comparisons were done is the mean response time of the jobs. We provide explanations for the comparative performance observed due to these schemes.

As was mentioned in the previous chapter, we decided to conduct two main sets of experiments denoting ideal case situation with reference to the workload as well as the realistic case. We use the term ideal case to mean a workload composed of a single application type, namely, the popular fork-and-join model. Though this model has been widely used as part of many studies, it definitely is not comprehensive to reflect the application structures usually scheduled on high performance systems around. This chapter details the results for this single workload model. Since the motivation was to gain a better understanding of the behavior of the three

¹The results presented in this chapter appear in the *IEEE International Conference on High Performance Computing* [DT97].

policies under our simulation environment, we conducted a number of experiments to study the effect of various parameters. In this regard, using a workload type the knowledge about which is known, assumes significance as resulting behavior of the schemes could be analyzed as to whether being due to the effect of workload or driven by certain parameters. As part of this, we studied the effect of scheduling and contention overheads, the effect of coefficient of variation of the cumulative job service demand, the effect of load imbalance due to a job, and other related ones. All the configurations were examined under the basic workload as well as its variant consisting of multiple fork-and-join phases. The motivations stems from the fact that characteristic applications such as the Barnes and Hut algorithm [BH86] can be modeled by this structure. The default system and workload parameters for the experiments are those that are detailed in Table 3.2. For a better understanding of the performance of each policy, we decided to present it at about 10 different load levels, that is, 5%, 10% to 90% in steps of 10. Note that these were represented as and when applicable and practical. Increased load on the system was achieved by increasing the job arrival rate λ while keeping the job service rate μ a constant. In the following sections, all the results corresponding to the simulation models of Chapter 3 are presented and analyzed.

4.1 Characteristics of MAP2, RRTS, and HSP

It is imperative that one understands the basic characteristics of the scheduling schemes in order to explain their behavior due to external parameters. Figure 4.1 shows the performance of the three policies as a function of offered system load.

Note that HSP is different from the other two schemes in the absence of a centralized queue. Over the entire range of system loads, space sharing performs much better than time sharing. In fact the relative performance of the former seems to improve to greater extents with increasing loads. This is because of two reasons. One, all the tasks of the job have equal service demand. Therefore when these are scheduled onto partitions, no fragmentation occurs, and all tasks finish at the

Figure 4.1: Performance of the three policies under ideal workload

same instant without undue overheads. Two, time sharing suffers due to multiprogramming the tasks and the corresponding context switch delays. Note that for all experiments we set the MPL at 2.

The hierarchical policy performs better than space sharing mainly because of its hybrid nature. By this, the extraneous waiting time experienced by arriving jobs in MAP2 is avoided in HSP. This becomes critical at medium to heavy loads wherein number of jobs arriving during the execution of a scheduled job is large. HSP avoids this due to the dynamics of the job and task transfer process combined with multiprogramming these within each partition. It is worth noting that the superior performance of HSP further indicates that context switching overheads are negligible. From a second viewpoint it can also be argued that MAP2 suffers from the additional disadvantage due to the presence of a centralized queue. At the load range of interest to us, that is, medium to heavy, this becomes a ready bottleneck.

The results further show that workload characteristics play a decisive role in the performance of the schemes. Sometimes the effects of which may even be counter intuitive. This is the motivation for us to study the impact of the multiple fork-and-join application structure on the performance of each of these schemes in the following sections.

4.2 Relative Performance under Workload Imbalance – 50-50

As was noted before we introduced an imbalance in the workload to study the behavior of the three schemes under varying parallelism conditions. We introduced the imbalance in three forms, one of which, termed as 50-50, is presented in this section. Recall that with respect to a job 50-50 refers to 50% of the tasks of a job getting 50% of their service demand while the remaining is shared equally amongst the other tasks. Figure 4.2 shows the relative performance of the three schemes under this workload at service time CV equal to 10.

Severe performance problems are experienced by MAP2 due to the imbalance and hence its performance is the worst in relation to RRTS and HSP. This is brought out in the following example. Consider a job consisting of 4 tasks being allocated a 4 processor partition. If the job service demand is 16 mins. Then 2 tasks (50%) will each have a demand of 2 mins (50% of 4 mins) while the remaining two tasks will have a demand of 6 mins each. As a partition is allocated for the entire length of a jobs execution, two processors out of four will idle for 4 mins. In other words, the overall utilization for this partition is $\frac{16}{24} = 67\%$. This scenario only worsens when larger partitions are considered. As a point to note, the average parallelism of the jobs was found to be 16, and the partition utilization in some cases was as low as 40%.

The performance of RRTS and HSP is appreciably better than MAP2. This is mainly achieved because of multiprogramming the tasks on a processor. Amongst the two, RRTS performs better than HSP at low loads. This is due to the space

Figure 4.2: Performance of the three policies under 50-50 distribution (service time $CV = 10$)

sharing nature exhibited by HSP at low system loads. Note that performance at these load levels are not of interest to us, and it is not useful in general.

With the increase in system load a noticeable difference is observed between the behavior of HSP and RRTS at medium to heavy loads. The deterioration of the latter is a direct consequence of the contention for the global queue. Note that at these loads, both HSP and RRTS are affected in a similar manner due to multiprogramming. That is, jobs tend to be subjected to a FCFS policy at these loads. Hence, the centralized queue is accessed by each processor after each task is executed. In contrast, the self scheduling nature overcomes this by reducing the contentions for each queue. To illustrate this, we could assume that the global queue will be accessed by all 64 processors under RRTS at high loads where as each of the

Figure 4.3: Performance of the three policies under 50-25 distribution (service demand $CV = 10$)

16 queues at level $L - 1$ in the hierarchy (as depicted in Figure 3.1) will be accessed at most 4 times. Hence the notable difference.

4.3 Relative Performance under Workload Imbalance – 50-25

A second variation in the imbalance was introduced as follows. Here, 50% of the tasks of a job received 25% of the service demand while the remaining job service demand was distributed evenly amongst the remaining 50% of the tasks. Figure 4.3 illustrates the relative performance of the three policies under this consideration.

As in the previous case, the performance of space sharing is substantially worse

than time sharing and hierarchy. The reason for this is the extensive fragmentation caused by the imbalance. To see this, consider the example introduced in the previous section. For this job, due to the 50-25 criterion, two tasks will have a service demand of 1 min (25%) while the remaining two tasks (50%) will have a service demand of 7 mins each. The utilization of the partition in this case would be $\frac{16}{28}$ which is about 57%. Extrapolating this argument to encompass the whole system, we see that more than 40% of the processors idle even when jobs are queued for service. In fact, this effect is the sole cause for the almost exponential rise in mean response times of the job even at low loads for MAP2.

HSP and RRTS, on the other hand, are more stable and their relative performance is much better due to multiprogramming. Notice, again that the overhead due to context switches can be ignored as it doesn't seem to have any major impact on the response times. In addition, the inherent self-scheduling nature of HSP overcomes the bottleneck problem for the global queue as perceived in RRTS as this distributed the effective system load within the hierarchy. This is the prime reason for the better performance of HSP.

4.4 Relative Performance under Workload Imbalance – 50-75

In order to comprehensively notice the behavior and recognize trends due to load imbalance, we decided to experiment with yet another ratio in the imbalance. Figure 4.4 shows the behavior of the three policies in this case.

Similar to the previous two experiments, we notice that HSP performs the best followed by RRTS and then by MAP2. The reasons for this behavior are again due to similar reasons as was cited in the previous two sections; fragmentation for MAP2 and global queue access contention for RRTS. The effect of the changes are not as pronounced as the 50-25 case though, due to higher utilization of the partitions by MAP2. Based, on the three observations for the case of load imbalance, we notice that fragmentation effects can significantly deteriorate the performance of MAP2.

Figure 4.4: Performance of the three policies under 50-75 distribution (service demand $CV = 10$)

Moreover, the extent of this effect seems to have a direct correlation to the amount of fragmentation. These results are convincing enough to avoid using pure space sharing policies in the presence of workloads as considered here. As has been mentioned so far, it is definitely an advantage to combine this with multiprogramming. This in essence supports the concept of developing hybrid scheduling schemes as has been conceptually proposed by a number of studies [SST93, DC95, AD96, FR96].

At this point, one might argue about certain recent studies which indicate performance benefits due to idling of processors in space sharing schemes [SRS⁺95]. However, it can be countered that the kind of fragmentation that is known to arise in our set up does only to degrade the performance. In fact, in the study by Smirni *et. al.* [SRS⁺95], processors would idle if they were not part of any partition. This

50-25	50-50	50-75
$t1 = 1 \text{ min}$	$t1 = 2 \text{ mins}$	$t1 = 3 \text{ mins}$
$t2 = 1 \text{ min}$	$t2 = 2 \text{ mins}$	$t2 = 3 \text{ mins}$
$t3 = 7 \text{ mins}$	$t3 = 6 \text{ mins}$	$t3 = 5 \text{ mins}$
$t4 = 7 \text{ mins}$	$t4 = 6 \text{ mins}$	$t4 = 5 \text{ mins}$
Util = 57.1%	Util = 67.0%	Util = 80.0%

Table 4.1: Example of service time divisions and corresponding utilization

is distinctively different than our case wherein idling is induced as part of fragmentation. Finally, we present the extent of fragmentation possible due to the three cases we have encountered so far in Table 4.1. We show this through the example referred to in the last two sections. That is, task execution times within a partition of size 4. The job being composed of four tasks $t1$, $t2$, $t3$, and $t4$ with its cumulative service demand equal to 16 mins. Taking a second look at MAP2, we go to the extent of making the conjecture that even of its performance is significantly improved due to modifications, it nevertheless would not be able to deliver good performance specially for DMMs as one has to contend with the problems of the global queue.

4.5 Relative Performance under Varying Job Service Demand

Recent studies in workload characterization have observed very widely varying service demands at high performance centres [FN95, Hot96, JPF⁺97]. This gives rise to workloads composed of varying job sizes. Figure 4.5 shows the performance of the three policies at low service demand $CV_d = 1$.

Performance under high service demand CV is illustrated in Figure 4.6.

The CV values used here were based on observations in a number of studies [CMV94, FN95, JPF⁺97]. Note that $CV_d = 1$ means that the demands are drawn from an exponential distribution while $CV_d > 1$ represents samples drawn from a hyper-exponential distribution. We were constrained to the maximum value 15 in

Figure 4.5: Performance of the three policies under 50-50 distribution at low service time CV ($= 1$)

order to incorporate MAP2 in the study. From the figures, it is noticeable that with increasing service demand, the performance of MAP2 deteriorates at a faster rate. This is a common trend with the effect more pronounced at high loads. This behavior is due to the fact that in a mixture of jobs consisting of many small ones and a few large ones, there is a tendency for the smaller jobs to be enqueued for long durations because of larger jobs in front of them.² This leads to premature queuing as there will arise situations wherein the number of free processors may not meet the partition requirement of the large job for a very long time though they might easily satisfy the smaller succeeding ones.

²It is to be brought to light that large jobs correspond to having many tasks with greater demands than corresponding small jobs or those with fewer tasks with very high demand per task.

Figure 4.6: Performance of the three policies under 50-50 distribution at high service time CV (= 15)

The inferior performance of RRTS as against HSP is again attributed to the presence of the central queue, though from a different perspective than contention. In this case, the increased response times under RRTS is perceived because of the duration it takes to completely execute the large job. This is possible in two ways as shown by the following two examples.

- Note that, higher variance in service demand implies a large number of small jobs and a few large jobs. Assume two job streams consisting of six jobs with the following service demands (in mins): 1, 1, 1, 1, 1, 7, and 1, 1, 2, 2, 3, 3. Even though both job streams have an average service demand of 2 mins, the first stream has higher variance and causes the inter access times of the central queue to be small for five jobs as there are five jobs with service demand 1 as

compared to the second job stream.

- If we were to consider a job with 64 tasks being enqueued at high load. If, also, we assume that the service demand of each task is 1 minute and the same for the quantum length. Then, this job gets executed after about 64 mins mainly because the system will operate in a quasi-FCFS mode. Upon scheduling this job, all jobs arriving later will have at most 1 slot on each processor assuming MPL to be 2. This leads to increased response times. The same large job gets executed in about 16 mins under HSP even at high loads. This is possible because the tasks are allocated processors much more quickly under HSP due to the presence of multiple task queues.

The advantage of the tree of schedulers is thus shown by the least response times experienced by the jobs even at high loads.

Another important point to note is the sensitivity of HSP under these varying demands. Higher CV demands tend to make the accesses “bursty”. The effect of clustered arrivals is very significant even though their time averaged behavior does not indicate any drastic change. Given this situation, it is easily observed in all the figures that HSP is insensitive to these “bursty” arrivals when compared with MAP2 and RRTS.

4.6 A Summary of the Relative Performance

This chapter discussed the fundamental behavior of the three scheduling policies in the face of known workload model, viz-á-viz, fork-and-join. These were tested with respect to a number of involved parameters. In summary, the following observations can be drawn about the performance of the schemes from the presented results.

- HSP is seen to perform the best under the basic single stage fork-and-join workload. This is especially noticeable at medium to high loads, which is in fact the range of interest to us. In addition, it is seen to be insensitive over the entire system load scale.

- For the case of multi-phase fork-and-join workload model, the imbalance in task service demands within each job induces corresponding fragmentation under space sharing. This proves to be detrimental in achieving low response times. In addition, based on the extent of imbalance, sensitivity at even medium loads is not observed in many cases.
- Time sharing and HSP do not experience the problem of fragmentation. Given this fact, the observed deterioration in performance of the former is a direct consequence of the presence of the global queue which becomes a certain bottleneck at medium to heavy system load considerations.
- In the presence of a large variation in the job service times, it can be seen that the HSP exhibits least sensitivity primarily due to the underlying tree of schedulers as well as the adaptive nature of the job and task transfer mechanisms. Time sharing, on the other hand, is unable to cope up with the increased access to the central queue, and thus exhibits poor performance.

Based on these conclusions, it seems that time sharing tasks within partitions together with demand based scheduling provides significant benefits as against either pure space- or time sharing centralized schemes. It seems interesting to study the behavior of the three schemes under realistic workloads. This is more so under workload combinations consisting to varying application structures. These form the basis of the results presented in the next chapter.

Chapter 5

Performance Evaluation for Realistic Workload Models

In this section the results of the simulation experiments to analyze the relative performance of MAP2, RRTS, and HSP under multiple realistic workloads are presented. All comparisons were done with reference to the mean response time as the performance metric. Several experiments were conducted to evaluate different aspects such as effect of contention overhead on the response times, effect of variation of the service demand CV, effect of variation of the coefficient of arrivals, etc. As discussed in Chapter 3, we conducted experiments under four different workload models WK1, WK2, WK3, and WK4. The first three represent distinct application characteristics, that is, matrix multiplication, mean values analysis, multi-phase fork-and-join. The fourth is a combination of the three in equal proportions. The nature of these workloads induce very different overheads for parallelization and communication and congestion delays. These are taken care of in the execution time function as introduced in Chapter 3 also. For all experiments, the average response time was noted against relevant system loads, which was obtained by manipulating the arrival rate.

Figure 5.1: Relative performance of the three schemes under WK1 (Matrix Multiplication) with overheads

5.1 Relative Performance under Workload WK1

Initially, we examined the performance of the policies under WK1 as discussed previously. Figure 5.1, shows the relative performance for mean service demand 16.06 and CV of demand 10.

Note that WK1 represents the best case speedup amongst the applications. At low loads, MAP2 tends to perform better than the others. This is because, RRTS incurs a penalty due to the MPL of 2. This leads to situations where some processors will have 2 tasks to execute while others may be idling. The same is true with HSP as well. In addition, due to its self scheduling nature, it incurs additional overhead during task transfers. During the experiments, it was observed that the penalties

due to context switching in RRTS and HSP was negligible. This is also evident from the figure. Between HSP and RRTS at this load, the former performs better as its space sharing nature induces less overhead than the latter. That is, tasks tend to be constrained to fewer processors under HSP than RRTS.

Turning our attention to higher loads, we would like to mention that the range of interest for comparisons for this study like any other is between 60% to 90% of the system load. At these loads, it is clearly evident that HSP outperforms the other two. Its hybrid nature involving job and task scheduling is able to transfer work along the appropriate branch at the root much more efficiently at these loads. Once the transfer is initiated, the process of self scheduling tends to maintain almost uniform load amongst all the task queues. As far as RRTS is concerned, the primary reason for its poor performance is the high access contention costs for the central queue. At these loads, even with MPL of 2, it behaves more or less like a FCFS policy. Previous studies indicate similar performance for RRTS at these loads [CMV94, DC95, AD96]. The behavior of space sharing is a direct consequence of the workload and system load. From the execution time function, it can be seen that the communication overhead is a function of the allotted processors. This means that jobs allotted larger partitions than their maximum parallelism tend to experience greater delays. In addition, the queuing delays at the scheduler tend to be large when larger partition sizes are allocated to jobs. These combine to raise the execution time of each job. As the load increases, the partitions tend to become smaller thus incurring less overhead due to message passing. At the same, this leads to limited distribution of work due to the $\frac{W}{p}$ term in the execution time function, thus experiencing large delays. This problem is even more compounded due to the fragmentation effect. These are the outstanding reasons for the performance of the individual policies.

5.2 Relative Performance under Workload WK2

The relative performance of the policies are shown in Figure 5.2 for workload WK2.

Note that WK2 exhibits slightly inferior speed up compared to WK1. Due

Figure 5.2: Relative performance of the three schemes under WK2 (Gravitation Dynamics) with overheads

to this the performance of the three policies are relatively inferior than for the case of WK1. The trend of each individual policy is as before. At this time, we would like to bring a point to note. In as much as the superior performance of the HSP is observed in this case, an equally important requirement is concerned with its stability. This means that the behavioral characteristics of HSP should be similar under different workloads. This is a very important characteristic and is an indication of the *robustness*¹ in the face of changing workloads. Based on the observations of HSP under WK1 and WK2, it can be seen to be exhibiting an appreciable degree of robustness.

¹Robustness is a measure of the stability of any policy under varying workload considerations [RSS⁺94].

In terms of performance, we notice a relative widening between RRTS and HSP. In comparison to Figure 5.1, the performance of RRTS and MAP2 tend to have deteriorated. The reason behind this is the contention for the scheduler especially under higher service demand requirements. MAP2, in addition, experiences greater queuing and communication delays due to fragmentation at about the 70% system load.

5.3 Relative Performance under Workload WK3

Workload WK3 is the worst application in terms of speedup. The behavior of the three policies are shown in Figure 5.3. Similar to the previous instance, we notice rapid deterioration of MAP2 as against RRTS and HSP. The robustness of HSP is again evident.

The performance degradation of MAP2 is quite rapid in this case. This is because of the high service demand CV of 10 that we are using together with the fragmentation. The average size of the partition allocated to jobs ranged between 16 and 9 till about 40% of the system load. This induces a lot of queuing and overhead, and hence the rapid performance degradation. At about 50%, the partition size was 4. This means that at most 6 jobs were scheduled on the system. In fact, this is the cause for the enormous queuing delays which leads to total performance degradation.

5.4 Relative Performance under Workload Combinations

Though the previous workload models were generated as per observations at various computing centres, it is commonly the case that the workloads are composed of many jobs with very differing characteristics. This fact prompted us to evaluate the three schemes under a combination of WK1, WK2, and WK3 to make up workload

Figure 5.3: Relative performance of the three schemes under WK3 (Mean Value Analysis) with overheads

WK4. We assumed that these would occur with equal probability during generation. Figure 5.4 illustrates the relative performance of the three disciplines.

As can be seen, HSP performs very well under all load levels. The behavior of the disciplines is similar as in the previous cases. The interesting point here is to note the marked improvement of space sharing in comparison with its performance under WK2, and WK3. This is due to presence of jobs belonging to WK1 in the workload mix. This behavior can be further generalized to mean that, individual workload types have a deterministic influence on scheduling policies even when occurring in a combination of heterogeneous workloads. In fact, this drives home the point that performance of any scheduling policy has to be studied under workload mixes. Thus, our motivation to consider them have been justified at this point. It

Figure 5.4: Relative performance of the three schemes under WK4 (workload combination, $CV = 10$)

is interesting to note that a similar effect has been observed due to multiple workloads in the realm of network of workstations as well [AVAP94]. Thus presence of some characteristic applications tends to have a definitive effect on the scheduling schemes. In this regard, we go as far as to say that certain salient job characteristics influence even general schemes such as HSP. Therefore it is very important that extensive studies involving workload combinations be performed primarily as part of simulation experiments before implementations to save costs to a large extent.

Additionally, in our case, observations of the performance of RRTS and MAP2 suggest that single schedulers are a major source of bottleneck for real instances as against HTQ type organization.

Figure 5.5: Response time versus utilization for high variance in service demand under WK4 ($CV = 15$)

5.5 Considerations for High Variance in Service Demands

Many recent studies have found the CV of demand in high performance computing centres to be very high, often ranging from 3 to even 40 [CMV94, FN95, PS95]. Figure 5.5 shows the performance of the three policies as a function of system utilization when the service demand variance is 15.

We limited the demand to 15 so as to include MAP2 for analysis. We present the results just for WK4 as it represents very realistic properties of applications. As a result of this the scheduling policies are tested thoroughly under the worst case. The impact of drastically changing service demand CV can be gauged by comparing the

results of this section with Figure 5.4 wherein the variance was fixed at 10. On the basis of these two figures we notice that there is definite performance deterioration in the three policies. However, MAP2 seems to suffer the most as it is very sensitive to job mixes. The drastic degradation in the performance can be attributed to the inherent nature of MAP2 to possess a run-to-completion method for executing jobs as they arrive in a strictly FCFS nature. By this, if a large job was scheduled before a smaller one at higher load ranges then the latter has to wait for the former to complete processing.

An additional point here is that, in comparison to HSP, the performance of both RRTS and MAP2 deteriorate dramatically at about 30% and 50% utilization respectively. This can be attributed to the queue access contention in these disciplines leading to the FCFS situation. As job mixes at these high CVs consist of many small jobs with lesser service demands as well as a few large jobs with very high demands, the access pattern to the queue is “bursty” though the time averaged behavior does not change much. HTQ overcomes these problems due to its structure and organization and is the least sensitive over all values. For instance, even if a large job is enqueued before smaller ones at the root, the first transfer restricts it to a partition of size 16 from 64 with respect to Figure 3.1. By this, the succeeding smaller jobs could well be transferred to the remaining three nodes. When this scenario is compared to that under MAP2 wherein it might even be difficult to get enough processors in the first place as typical partition sizes will be between 10 and 2 approximately at medium to high loads. With reference to RRTS, tasks tend to be scheduled by the global queue in almost FCFS fashion at these loads, thus causing smaller jobs to wait extensively.

5.6 Considerations for Parallelism Variation

It is our view that considering job mixes presents but one dimension with reference to real workloads. However from the viewpoint of individual applications in the amalgamation, there could be many variations in terms of their sizes, demands

Figure 5.6: Effect of variation of maximum parallelism at low loads under WK4

etc. To take this into account, we compared the three schemes under the mixed workload model on the basis of limiting the maximum parallelism of the jobs. In order to observe an individual job's effect explicitly, we first compared the policies at low loads. The results are illustrated in Figure 5.6.

This consideration was motivated by the fact that it is worthwhile to note the behavior of the disciplines as it will be mainly due to the properties of individual applications. In a workload mix, the behavior of a few individual applications with widely differing properties could have a pronounced effect on the scheduling discipline as observed in Section 5.5 and which time averaged statistics do not make explicit.²

²For instance, we observed a few jobs to experience overheads scaled up to 4000 while the time average was about 16.

From this we notice that at low loads HSP and MAP2 perform worse than RRTS. The behavior of MAP2 is bad because of extreme fragmentation problem. For instance, during experiments it was noticed that on the average there were 3 jobs in the input queue at any given time. Assuming all of them arrived at the same time, then each of them would be scheduled on a partition size of 21. Of these, up to 20 processors could potentially be idle in each partition. HSP's performance is better than MAP2 but worse than RRTS due to the space sharing component during scheduling. However, this cost is very minimal and is discernible only at low loads.

For increasing parallelism, we notice that hierarchy performs better than space sharing and time sharing. For these parallelism values, all the processors in a partition will be busy under MAP2. In this case, the only cost a job accrues is due to waiting if the system is already occupied by the preceding jobs. In the case of timesharing, multiprogramming induces additional costs and thus degrades its performance. As far as hierarchy is concerned, its enhanced performance is due to the adaptive nature of the transfer process. In the presence of a few jobs with large maximum parallelism, due to the task transfer process initiated at the root, and with the MPL fixed at 2, every individual processor's requirement is optimally satisfied while being able to balance load fairly evenly amongst the task queues.

Based on observations at a number of high performance computing centres, it can be argued that the percent of sequential and very small jobs is, but, a very miniscule portion of the total workload [FN95, JPF⁺97, Hot96]. Therefore we conclude HSP exhibits the best performance overall at low loads.

The results for the case of high loads is illustrated in Figure 5.7.

In this case MAP2's performance is the worst. As it tends to allocate smaller partitions at very high loads, the execution time on each processor is greatly increased even though the resulting overheads tend to be small. Time sharing and hierarchy perform much better than this. Amongst them, RRTS performance is bad mainly because of access contention costs at these loads. And the absence of the same is the reason for the superior performance of HSP. Here again the robust

Figure 5.7: Effect of variation of maximum parallelism at high loads under WK4

feature of HSP is evident due to its conservative mean response time values over the entire system load range. Based on the observations, one can naturally conclude that the hierarchical policy provides the best performance gains under a plethora of conditions.

5.7 Summary of the Relative Performance under Real Loads

In this chapter, we conducted relative performance evaluations of the three schemes under realistic workloads composed of different application characteristics and combinations of the same. Influence of a number of parameters affecting response time was observed and a summary of the same is given below.

- Under workload WK1, WK2, and WK3 the performance of MAP2 was substantially worse than that of RRTS and HSP. A number of reasons contribute to this behavior such as overheads due to partition allocations of widely varying sizes than the applications maximum parallelism, fragmentation within partitions inherent job characteristics inducing high overheads, widely varying parallelism, and so on.
- The performance of RRTS was much better comparatively though worse than HSP. This is mainly due to two reasons: increased access contention for the global queue and conservative processor allocation mechanisms especially under high variance in service demands and different parallelism profiles of the applications. Its nature to schedule jobs in an almost FCFS fashion under these conditions causes high response times for individual workloads and combinations as well.
- The best performance of HSP is mainly achieved due to even load distribution within the hierarchy especially under high loads and widely varying service demand CV. Its inherent nature to constrain jobs to deterministic partitions during scheduling adds to the improved performance. In addition, it is seen to be the most insensitive to the varying application characteristics. This is possible due to demand driven nature of work transfer as well as the hybrid nature of the policy. In comparison, RRTS is seen to be much more insensitive than MAP2 though inferior to HSP. This is mainly because of time sharing the tasks within each processor.

In the face of these observations, it is safe to assume that HSP's ability to time share tasks within processor partitions makes it more robust to various workloads as against RRTS and MAP2. This in combination with the tree of schedulers is seen to provide the best performance under realistic workload considerations. This strongly suggests a means of designing future scheduling schemes for multiprocessor systems.

Chapter 6

Conclusions

6.1 Epilogue

Parallel job and task scheduling has emerged as a very important area in theoretical as well as experimental studies. Numerous researchers have applied themselves to developing approximate solutions to this problem. This is because, an exact solution to the problem is proven to be NP-Complete [GJ79]. This thesis deals with scheduling strategies for multiprocessor operating systems from a 'systems' viewpoint. During development of multiple techniques for solving the problem, it has become apparent that a few issues decide the feasibility and applicability of the solution approach. These are:

I0 Presence of multiple job queues.

I1 Combined space partitioning and time sharing.

I2 Adaptive load distribution.

Presence of multiple job queues For large scale systems that has been the focus of this thesis (typically containing many hundreds to a few thousands of processors), there is a requirement for having multiple job queues for efficient scheduling of tasks. Otherwise, presence of a single queue as in many traditional systems makes it a point of contention especially at medium to high loads.

Combined space partitioning and time sharing Numerous studies have indicated that a combination of space- and time- sharing is beneficial during scheduling when there is a large variance in the service demand. If instead, only run-to-completion space partitioning is used, there is a possibility for resources becoming scarce for smaller jobs as larger jobs tend to hold them for long durations, thus degrading performance to a large extent. Also, the size of allocated partitions tends to be smaller in most cases. This would deny performance benefits due to lack of processing power. In addition, there is the potential problem of fragmentation, during which processors idle even though there may be enqueued jobs waiting for processors. Under time sharing,

Adaptive load distribution When multiple queues schedule jobs, random or greedy approaches to job distribution potentially threaten the notion of load balance in the systems. Due to this imbalance, instances may arise wherein a few of the queues may be overly loaded with jobs while others may be starving at the same time. Naturally performance is greatly affected due to uneven utilization of processors. In the face of this, there is a need to distribute jobs in such a manner that the load on every queue is close to the average.

6.2 Contributions of this Thesis

In this thesis, we have made a preliminary attempt at deriving a novel scheduling policy called Hierarchical Scheduling Policy (HSP) that is (i) based on a hierarchy of task queues (I0), (ii) hybrid in nature – this is, time sharing multiple jobs on deterministic processor partitions (I1), and (iii) adaptively transfers work on a demand driven basis depending on the load and job characteristics (I2).

Based on the recent advances in workload characterization at high performance computing centres, we conducted a study to understand the impact these have on the scheduling schemes for achieving good performance.

A brief look at the different research initiatives into scheduling makes it apparent

that results are presented based on using different types of information for the various policies. This was the motivation for our desire to test HSP against one of the best known space partitioning policy of Rosti *et. al.* [RSS⁺94] (MAP2) and a time sharing policy of Leutenegger and Vernon [LV90] (RRTS) with appropriate matching features for fair comparison purposes. We made suitable modifications to these to fit into our system model. The modifications in fact go a long way into improving their performance when compared to their original versions.

For a unique method to generate workloads with features representative of recent observations and analyses, we used Sevcik's execution time function with proper choices of parameters. This made it possible to generate a wide range of workloads with varying characteristics that increased the validity of the results. Particularly, we generated four workloads WK1, WK2, WK3, and WK4 of which the first three had very different speedup characteristics while the fourth was a combination of them. The application structures we considered were matrix multiplication (WK1), mean value analyses (WK2), and NAS parallel benchmarking suite (WK3).

In an attempt to understand the numerous mechanisms of HSP, we decided to test it under a unitary workload model initially. For this the popular fork-and-join type of workload was chosen since it has been analyzed to a very large degree as well as used as a representative workload in many applications. Relative performance of HSP under this workload was made with RRTS and MAP2.

For evaluations under realistic conditions, a second set of experiments considered all the related overheads due to workloads such as parallelism overhead and communication and congestion delays. Performance evaluations were conducted to study the effectiveness of the three schemes on variations of a number of parameters. As part of this evaluation, experiments were conducted to evaluate the sensitivity of the schemes to large variations in service demand CVs, and application structures, etc. Based on results of extensive simulation experiments, we drew the following conclusions under medium to heavy system load conditions.

- MAP2 exhibited the worst performance for all the workloads and had the least sensitivity amongst the policies. This is because of the following three

reasons. One, jobs with super liner speedup, such as in WK1, are not provided sufficient processors. Hence they experience larger response times as their potential for using optimal number of processors is not generally satisfied. Two, internal fragmentation leads to poor utilization even when jobs are being enqueued waiting for processors. Three, additional overheads when number of processors allocated to jobs are either less or more than the required number inflate individual job execution times to a great extent. These are especially severe under large variation in service demands.

- RRTS performed much better in comparison to MAP2. This can be expected as it overcomes all the problems of space sharing implicitly. Its performance, however, suffers, due to increased access contention for the global queue. This becomes prominent at high loads with large variance in service demand CV. Finally, FCFS job transfer mechanism leads to large synchronization times especially for jobs with higher levels of parallelism at these loads. For both MAP2 and RRTS we have given the benefit of doubt in that processors in the system are treated as a 'pool of processors'. For instance, if the partition size is four and there are four processors in the system, the space sharing policy allocates these four processors to the partition no matter where they are located in the system. This type of allocation is fair for bus-based systems. For other kinds of interconnects such as mesh, etc. processor proximity is important even if high speed routing such as wormhole is employed.
- HSP tends to provide the best performance for all the parameters considered here. This is due to the presence and exploitation of the underlying HTQ organization. It is to be strongly emphasized that such an organization need not be present physically. For all purposes a logical representation serves as good a purpose as the physical one. This can be easily achieved through a software implementation, which, in fact is our aim. Lower response times for all workloads is achieved by scheduling jobs onto processors in a deterministic fashion. The identity and number are however determined adaptively based

on workload

6.3 What Lies Ahead?

For deriving and testing a hybrid scheduling scheme for distributed memory multi-computers, this thesis has been but a preliminary attempt. In order to implement this scheme and put it to use, a number of issues have to be taken into consideration. Below is a description of the aims and directions we want to embark upon.

- As much as processors are an important resource, due to the increasing number of applications and sizes being attempted, the memory requirements at individual nodes should also be considered during design of scheduling policies.
- The adaptive transfer of work has been shown to adjust to varying workload conditions. A natural way for further improving the performance of HSP is to test it under different values of T_r depending on the workload.
- There is a strong possibility for HSP to be applicable to workstation clusters as they can be looked at as loosely coupled distributed memory systems. In addition, from an organizational view point, it may be worthwhile to study the feasibility and performance benefits that may result if the task queues at each level were connected in some fashion, say ring. This provides a contingency for dynamic load sharing whenever the load within the system is severely imbalanced, possibly due to user interaction.
- In order to verify the simulation predictions, an implementation of the proposed algorithm on a message passing system is necessary. At the same time, this would provide additional knowledge for constructing future simulation experiments thereby saving enormous costs.

- Though Sevcik's execution time function is seen to represent many widely varying workloads, it has not been implemented so far. Potential sources for improvements may come up after an attempt in this direction.
- Since scheduling policies are very much influenced by the workload parameters, it seems advantageous to test it under additional workload types and mixes.
- Heterogeneous considerations could be incorporated especially when applying HSP in workstation clusters.

Researching these points would further enhance the behavior of HSP as well as take it many steps closer to using it as a general purpose scheme in a class of high performance computing systems.

References

- [AD96] S. Ayachi and S.P. Dandamudi. A hierarchical processor scheduling policy for multiprocessor systems. In *Proceedings of Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 100–109. IEEE Computer Society, October 1996.
- [Ana96] S.V. Anastasiadis. Parallel application scheduling on networks of workstations. Technical Report 342, Computer Systems Research Institute, University of Toronto, Toronto, Canada, 1996.
- [AVAP94] R. Arpaci, A. Vahdat, T.E. Anderson, and D.A. Patterson. Combining parallel and sequential workloads on networks of workstations. Technical Report 342, University of California, Berkeley, California, 1994.
- [BCG⁺96] M. Beirne, J.M. Chapin, M. Gantly, J.R. Matarese, and E. Sarto. Personal communication. 1996.
- [BH86] J. Barnes and P. Hut. A hierarchical $O(n \lg n)$ force calculation algorithm. *Nature*, 324(4):446–449, December 1986.
- [BKN95] R. Bhatia, S. Khuller, and J. Naor. The loading time scheduling problem (extended abstract). In *36th Annual Symposium on Foundations of Computer Science*, pages 72–81. IEEE Computer Society, October 1995.
- [CMV94] S.-H. Chiang, R.K. Mansharmani, and M.K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the ACM SIGMETRICS*

- Conference on Measurement and Modeling of Computer Systems*, pages 33–44. ACM Press, May 1994.
- [DC95] S.P. Dandamudi and P.S.P. Cheng. A hierarchical task queue organization for shared-memory multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(1):1–16, 1995.
- [DCDP90] K. Dussa, B. Carlson, L. Dowdy, and K.-H. Park. Dynamic partitioning in a transputer environment. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 203–213. ACM Press, May 1990.
- [DT97] S.P. Dandamudi and T.K. Thyagaraj. A hierarchical processor scheduling policy for distributed memory multicomputer systems. In *IEEE International Conference on High Performance Computing*, Bangalore, India, 1997. IEEE Computer Society.
- [Fei94] D. G. Feitelson. A survey of scheduling in multiprogrammed parallel systems. Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Yorktown Heights, NY, October 1994.
- [Fly66] M.J. Flynn. Very high speed computing systems. In *Proceedings of the IEEE*, pages 1901–1909. IEEE, 1966.
- [FN95] D.G. Feitelson and W. Nitzberg. Job characteristics of a production parallel scientific workload on the NASA Ames iPSC/860. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 337–360. Springer Verlag, 1995. D.G. Feitelson, and L. Rudolph (eds.).
- [FR90] D.G. Feitelson and L. Rudolph. Distributed hierarchical control for parallel processing. *IEEE Computer*, 23(5):65–77, May 1990.

- [FR96] D. G. Feitelson and L. Rudolph. Evaluation of design choices for gang scheduling using distributed hierarchical control. *Journal of Parallel and Distributed Computing*, 35(1):18–34, May 1996.
- [GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.
- [HL90] L.A. Herrbach and J.Y.-T. Leung. Preemptive scheduling of equal length jobs on two machines to minimize mean flow time. *Operations Research*, 38:487–494, 1990.
- [HLV92] J.A. Hoogeveen, J.K. Lenstra, and B. Veltman. Three, four, five, six, or the complexity of scheduling with communication delays. Technical Report MATH CENTRUM BS-R9229, Mathematisch Centrum. Centre for Mathematics and Computer Science, Dept. of Operations Research, Statistics, and System Theory, October 1992.
- [Hot96] S. Hotovy. Workload evolution on the Cornell Theory Center IBM SP2. In *Job Scheduling Strategies for Parallel Processing*, volume 1162 of *Lecture Notes in Computer Science*, pages 27–40. Springer Verlag, 1996. D.G. Feitelson, and L. Rudolph (eds.).
- [HvdVV92] J.A. Hoogeveen, S.L. van de Velde, and B. Veltman. Complexity of scheduling multiprocessor tasks with prespecified processor allocations. Technical Report MATH CENTRUM BS-R9211, Mathematisch Centrum. Centre for Mathematics and Computer Science, Dept. of Operations Research, Statistics, and System Theory, June 1992.
- [JPF⁺97] J. Jann, P. Pattnaik, H. Franke, F. Wang, J. Skovira, and J. Riordan. Modeling of workload in MPPs. In *3rd Workshop on Job Scheduling Strategies for Parallel Processing*, 1997.
- [LV90] S.T. Leutenegger and M.K. Vernon. The performance of multiprogrammed multiprocessing scheduling policies. In *Proceedings of the*

- ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 226–236. ACM Press, May 1990.
- [MEB88] S. Majumdar, D.L. Eager, and R.B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 104–113. ACM Press, May 1988.
- [MTR95] P. K. McKinley, Y.-J. Tsai, and D. Robinson. Collective communication in wormhole-routed massively parallel computers. *IEEE Computer*, 28(12):39–50, December 1995.
- [NSS95] V.K. Naik, S.K. Setia, and M.S. Squillante. Processor allocation in multiprogrammed, distributed-memory parallel computer systems. Technical Report IBM RC 20239, IBM Research Center, Yorktown Heights, NY, October 1995.
- [Ous82] J.K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of Third International Conference on Distributed Computing Systems*, pages 22–30, October 1982.
- [PS95] E.W. Parsons and K.C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In *Job Scheduling Strategies for Parallel Processing*, volume 949 of *Lecture Notes in Computer Science*, pages 127–145. Springer Verlag, 1995. D.G. Feitelson, and L. Rudolph (eds.).
- [RLA94] H. Rewini, T.G. Lewis, and H.H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Prentice Hall Inc., 1994.
- [RSS⁺94] E. Rosti, E. Smirni, G. Serazzi, L.W. Dowdy, and B.M. Carlson. Robust partitioning policies of multiprocessor systems. *Performance Evaluation*, 19:141–165, 1994.

- [Sev89] K.C. Sevcik. Characterizations of parallelism in applications and their use in scheduling. In *Proceedings of the ACM SIGMETRICS Conference*, pages 171–180. ACM Press, May 1989.
- [Sev94] K.C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19:107–140, 1994.
- [Sev96] K.C. Sevcik. Personal communication. 1996.
- [SRS⁺95] E. Smirni, E. Rosti, G. Serazzi, L.W. Dowdy, and K.C. Sevcik. Performance gains from leaving idle processors in multiprocessor systems. In *Proceedings of International Conference on Parallel Processing*. IEEE Computer Society, 1995.
- [SST93] S.K. Setia, M.S. Squillante, and S.K. Tripathi. Processor scheduling on multiprogrammed, distributed memory parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 158–170. ACM Press, May 1993.
- [SST94] S.K. Setia, M.S. Squillante, and S.K. Tripathi. Analysis of processor allocation in multiprogrammed, distributed-memory parallel processing systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):401–420, 1994.
- [TG89] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, December, pages 159–166. ACM Press, 1989.
- [TRS90] D. Towsley, C.G. Rommel, and J.A. Stankovic. Analysis of fork-join program response times on multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):286–303, 1990.

- [vECGS92] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 256–267. IEEE Computer Society Press, May 1992.
- [Wu93] C.-S. Wu. Processor scheduling in multiprogrammed shared memory NUMA multiprocessors. Technical Report 341, Computer Systems Research Institute, University of Toronto, Toronto, Canada, 1993.
- [XH96] Z. Xu and K. Hwang. Modeling communication overhead: MPI and MPL performance on the IBM SP2. *IEEE Parallel and Distributed Technology*, pages 9–23, 1996. Spring Issue.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved