1

PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A Hierarchical Processor Scheduling Policy for NUMA Systems

by

Samir Ayachi, B.E.(Hons)

**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

June 26, 1995.

©1995, Samir Ayachi

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

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

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

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

ISBN 0-612-08954-1

Canada

Name SAMIR AYACHI

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

0984

U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tesis and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0677
General 0289
Ancient 0290
Linguistics 0291
Modern
Literature 0401
General 0294
Classical 0295
Comparative 0297
Medieval 0298
Modern 0315
African 0591
American 0305
Asian 0352
Canadian (English) 0355
Canadian (French) 0593
English 0311
Germanic 0312
Latin American 0315
Middle Eastern 0313
Romance 0314
Slavic and East European

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical
Business Administration 0310
General 0272
Accounting 0770
Banking 0454
Management 0338
Marketing 0385
Canadian Studies
Economics 0501
General 0503
Agricultural 0505
Commerce Business 0508
Finance 0509
History 0510
Labor 0511
Theory 0358
Folklore 0366
Geography 0351
Gerontology
History 0578
General

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 012
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology 0306
General 0287
Anatomy 0308
Biostatistics 0309
Botany 0379
Cell 0329
Ecology 0353
Entomology 0369
Genetics 0793
Immunology 0410
Microbiology 0307
Molecular 0317
Neuroscience 0416
Oceanography 0433
Physiology 0821
Radiation 0778
Veterinary Science 0472
Zoology
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0792
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry 0485
General 0749
Agricultural 0486
Analytical 0487
Biochemistry 0488
Inorganic 0738
Nuclear 0490
Organic 0491
Pharmaceutical 0494
Physical 0495
Polymer 0754
Radiation 0405
Mathematics
Physics 0605
General 0986
Acoustics
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463
Applied Sciences
Applied Mechanics 0346
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of the Thesis

“A Hierarchical Processor Scheduling Policy for NUMA Systems”

submitted by Samir Ayachi, B.E.(Hons) in partial fulfillment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Director, School of Computer Science

Carleton University

July 12, 1995

Abstract

In a multiprogrammed multiprocessing system it is necessary to devise a mechanism for allocating processors to arriving jobs as well as for determining the policy to be used by the processors to execute the separate tasks in its local ready queue; the objective being the minimization of the overall response time and maximization of average processor utilization.

Several policies have been suggested for achieving the above goal. These policies are either time sharing policies, in which processors execute tasks for fixed time quantum, in a round robin manner, or space sharing policies in which each job is assigned to a set of processors.

In this thesis a new processor scheduling policy is proposed, which uses a hierarchical task queue organization and combines the desirable features of space sharing and time sharing to achieve better performance than most other existent policies.

Acknowledgments

I would like to take this opportunity to express my gratitude to Dr. Sivarama Dandamudi, my thesis supervisor, for his support and guidance during the course of my work. He devoted a lot of his time for regular discussions about the work presented here, without which this thesis could not have been completed.

I would also like to thank the School of Computer Science and Carleton University for giving me the opportunity to work here and providing me with financial support.

Contents

1	Introduction	2
1.1	Background	2
1.1.1	Multiprocessor Systems	2
1.1.2	Task Queue Organization	8
1.1.3	Workload Models	12
1.1.4	The Multiprocessor Scheduling Problem	14
1.2	Organization of the Thesis	16
2	Process Scheduling in Multiprocessors	17
2.1	Time Sharing Policies for Shared Memory Systems	18
2.1.1	Co-scheduling Policies	19
2.1.2	Round Robin Policies	21

2.1.3	Shortest Job First Policies	25
2.2	Space Sharing Policies for Shared-Memory Systems	27
2.2.1	Static Partitioning Policies	27
2.2.2	Dynamic Partitioning Policies	32
2.3	Scheduling Policies for Distributed Memory Systems	37
2.3.1	Adaptive Multiprogrammed Partitioning	38
2.3.2	Equipartition and Folding	40
2.4	Motivation for this Thesis	42
3	A Hierarchical Scheduling Policy for Shared Memory Systems	45
3.1	The System Model	47
3.2	The Workload Model	48
3.2.1	The MCS Lock Model	52
3.3	The Hierarchical Queue Organization	55
3.4	A Simulation Model for the Hierarchical Scheduling Policy	58
3.5	Simulation Models for Other Policies	65
3.5.1	Modified RRjob	65
3.5.2	Equipartitioning	67

3.6	Original RRjob	69
3.7	Validation of Simulation Models	70
4	Independent Workload Model Results	71
4.1	Effect of varying the scheduling overhead	73
4.2	Effect of varying the C_{v_d}	80
4.3	Effect of ERF factor	84
4.4	Effect of average parallelism	87
4.5	Effect of the round robin quantum size	89
4.6	Effect of the multiprogramming level	92
4.7	Effect of branching factor on HSP	95
4.8	Summary of Results	96
5	Dependent Workload Model Results	98
5.1	Effect of varying the scheduling overhead	100
5.2	Effect of varying the C_{v_d}	104
5.3	Summary of results	108
6	Conclusions	110

6.1	Summary	110
6.2	Contributions of this thesis	112
6.3	Areas for future work	113
Bibliography		115

List of Figures

1.1a	A typical shared memory architecture	5
1.1b	MIMD Distributed memory architecture	5
1.2	A shared memory UMA architecture	6
1.3	An 8×8 Omega MIN, routing a P3 request to M3	7
1.4	The BBN Butterfly interconnection network	8
1.5	Centralized queue organization	11
1.6	Distributed queue organization	11
1.7	Hierarchical queue organization for 4 processors with binary branching	12
1.8	The Fork and Join job structure	14
2.1	Mean response times for some time sharing policies	25
3.1	Example curves for ERF $\gamma(k) = \frac{(1+\beta)k}{k+\beta}$	50
3.2	A flow chart depicting one iteration of a task's execution	52

3.3	The MCS locking model	54
3.4	Task transfer process in the hierarchical organization	56
3.5	Hierarchical queue organization for four processors	58
4.1	Response Time Vs. Utilization for low overhead.	75
4.2	Response Time Vs. Utilization for medium overhead.	77
4.3	Response Time Vs. Utilization for high overhead.	77
4.4	Response Time Vs. Overhead, at 72% utilization.	79
4.5	Response Time Vs. Utilization for Demand $C_v = 1$	81
4.6	Response Time Vs. Utilization for Demand $C_v = 3$	82
4.7	Response Time Vs. Utilization for Demand $C_v = 5$	82
4.8	Response Time Vs. Demand C_v , at 72% utilization.	84
4.9	Response Time Vs. ERF Factor, (β) , at 50% utilization.	85
4.10	Response Time Vs. ERF Factor, (β) , at 75% utilization.	86
4.11	Response Time Vs. Avg. Parallelism, (avg) , at 50% utilization.	87
4.12	Response Time Vs. Avg. Parallelism, (avg) , at 75% utilization.	89
4.13	Response Time Vs. RR Quantum at 50% utilization.	90
4.14	Response Time Vs. RR Quantum at 75% utilization.	91

4.15	Response Time Vs. MPL at 50% utilization.	93
4.16	Response Time Vs. MPL at 75% utilization.	94
4.17	Response Time Vs. Utilization for different BR values.	95
5.1	Response Time Vs. Utilization for low overhead.	100
5.2	Effect of synchronization on task execution times.	101
5.3	Response Time Vs. Utilization for medium overhead.	102
5.4	Response Time Vs. Utilization for high overhead.	103
5.5	Response Time Vs. Overhead, at 72% utilization.	104
5.6	Response Time Vs. Utilization for $C_{v_d} = 1$	105
5.7	Response Time Vs. Utilization for $C_{v_d} = 3$	106
5.8	Response Time Vs. Utilization for $C_{v_d} = 4$	107
5.9	Response Time Vs. C_{v_d} at 72% utilization.	107

Chapter 1

Introduction

1.1 Background

This thesis deals with the problem of scheduling in multiprogrammed multiprocessing systems. This chapter presents a brief introduction to the main features of multiprocessing systems that are relevant to the work presented here.

1.1.1 Multiprocessor Systems

Medium to large scale multiprocessing systems are now commonly used in numerous scientific and industrial applications. These systems can have anywhere from less than a hundred to several hundred processors, and are typically multiprogrammed, i.e. several applications can be executed simultaneously on the system.

Numerous classifications of parallel processing systems have been proposed but the most commonly used taxonomy scheme was introduced by M. J. Flynn in 1972 [14]. Since that time there have been revolutionary changes in the architecture of parallel or multiprocessor systems and Flynn's scheme can no longer be used to differentiate clearly between the various architectures now present [31, 12]. However, his classification scheme is still commonly used in the literature and will be briefly described in the following paragraphs.

Flynn's taxonomy divides systems into four broad classes based on the number of instruction streams and data streams that they can operate on at the same time. These classes are the following [12] :

- *SISD* (single instruction, single data stream): This class consists of sequential computers.
- *MISD* (multiple instruction, single data stream): This a hypothetical class of computers in which each processor in the system executes a different set of instructions on the same datum. Such systems are not considered practical to implement.
- *SIMD* (single instruction, multiple data streams): This is a class of multiprocessors in which, typically, a central control unit broadcasts a single instruction to all processors, which execute the instruction in lockstep fashion on local data. Array processors like the Illiac-IV are examples of this class of multiprocessors.

- *MIMD* (multiple instruction, multiple data streams): This is the class of multiprocessors that are the subject of this thesis. MIMD systems consist of processors that operate in an autonomous manner. They are asynchronous computers, characterized by decentralized hardware control, in which the software processes executing on the individual processors may be synchronized by *passing messages* through an interconnection network or by accessing data in *shared memory* units.

As can be seen from the above definition, MIMD architectures can be further classified as *shared memory systems* or *distributed memory systems* with the main difference between them being that in the latter category processors can access contents of memory modules associated with other processors by message passing only. The Intel hypercube, iPSC/860 is an example of such a system. In shared-memory systems, on the other hand, all processors share a common address space even though the various memory modules may or may not be physically distributed. Several commercial and experimental machines belong to this category. BBN's Butterfly, IBM's RP3, NYU's Ultracomputer, DEC's Firefly, the Stanford DASH, and the Illinois Cedar are some common examples. A schematic of a typical shared memory system, also referred to as a multiprocessor, is shown in Figure 1.1(a). A distributed memory system, also sometimes referred to as a multicomputer is shown in Figure 1.1(b).

Figure. 1.1(a) A typical shared-memory architecture.

Figure 1.1(b). MIMD distributed memory architecture, [12].

In figure 1.1(a) a block labeled “interconnection network” is shown to provide the connection between the processors and memory modules. Actually, there are three main mechanisms for connecting processors to memory modules in shared memory

systems [12]. Based on the manner in which the memory modules of shared memory systems are organized, these systems are further classified as Non-Uniform Memory Access (NUMA) or Uniform Memory Access (UMA) systems.

Bus based architectures might be thought of as being UMA systems because in these systems all processors have the same memory access times. A typical UMA architecture is depicted in Figure 1.2.

Some UMA systems also use crossbar interconnections, in which a crossbar switch of n^2 crosspoints is used to connect n processors to n memories. A crossbar interconnection scheme prevents contention for communication links by providing a dedicated pathway between each possible processor-memory pair. However, power, pinout and size considerations limit the scalability of systems that use this interconnection mechanism.

Figure. 1.2. A Shared Memory UMA Architecture.

The third interconnection mechanism used in shared memory systems, is referred to as *multistage interconnection network*, or MIN, and systems that use this interconnection scheme fall in the class of shared memory NUMA systems if each processor has a local memory and is able to access the local memories of other processor with the help of the MIN. In a such a system, memory modules are interconnected with

processors in a manner that the delay involved in accessing a module is different for different processor-memory pairs, with the minimum delay being incurred when a processor accesses its own associated memory. One example of a MIN is an omega network, depicted in Figure 1.3. In a $N \times N$ MIN, N processors are connected to N memories using multiple stages of switches in the interconnection network pathway. For instance, if N is a power of 2, $\lg_2 N$ stages of $N/2$, 2×2 switches can be used. A processor that requires to access a memory module specifies the desired destination by issuing a bit-value that contains a control bit for each stage. The switch at stage i examines the i th bit to determine whether the request is to be routed to the upper or lower output of that switch.

Figure 1.3. An 8 X 8 Omega MIN, routing a P3 request to M3, [12].

The omega network shown in Figure 1.3 consists of eight processors and eight memories. A control bit of zero indicates a connection to the upper output. In the

figure, the pathway indicated by the dark line indicates how processor P_3 accesses the memory module M_3 .

An example of a commercial shared memory NUMA system is the BBN Butterfly. Figure 1.4 depicts how the processors and memory modules are connected in the BBN Butterfly. Only five processor-memory pairs are depicted, though it should be noted that this system can be configured with up to 256 processors.

Figure.1.4 The BBN Butterfly Interconnection Network, [2].

In this thesis our focus is on medium sized shared-memory NUMA systems.

1.1.2 Task Queue Organization

Consider a scenario of a shared-memory multiprocessing system where parallel applications (jobs) arrive continuously for service. In this thesis, the term *job* refers to any

parallel application. Each job might consist of several sequential units of computation. The smallest independently schedulable segment of computation is referred to as a *task*. When the system load is moderate to heavy, jobs will have to wait for some time before they can be assigned to processors. This means that arriving jobs need to be queued somewhere in the system while they are waiting. There are three main strategies used for queueing arriving jobs in a multiprocessing system. These are :

- *Centralized organization*, in which all arriving jobs are enqueued in a single shared queue (see Figure 1.5). Tasks are allocated to the processors from this queue. In the absence of access contention, the performance of this queue is superior to other queue organizations due to perfect load sharing. However, it is impossible to eliminate queue access contention in real systems, and in such a case the central queue becomes a bottleneck due to mutually exclusive access and its performance deteriorates considerably for moderate to heavy system loads and for large systems.
- *Distributed organization*, in which there is a job queue associated with each processor, see Figure 1.6. Central queue access contention is eliminated in this organization, but this scheme has several disadvantages. The main drawback with this scheme is that it causes load imbalance due to the distributed nature of the queues. Also, studies in [22] and [8] have shown that in some implementations, the overheads associated with the additional data structures and task allocation policies required for this scheme might outweigh the benefit that is

obtained by eliminating queue access contention.

- *Hierarchical organization*, has been recently proposed in [9]. In this scheme the central queue is replaced by a tree of queues, see Figure 1.7. All jobs arrive and are queued at the root node. The processors comprise the leaf level of the tree. The tree nodes serve as intermediate queues. Whenever a processor becomes idle, it searches the tree node immediately above it in the hierarchy, for work. The search continues up the tree till tasks are found. Each processor that finds tasks carries down more tasks than it can process, so that other processors in its sub-tree do not have to traverse the entire hierarchy again. This organization eliminates contention for a single queue and is scalable to a large number of processors. Experiments in [9] indicate that the performance of this organization is superior to the distributed organization and is very close to the performance of the ideal situation of the centralized organization. This is expected, since the hierarchical organization provides excellent load sharing, similar to the centralized organization, but does not have the queue access contention that is necessarily associated with the latter scheme. Also, even though there are multiple queues in the hierarchical scheme, as there are in the distributed organization, the hierarchical nature of the queue reduces the number of times a processor needs to access a queue in order to schedule a task.

Figure. 1.5 Centralized Queue Organization.

Figure. 1.6 Distributed Queue Organization

Figure. 1.7 Hierarchical Queue Organization for Four Processors with Binary branching.

Scheduling policies that use centralized and hierarchical queue organizations are considered in detail in the thesis.

1.1.3 Workload Models

So far, two components of a multiprocessing system have been discussed — the multiprocessor and the job queueing models. A third component is the job itself. There are many different kinds of parallel applications and several studies have been done to characterize the various classes of parallel applications for use in evaluation of scheduling policies [28, 3]. For the purposes of this thesis it will suffice to consider the following classification of parallel jobs:

- *Independent Job Model*, in which a job can be thought of as being composed of N equivalent, independent tasks that do not communicate during the course of their execution and synchronize with the other tasks on completion, as shown in Figure 1.8. According to Towsley [33], this *fork-join* type of job structure is a reasonable model for a large number of parallel applications, e.g. parametric investigation of numerical models.

Figure 1.8 The Fork and Join Job Structure.

- *Dependent Job Model*, in which the tasks belonging to an application must communicate or synchronize with each other, or require exclusive access to a shared data structure. The former case is often referred to as *barrier synchronization*, where a task must wait after reaching a certain point in its computation, until all its peer tasks have reached that point or *barrier*. In the case where the tasks need access to a shared data structure a mechanism for locking the data in order to ensure mutually exclusive access is required [21]. Tasks waiting to acquire the lock can go into a busy-wait loop till the lock is available or can be blocked.

Studies have been done to compare busy-waiting (spinning) and blocking in [16] and [5]. It has been observed that in a multiprogrammed environment, blocking synchronization is preferred over busy waiting or spinning since it allows tasks of other jobs to be scheduled while the current task is not able to proceed due to synchronization constraints. The study in [16] indicates that an initial, brief period of spinning followed by blocking improves the performance of most applications.

Both classes of jobs mentioned above will be used to evaluate the performance of the scheduling policies that are studied in this thesis.

1.1.4 The Multiprocessor Scheduling Problem

Now that all the relevant components of a multiprocessing system have been discussed, the problem of multiprocessor scheduling can be described in greater detail. Given a multiprocessor system, possibly consisting of a large number of processors, and a steady stream of parallel applications arriving to be serviced, an efficient mechanism is required to allocate the tasks of each application to processors in a manner that the average response time of the applications is minimum and the average utilization of the system is maximum. Two main approaches to scheduling — time-sharing and space-sharing — have been studied and analyzed quite extensively in the literature.

In time-sharing systems a set of processors are allocated to each waiting job for a fixed period of time or *quantum*, in a round robin manner. Several variations of the

basic round robin policy exist. For example, the quantum length could be a constant for all tasks or could be dependent on the size of the job. Some common time-sharing policies will be reviewed in the next chapter.

In space-sharing, also known as partitioning, processors are divided between all waiting jobs. Usually all jobs are allocated at least one processor and if all processors are exhausted, the remaining jobs are queued, but again there can be several variations to this policy. For example, allocating at least a certain minimum number of processors (maybe greater than one) to each job. Also, partitioning could be *static*, in which the allocation, once made, does not change until the job departs, or *dynamic*, in which periodic re-allocation of processors is done as jobs arrive, depart or their requirement for processors changes. In [4], Brecht considers partitioning to be composed of two separate problems :

- The allocation problem, in which the scheduler must decide how many processors are to be allocated to each application, and
- The placement problem, in which a decision must be made regarding which processors to allocate to each application.

Some common space-sharing policies will be studied in some detail in the next chapter.

It can be seen from the discussion so far that multiprocessor scheduling is not a problem that lends itself to a straightforward solution. Several factors determine the overall performance of any policy and there have been a number of studies to evaluate

the performance of different scheduling policies on a variety of multiprocessing systems and workloads. As mentioned earlier, some of these studies will be reviewed in the following chapter.

1.2 Organization of the Thesis

The rest of this thesis is organized as follows.

Chapter 2 describes the previous research in the area of multiprocessor scheduling. Greater emphasis is placed on the studies that are directly related to the work presented here.

In Chapter 3, a detailed description of the hierarchical scheduling policy (HSP) is presented. The system, workload and locking models used are described and the actual steps involved in making task allocation decisions are explained.

Chapter 4 presents the results of simulation experiments carried out to evaluate the performance of the hierarchical scheduling policy for the independent job model. Chapter 5 contains results for the dependent job model.

Contributions of this thesis are summarized in Chapter 6. Suggestions for future research and concluding remarks are also presented in Chapter 6.

Chapter 2

Process Scheduling in Multiprocessors

Due to the existence of a large variety of multiprocessing systems and numerous kinds of parallel applications, the problem of scheduling jobs on multiprocessors has become extremely diverse. This implies that it is difficult, if not impossible, to find a general algorithm that will be optimal for all systems and workloads. Therefore, any research that attempts to provide a scheduling policy for a multiprocessor system, must limit its scope to a subset of existing systems and applications.

Work in this area has focused mainly on shared-memory and to some extent on distributed-memory systems. Various kinds of real and synthetic workloads have been used to evaluate the performance of the proposed policies. The scheduling policies can be classified into space sharing or time sharing policies. Most studies have assumed

a centralized queue organization, as described in the previous chapter.

In the following sections a review of the research that has been done in the last few years in the area of multiprocessor scheduling is presented.

In all the work that has been done in this area as well as in this thesis the term “performance” refers specifically to two metrics — the average job response time and the average system utilization. How these metrics are actually calculated is described in the next chapter.

2.1 Time Sharing Policies for Shared Memory Systems

In this section some scheduling policies that have been proposed for both UMA and NUMA shared-memory multiprocessing systems will be reviewed. The simplest policy is First Come First Serve (FCFS), in which tasks of arriving jobs are appended to a central queue. When a processor becomes idle it removes the task at the head of the queue and runs it to completion. This policy has the worst performance for most offered workloads because jobs have to wait for long durations before they can receive any processor time especially if larger jobs are in the queue in front of smaller jobs [17]. Since there is no sharing of resources between jobs in FCFS it also leads to low system utilization.

2.1.1 Co-scheduling Policies

One of the earliest attempts at devising a scheduling policy for systems with several processors, executing more than one parallel application at a time was made by Ousterhout, [23]. He proposed a scheme that is known as *co-scheduling*, intended initially for the Medusa OS on the Cm* multiprocessor. In this scheme, all the tasks of a parallel application were scheduled to run at the same time, for a fixed quantum ¹. At the end of the quantum, all tasks were preempted simultaneously and the tasks of the next application were then scheduled. Thus all waiting applications were scheduled on the available processors in a round robin manner. The advantage of this scheme was that, since all the tasks of an application were running at the same time, synchronization delays were reduced to a minimum.

However, as it stands, there are several obvious drawbacks in this scheme. For instance, if the number of tasks in a job exceed the number of available processors ², then any synchronization between tasks will be delayed and performance will deteriorate considerably. On the other hand, if the number of tasks in a job are less than the number of processors available and the next job requires more than the remaining processors then part of the system will remain idle for that time slice. Another drawback, that has been mentioned earlier with reference to centralized queues, is that at moderate to heavy system loads, the central scheduler becomes a bottleneck.

¹Note that the terms quantum and time slice are used interchangeably throughout this thesis.

²Most studies bound the number of tasks per job by the number of processors in the system, to eliminate the mentioned problem.

The original co-scheduling policy was modified in [17]. In this study, Leutenegger and Vernon compared the performance of two modified versions of Ousterhout's scheme with several other policies. Their first modified scheme, known as *Cosched*, works in the following manner. The tasks of all arriving jobs were appended to a single central queue. A slotted window in which the number of slots was equal to the number of processors was moved along the list and all tasks that fell under the window were allotted one time slice on a processor. At the end of the time slice the window was moved till it coincided with the first task of an application that was not completely scheduled in the previous time slice. This modified scheme attempted to reduce some problems with the original scheme. In this version of coscheduling, none of the processors were idle during any quantum, and since jobs that were not completely scheduled in one quantum were scheduled again in the next quantum, response times of these jobs was reduced as they were getting more processor time than before.

The second modified co-scheduling scheme in [17], known as *Cosched2*, was the same as *Cosched*, except that at the end of a time slice the processor window was moved to the first task of the next job in the list even if that job had been completely co-scheduled in the previous time slice. The effect of this modification was that smaller jobs (i.e. those with fewer tasks), received multiple time slices and this offset the detrimental effect of larger jobs dominating the processors.

Simulation experiments in [17] showed that for several workloads, which will be

described later in the thesis, *Cosched2* performed better than *Cosched*, but in general they both performed worse than other policies that allocated equal processing power to all jobs, irrespective of the size of the job.

Another study that looks at the performance of co-scheduling, also referred to as *gang-scheduling*, is [16]. In this study co-scheduling was compared with several other policies, including priority scheduling, process control with processor partitioning, handoff scheduling and affinity-based scheduling. This study noted that a major cause of performance deterioration was the number of tasks in a job exceeding the number of processors allocated to it and so the main aim of this study was to try and improve the performance of such applications. Their proposed solution will be examined in a later section. In the group of policies studied in [16], gang-scheduling demonstrated better performance than most other policies since it was not as affected by blocking or spinning synchronization primitives as the other policies. This was due to the fact that all tasks that were coscheduled were preempted at the same time.

2.1.2 Round Robin Policies

This is an important class of time sharing policies with a few variants that in general exhibit better performance than co-scheduling and are competitive with some of the better policies known till now.

Majumdar et. al., studied the performance of a round robin policy, that they termed *RR*, now generally referred to as *RRprocess*, in [18]. This study was an

early attempt at evaluating the performance of several scheduling policies for multi-programmed systems. A number of policies were studied and compared by varying factors like the job parallelism, variability in cumulative job demand, and context switch rate.

In the *RR* policy, all tasks of arriving jobs were appended to a central queue, and processors removed a task from the head of the queue and ran it for a fixed quantum, at the end of which the task was placed at the end of the list and the next task at the head was removed for servicing.

An important contribution of this study was the definition of two workload classes that were used in later studies like [17, 7], and is also used by us, with some minor modifications. The two types of workloads used were the following :

- *Workload A*, that generated jobs given the input parameters — the mean number of tasks per job, the coefficient of variation of the mean number of tasks, the mean cumulative demand per job and its coefficient of variation. The cumulative demand for the job was generated independently of the number of tasks in the job, and then divided uniformly among all tasks. Previous research has shown that for many scientific applications, e.g. those that involve matrix operations, uniform task demands are fairly accurate representations of real workload conditions [19].
- *Workload B*, was similar to the above, except that there was a linear correlation between the number of tasks in the jobs and its cumulative demand. The

demand was generated as the product of the number of tasks and an input parameter, t .

The number of tasks and the cumulative demand for each job were generated using a two-phase hyperexponential distribution that will be described in greater detail in the next chapter.

Scheduling policies were also classified in [18] as those that used no information about job characteristics, like FCFS and RR, and those that did, like Smallest Number of Processes First (SNPF) and Shortest Cumulative Demand First (SCDF).

Several simulation experiments were carried out, each varying a different parameter and studying its impact on the policies. The study concluded that the RR policy was better than FCFS but performed worse than the other policies studied, i.e., SNPF and SCDF, indicating that use of job characteristics in scheduling decisions improved performance to a great extent.

Later, in [17], another round robin policy, referred to as *RRjob*, was studied. Again, like in the previous work, a comparison of several policies was made under varying conditions and for different workloads. As mentioned earlier, the workload from [18] was used here and the same set of policies as [18] were compared with the addition of *RRjob* and a space sharing policy known as *dynamic partitioning* [34].

In the *RRjob* policy, studied in [17] for the first time, processing power was allocated equally to all jobs, unlike *RRprocess*, in which all tasks were allocated equal

time slices. Round robin quanta of different sizes were allocated to each job. The quantum size for each job was decided based on the number of tasks in the job. For example, if a job had N tasks, and was running on a system with P processors, each task was allotted a quantum of size Q , where Q was determined as follows:

$$Q = (P/N) \times q,$$

Here q is the fixed quantum size allotted to jobs in which $N = P$. It can be seen from the above equation that jobs with smaller parallelism get a larger share of processor time, while jobs with a large number of tasks were proportionately allotted smaller time slices.

The effect of this modification was that jobs with smaller number of tasks finished faster than in RRprocess where large jobs held the processors for most of the time and smaller jobs were starved.

Providing equal processing power to all jobs had a great impact on system performance and simulation experiments in [17] showed that of all policies studied, RRjob exhibited the best performance, except for the space sharing dynamic partitioning policy.

In Fig. 2.1, the performance of all time sharing policies that have been discussed so far are compared. It can be seen from the figure that in order of performance the policies can be ranked from best to worst as — RRjob, Co-scheduling policies, RRprocess and FCFS. The workload on which this graph is based is a lock accessing or

synchronizing workload with job parameters like parallelism and cumulative demand being generated from a hyperexponential distribution, [17].

Figure. 2.1 Mean Response Times for some time-sharing policies, [17].

2.1.3 Shortest Job First Policies

Two policies that have been mentioned briefly in the previous sections, but not discussed in detail are Smallest Number of Processes First (SNPF) and Shortest Cumulative Demand First (SCDF) and their corresponding preemptive versions — PSNPF and PSCDF. Both these policies were first suggested in [18] and analyzed further in [17].

In the SNPF policy, the processors are allocated to the job that has the smallest number of runnable processes. Processes of all jobs are inserted into a central process queue, which is organized as a priority queue. The highest priority is given to jobs

with the smallest number of tasks. When processors become idle, they simply remove the task at the head of the queue and execute it to completion. In the preemptive version of this policy, if a new job has a smaller number of tasks than the currently executing job, then the latter is preempted and the new job is scheduled to run.

SCDF and PSCDF are similar to SNPF and PSNPF, except that the criterion for scheduling jobs is their remaining cumulative service demand, not the number of tasks in the job.

Analysis of these policies in [17] indicates that SNPF performs very poorly under most conditions and is better only than FCFS. On the other hand, PSCDF has the best overall performance of all policies studied in [17], but is of theoretical interest only, because it is usually not possible to satisfactorily estimate the cumulative demand of parallel applications.

So far, several time-sharing policies have been discussed in this section and results from [17] indicate that of all policies considered till now, *RRjob* exhibits the best performance, for all workload classes used. However, it is worth mentioning here that in [17], the effect of overheads like context switching delay and queue access contention are not considered and the study assumes an ideal multiprocessor in which the central queue is not a bottleneck. In a real system, however, access contention is the major cause of performance deterioration [9]. In our work, we study the effect of queue access contention and scheduling overhead on a modified version of the *RRjob* policy. The original *RRjob* from [17] is also studied, but is used mainly to compare the

performance of other policies with an "ideal" system.

2.2 Space Sharing Policies for Shared-Memory Systems

Space sharing, or partitioning policies are the second class of scheduling policies for multiprocessor systems. Again, there are several such policies that have been proposed and evaluated in the literature [19, 36, 16, 34, 7, 39]. It has been observed that the best space sharing policies known exhibit better performance than their time sharing counterparts like *RRjob* when the variance in job service times is low.

The concept of space sharing is the division of the multiprocessor into slots, with each slot consisting of one or more processors. Arriving jobs are then allocated to the most suitable slot. The policy must decide the size of each slot ³, and which job to assign to which slot. How these decisions are made differentiates one policy from another.

2.2.1 Static Partitioning Policies

Obviously, the simplest way to partition a set of processors would be to divide them into groups of equal size at startup and then maintain the same organization through-

³All studies assume a homogeneous multiprocessing system, i.e., one in which all processors are equivalent.

out the time that the system is up. All arriving jobs would be assigned to the group with the shortest queue length, and once this allocation was made, it would not change.

A possible variation of this policy would be to allocate a fixed number of processors to each arriving job, based on the jobs requirement, i.e. the number of tasks in the job. This allocation would remain constant throughout the execution of the job. If no processors are available the remaining jobs would be queued. This scheme is similar to the one described above, except that the partitions are not necessarily of equal sizes, thus improving system utilization. This policy has been studied in [36], where it is referred to as *static allocation*.

In [36], Zahorjan and McCann compare the performance of two static allocation policies with a dynamic allocation scheme in which the number of processors allocated to a job might change if new jobs arrive or if the current job's requirement for processors increases or decreases.

Static schemes are attractive because they are easy to implement and have very low scheduling overheads. However, they are not usually favored due to their tendency to waste system resources leading to very poor response times at moderate to high system load. The study in [36] shows that static policies have poor performance compared to dynamic allocation. The two static allocation schemes that they look at are Run-To-Completion (RTC) and Round-Robin. In the RTC⁴ policy, a task that is allocated

⁴Note that the RTC policy is different from the FCFS policy in that the former is a space sharing policy in which several jobs might execute at the same time in different system partitions.

a processor, runs there till completion. In the Round-Robin scheme, the same set of processors are rotated among a set of waiting jobs in a round robin manner. RTC is shown to have better performance than Round-Robin at most parameterizations of the model in [36] due to the fact that it has no context switching overheads.

There are two other studies, [15] and [7], devoted in most part to the study of static allocation policies. The concept of *pws*, or Processor Working Set, is introduced in [15], as a means of characterizing parallel workloads. It is then used to compare a set of scheduling policies that make their decisions based on the knowledge of the *pws* for each job. The *pws* is defined as “the minimum number of processors that maximizes the speedup per unit of a cost function that incorporates the number of processors used and the associated speedup.”

The scheduling policies compared in [15] are the following:

- *First Fit (FF)*, in which the scheduler picks the first job from the queue of waiting jobs whose *pws* is less than or equal to the number of available processors, and schedules that job on the processors, where they run to completion. This policy is not *work conserving* since it has the tendency to leave processors idle while there are still jobs waiting for service.
- *FF + FIFO*, is a work conserving variation of the above scheme. After all jobs whose *pws* can be accommodated on the available processors are scheduled, the scheduler picks the first job from the central queue and allocates the remaining processors to that job.

- *FF + LA*, in which a job is allocated all available processors, even if the number exceeds the job's *pws*, provided that there are no other jobs behind it in the queue. (LA is an abbreviation for Largest Available partition).
- *FF + FIFO + LA*, an extension of the *FF + FIFO* policy with the added feature of providing the largest available partition to the job that is granted the remaining processors, if there are no more jobs behind it in the queue.
- Equal Partitions, in which the processors are divided statically into partitions of equal sizes and jobs are allocated to partitions irrespective of their *pws*.

This study concludes that the *pws* is a robust index for characterization of parallel programs. It allows an optimal number of processors to jobs at low to moderate loads and then fragments the system into smaller and smaller processor sets as the system load increases. The simulation experiments in [15] indicate that of all policies considered *FF + FIFO* demonstrates the best overall performance.

Another recent study that examines the effect of using application characteristics for improving the performance of RTC policies, and compares a number of static RTC policies with a dynamic equipartition policy is [7]. The effect of using the job's average parallelism (*avg*), or *pws*, in making scheduling decisions for RTC policies is analyzed in this paper, along with the effect of limited preemption.

The static policies that are looked at in this work are :

- *ASP*, or Adaptive Static Partitioning, [27], in which arriving jobs are allocated a

certain number of processors, with this number never exceeding the job's average parallelism. New jobs are queued behind all waiting jobs if no processors are available, and once processors become available all waiting jobs are allocated one processor each, starting at the first job in the queue.

- *PWS*, or *FF + FIFO*, described above.
- *AVG*, which is the same as above except that the index used for scheduling decisions is *avg*, not *pws*.
- $A+\infty M$, [28], in which jobs are scheduled in a FCFS manner. The number of processors allocated to a job is the minimum of the number available and a parameter α , which is a function of the job's average, minimum and maximum parallelism.
- Adaptive-AVG, a new policy proposed in [7], which is similar to ASP, except that when processors become available, they are allocated to jobs in proportion to their *avg* or *pws*.

This detailed study concludes that using a maximum allocation constraint on the number of processors assigned to a job, based on the system load, improves the performance of RTC scheduling, more than the use of application execution rate characteristics like *avg* or *pws*. It is further demonstrated that allowing the scheduler to preempt tasks at least once improves the response time of RTC policies, but does not make them competitive with dynamic allocation policies that are studied next.

2.2.2 Dynamic Partitioning Policies

This is an important class of scheduling policies for multiprocessor systems and many policies proposed under this category are known to exhibit the best performance measures till now.

In the previous section, reference was made to [36], in which static allocation policies were compared with a dynamic partitioning policy for varying levels of scheduling overhead. It was shown in this study that dynamic partitioning performed much better than its static counterparts for all levels of scheduling overheads.

The dynamic partitioning policy in [36] works in the following manner. When a job arrives, it makes its processor requirement known to the central scheduler. Idle processors, if any, are allocated to the job, not exceeding the number of processors requested. If there are no free processors then a processor is taken from a running task and allocated to the new job, so that to the extent possible all jobs have at least one processor. If more jobs arrive, they are added to the queue of waiting jobs. Processors might be freed when the requirement of a job decreases or when a job departs. In both these cases, the scheduler scans the list of waiting jobs and allocates one processor to each waiting job. If any more processors remain after this, they are allocated to the waiting jobs in a round robin manner, starting at the head of the job queue.

This policy is sensitive to the scheduling overhead, but demonstrates good performance for low overhead values. It is consistently better than the static allocation

policies.

An extension of the work in [36] can be found in [19]. In this work, a dynamic policy, known simply as *Dynamic* is proposed by the authors and compared with the *RRjob* policy of [17] and a modified version of the *process control* policy of [34], referred to as *Equipartition*. In the *Dynamic* policy the allocation of processors to jobs is altered based on changing requirements of jobs. Jobs cooperate with the scheduler in that they inform the scheduler that they have processors that can be taken away from them if required. If new jobs arrive, *Dynamic* forcibly takes away a processor from a currently executing job to give to the new job. However, preemptions are said to be *coordinated*, in that the job that was forced to relinquish a processor is explicitly informed of this action so that if the preempted thread was *critical* for the execution of the rest of the job, then it can re-schedule the preempted thread on another processor, by blocking one of its own *non-critical* threads. Another novel feature of this scheduling policy is that steps are taken to encourage jobs to cooperate with the scheduler by giving them credit for being "well-behaved", i.e., jobs are given a priority based on their processor usage histories. Jobs that hold processors for large periods of time, tend to have lower priorities, and processors can be preempted only from low priority jobs. According to the authors, this scheme prevents unscrupulous users from being greedy and holding on to more processors than required.

Equipartition, *RRjob* and *Dynamic* are compared in this study, using a workload mix of four parallel applications including parallel matrix multiplication and paral-

1el Mean Value Analysis. Experimental results in this study lead to the following conclusions in [19] :

- Space sharing is found to be superior to time sharing.
- Coordinated preemptions are found to be critical to the performance of dynamic policies, and
- Dynamic scheduling is preferable to static scheduling.

Now let us look at Tucker and Gupta's *Process Control* policy that has been referred to previously in this chapter, first proposed in [34]. It was used in both [17] and [19], for comparison with the *RRjob* and *Dynamic* policies.

Process control was initially implemented by the authors on an Encore Multimax under the UMAX operating system and its performance was studied using multi-threaded applications like Matrix Multiplication and FFT. In this scheme, a job controls the number of runnable threads it has, based on the system load and processor availability. The number of runnable threads are not allowed to exceed the total number of processors in the system. The basis for this scheme is the hypothesis that applications exhibit the best performance when the number of runnable tasks in the application is equal to the number of available processors. This approach reduces the need for preemption and also reduces degradation in performance due to cache corruption. According to this study, cache corruption is an important factor that causes performance deterioration in scheduling policies that frequently re-allocate

processors between jobs, e.g., round robin or coscheduling.

In [34], four parallel applications are executed using *process control* and it is shown that the hypothesis on which the idea of process control is based, is indeed true — parallel applications achieve maximum speedup when the number of processors is equal to the number of runnable threads in the system. As mentioned in the section dealing with coscheduling, a later study, [16], compares *process control* with other policies like coscheduling, affinity scheduling and handoff scheduling and it is demonstrated that under most conditions *process control* exhibits the best overall performance.

The last multiprocessor scheduling policy that will be considered in this section was first proposed in [39]. It is referred to as *Processor Pool-Based Scheduling* and is intended for large scale NUMA multiprocessors. This form of scheduling involves grouping of processors in NUMA systems into software constructs known as processor pools and then scheduling the tasks of an application on the processors within a single pool. Processor pools can be formed using processors that have the smallest delays involved in accessing one another's local memories and are in physical proximity. In this way, tasks of an application that need to communicate or access other memory modules can do so with smaller delays, thus decreasing the average response times.

However, using processor pools has the potential disadvantage of causing load imbalance, especially in large systems. In small systems, with low to moderate system load, a central queue would eliminate load imbalances, but in large systems the central

queue would become a bottleneck, negating the performance gains due to processor pools.

Simulation experiments were performed in [39] to study the performance of processor pool based scheduling and the effect of factors like overhead sensitivity, CPU quantum sizes, degree of pool spanning by tasks of the same application, system size, and different load levels, was evaluated. The study concluded that processor pool based scheduling is an efficient scheduling policy that reduces average response times by reducing system bottlenecks. An observation was made in this study that the pool size that resulted in the best performance had a positive correlation with the average parallelism of the job. Note that this observation is consistent with the hypothesis in [34], mentioned earlier.

In this section a large number of space sharing policies for shared memory systems were surveyed. It was seen that space sharing policies can be classified as either static or dynamic, based on whether they perform processor reallocations, when the system state changes due to increasing or decreasing load. Most studies indicate that static policies are in general, inferior to dynamic policies.

It was also observed that dynamic policies that allocated the same number of processors to a job as the number of tasks in that job, had the best overall performance, [19, 34, 39].

2.3 Scheduling Policies for Distributed Memory Systems

This thesis is concerned mainly with scheduling policies for UMA and NUMA shared memory systems, but, for the sake of completeness, this section will briefly look at some work that has been done in the area of scheduling in message passing, or distributed memory systems. There has been relatively little research in the area of scheduling on message passing systems, and of the few studies that deal with this topic, this section will concentrate on [25] and [20].

A message passing system consists of a set of processors connected together by a high speed communication network. Some examples of such systems are Intel's iPSC/860, Intel's Paragon, NCUBE Corporation's NCUBE/2 and Thinking Machines Corporation's CM-5. Usually, in such systems, scheduling is done statically by partitioning the processors into groups and then assigning each waiting job to one group, where it runs to completion. This scheme leads to low processor utilization and high response times, since most of the tasks of an application spend a lot of time waiting for messages from other tasks, and since each processor executes only one task at a time⁵, processor idle time is increased. Also, if system load is moderate to high, a large number of applications might have to wait for a long time before they can be allocated to a partition.

⁵This is because it is purely a space sharing policy without any time sharing mechanism.

Dynamic allocation schemes are not yet considered feasible because any dynamic scheme would require reassignment of jobs between processors, involving task and data migration and application reconfiguration. The costs involved in moving applications from one set of processors to another would probably outweigh the benefits of dynamic scheduling.

2.3.1 Adaptive Multiprogrammed Partitioning

In [25], a policy known as *adaptive multiprogrammed partitioning*, (MP), is suggested to try and overcome the problems with static partitioning, (SP), without requiring application reconfiguration or data migration.

In this policy, the system is statically partitioned into groups of equal sizes ⁶, like in the static policy, but more than one job is scheduled in each partition, i.e., each partition is time-shared between a certain number of jobs. This modification leads to increased system utilization and has the potential to improve job response times, but there are several drawbacks as well. Time sharing of processors involves context switching and cache reloading overheads. It also increases synchronization costs, since processors might suspend tasks that were cooperating with tasks on other processors.

It can be seen, therefore, that there is a tradeoff between increased system utilization and the costs introduced due to time sharing. In [25] the static and adaptive

⁶In [24] it was demonstrated that for a workload in which all jobs have the same maximum parallelism, adaptive static partitioning performs best under equal sized partitions.

multiprogrammed policies are compared to determine the regions over which one policy outperforms the other.

Scheduling in message passing environments can be thought of as a two level process — the system level, at which the global scheduler decides how to allocate processors to competing programs, and the node level, where each processor must decide how to schedule each task in its local run queue.

In the MP policy, the system level scheduler decides the size of each partition, Z , the number of programs that can be time-shared in each partition (multiprogramming level), D , and the specific partition to allocate to an application. At the node level, the scheduler needs to determine the length of the quantum for which each task is allowed to execute. In [25] each task is assumed to consist of alternating phases of computation and communication, and a task blocks if the message it is waiting for has not arrived. Such a task is descheduled till its message arrives, and another task is scheduled in its place.

Based on this assumption two node level scheduling strategies are studied — one in which the time slice is fixed and a task is necessarily blocked at the end of the time slice, and the other in which a task is blocked only when it executes a blocking communication primitive.

Simulation studies are done to study the performance of MP by varying the values of Z , D , and the quantum size, and compared with the performance of SP for different values of Z . Results indicate that for a system size of 64 processors, values of Z that

are equal to 64 exhibit lower response times and a lower rate of saturation due to increased load, than the for smaller Z values, for both SP and MP. It was also observed that SP policies tend to saturate faster than their MP counterparts.

Quantum size was also observed to have a significant impact on response time in the MP policy. Very low quantum lengths incurred high context switching overheads that increased response time. As the quantum length was increased, response time decreased for a while till the synchronization overheads caused performance to deteriorate again. However, when the quantum size was increased to a level where most computation could be completed within a single quantum, response times were observed to fall again.

This study concluded that the best system scheduling policy for message passing systems is one that adaptively selects optimal values of Z and D based on the current state of the system.

2.3.2 Equipartition and Folding

Another paper that proposes scheduling policies for distributed memory systems is [20].

An assumption on which the policies proposed in this paper are based is that with hardware advances, bandwidths of interconnection networks will be increased in the next generation of computers and communication delays, that have discouraged dynamic scheduling policies for message passing systems will no longer be important.

The authors claim that their "goal is to lay the foundation for processor allocation policies in the next generation of multiprocessor kernels, which will enjoy high network bandwidths and low message startup costs."

The contents of [20] will not be discussed, due to its limited relevance to our work. This section is concluded with a brief overview of the two main policies proposed in this paper; the reader is referred to [20] for further information. The first policy proposed in [20] is known as *Folding*. In this scheme the size of partitions into which the system has been divided is changed each time a job arrives or departs. In the case of job arrival, if the system is idle, the job is allowed to use the entire system. Otherwise, the largest currently allocated partition is divided equally into two parts and the new job is assigned to one half. The job that was previously executing in this partition is now restricted to the other half. In the case of job departure, two or more partitions are combined and the applications that were executing in those partitions are reconfigured. Several mechanisms for handling job departures are considered in the study. It can be seen from this description that unless the number of jobs in the system is a power of two, the partitions will be of unequal sizes, leading to an unfair allocation of resources between jobs. To overcome this drawback, the authors propose an involved mechanism for rotating jobs between unequal sized partitions so that all jobs receive their fair share of processor time. Variations of the policy are suggested to overcome some of the problems, and details regarding these can be found in [20].

The second policy proposed in this study is referred to as *Equipartition*, which

attempts to allocate equal processing power to each job as much as possible. No rotation is carried out and partitions are reconfigured only on job arrivals and departures. Simulation and analytical studies in [20] indicate that for most conditions studied, the family of *Folding* policies outperform the *Equipartition* policies. No performance comparisons are done with other known policies for message passing systems so it is not clear how these policies compare with static partitioning and the adaptive multiprogrammed partitioning policy of [25].

This concludes our review of scheduling policies for multiprocessing systems. Based on what has been seen in this review, the need for the policy that is proposed in this thesis is now explained.

2.4 Motivation for this Thesis

In this chapter a number of time sharing and space sharing policies have been studied in a fair amount of detail and it can be concluded that policies that perform well have some features in common. They attempt to allocate equal processing power to all jobs, and they all use some information about the jobs to make scheduling decisions, like the jobs' average parallelism, average service demand, or processor working set, (*pws*). Also, performance is enhanced if the number of processors allocated to a job is equal to its parallelism.

However, another feature that is common across most policies for shared-memory

systems is that they use a central scheduler with centralized queueing for all jobs. It has been demonstrated in [9] that in real systems, such an organization becomes a bottleneck at moderate to heavy loads, irrespective of the scheduling policy used. The effect is more pronounced in time sharing policies where all jobs have to wait for longer times before they are re-scheduled, and by then their cache usually needs to be reloaded. Performance deteriorates even more if tasks that are holding locks are preempted due to expiration of their quantum. Even in dynamic allocation schemes where partition sizes are reduced with increasing system load the effect of scheduler saturation can be seen at high system loads, and if the scheduling overheads are not negligible, then the deterioration in performance can be observed even at moderate load levels.

It is known from [9] that the hierarchical queue organization has a performance very close to the "ideal" centralized queue, with similar load balancing features and the added capabilities of scalability and elimination of queue access contention. Therefore, it is felt that a scheduling policy based on the hierarchical policy should exhibit performance that is better than policies that use central queues and schedulers, at moderate to heavy loads.

In this work, a policy that eliminates the need for a central scheduler and uses a hierarchical queue structure is proposed. In this policy each queue is accessed by no more than BR processors where BR is the branching factor of the hierarchical queue. This eliminates delays due to queue access contention at moderate to heavy system

loads. This policy also improves performance by allocating processors to a job, equal in number to its parallelism.

Several simulation experiments are done to study the performance of the Hierarchical Scheduling Policy (HSP), and look at the effect of parameters like scheduling overhead, access contention, job service demands and average parallelism on the mean response time. Similar studies are also done for other well-known policies to compare their performance with HSP under the same conditions.

The results of this thesis indicate that HSP exhibits better performance than a dynamic allocation policy and a Round Robin policy at moderate to heavy loads, and at low system loads its performance is very close to that of the dynamic policy and the ideal *RRjob* policy of [17].

In the chapters that follow the hierarchical policy is described in considerable detail along with the system and workload models that we use. The results of simulation experiments that compare the performance of HSP with other policies under various parameterizations of our model will also be examined.

Chapter 3

A Hierarchical Scheduling Policy for Shared Memory Systems

The previous chapters have examined the various components of a multiprocessing system and how parallel applications can be scheduled on such systems in a manner that maintains acceptable levels of performance.

As seen in the first chapter there are many classes of parallel systems and any scheduling mechanism can be designed to perform efficiently only on a subset of these systems.

The second chapter described a number of these scheduling mechanisms and outlined the specific systems for which they were designed and the range of system parameters for which the policies exhibited good performance.

It was also pointed out that a common feature in most known scheduling policies was the presence of a central job queue that led to performance deterioration due to access contention and scheduler saturation at moderate to high load levels and for large system sizes.

In this thesis a new scheduling mechanism is proposed that eliminates the above problem while keeping the benefits of centralized queueing, and at the same time incorporating the positive features of both time sharing and space sharing.

In the following sections, several aspects of the hierarchical scheduling policy (HSP), proposed in this thesis, are described. To study and evaluate the performance of HSP and to compare it with other policies under similar system conditions, simulation models were developed for all policies under study. All the models were discrete time event driven simulation models and were developed using the C programming language. Each of the programs contained approximately three thousand lines of code.

This chapter defines the system and workload models used to study the various scheduling policies. It also describes in detail the behaviour of HSP and the manner in which scheduling decisions are made.

3.1 The System Model

HSP is currently intended for a shared memory NUMA multiprocessor like the one shown in Figures 1.1(a) and 1.4. Specifically, the system consists of P homogeneous processors that are interconnected with P memory modules using a multistage interconnection network, as shown in Figure 1.3.

Jobs arrive for service at the system in a Poisson stream with rate λ and are queued. The hierarchical queueing model, which is described in a later section, is used. Initially all jobs are queued at the *root node* (see figure 1.7).

Several overheads need to be factored into the system for a realistic analysis. Some system specific overheads that are incorporated into the model are: the delay involved in searching the task queues, the delay involved in removing a set of tasks from the queue and the preemption overhead when a task is descheduled at the end of its quantum.

These delays are usually very system specific and difficult to generalize, but normally do not exceed a small fraction of the average task service time. The performance of HSP is studied for a range overheads, varying between 2% to 8% of the average task service time. It is believed that this is a realistic estimate for the system model used [19, 36].

If the scheduling policy being studied uses the round robin mechanism to schedule tasks in the processor ready queues, then another system specific factor that comes

into play is the length of the round robin quantum. Simulation experiments are performed in this study to examine the effect of the round robin quantum length on the average job response times.

3.2 The Workload Model

The workload used in this thesis is similar to the workloads of [7], [17] and [18].

The two main features of parallel applications that are of interest in this study are the average parallelism and the total service demand. Both workload classes that were described in the first chapter are used in the simulation model.

For the independent job model or Fork-Join type of job, the job is described completely by the following random variables:

1. The average parallelism of the job, N , i.e., the average number of tasks in a job,
2. The cumulative service demand for the job, D ,
3. The job's ERF, or *execution rate function*, first described in [10], and used by us to divide the cumulative service demand among the tasks of the job.

As mentioned earlier, a two-stage hyperexponential distribution is used to generate both N and D . To generate random variables with this distribution requires three input parameters — the mean m , the coefficient of variation of the mean C_v , and a parameter α . The first two parameters are varied according to the requirements of the

simulation experiment being performed. In this thesis, α is fixed at 0.95. The two-stage hyperexponential distribution calculates a small mean s and a large mean l using the three parameters supplied. α is used to determine the proportion of jobs that are picked from the distribution with the smaller mean, implying that the proportion of jobs picked from the distribution of the larger mean would be determined by $(1 - \alpha)$.

The small mean is calculated using the following formula:

$$s = m - \frac{\sqrt{(C_v^2 - 1)}}{2\alpha(1 - \alpha)}(1 - \alpha)m$$

Similarly, the large mean is calculated using the formula:

$$l = m + \frac{\sqrt{(C_v^2 - 1)}}{2\alpha(1 - \alpha)}(\alpha)m$$

Another feature of the workload model is that, it is assumed that the cumulative demand of the job is not split equally among all tasks. This is because, there are always overheads associated in splitting the total work of an application into smaller units of computation. Therefore, the execution rate function described in [10] and also used in [7], is used here to determine the service demand of each task in a job, given the cumulative service demand of the job. The calculation is done using the following formula:

$$\gamma(k) = \frac{(1 + \beta)k}{k + \beta}, k = 1, 2, \dots, P.$$

In the above equation, β determines how the total service demand is divided among

the tasks. Lower β values lead to higher task service demands and vice versa. k is the parallelism of the job. The task service demand, d , is obtained using:

$$d = \frac{D}{\gamma(k)}$$

Figure 3.1 shows a graph of $\gamma(k)$ as a function of the average parallelism k for various values of β . From the graph it can be seen that small values of β result in small values of $\gamma(k)$, resulting in large task service demands. The dashed line in the figure shows the case when the cumulative demand of the job is split equally among all the tasks.

Figure 3.1 Example curves for ERF $\gamma(k) = \frac{(1+\beta)k}{k+\beta}$

In the simulation experiments, the effect of varying several of the parameters

mentioned above, on the performance of the scheduling policies, is studied.

The second workload model used in this thesis is the dependent workload model. In this, each task is assumed to consist of a certain number of iterations. During each iteration, a task requires to exclusively access a data item that is shared between all the tasks. In order to do this it must exclusively acquire the lock on that datum. If the lock is not available immediately, the task is blocked and another task in the processor's ready queue is scheduled to run in its place. Other job characteristics like N and D are generated in a manner similar to the independent workload model.

Two simplifying assumptions are made regarding the acquisition and holding of locks:

- It is assumed that the requirement for the lock is at the beginning of each task, i.e., no computation is done by the task till it has acquired the lock.
- It is also assumed that the lock holding period is a fraction of the round robin quantum, in policies that use the round robin scheduling mechanism. This assumption is similar to the assumption made in [17] where the lock was held for 1% of the round robin quantum in the *RRprocess* policy.

It is believed that both these assumptions are reasonable and do not cause any loss of generalization since locks are typically held for extremely brief periods of time, e.g. to increment or decrement counters, requiring just a few machine level instructions. The second assumption also eliminates the delays that could have been caused by

tasks that are blocked at the end of their quantum, while they are still holding on to the lock. In the model used here, if the remaining part of the quantum is less than the lock holding time, then a new iteration is not started and a new quantum is initiated for the next task in the ready queue.

The number of iterations per task are generated from a bounded uniform distribution. Figure 3.2 shows the execution cycle of one iteration of a task. The task may contain several such iterations or cycles.

Figure 3.2 A flow chart depicting one iteration of a task's execution.

3.2.1 The MCS Lock Model

The locking model used to implement critical sections in jobs with multiple tasks is an extension of the MCS lock model [21]. An advantage of this lock model is that it eliminates the memory contention associated with accessing locks. This is achieved

by associating a local flag with each processor that is executing tasks belonging to a job. This implies that each processor that needs to check for the availability of a lock needs only to check its local flag, thus eliminating the hot spot that might be caused by several processors trying to access a common memory location to try and acquire the lock.

For instance, consider a job J , consisting of n tasks, $t_{j_1}, t_{j_2}, \dots, t_{j_n}$, and let each of the tasks be executing on processors $p_1, p_2, \dots, p_n$, respectively. Then each processor will have a flag f_i , where i denotes the task executing on the processor p_i . Each time the task t_{j_i} needs to acquire the shared lock, the processor p_i checks its local flag f_i , and if the flag is set to *TRUE*, the task acquires the lock and enters the critical region. The MCS model ensures that only one task at a time can enter the critical region.

Figure 3.3 shows how the MCS lock can be implemented in software. Each job contains a lock structure (or record) that consists of a pointer to the tail of a queue of processors that are waiting to acquire the lock. The processor at the head of the queue is the one that is currently inside the critical region. When this processor is ready to exit the critical region and release the lock, it simply follows the link to the next processor in the queue and sets the local flag in the latter to *TRUE* and removes itself from the queue.

(a) A job J, with a free lock, L.

(b) The Lock now acquired by processor P1

(c) Processors P2 and P3 waiting for the lock held by P1.

(d) P1 releases the lock, sets local flag of P2. P2 acquires lock and begins execution.

Figure 3.3. The MCS locking model.

Each time a processor needs to acquire the lock it checks to see if the pointer in the job's lock structure is *NULL* or not. If the pointer is *NULL*, it implies that the lock is available. So the processor adds itself to the list and acquires the lock. If the pointer is not *NULL*, the processor simply adds itself to the end of the list with its local flag set to *FALSE*. Once in the list, the task that required the lock is blocked till the processor's flag for that task is set to *TRUE*.

A small modification is made to the original MCS model of [21] to make it work for the system model used in this thesis. Since more than one task can be executing at a processor at a time in our model, each link in the lock queue is actually a unique processor-task pair, and not a processor like in the original scheme.

3.3 The Hierarchical Queue Organization

The hierarchical task queue organization was proposed in [9] and was described briefly in the first chapter of this thesis. HSP is based on this queue organization, shown in Figure 1.7.

In the hierarchical organization a set of task queues is organized as a tree. The degree of each node in the tree is the same and is referred to as the branching factor, or *BR factor*, B , of the tree. The leaf nodes of the tree are the processors of the system. All the tree nodes, each of which holds one task queue, is mapped to a separate memory module of the multiprocessor. To see that this is indeed possible, consider a multiprocessor like the BBN Butterfly. Suppose there are N processors and N memory modules in the system and the hierarchical queue is organized as a binary tree, i.e., with a *BR factor* of 2. In this case the number of tree nodes is $N - 1$, implying that each tree node can be mapped to a separate memory module with one module to spare. This is in fact the worst case situation, since a binary tree has the greatest number of tree nodes relative to trees of greater degree but the same number of leaf nodes.

As shown in figure 1.7, all arriving jobs are initially queued at the root node and move down the levels of the tree when they are picked up by idle processors that are looking for work. A transfer factor, or *TR factor*, Tr , determines how many tasks are moved down one level of the tree at a time. Figure 3.4 illustrates how the transfer factor operates. Initially, the root node in the figure has 100 tasks and the other

queues are empty.

Figure 3.4. Task transfer process in the hierarchical organization for $N = 64$ processors with a branching factor $B=4$. (a) $Tr = 1$. (b) $Tr = 2$. (for simplicity, processor local queues are not shown), [9]

In the above figure, it can be seen that when the transfer factor Tr is one then the number of tasks carried down a level of the tree equals the number of processors under that branch of the tree as shown in figure 3.4 (a). Similarly, when $Tr = 2$, the number of tasks carried down a branch is equal to twice the number of processors in the subtree under that branch as shown in figure 3.4 (b).

Simulation and analysis done in [9] indicate that the number of queue accesses required per task scheduled decreases as the branching factor increases. This is to be expected, since the number of levels in the tree are reduced as the branching factor is increased. However, it was also demonstrated in [9] that varying the transfer factor, given a fixed branching factor, did not affect the performance very much, though a

transfer factor of 2, for all values of branching factors studied, required the minimum number of queue accesses.

The simulation experiments done in this thesis assume a branching factor of 2 and a transfer factor of 1 because it has been shown in [9] that the hierarchical organization is not very sensitive to BR and Tr values and higher values do not yield significant performance improvements.

Since the hierarchical queue organization forms the basis for the hierarchical scheduling policy proposed in this thesis, some of the observations made in [9], regarding this queue organization are briefly enumerated here:

- The hierarchical organization can be designed to provide performance similar to that of the centralized organization while eliminating the ready queue bottleneck.
- The overhead of accessing more task queues in the hierarchical organization can be reduced close to that associated with the centralized and distributed organizations.
- The hierarchical organization is fairly insensitive to the transfer factor Tr .

In the next section, the hierarchical scheduling policy (HSP) is described, using the system, workload, locking and queuing models described here.

3.4 A Simulation Model for the Hierarchical Scheduling Policy

A discrete time, event driven simulation model was developed to study the performance of HSP. After the system is powered on and jobs begin to arrive for service, a set of events continuously take place as long as the system remains up.

The description of system behaviour, presented now, is based on the system model depicted in figure 1.7. For the sake of convenience figure 1.7 is reproduced below as figure 3.5.

Figure. 3.5 Hierarchical Queue Organization for Four Processors with Binary branching.

Jobs arrive into the system as a result of the *job-arrival* event and are queued initially at the root node. Job arrival is modeled as a Poisson process with a rate

λ . The load on the system depends on the value of λ , which can thus be varied to study the effect of increased load on the system performance. Each job that arrives for service must contain the characteristics mentioned in Section 3.2. The jobs average parallelism avg , and its cumulative service demand D , are generated using the hyperexponential distribution. The cumulative service demand is then distributed among the avg tasks based on the execution rate function (ERF). HSP uses a fixed quantum length, independent of the size of the job, therefore, once the task service demand is known, the number of quantum required for the task to complete can be calculated. It should be noted that a fixed quantum size does not cause the performance deterioration observed in [17] for the *RRprocess* policy, because HSP uses a combination of spatial and temporal sharing to allocate processing power to each job.

Initially, all processors in the system are idle. HSP uses the *self-scheduling* mechanism to allocate tasks to processors. This implies that idle processors search the task queues for work and there is no central scheduler involved to make task placement decisions. Self scheduling is simulated with the help of the *job demand* event. Each processor that needs to obtain tasks, searches the node immediately above it in the hierarchy (see Figure 3.5). Since the system model in this study assumes a binary branching factor, each tree node in the hierarchy can be accessed by only two processors. There is a lock associated with each task queue¹ that ensures mutually exclusive

¹There is a one-to-one correspondence between the tree nodes and task queues, i.e., each tree node contains one task queue.

access to the queue for each polling processor. A processor looking for tasks, first checks the lock status and if the lock is free, it sets to lock status to *in-use*. The other processor that polls the same queue and finds the lock *in-use* simply queues itself in a wait queue that is associated with the tree node.

The processor that manages to acquire the lock on a queue, searches the queue for any available tasks. If there are no tasks available in the current queue, the queue directly above it in the hierarchy is checked. Again, the same process is repeated. It can be seen that continuing in this manner, when the system starts up, only two of the processors will reach the root node to access the tasks of the first jobs that arrived. All other processors would have been queued at some intermediate tree node after failing to acquire the lock. This mechanism of locking to prevent all processors from searching the entire hierarchy above them, greatly reduces the access contention that is observed in a centralized queueing model.

Once a non-empty task queue is located by a processor, it must decide how many tasks or jobs to carry down with it to the lower levels. Each processor that finds tasks, retraces its path back down the tree, and at each level it may leave behind a certain number of tasks or jobs. The spatial partitioning feature of HSP comes into play at this point. The decisions that need to be made by the processors are -- can a job be carried down a level in its entirety or should only a certain number of tasks belonging to the job be moved down, and how many tasks, in all, are to be carried down one level of the queue. The first decision is made on the basis of the average

parallelism of the job. If the parallelism of the job in question is less than or equal to the number of processors in the sub-tree below its current level then the entire job is picked up and moved down the tree. In this manner, a job moves down the hierarchy till it reaches a partition equal in size to its parallelism. The second decision is made based on the transfer factor that was mentioned in Section 3.3. If the transfer factor is one, as is the case in most experiments in this thesis, the number of tasks that are carried down a level of the tree is equal to the number of processors below that level.

The above description assumes the situation in which there are enough tasks to move down the tree. However, at low to moderate system loads, a situation that commonly occurs is that a processor might not have been able to carry down enough tasks to leave some behind at each level. In such a case, some of the task queues in its return path might remain empty.

Recall that processors that were not able to acquire the lock on a task queue had been queued at the corresponding node, waiting to be notified when the lock became available. Processors on their way down the hierarchy after obtaining tasks fulfill this requirement with the help of a *free-queue* event. The lock on the queue is reset and the processor in the wait queue is notified of this action. In shared memory systems this can be achieved with the help of a semaphore variable.

The processor that is notified of the queue being free might be busy at the time of notification, executing tasks in the ready queue, thus the semaphore variable is checked at the next instant when the processor becomes free, i.e., when a quantum

on the processor expires. At this point the processor that was "woken-up" in this manner, directly searches the queue where it was waiting (note that this may be at any level of the tree). Thus having processors enqueued at the node that they last polled eliminates the need to search the entire hierarchy again, and instead the level at which tasks are most likely to be found, is searched. At moderate to high load levels, tasks will be found by the processor at the level where it was notified of a free queue, since the processor that sent the notification would have left a certain number of tasks at that level. At low load levels, if tasks are not available, the same process of searching the hierarchy is repeated starting from the node at which the processor had been queued.

It must be mentioned here that in the system model used for HSP, each processor is multiprogrammed i.e., there may be more than one task in each processor's ready queue, depending on the system load. For the simulation experiments the multiprogramming level (MPL) has been set to five. This implies that each processor searches for tasks in the hierarchical queue, not just when it is idle, but also as long as the number of tasks in its ready queue are less than the MPL. The tasks in the ready queue of each processor are executed in a round robin manner, with a fixed round robin quantum mentioned above. Task execution is simulated by a series of *start-task* and *block-task* events, till the task's total service demand has been met. When the service demand has been met, the task execution is complete. When all *avg* tasks of a job complete their execution, the job then leaves the system. The difference between the time that the job joined the queue at the root node and the time at which its

final task was completed is recorded as the job's response time.

The *average response time*, calculated by summing all the job response times and dividing by the number of jobs is our metric of interest. The other performance metric that needs to be calculated is the system utilization. To determine this metric, the time that each processor spent to execute tasks, (the processor's *busy-time*) is summed and averaged over the number of processors. This mean *busy-time* is then divided by the duration of the entire simulation . The figure obtained denotes the fraction of the time the processors were busy doing "useful" work, and is termed the *system utilization*.

To obtain confidence intervals for the metrics that are calculated, each experiment consists of thirty batches, with each batch consisting of 10,000 jobs. The *student's t-distribution* is used to obtain confidence intervals. The large number of jobs in each batch ensures tight statistical bounds. All results obtained have errors of less than 10% at a 90% level of confidence, though in most cases the errors are much smaller, often less than 1%, ensuring the statistical reliability of the results.

The description above has detailed the important features of the hierarchical scheduling policy. A recap of the main features of HSP is presented below:

- HSP makes its scheduling decisions based on a combination of the principles of space sharing and time sharing.
- At higher levels of the tree, small jobs move down the hierarchy till they have

been allocated completely to a partition with size equal to or greater than the job's parallelism.

- Due to a multiprogramming level associated with the system, at the lowest level of the tree, time sharing is used to schedule the tasks in a processor's ready queue.
- HSP ensures that each jobs is allocated a number of processors equal to its parallelism. It has been shown in several studies that were reviewed in the second chapter that this greatly improves performance.
- Since demand driven or *self scheduling* is used to allocate tasks to processors, the need for a central scheduler is removed. This eliminates a possible system bottleneck.
- The hierarchical queue organization provides excellent load balancing and does not have the access contention associated with centralized queues, or the overheads associated with the distributed queue organization.

This completes the description of the hierarchical scheduling policy. In the next section the models developed for other policies, used for comparing the performance of HSP with other known schemes, will be described.

3.5 Simulation Models for Other Policies

One space sharing and one time sharing policy were used to evaluate the performance of HSP relative to standard scheduling policies. A modified version of the *RRjob* policy of [17] and a dynamic equipartitioning policy proposed in [36] were used for the sake of comparison. The original *RRjob* of [17] was also simulated to see how close the performance of HSP was to the ideal situation in which there are no scheduling overheads or access contention delays.

3.5.1 Modified RRjob

All simulation details like job arrival, job characteristics generation, implementation of self scheduling and the multiprogramming level of the processors are identical to what was described for the hierarchical policy.

In the modified scheme, the centralized queue organization is used. Unlike the original scheme of [17], however, there is no scheduler that makes allocation decisions. Instead of having a central scheduler, the demand driven approach of self scheduling is used, similar to HSP. Also, in the original scheme of [17] there was no MPL associated with the processors, as it is in this case.

The system model used is the same as that used in HSP, i.e., a shared memory NUMA multiprocessor that incorporates scheduling overheads and the effects of access contention.

When a job arrives at the system, it is added to the queue behind all waiting jobs. Processors pick up tasks from the queue, equal in number to their multiprogramming level. Depending on the parallelism of the job, the quantum of service given to tasks from different jobs is different. Smaller jobs are given larger processor quantum in order to allocate equal processing power to each job. The quantum size Q for the tasks of a job are calculated using the formula:

$$Q = \frac{P}{N} \times q,$$

where P is the number of processors in the system, N is the parallelism of the job and q is the basic quantum size. The parallelism of the job is bounded by the system size so that q becomes the minimum quantum length for the system. At moderate to high load levels processors are busy most of the time and check the job queue only when a task blocks or finishes execution. Whenever the job queue is checked, it is locked to prevent other processors from accessing the queue at the same time. Each time a processor accesses the job queue, it attempts to remove enough tasks so that it maintains the number of tasks in its ready queue at its MPL. However, when the system is in a steady state, each access brings down only one task, since queue accesses are made each time a task departs.

As in the case of HSP, the main performance metrics gathered are the mean job response time and the mean system utilization. All experiments are performed in 30 batches, with 10,000 jobs in each batch, similar to the HSP experiments.

3.5.2 Equipartitioning

This is a modified version of the dynamic policy first outlined in [36], where it was compared with static allocation policies. As described in the previous section this policy also uses a central queue and a central scheduler for making allocation and placement decisions.

Each job informs the system of its processor requirement. This requirement is usually equal to the parallelism of the job. The scheduler attempts to satisfy the jobs requirement if there are processors available. If no processors are available, one is taken away from a job that is holding more than one processor.

At moderate to high system loads this leads to a situation where each job has at least one processor. If there are more jobs in the system than there are processors then the extra jobs have to wait in the central queue.

This policy is a dynamic space sharing policy. It is dynamic in that the processor allocation changes with job arrival, job departure, or a change (decrease) in a job's requirement. It is a space sharing policy since each job is exclusively allocated a certain number of processors.

A modification has been made in this study from the original scheme in [36]. In the original scheme, when a new job arrived and there were no processors available, a processor was taken away from an executing task and allotted to the new job. This led to a high frequency of pre-emption at all but low system loads. In this

study, all tasks that are allotted a processor, run on that processor, to completion. This is expected to improve performance, especially if pre-emption overheads are not negligible. However, to ensure that new jobs do not have to wait for too long, the central scheduler gives preference to jobs that have not been allotted any processors so far. This implies that whenever processors become available, the scheduler first checks to see if there are any jobs in the queue that have not been allotted any processors. If such jobs are present then the processor is given to that job, which might not necessarily be at the head of the queue. Note also that a job's processor allocation changes during its execution. For instance, if a job required n processors, but was allocated only p , where $n > p$, then once the p tasks of the job are done, it is not necessary that the same processors will be reallocated to the same job -- because, in the meantime, new jobs might have arrived in the system and these jobs are given preference over older jobs that have received some service already.

The equipartition policy was included in this study for several reasons. It is a purely space sharing policy. It exhibits excellent performance at low system overheads for all load levels, but is extremely sensitive to the scheduling overhead especially as system load increases.

All details regarding the actual simulation are identical to the policies described above. Note that since this is a Run-To-Completion, or RTC policy, there is no multiprogramming level associated with the processors. Two main overheads are associated with this policy:

- An allocation delay D_{alloc} is incurred when a task is allocated to an idle processor and,
- A reallocation delay $D_{realloc}$ is incurred if a process is taken away from a job and allocated to a new job.

3.6 Original RRjob

The simulation model developed for the original RRjob policy, adheres exactly to the description of this policy in [17], where it was first studied.

All jobs that arrive for service are appended to a central queue and a central scheduler allocates tasks to processors in a round robin fashion. The quantum for which each task executes is calculated as described in Section 3.5.1. The original RRjob policy differs from the modified policy of Section 3.5.1 in that it is not a self-scheduling policy since a central scheduler makes placement and allocation decisions, and that it uses an ideal system model that has no delays associated with scheduling, preemption or queue access contention.

The original RRjob policy has been included in this study to see how close the performance of HSP comes to an “ideal” policy that incurs no overhead penalties.

The next chapter presents the results of simulation experiments performed to study the performance of HSP under varying system conditions and parameters, and to compare its performance with the other policies described above.

3.7 Validation of Simulation Models

The models for the RRjob and Equipartitioning policies were validated by comparing performance results with the results obtained by researchers who first proposed these policies [17, 36]. The models developed in this thesis were tested with the same system and workload parameter values as the original research and it was observed that the results obtained were very close to the results in the previous work.

Since HSP is being studied for the first time in this thesis the validation for this model was done by analytically estimating the results of a corresponding deterministic model and then comparing the simulation results with the estimated values. This ensured that the simulation model performs correctly.

Chapter 4

Independent Workload Model

Results

In this chapter the results of simulation experiments carried out to evaluate the performance of HSP are presented. The performance of HSP under different simulation parameters, was studied and compared to the performance of Equipartition, Modified RRjob and the original RRJob. Several classes of experiments were performed; these include experiments that study the effect of scheduling and contention overheads on the response time, the effect of the coefficient of variation of the cumulative job service demand, C_{vd} , on the response time, and the effect of other factors like the round robin quantum, the ERF factor (β), and the average parallelism, *avg*.

The workload model used in these experiments has been detailed in the previous chapter. The default system and workload parameters for the simulation experiments,

are presented in Table 4.1.

Parameter	Description	Default Values
P	System size (number of processors)	64
D_{qp}	Queue polling overhead	0.04%
D_{qa}	Queue access overhead	0.04%
D_{cs}	Context switching overhead	0.02%
D_{no}	Notification overhead	0.001%
D_{alloc}	Processor allocation delay	0.1%
$D_{realloc}$	Processor reallocation delay	0.7%
Tr	Transfer factor	1
B	Branching factor	2
λ	Job arrival rate	0.75
μ	Job service rate	10.00
avg	Avg. parallelism (number of tasks)	8
β	ERF factor	30
C_{vd}	Coeff. of var. of demand	1
C_{vn}	Coeff. of var. of parallelism	1
MPL	Multiprogramming Level	5
RR_q	Round robin quantum	0.025
N_j	Number of jobs per batch	10000

Table 4.1 Default parameter values for simulation experiments.

Note that in the above table, all overhead values are expressed as a percentage of the average task service time. The *notification overhead* mentioned above, is the delay involved in informing a processor in a wait queue of the availability of the lock it was waiting on. Also, the default job arrival rate is set such that the minimum system utilization is approximately 12%. This rate is increased, keeping μ constant, to increase the load on the system.

In the following sections, results for all the experiments performed, will be presented and analyzed.

4.1 Effect of varying the scheduling overhead

Since the goal of this study is to develop a scheduling policy that outperforms the centralized queue policies, assuming that there is always an overhead associated with queue access contention, the most important aspect of the policy that needs to be analyzed is the impact of overheads on its performance.

The main overheads involved are the delays for accessing and polling a queue and the context switching delay for the self scheduling and round robin policies, and the scheduling delay, processor allocation and processor reallocation delays for the scheduler based and RTC policies.

For the purpose of this experiment, the overheads were varied from the low default values shown in the table to high overhead levels that were a factor of three greater

than the default values and the performance of the policies was compared at three different overhead levels i.e. low, medium and high, for a range of system load levels.

The delays associated with the HSP and the modified RRjob policy are D_{qp} , D_{qa} , D_{cs} and D_{no} and the delays associated with the equipartitioning policy are D_{alloc} and $D_{realloc}$.

Figure 4.1 shows the performance of the policies at the lowest overhead level where the delay values are the default values shown in Table 4.1. Note that the performance of the original RRjob is at 0% overhead since this indicates the performance of the policy on an "ideal" system with no access contentions or scheduling delays. Also note that the range of interest with respect to the system utilization is 50% to 85% for all experiments performed in this thesis. For system utilizations higher than 85% all policies tend to saturate and the results that are obtained for these extremely high loads are not of much interest.

Figure 4.1. Response Time Vs. Utilization for low overhead.

From the above figure it can be seen that the equipartition policy has the best performance when the associated overheads are low and the system load is low to moderate. However, even at low overheads, when the system load increases there is a rapid deterioration in the performance of this policy. This is because equipartition has a central scheduler that saturates when a large number of jobs are waiting in the queue.

Looking at the curve for HSP in figure 4.1, it can be seen that even though, at low to moderate loads, the average response time of the policy is greater than the equipartition policy, as the load increases above the 75% level, the deterioration in performance occurs at a much slower rate than the equipartition policy. In fact, it can be seen from the graph that HSP does not show the effects of saturation even at

very high system utilization ($> 90\%$), due to the absence of a central scheduler.

The third policy, modified RRJob, also uses a centralized queue, but is a self scheduling policy. Hence, expectedly, the response times at low to moderate utilization remain almost constant, though higher than the other policies due to the higher context switching overhead involved. At high utilization, the central queue saturates faster than HSP, but slower than equipartition, due to the self scheduling nature of the policy.

Quantitatively, at this overhead level Equipartition has response times approximately 30% lower than HSP, while HSP provides an approximately 55% improvement over the modified RRjob policy.

The performance of the ideal policy is also shown in the graph. The ideal policy is studied only in the first few experiments because the aim of including this policy in the study is simply to compare the performance of other policies to the “ideal” case.

In the next two graphs, figure 4.2 and figure 4.3, the impact of increasing the overhead is presented.

Figure 4.2 Response Time Vs. Utilization for medium overhead.

Figure 4.3 Response Time Vs. Utilization for high overhead.

Figure 4.2 shows the effect of increasing all the associated delays by a factor of two

over the default values and in Figure 4.3 the effect of increasing the overheads by a factor of three over the default values is shown. It can be seen from the above figures that the equipartition policy is extremely sensitive to relatively small increases in the scheduling overhead and its performance is not comparable with the other policies at moderate to high load levels. This is expected, since as more and more new jobs arrive in the system, processors need to be reallocated in a manner that ensures that new jobs get at least one processor, as soon as possible. If the overhead associated with this reallocation is increased, the effect is quite pronounced, as can be seen from the graphs. Also, as the load increases, processors might not be available for substantial lengths of time and the queue length soon causes the scheduler to saturate. The combined effects of high system load and processor reallocation overhead account for the rapid deterioration in performance.

HSP and the modified RRJob policy, however, are not as sensitive to increasing overheads as the equipartition policy. Response times increase slightly as the delay level is increased and the effect is not noticeable at low to moderate loads, but at high load, it can be seen that performance deterioration occurs more rapidly at high overhead levels. By comparing the curves of HSP and modified RRjob, it can further be noted that modified RRJob is more sensitive to delay than HSP, especially at high system utilization. In terms of relative performance HSP exhibits a performance improvement of about 55% over the modified RRjob policy over a wide range of system utilization.

The results of this experiment indicate that scheduling overheads play an important role in determining system performance. If these overheads are almost negligible then the space sharing policy exhibits the best performance over a wide range of system utilization values. However, as scheduling overheads are increased, it can be seen that the performance of HSP remains stable and it considerably outperforms both the space sharing equipartition and the time sharing RRJob policies over a wide range of system load levels.

This can be seen more clearly in figure 4.4, where the offered load to the system was fixed at approximately 72% and the effect of increasing the overhead was studied. It can be seen from the graph that the average response times of HSP over the range of overheads are consistently better than the other policies at this load level.

Figure 4.4. Response Time Vs. Overhead, at 72% utilization.

In the above figure the horizontal axis is the sum of all overheads expressed as a percentage of the task service times. As mentioned earlier, for the HSP and the modified RRjob policies these delays include D_{qp} , D_{qa} , D_{cs} and D_{no} while for the equipartitioning policy the delays involved are D_{alloc} and $D_{realloc}$.

4.2 Effect of varying the C_{vd}

The previous set of experiments was performed with the coefficient of variation of demand fixed at 1. This implied that the cumulative service demand of jobs was taken from an exponential distribution. However, in real systems there is a possibility that some of the jobs might have very high cumulative service demands [7]. To simulate this situation the coefficient of variation, C_{vd} , of the jobs can be increased, so that in the jobs that arrive there might be a small proportion of jobs that have very high service demands.

The performance of the scheduling policies was studied at three levels of C_{vd} , and this section presents the results of these experiments. The scheduling overhead was fixed at the lowest level for all these experiments.

Figure 4.5 Response Time Vs. Utilization for Demand $C_v = 1$.

The above figure is the same as figure 4.1. It shows the relative performance of the four policies when the $C_{v_d} = 1$. In figure 4.6 the C_{v_d} is increased to 3, and in figure 4.7 the C_{v_d} is fixed at 5.

Figure 4.6 Response Time Vs. Utilization for Demand $C_v = 3$.

Figure 4.7 Response Time Vs. Utilization for Demand $C_v = 5$.

From the above figures it can be seen that as C_{v_d} increases, the performance

of equipartition deteriorates at a faster rate, and the effect is more pronounced at high system load. This is due to the fact that not only do the large jobs have high response times, but they also delay the other smaller jobs by holding onto processors for a longer time. Since equipartition and modified RRjob use centralized queueing, the smaller jobs are queued behind the large jobs for longer times than in HSP. The effect of this situation is visible in the above graphs, where as C_{v_d} increases, the HSP curve clearly falls below both the equipartition and the modified RRjob curves. In fact at high load levels the performance of HSP is approximately 2% better than the equipartitioning policy and 18% better than the modified RRjob policy even at low overhead values. The ideal original RRjob curve is also shown to indicate the best achievable performance.

The reason HSP has an advantage over the other policies is because the larger jobs are removed from the root node queue relatively faster than in the other policies, so that other jobs do not have to wait as long for service.

Figure 4.8 compares the effect of increasing C_{v_d} while keeping the job arrival rate constant to maintain the system load at approximately 72%. From the figure it can be seen that the equipartition policy is much more sensitive to increasing C_{v_d} values than HSP, since the curve for the former policy rises at a much faster rate and crosses the curve for HSP at $C_{v_d} = 5$.

Figure 4.8. Response Time Vs. Demand C_v , at 72% utilization.

4.3 Effect of ERF factor

The ERF factor, β , is used to divide the cumulative service demand of a job among the tasks of the job. As explained in the previous chapter, a small value for β causes the task service demand to be large. The need for using the execution rate function in this study stems from the assumption that in realistic workloads splitting a parallel application into several sequential tasks involves an overhead that can be factored into the analysis by increasing the task service requirement.

This experiment allows us to study the effect of increasing the overhead involved in splitting a parallel job into sequential tasks. For the purpose of this experiment β

2 of/de 2

PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

was varied from 5 to 60. The lower value of β caused an overhead of approximately 60% of the average task service time and the higher value of β caused the overhead to be about 11% of the average task demand.

The experiments were performed at two levels of offered load and the results are presented in figures 4.9 and 4.10 for 50% and 75% load, respectively. The scheduling overhead in the experiment was fixed at the 0-2% level in order to include the equipartition policy in the analysis¹.

Figure 4.9 Response Time Vs. ERF Factor, (β), at 50% utilization.

¹It was seen in previous experiments that at higher overheads the performance of the equipartition policy deteriorated considerably.

Figure 4.10 Response Time Vs. ERF Factor (β), at 75% utilization.

The above graphs indicate that for very low values of the ERF factor, the impact on the response times for all policies is quite significant but the response times improve rapidly as the overhead is reduced to about 20% of the average task service time, ($\beta = 30$). Increasing β beyond this point does not cause significant improvement in performance.

The slopes of the curves in the 4.9 and 4.10 indicate that the equipartition policy is more sensitive to lower β values, especially at high system utilization. HSP and modified RRjob are similar in their behaviour with respect to changes in the ERF factor, though the mean response times for HSP are smaller than those for the modified RRjob policy at all values of β .

4.4 Effect of average parallelism

This experiment was performed to study the impact of increasing the number of tasks per job on the performance of the scheduling policies, at moderate and high system loads. The scheduling overhead was fixed at the lowest level, like in the other experiments and the ERF factor was fixed at 30. The round robin quantum was also fixed at levels at which the response times were minimum.

Figures 4.11 and 4.12 present the results of this experiment.

Figure 4.11 Response Time Vs. Avg. Parallelism, (*avg*), at 50% utilization.

From the above graph it can be seen that, expectedly, as the average parallelism of the jobs increases, the response times begin to increase, as each job now requires more and more processor time. The effect is most pronounced in the equipartition

policy due to the fact that as more jobs arrive each job is allocated fewer and fewer processors till all scheduled jobs have only one processor each and new jobs are added to the central queue. The scheduler saturation and queue access contention add to the effect of large service demands and the response time rapidly deteriorates as the average parallelism increases, even at moderate system load.

Of the other two policies HSP appears to be the least sensitive to increasing parallelism at moderate system load, although the response times of both HSP and modified RRjob remain more or less constant for a wide range of parallelism values. However, RRjob saturates faster as the parallelism is increased to a level where the number of tasks per job are equal to or greater than half the number of processors in the system. This difference can be attributed largely to the contention associated with the central queue.

In figure 4.12 the results of the experiment at high system load are presented. It can be observed that the performance trends are similar to the previous experiment, though more pronounced. At high system load, with large jobs, all the policies demonstrate rapid deterioration in performance as the average parallelism of the job increases beyond a certain level. This is expected since processors are busy most of the time and the number of tasks waiting for service is large. Quantitatively, the performance of HSP is approximately 12% better than that of the modified RRjob policy over a wide range of *avg* values.

When the parallelism is very low the response times are observed to be slightly

higher in the RRjob policy. This is mainly because when the parallelism is lower than the MPL entire jobs might move to the same processor. If this occurs, the tasks of a job cannot be executed simultaneously causing an increase in job response times. As the parallelism gradually increases and exceeds the MPL response times are seen to decrease.

Figure 4.12 Response Time Vs. Avg. Parallelism.(avg), at 75% utilization.

4.5 Effect of the round robin quantum size

In this section, the effect of the round robin quantum size is studied on the policies that use the round robin scheme to schedule tasks in the processors' ready queues. This includes the modified RRjob and HSP. The ideal, original RRJob is not included in this experiment.

For the purpose of this experiment, the round robin quantum² was varied from approximately 0.2% to 7% of the task service time. The experiment was performed for two levels of system load — 50% and 75%. Figure 4.13 presents the results for the case where the system load was fixed at 50% and figure 4.14 presents the results for system load equal to 75%.

Figure 4.13 Response Time Vs. RR Quantum at 50% utilization.

²Note that in the case of the RRjob policy, the round robin quantum is the minimum quantum allocated to a task, since in this policy the quantum size depends on the job's average parallelism.

Figure 4.14 Response Time Vs. RR Quantum at 75% utilization.

It can be seen from the above figures that the behaviour of both the policies is similar, as the round robin quantum is varied. For very small quantum sizes, the number of preemptions per task is large and so the delay incurred due to frequent context switching causes the response time to increase. As the quantum size is increased, the number of context switches per task, are reduced and the response time also decreases. As the quantum size is increased further an increase in response times is observed because tasks in the processors' ready queue have to wait for a longer time before receiving service. This can cause deterioration in performance specially if larger tasks are present in the queue in front of smaller tasks.

It can also be seen from the graph that the modified RRjob policy is more sensitive to increase in the minimum quantum size than HSP. This can be attributed to the

fact that in the RRjob policy the actual quantum size is normally several times greater than the minimum quantum size, since the policy allocates larger round robin quanta to smaller jobs. For example, if a job's parallelism is 8, the number of processors in the system is 64 and the minimum quantum size is q , then the actual quantum size allocated to the tasks of this jobs would be $64/8 \times q$, or $8 \times q$. On the other hand in HSP each task of the job would be scheduled for a quantum length equal to q .

The above feature of the RRjob policy explains the rapid deterioration in performance of the policy at high system loads and large quantum sizes. This experiment was useful for determining the best quantum size for the workload model used. For both policies the minimum response times were observed when the quantum size was in the range of 2-4% of the task service time.

4.6 Effect of the multiprogramming level

The multiprogramming level (MPL) is the maximum number of tasks that can be present in a processor's ready queue. The two policies that use the round robin discipline i.e. HSP and modified RRjob have an MPL associated with the processors. Each processor continues to look for tasks till its MPL is reached. In all the previous experiments the MPL was fixed at five.

The following experiments were done to study the effect of varying the MPL on

the average response times for the two policies. The experiments were performed for a range of MPL values between 2 and 10 and a branching factor of 8. Figures 4.15 and 4.16 show the effect of varying the MPL for system loads of 50% and 75% respectively.

It can be seen from the figures that lower MPL values lead to lower response times for both the policies. The reason for this is that tasks have to wait for shorter lengths of time before they can be rescheduled.

Figure 4.15 Response Time Vs. MPL at 50% utilization.

Figure 4.16 Response Time Vs. MPL at 75% utilization.

As the MPL is gradually increased the response time also increases since longer ready queues mean longer waiting times for tasks. The effect is more pronounced in the case of the modified RRjob than for HSP and again this can be attributed to the centralized queue organization. If tasks in a processor's ready queue have to wait for longer and longer times to receive service the impact would be felt by the newly arriving jobs that will have to wait for longer durations as well, causing the average response times to go up. In the case of HSP this effect is less pronounced since newly arriving jobs are distributed to different queues soon after arrival and thus on an average jobs have to wait for shorter durations to receive service than in the modified RRjob policy.

4.7 Effect of branching factor on HSP

The final experiment that was performed with the independent workload was to study the effect of the branching factor on the average response time in the hierarchical policy. It has been shown in [9] that the hierarchical queue organization is not very sensitive to the branching factor, but response times are reduced for higher BR values.

The following experiment was performed for BR values of 2, 4 and 8 at low overhead. Figure 4.17 shows the results of this experiment and compares it to the performance of the equipartition policy at low overhead over a range of system utilization levels. For the sake of clarity the curve for $BR = 4$ is not shown in the figure as it lies between and very close to the curves for $BR = 2$ and $BR = 8$.

Figure 4.17 Response Time Vs. Utilization for different BR values.

From the figure it can be seen that HSP is not very sensitive to the branching factor. However, for a higher BR values the response times for HSP are slightly lower than for lower BR values. A comparison with the curve for the equipartition policy shown in the figure demonstrates that for $BR = 8$ the response time for HSP at low utilization is very close to that of the equipartition policy. In fact, at this branching factor the response time of HSP is only slightly worse than the ideal original RRjob policy (see Figure 4.1).

The above result is in accordance with the results of [9]. The main reason for the slight improvement in performance for higher BR values is that fewer levels of the tree have to be traversed by processors looking for work.

4.8 Summary of Results

In this chapter, results were presented for a large number of experiments performed to study the performance of the hierarchical policy and to compare it with other space and time sharing policies.

The following observations can be made about the performance of HSP based on the results presented above :

- HSP is relatively insensitive to scheduling and queue access overheads and its performance at all overhead levels studied, is better than the time sharing policy that uses a centralized queue. The performance of HSP is better than the

space sharing equipartition policy when the overheads are in the 4-6% range. Equipartition performs best when the overheads are negligible but is extremely sensitive to higher overhead values.

- Of the three policies that were compared, HSP is also the least sensitive to increasing variance in job service times.
- Varying other parameters like the ERF factor, the round robin quantum and the average parallelism of jobs has an impact on the performance of all the policies, but the sensitivity of HSP to these factors, is the minimum of the policies considered.
- The hierarchical queue organization also reduces the sensitivity of HSP to the multiprogramming level.

These results indicate that using a hierarchical queue organization and a combination of spatial and temporal sharing, along with self scheduling by processors, provides significant performance benefits over policies that use a central scheduler and a centralized queue organization.

The independent workload model in which tasks do not synchronize during execution, was used in the experiments described so far. In the next chapter results of similar experiments performed with the dependent workload model will be presented.

Chapter 5

Dependent Workload Model

Results

In a large number of parallel applications the individual tasks of the application must synchronize in some way in order to complete their execution. These applications belong to the dependent workload model that was described in Section 3.2.

This chapter presents the results of simulation experiments performed to study the performance of HSP using the dependent workload model. Most workload and system parameters are the same as those described in Table 4.1. Those that are different and some additional parameters are presented in Table 5.1.

Parameter	Description	Default Values
B	Branching factor	8
MPL	Multiprogramming Level	2
MAX_{iter}	Max. number of iterations	10
D_{lock}	Delay for accessing lock	0
T_{lock}	Lock hold time	$0.1 \times RR_q$

Table 5.1 Additional parameters for the dependent workload.

In this job model, the number of iterations per job are generated from a bounded uniform distribution, using the MAX_{iter} parameter specified above. There is no delay associated with accessing the lock [21] because each processor needs only to test a local flag to determine if the lock is available. As mentioned earlier, the lock is held only for a fraction of the round robin quantum [17] to avoid the problem of tasks blocking while they are still holding locks.

The experiments performed with the dependent workload are the same as those performed for the independent workload. The performance of HSP, modified RRjob and equipartitioning is studied under varying levels of scheduling overhead and for varying levels of cumulative demand variance.

The effect of parameters like the round robin quantum and the ERF factor is not examined in this chapter, since these parameters are not workload specific and the system model used for these experiments is the same as that used for the independent workload model.

5.1 Effect of varying the scheduling overhead

Figure 5.1 shows the behaviour of the scheduling policies under increasing load levels for low scheduling overhead. It can be seen from the graph that the behaviour of the policies under this workload is similar to the behaviour for the independent workload. However, the response times for jobs, in all policies is greater than what is shown in Figure 4.1.

Figure 5.1 Response Time Vs. Utilization for low overhead.

This result is expected because no two tasks belonging to the same job can execute simultaneously due to the synchronization constraint. This delays all the tasks of a job in a manner shown in Figure 5.2. Figure 5.2 (a) shows the execution phases of tasks in which there is no synchronization required. In Figure 5.2 (b) tasks belong-

ing to a synchronizing workload are depicted, at low load levels, and Figure 5.2 (c) demonstrates the effect of increased load on the execution of tasks.

Figure 5.2 Effect of synchronization on task execution times (a) Non-synchronizing workload (b) Synchronization at low system load (c) Synchronization at high system load

As the number of tasks in a processor's ready queue increases, tasks that are not able to acquire their shared lock have to wait for longer times before they are rescheduled. This leads to a faster deterioration in response times as the system load increases. This effect is clearly exhibited in the graphs presented in this section. A comparison with the corresponding graphs of the previous chapter will show that the system saturates faster for this workload in all the policies.

However, it can also be observed that of the three policies HSP demonstrates the best performance and has acceptable response times for system loads as high as 90%. The other policies exhibit very poor performance after the load exceeds 70% utilization. Specifically, an improvement of approximately 40% is observed over the RRjob

policy. At this low overhead level equipartition has response times approximately 50% lower than HSP.

Figures 5.3 and 5.4 depict the performance of the policies under increasing levels of overhead and it can be seen from the graphs that the performance trends are similar to what was observed for the independent workload, although the average response times are higher. The equipartitioning policy is very sensitive to the scheduling overhead and is not comparable with the other policies at moderate to high system utilization. Again HSP depicts the best performance of all three policies. It is relatively insensitive to increases in scheduling overhead due to its hierarchical queue organization and self-scheduling nature.

Figure 5.3 Response Time Vs. Utilization for medium overhead.

The modified RRjob policy also exhibits reasonable performance for light to moderate load levels but performance degrades rapidly as the system load exceeds approximately 50%. At these overhead levels HSP exhibits about a 50% improvement over the other policies.

Figure 5.4 Response Time Vs. Utilization for high overhead.

Figure 5.5 compares the response times of all policies at different levels of scheduling overhead when the system utilization is fixed at about 72%. From this graph it is clear that HSP is quite stable for a large range of overhead levels and its response time remains almost constant as the overhead is increased. Modified RRjob and equipartition are far more sensitive to the overheads, mainly due to the centralized queue organization.

Figure 5.5. Response Time Vs. Overhead, at 72% utilization.

5.2 Effect of varying the C_{v_d}

As mentioned earlier, increasing the coefficient of variation of cumulative service demand, has the effect of introducing a few jobs with very high cumulative demands into the job stream. It is important to study the effect, on performance, of the policy in a scenario where there might be a few very large jobs because in any real multiprocessor installation it is reasonable to assume that such jobs would arrive from time to time.

Figure 5.6 depicts the graph of response time vs. utilization when the C_{v_d} is one. This graph is the same as Figure 5.1, and it can be seen from this graph that the

performance of all the policies is fairly stable over a large range of system utilization. HSP is least sensitive to increasing system load, even though at this level of negligible overhead its response times are slightly higher than the equipartition policy.

Figure 5.6 Response Time Vs. Utilization for $C_{v_d} = 1$.

Figures 5.7 and 5.8 demonstrate the effect of gradually increasing the variance of demand. It can be seen from the graphs that again, HSP is fairly stable and relatively insensitive to increasing values of C_{v_d} . This observation is the same as what was made for the independent workload and can be attributed to the fact that due to the hierarchical queue organization, HSP quickly moves tasks from large jobs to lower levels of the tree and its combined space and time sharing capabilities overcome the effect of a few tasks with large service times. At this level of overhead the equipartition policy is also relatively insensitive to increasing C_{v_d} values, although

the effect of scheduler and central queue saturation can be observed even at moderate system loads, whereas for HSP the response times are fairly reasonable even at high system loads.

Figure 5.7 Response Time Vs. Utilization for $C_{v_d} = 3$.

In the dependent workload, increasing the variance of demand has the most pronounced effect on the modified RRjob policy. The manner in which processor time is allocated to tasks is to a great extent responsible for this behaviour. Tasks with large service demands remain in the processors' ready queues for much longer durations since they are allocated the smallest round robin quanta. Also, since the service demand of some tasks might be very large, other tasks are also forced to wait for service for longer times.

Figure 5.8 Response Time Vs. Utilization for $C_{v_d} = 4$.

Figure 5.9 Response Time Vs. C_{v_d} at 72% utilization.

The synchronization requirement further causes performance to deteriorate since at

moderate to high system loads large tasks that cannot acquire locks remain in the queue for longer times. Since the processors are busy most of the time and tasks depart at slower rates, the central queue soon becomes very large and this further increases response times.

In figure 5.9 the effect of increasing C_{v_d} on the response time is presented at a fixed system load of 72%. It is clear from this graph that HSP is the least sensitive to variance in job demand and is superior to the other two policies at moderate to high system utilization.

5.3 Summary of results

The results of this chapter are similar to those obtained for the independent workload.

It can be observed from the graphs presented in this chapter that:

- At low overhead values, the response times for the space sharing policy are the lowest for low to moderate system load levels.
- As the overhead is increased the performance of HSP is clearly superior to the other policies over a wide range of system utilization values.
- HSP is the least sensitive to increasing overhead values.
- HSP is also insensitive to high variance in service demand, whereas the time sharing modified RRJob, and the space sharing equipartition policy are affected

to a great extent as the C_{v_d} is increased.

The results of this chapter, along with the results of the previous chapter support the conclusion that a hierarchical queue organization coupled with self scheduling causes significant performance improvement for both synchronizing and non-synchronizing parallel workloads compared to policies that use centralized queuing with either spatial or temporal sharing of processing power.

In the next chapter the results of this thesis are briefly summarized.

Chapter 6

Conclusions

6.1 Summary

The goal of this thesis was to devise a scheduling policy for shared memory NUMA systems that eliminated some serious drawbacks of other well-known time sharing and space sharing policies. Details regarding the behaviour and performance of the proposed hierarchical scheduling policy were presented in the preceding chapters of the thesis. The main features of the new policy, referred to as HSP are the following.

- HSP eliminates the central job queue that becomes a performance bottleneck in other common scheduling policies. The removal of the centralized queue eliminates the queue access contention and its associated delays, leading to marked improvements in performance. Quantitatively, performance improvements of upto 55% are observed over the modified RRjob policy. At moderate over-

head levels the performance improvement over the equipartitioning policy is also greater than 50%.

- HSP uses a hierarchical queue organization that among its many benefits also has the extremely attractive feature of scalability. If a centralized queue structure is used for all incoming jobs, the system size cannot be increased beyond a modest number of processors as that would lead to a large contention overhead for the central queue. On the other hand, in HSP this limitation does not exist.
- HSP is a self scheduling policy and processors look for work whenever they are idle. This eliminates the need for a central scheduler, making the system more fault tolerant. Another advantage of self scheduling is that at moderate to high system loads a central scheduler might get saturated, but this does not occur in self scheduling.
- Purely time sharing policies are in general inferior to purely space sharing policies when the service time variance is not high. On the other hand space sharing policies perform poorly when variance in service times is high. HSP combines space sharing with time sharing, and the results of the previous chapter indicate that as a result its performance is superior to the other policies over a wide range of system parameters.

In the preceding chapters the performance of HSP was studied for a wide a range of system parameters. Observations regarding the performance of HSP are presented below.

- HSP is relatively insensitive to scheduling and other system overheads compared to the other policies that were studied. Acceptable response times are obtained with HSP over a wide range of system utilization levels and overhead levels for both the independent and the dependent workload models.
- HSP is also insensitive to the variance in job service demands for both workloads and over a wide range of system load levels. This is unlike the RRjob policy and the equipartitioning policy.
- The behaviour of HSP under varying conditions of job parallelism, round robin quantum size and the execution rate function was also studied and it was observed that in general, if the scheduling overheads were not negligible and the system load was not very light, HSP exhibited better performance than equipartitioning and RRjob.

6.2 Contributions of this thesis

A new scheduling policy for shared memory multiprocessor systems was proposed in this thesis. This scheduling policy, known as HSP was evaluated using simulation experiments and compared with other known policies. The results of the experiments indicated that HSP has several advantages over the policies known till now.

It is evident from the results of this thesis that HSP, which uses a hierarchical queue organization and self scheduling to combine the features of both spatial and

temporal sharing is capable of significant performance improvements for parallel applications executing on medium to large sized shared memory multiprocessor systems.

6.3 Areas for future work

There are several aspects of HSP that need to be studied or expanded in the future. With increasing work in the characterization of parallel workloads, the performance of HSP needs to be analyzed under a wider range of workload conditions.

One of the features of the space sharing equipartitioning policy that improves its performance is the fact that newly arriving jobs are given priority over older jobs that have already received some processor time. In HSP this feature is absent. Tasks are removed from the root queue in a first come first served manner. This priority scheme needs to be built into HSP in the future to study how it affects performance.

Currently the transfer factor TR is fixed at one i.e., only as many tasks are moved down a level of the tree as there are processors under that level. It is felt that making the TR dependent on the system load might improve the performance of HSP by achieving a higher degree of load balancing. In other words, if the load in one partition, say P_a is higher than the load in another partition, say P_b , then TR for P_a would be reduced and the TR for P_b would be correspondingly increased. It is expected that this might improve the performance of HSP especially at higher system loads.

It would also be interesting to study how the performance of HSP is affected under memory constrained situations i.e., situations where a certain amount of main memory must be available for a task to execute on a processor. In most research done so far the memory constraint has generally been ignored and to do a more accurate analysis of scheduling policies it is necessary to consider the memory requirements of tasks.

The concepts of HSP can also be easily extended to distributed memory systems and networks of workstations and this is intended to be the main focus of future research in the area of hierarchical scheduling.

Bibliography

- [1] I. Ahmad, A. Ghafoor, and G. C. Fox, "Hierarchical Scheduling of Dynamic Parallel Computations on Hypercube Multicomputers", *Journal of Parallel and Distributed Computing*, 20, 1994, pp. 317-329.
- [2] G. S. Almasi, and A. Gottlieb, *Highly Parallel Computing*, Benjamin Cummings, Redwood City, CA., 1989.
- [3] T. S. Au, *Characterization of Classes of Parallel Programs and their Use in Performance Evaluation of Scheduling Policies*, Master's Thesis, School of Computer Science, Carleton University, Ottawa, Canada, 1992.
- [4] T. B. Brecht, "Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors", Technical Report CSRI-303, Computer Systems Research Institute, University of Toronto, Toronto, Canada, 1994.
- [5] S. P. Cheng, *Performance Evaluation of a Hierarchical Task Queue Organization for Multiprocessor Systems*, Masters Thesis, School of Computer Science, Carleton University, Ottawa, Canada, 1992.

- [6] S. P. Cheng, and S. P. Dandamudi, "Scheduling in Parallel Systems with a Hierarchical Organization of Tasks", *Proceedings of the 12th International Conference on Distributed Computer Systems*, Jun. 1992.
- [7] S. Chiang, R. K. Mansharamani, and M. K. Vernon, "Use of Application Characteristics and Limited Pre-emption for Run-To-Completion Parallel Processor Scheduling Policies", *Proceedings of the ACM SIGMETRICS Conference*, 1994, pp. 33-44.
- [8] S. P. Dandamudi, "A Comparison of Task Scheduling Strategies for Multiprocessor Systems", *IEEE Symposium on Parallel and Distributed Processing*, Dallas, TX, Dec. 1991, pp 423-426.
- [9] 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*, Vol. 6, No. 1, Jan. 1995, pp. 1-16.
- [10] L. Dowdy, "On the Partitioning of Multiprocessor Systems", Technical Report, Vanderbilt University, Jul. 1988.
- [11] K. Dussa, B. Carlson, L. Dowdy, and K.-H. Park, "Dynamic Partitioning in a Transputer Environment", *Proceedings of the ACM SIGMETRICS*, May 1990, pp. 203-213.
- [12] R. Duncan, "A Survey of Parallel Computer Architectures", *IEEE Computer*, Feb. 1990, pp 5-16.

- [13] D. G. Feitelson, and L. Rudolph, "Distributed Hierarchical Control for Parallel Processing", *IEEE Computer*, 23(5), May 1990, pp. 65-77.
- [14] M. J. Flynn, "Some Computer Organizations and their Effectiveness", *IEEE Transactions on Computers*, C-21, No.9, Sept. 1972, pp. 948-960.
- [15] D. Ghosal, G. Serazzi, and S. K. Tripathi, "The Processor Working Set and Its Use in Scheduling Multiprocessor Systems", *IEEE Transactions on Software Engineering*, Vol. 17, No. 5, May 1991.
- [16] A. Gupta, A. Tucker, and S. Urushibara, "The Impact of Operating System Policies and Synchronization Methods on the Performance of Parallel Applications", *Proceedings of the ACM SIGMETRICS Conference*, May 1991, pp. 120-132.
- [17] S. T. Leutenegger, and M. K. Vernon, "The Performance of Multiprogrammed Multiprocessing Scheduling Policies", *Proceedings of the ACM SIGMETRICS Conference*, May 1990, pp. 226-236.
- [18] S. Majumdar, D. L. Eager, and R. B. Bunt, "Scheduling in Multiprogrammed Parallel Systems", *Proceedings of the ACM SIGMETRICS Conference*, May 1988, pp.104-113.
- [19] C. McCann, R. Vaswani, and J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors", *ACM Transactions on Computer Systems*, Vol. 11, No. 2, May 1993, pp. 146-178.

- [20] C. McCann, and J. Zahorjan, "Processor Allocation Policies for Message-Passing Parallel Computers", *Proceedings of the ACM SIGMETRICS Conference*, May 1994, pp. 19-32.
- [21] J. M. Mellor-Crummey, and M. L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", University of Rochester, Computer Science Technical Report 342, Jan. 1991.
- [22] L. M. Ni, and C. E. Wu, "Design Tradeoffs for Process Scheduling in Shared-Memory Multiprocessor Systems", *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 3, Mar. 1989, pp. 327-334.
- [23] J. K. Ousterhout, "Scheduling techniques for Concurrent Systems", *Proceedings of the 3rd International Conference on Distributed Computing Systems*, 22-30, Oct. 1982, pp. 22-30.
- [24] E. Rosti, E. Smirni, L. W. Dowdy, G. Serrazi, and B. M. Carlson, "Robust Partitioning Policies of Multiprocessor Systems", *Performance Evaluation* 19, 1994, pp. 141-165.
- [25] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "Processor Scheduling on Multiprogrammed, Distributed Memory Parallel Computers", *Proceedings of the ACM SIGMETRICS Conference*, May 1993, pp. 158-170.
- [26] 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, Vol. 5, No. 4, Apr. 1994, pp. 401-420.

- [27] S. K. Setia, and S. K. Tripathi, "An Analysis of Several Processor Partitioning Policies on Parallel Computers", Technical Report CSTR-2684, University of Maryland, May 1991.
- [28] K. C. Sevcik, "Characterizations of Parallelism in Applications and their Use in Scheduling", *Proceedings of the ACM SIGMETRICS Conference*, May 1989, pp. 171-180.
- [29] K. C. Sevcik, "Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems", *Performance Evaluation - An International Journal* 19, 1994, pp. 107-140.
- [30] K. C. Sevcik, and S. Zhou, "Performance Benefits and Limitations of Large NUMA Multiprocessors", *Performance Evaluation - An International Journal* 20, 1994, pp. 185-205.
- [31] D. Skillicorn, "A Taxonomy for Computer Architectures", *IEEE Computer*, Nov. 1988, pp. 46-57.
- [32] M. S. Squillante, and R. D. Nelson, "Analysis of Task Migration in Shared Memory Multiprocessor Scheduling", *Proceedings of the ACM SIGMETRICS Conference*, May 1991, pp. 143-155.

- [33] 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*, Vol. 1, No. 3, Jul. 1990, pp. 286-303.
- [34] A. Tucker, and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors", *Proceedings of the 12th ACM Symposium on Operating System Principles*, Dec. 1989, pp. 159-166.
- [35] R. Vaswani, and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared-Memory Multiprocessors", *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, ACM, New York, May 1990.
- [36] J. Zahorjan, and C. McCann, "Processor Scheduling in Shared Memory Multiprocessors", *Proceedings of the ACM SIGMETRICS Conference*, May 1990, pp. 214-225.
- [37] J. Zahorjan, E. D. Lazowska, and D. L. Eager, "Spinning versus Blocking in Parallel Systems with Uncertainty", *Proceedings of the International Symposium on Performance of Distributed and Parallel Systems*, Kyoto, Japan, Dec. 1988.
- [38] J. Zahorjan, E. D. Lazowska, and D. L. Eager, "The Effect of Scheduling Discipline on Spin Overhead in Shared-Memory Parallel Systems", *IEEE Transactions on Parallel and Distributed Systems*, 2, 2, Apr. 1991, pp. 180-198.

- [39] S. Zhou, and T. Brecht, "Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors", *Proceedings of the ACM SIGMETRICS Conference*, May 1991, pp. 133-142.

END

0|4-06-9|6

FIN