PhD Dissertation, Technical Report CSRI-303, Computer Systems Research Institute, University of Toronto.

Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors

Timothy Benedict Brecht

Technical Report CSRI-303
June 1994

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors

Timothy Benedict Brecht

Technical Report CSRI-303
June 1994

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

The Computer Systems Research Institute (CSRI) is an interdisciplinary group formed to conduct research and development relevant to computer systems and their application. It is an Institute within the Faculty of Applied Science and Engineering, and the Faculty of Arts and Science, at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

This report is an adaptation of Tim Brecht's Ph.D. dissertation.

Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors

Timothy Benedict Brecht

A thesis submitted in conformity with the requirements for the
Degree of Doctor of Philosophy, Graduate Department of Computer Science,
University of Toronto, April, 1994

ABSTRACT

The invention, acceptance, and proliferation of multiprocessors are primarily a result of the quest to increase computer system performance. The most promising features of multiprocessors are their potential to solve problems faster than previously possible and to solve larger problems than previously possible. Large-scale multiprocessors offer the additional advantage of being able to execute multiple parallel applications simultaneously.

The execution time of a parallel application is directly related to the number of processors it is allocated and, in shared-memory non-uniform memory access time (NUMA) multiprocessors, which processors it is allocated. As a result, efficient and effective scheduling becomes critical to overall system performance. In fact, it is likely to be a contributing factor in ultimately determining the success or failure of shared-memory NUMA multiprocessors.

The subjects of this dissertation are the problems of processor allocation and application placement. The processor allocation problem involves determining the number of processors to allocate to each of several simultaneously executing parallel applications and possibly dynamically adjusting those allocations to improve overall system performance. The performance metric used is mean response time. We show that by differentiating between applications based on the amount of remaining work they have to execute, performance can be improved significantly. Then we propose techniques for estimating an application's expected remaining work along with policies for using these estimates to make improved processor allocation decisions. An experimental evaluation demonstrates the promise of this approach.

The placement problem involves determining which of the many processors to assign to each application. Using experiments conducted on a representative system, we demonstrate that in large-scale NUMA multiprocessors the execution time of parallel applications is significantly affected by the placement of the application. This motivates the need for new techniques designed explicitly for NUMA multiprocessors. We introduce such a technique, called processor pool-based scheduling, that is designed to localize the execution of parallel applications within a NUMA architecture and to isolate different parallel applications from each other. An experimental evaluation of this scheduling method shows that it can be used to significantly reduce mean response time over methods that do not consider the placement of parallel applications.

Acknowledgments

I would like to express my gratitude to my supervisor; Songnian Zhou, for introducing me to this area of research and for providing me with guidance, support, and understanding. I would also like to thank Ken Sevcik, who in many ways served as a co-supervisor (without making me feel like I had to serve two masters). His insightful comments and suggestions regarding the presentation and content and of this dissertation have improved it greatly.

My committee members: Christina Christara, Michael Stumm, and Dave Wortman provided valuable comments during the various checkpoints required for the completion of this degree. As well, I am thankful to my external examiner John Zahorjan for his quick and careful reading of the dissertation and for his suggestions for improvements.

I would like to thank Brian Carlson for many enjoyable hours spent discussing scheduling, statistics, and various proof approaches, as well as for providing valuable feedback on early drafts of this work. Thanks also to Tom Fairgrieve, to whom I often turned for help, most often with only a vague description of a minimization problem.

This research could not have been completed without the Hector and HURRICANE groups. Fortunately, the members of these groups have dedicated themselves to designing and building a multiprocessor and an Operating System, and to ensuring that they run and continue to run. Michael Stumm deserves special credit for essentially overseeing these projects and for his design and implementation contributions. The system would not have existed if it had not been for Michael, Ron White, Ron Unrau, Orran Krieger, Ben Gamsa, and Jonathan Hanna. Thanks also to James Pang, Deepinder Gill, Thomas Wong, and Ron Unrau for contributing some of the parallel applications used in the experiments.

Fortunately, my time at the U of T has not all been spent doing research. I wish to thank my friends, especially but not exclusively, Lucia, Tom, Diane, Rich, Toni, Sheila, Gord, Jim, Jennifer, and the DGP gang for providing me with lots of fun, food, and moral support during these years. Thanks also to my parents for instilling in me the drive to set new goals and the belief that I can achieve them.

I am also grateful to the Friday afternoon hockey gang for providing a stressed out graduate student with a place to take out some pent up frustrations. Tom Fairgrieve deserves special recognition for allowing me to beat him with the same dumb move for so many years, just so he could shut me down this year.

NSERC and OGS provided some of the financial support without which this work would not have been possible.

Portions of Chapter 3 have been developed from a paper originally published in the USENIX Association Conference Proceedings, 1993. ‘‘On the Importance of Parallel Application Placement in NUMA Multiprocessors’’, from the *Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS-IV)*, September, 1993, pp. 1-18. Permission has been granted by the USENIX Association to reprint this paper. Copyright © USENIX Association, 1993.

Table of Contents

Chapter 1

Introduction 1

1.1. Motivation 1

1.2. Goals 3

1.3. Contributions 4

1.4. Overview of the Dissertation 5

Chapter 2

Background 7

2.1. Introduction 7

2.2. Multiprogramming Parallel Applications 7

2.2.1. Static versus Dynamic Scheduling 8

2.2.2. Time Sharing versus Space Sharing 9

2.3. Characterization of Parallel Programs 9

2.3.1. Parallelism Profile 10

2.3.2. Speedup and Efficiency 10

2.3.3. Work to be Executed 14

2.4. Scheduling in NUMA Multiprocessors 15

2.4.1. Large-Scale Shared-Memory Multiprocessors 15

2.4.2. Software Scalability 17

2.4.3. Localizing Data Accesses 18

2.5. The Experimental Environment 20

2.6. Applications Used in Experiments 22

2.7. Summary 27

Chapter 3

Processor Partitioning 29

3.1. Introduction 29

3.2. Motivation 29

3.3. Simplifying Assumptions 30

3.4. Scheduling without Reallocations (Static Partitioning) 31

3.4.1. Equal Partitions (EQUI) 32

3.4.2. Proportional Partitions (PROP) 33

3.4.3. Comparing PROP with EQUI 34

3.4.4. Optimal Partitions (ROOT) 35

3.4.5. Comparing ROOT with EQUI 37

3.5. Scheduling with Reallocations (Dynamic Partitioning) 39

3.5.1. Dynamic Equipartition (DYN-EQUI) 39

3.5.2. Least Work First (LWF) 40

3.5.3. Comparing LWF with DYN-EQUI 41

3.5.4. Substantiating the Analysis 47

3.6. Considering New Arrivals 49

3.7. Relaxing Perfect Efficiency 53

3.8. Summary 54

Chapter 4

Practical Partitioning Strategies 55

4.1. Introduction 55

4.2. Reevaluating Dynamic Equipartition 55

4.2.1. Considering New Arrivals 55

4.2.2. Considering Imperfect Efficiency 57

4.3. Estimating Expected Remaining Work 60

4.3.1. Using User Supplied Estimates 61

4.3.2. Using System Maintained Logs 62

4.3.3. Using the Run-Time System 62

4.3.3.1. Applicability 65

4.3.3.2. Experimental Results 66
4.3.3.3. Sources of Inaccuracy 71
4.3.4. Using Past Execution 71
4.3.4.1. Cumulative Execution Time 72
4.3.4.2. Generalizing the Partitioning Policies 75
4.4. Summary 78

Chapter 5

Application Placement 79
5.1. Introduction 79
5.2. Application Placement for Localization 79
5.3. Impacts of Placement on Performance 81
5.3.1. Increasing Memory Latencies 84
5.3.2. NUMAness 86
5.3.3. System Size 88
5.3.4. Placement of Processes within the Application 90
5.4. Summary 91

Chapter 6

Processor Pool-Based Scheduling 93
6.1. Introduction and Motivation 93
6.2. Processor Pools 94
6.3. Assumptions 95
6.4. Forming Processor Pools 97
6.4.1. Experimental Setting 97
6.4.2. Determining Processor Pool Sizes 98
6.4.3. Application Influences on Pool Size 100
6.4.4. Architectural Influences on Pool Size 103
6.5. Using Processor Pools 107
6.5.1. Initial Placement 107
6.5.2. Expansion 110

6.5.3. Repartitioning 110
6.6. Summary 113

Chapter 7

Conclusions 115
7.1. Introduction 115
7.2. Summary of Contributions 115
7.3. Future Work 118

Appendix 120
Bibliography 123

Glossary

J_i - job i

a_i - arrival time of job i

P - number of processors in the system

p_i - number of processors allocated to job i

$p_i(t)$ - number of processors allocated to job i at time t

N - number of jobs being executed

$n(t)$ - number of jobs being executed at time t

W_i - work executed by job i

$W_i(t)$ - work remaining to be executed by job i at time t

R_i - response time of job i

$\overline{R}$ - mean response time $= \frac{1}{N} \sum_{i=1}^{N} R_i$

$R_i^{(E)}$ - response time of job i using EQUI scheduling policy

$\overline{R}^{(E)}$ - mean response time using EQUI scheduling policy

$R_i^{(R)}$ - response time of job i using ROOT scheduling policy

$\overline{R}^{(R)}$ - mean response time using ROOT scheduling policy

$R_i^{(P)}$ - response time of job i using PROP scheduling policy

$\overline{R}^{(P)}$ - mean response time using PROP scheduling policy

$R_i^{(D)}$ - response time of job i using DYN-EQUI scheduling policy

$\overline{R}^{(D)}$ - mean response time using DYN-EQUI scheduling policy

$R_i^{(L)}$ - response time of job i using LWF scheduling policy

$\bar{R}^{(L)}$ - mean response time using LWF scheduling policy

$R_i^{(LR)}$ - response time of job i using LRWF scheduling policy

$\bar{R}^{(LR)}$ - mean response time using LRWF scheduling policy

$\bar{R}^{(P/E)}$ - the mean response time ratio $\frac{\bar{R}^{(P)}}{\bar{R}^{(E)}}$

$\bar{R}^{(R/E)}$ - the mean response time ratio $\frac{\bar{R}^{(R)}}{\bar{R}^{(E)}}$

$\bar{R}^{(L/D)}$ - the mean response time ratio $\frac{\bar{R}^{(L)}}{\bar{R}^{(D)}}$

$\bar{R}^{(LR/D)}$ - the mean response time ratio $\frac{\bar{R}^{(LR)}}{\bar{R}^{(D)}}$

H_k - the k-th harmonic number $= \sum_{i=k}^{N} \frac{1}{k}$

$T(1)$ - execution time on 1 processor (sequential version)

$T(P)$ - execution time on P processors

$S(P)$ - speedup on P processors $= \frac{T(1)}{T(P)}$

$E(P)$ - efficiency on P processors $= \frac{S(P)}{P}$

Chapter 1

Introduction

1.1. Motivation

This dissertation is concerned with the scheduling of multiple parallel applications in large-scale shared-memory Non-Uniform Memory Access time (NUMA) multiprocessors. In particular, this dissertation focuses on problems of determining the number of processors to assign to each application (the allocation problem) and which processors to assign to each application (the placement problem).

Small-scale, shared-memory multiprocessors based on a single shared bus have become prevalent and the number of manufacturers building and selling such products continues to rise. The success of these systems can be partially attributed to the relatively simple parallel programming model they present, compared to strictly message-passing parallel systems. This simple programming model has allowed many applications to achieve substantial increases in performance by making effective use of all of the processors in the system.

The lure of substantial reductions in processing time through the use of multiprocessors continues to fuel the desire for greater and greater performance improvements. One approach to increasing performance is to simply build larger and larger systems. Single-bus systems, however, are not scalable because the bandwidth of the bus limits their size. As a result, research and design efforts in shared-memory multiprocessors have focused on scalable architectures. These architectures distribute memory modules throughout the system in order to optimize access times to some memory locations. The result is an important class of scalable shared-memory systems known as NUMA multiprocessors. (Alternatively all memory accesses could be made uniform, but then they would be uniformly slow.)

The emergence of large-scale shared-memory multiprocessors presents new opportunities and challenges. The opportunities are to solve much larger problems than previously possible by executing applications that are capable of effectively utilizing large numbers of processors, and to solve a number of different problems concurrently by simultaneously executing multiple parallel applications that may not execute efficiently using all of the processors. (One argument

for building large-scale multiprocessors, rather than using a number of smaller systems, is that large-systems offer a relatively simple, cost effective, and flexible means for providing and sharing expensive resources like processors, memory, and disks.) One of the main challenges is to productively utilize the processors while affording the efficient execution of multiple applications. The effective scheduling of the parallel processes of these applications is central to the performance of such systems. *Scheduling* involves determining the number of jobs to execute simultaneously (i.e., when to activate jobs), the number of processors to allocate to each of those jobs (allocation), and which processors should be used to execute those jobs (placement).

One method of multiprogramming parallel applications is to alternate their execution by assigning processors to each application, in turn, for a period of time called a time-slice. Applications are thus executed in a rotating round-robin fashion, ensuring that cooperating processes of each application execute simultaneously [Ousterhout1982]. That is, processors are synchronously time-shared among applications. Using this technique, each processor incurs context switching overheads at every time-slice. These overheads can be avoided by dividing the system across processors rather than across time; that is, to space-share rather than time-share processors. The space-sharing, or *partitioning*, of processors is accomplished by allocating different portions of the system to different applications [Tucker1989]. This eliminates the need to alternate execution between applications and avoids unnecessary context switching overheads. If processors are space-shared, a key factor in determining the execution time of an application and ultimately overall system performance is:

- **The Allocation Problem:**
  How many processors should be allocated to each application?
  That is, how large should the processor partition be for each application?

Scheduling techniques for small-scale, UMA (Uniform Memory Access time) multiprocessors have traditionally treated all processors equally, since the time to access any address in main memory is the same from any processor. As the gap between processor speeds and memory access times has grown, more recent work has examined scheduling techniques that exploit cache contexts when scheduling loops and threads within an application. An obvious but critical difference between scheduling in UMA and NUMA multiprocessors is that scheduling decisions in NUMA systems must also consider the time it takes to access different memory locations from different processors. Thus, NUMA scheduling policies must consider the latency incurred during remote communication (in some systems determined by the number of levels in

the memory access hierarchy) and to the extent possible preserve the locality of data references inherent in parallel applications. Therefore, an important aspect of scheduling in NUMA multiprocessors is:

- **The Placement Problem:**
  Which processors are assigned to each application?
  That is, how should the parallel processes of an application be placed or located in a NUMA multiprocessor?

The execution time of a parallel application is directly related to the number of processors it is allocated and, in NUMA systems, which processors it is allocated. As a result, efficient and effective scheduling becomes critical to overall system performance. In fact, it is likely to be a contributing factor in ultimately determining the success or failure of NUMA multiprocessors. Additionally, if scheduling techniques designed for such systems are to have a lasting impact they must be scalable, especially since one of the goals of many current architecture designs is scalability.

1.2. Goals

The main goal of this dissertation is to investigate and gain a better understanding of the factors involved in designing and implementing scheduling methods for NUMA multiprocessors.

Several analytic and simulation studies have demonstrated the importance and benefits of knowing and using application characteristics when scheduling parallel applications [Majumdar1988] [Eager1989] [Sevcik1989] [Zahorjan1990] [Sevcik1994]. However, most popular implementable scheduling techniques continue to treat all applications equally, mainly because of a lack of practical techniques for obtaining and using application characteristics (for example, the process control approach [Tucker1989]).

Prior to the emergence of NUMA multiprocessors, all processors could be treated equally because the execution times of applications were not affected by which processor they executed on (ignoring the affects of cache contexts). Thus scheduling decisions made by the operating system have traditionally amounted to simply determining how many processors to allocate to each application.

Consequently, an additional goal of this dissertation is to demonstrate that knowledge of application and architectural characteristics can be obtained and used when making scheduling decisions and that mean response time of parallel applications is improved as a result.

1.3. Contributions

The main contributions of this dissertation are:

Effective processor partitioning

We analytically compare currently popular processor partitioning techniques that allocate an equal portion of processors to all applications that have enough parallelism [Tucker1989] [Zahorjan1990] [McCann1993] to techniques that partition processors according to the amount of work each application will perform. Our results show that, under certain conditions, allocating processors in proportion to the amount of work they perform can significantly improve mean response time.

Obtaining and using application characteristics

The potential for significant performance improvements as a result of knowing the amount of work an application executes motivates the need for practical techniques for estimating expected remaining work. We propose, implement, and experimentally evaluate a number of such techniques along with several scheduling policies that use these estimates. The results of our experiments show that obtaining estimates of expected remaining work is feasible and that these estimates can be used in combination with policies that allocate unequal portions of processors to improve mean response time when compared with an equipartition policy.

Demonstrating the importance of application placement

Using a representative scalable multiprocessor, we demonstrate how the execution time of parallel applications is affected by which processors are used to execute the application. Our experiments show that proper application placement must consider the underlying architecture and the natural clusters of the system and that achieving data reference locality involves coordination between many parts of the system, especially between the memory manager and the operating system scheduler. The results of these experiments also show that the importance of application placement increases as the time required to access remote memory references increases.

Processor pool-based scheduling

A new class of scheduling techniques for performing effective application placement, called processor pool-based scheduling, is developed, implemented and evaluated using a representative prototype scalable multiprocessor. Experimental results show that processor pools are an effective method for improving application placement and reducing mean response time. The benefits from using processor pool-based scheduling increase with the size and NUMAness of the system.

1.4. Overview of the Dissertation

This dissertation investigates some of the main factors that determine the execution time of parallel applications:

- **Allocation:** How many processors should be allocated to each application?
- **Placement:** Which processors should be allocated to each application?
- **Scale:** Are the proposed techniques applicable to increasingly larger systems?

The number of processors to allocate to an application and the placement of an application are scheduling decisions that depend mainly on the following application characteristics:

- **Work:** The total amount of work to be executed by an application.
- **Efficiency:** The efficiency with which processors can be effectively utilized by an application. (This includes overheads such as those due to communication, synchronization, and load imbalance.)

In this dissertation, we explore some of these factors in order to gain insight into effective scheduling techniques for shared-memory, NUMA multiprocessors. The main purpose for these explorations is not to determine optimal scheduling strategies but to improve upon existing implementable scheduling techniques.

The structure of the thesis is as follows: In Chapter 2 we present some of the background required to examine these problems, survey relevant multiprocessor scheduling techniques, and describe the Hector multiprocessor, HURRICANE operating system, and parallel applications used to conduct the experiments presented in this dissertation. In Chapter 3, we analyze a currently popular scheduling policy that proposes sharing processors equally among applications, provided they have sufficient levels of parallelism. We show that knowledge of a simple application characteristic, the amount of remaining work, can be used to derive new policies that substantially improve mean response time. The results obtained in Chapter 3 motivate the work in Chapter 4, in which new adaptive processor partitioning techniques are described. The idea behind adaptive partitioning techniques is to try to distinguish among different applications and assign unequal processor partitions in order to improve mean response time. An experimental evaluation of these strategies is performed. The results of Chapter 3 and Chapter 4 are not specific to NUMA multiprocessors and can be applied to UMA systems. As well, some of the techniques for determining processor allocations can be easily tailored for strictly message-passing parallel systems.

Chapter 5 examines the problem of application placement in NUMA multiprocessors. Experiments are conducted on a representative scalable, shared-memory, NUMA multiprocessor to show that placement substantially affects an application's performance. Since existing scheduling techniques have not considered scheduling applications to reduce the cost of required accesses to shared data, Chapter 6 proposes a class of scheduling algorithms designed to preserve the locality of data references inherent in parallel applications. The concept of processor pools is introduced to enhance the locality of data references within applications, reduce interference between applications, and reduce the likelihood of bottlenecks in the scheduling sub-system. A number of processor pool-based scheduling policies are proposed and experimentally evaluated. The dissertation is concluded in Chapter 7 with a summary of the results, the contributions of this work, and a discussion of future research.

Chapter 2

Background

2.1. Introduction

The invention, acceptance, and proliferation of multiprocessor computer systems are primarily driven by the desire for increased performance. The most promising features of multiprocessors are their potential to solve larger problems than previously possible and to solve problems faster than previously possible. Large-scale multiprocessors offer the additional benefit of being able to execute multiple parallel applications simultaneously.

In this chapter we first provide an overview of existing techniques for scheduling multiprogrammed parallel applications. Then we discuss parallel applications and the factors that determine their execution time and we briefly outline some of the research that advocates the use of these factors (or measures that somehow characterize an application's execution) when making scheduling decisions. Because most existing scheduling techniques are designed for UMA architectures, we then discuss how the new and emerging class of NUMA multiprocessors influences and complicates scheduling decisions. Then we describe some existing NUMA architectures and research that is relevant to scheduling for NUMA systems. The experimental environment is then described, including the Hector architecture, the HURRICANE operating system, and the parallel applications used.

2.2. Multiprogramming Parallel Applications

As the number of processors available in multiprocessors continues to grow it is increasingly likely that many applications will not be capable of effectively utilizing all of the processors in the system. The effective simultaneous execution of multiple parallel applications is, therefore, a research area of growing interest and importance as is evident by the number of recent studies concerned with multiprogramming parallel applications. We briefly describe different approaches to multiprogramming parallel applications while emphasizing those techniques that have gained general acceptance and are, consequently, the methods upon which our work is based.

2.2.1. Static versus Dynamic Scheduling

A range of approaches can be taken with the assignment of processors to applications. At one end of this range *static scheduling* policies assign processors to an application for the lifetime of the application. This approach is more popular in systems in which the cost of reallocating processors is high; message passing systems, for example. A static allocation of processors can result in the idling of processors when they are not being used by the application to which they were assigned.

A *dynamic scheduling* approach adjusts the number of processors assigned to each application during their execution. If the costs of dynamically reallocating processors is not too high, dynamic scheduling techniques can lead to performance improvements because of higher processor utilization. Dynamic scheduling techniques make use of user-level, application-initiated methods for decomposing the problem into a number of units of execution. The number of these units is independent of the number of processors (usually far exceeding the number of processors). Two examples of such methods are the WorkCrews approach [Vandevoorde1988] and the Supervisors approach [Junkin1989]. These techniques are similar in that the application creates some number of *worker* processes that are responsible for the execution of conceptual units of work (called threads). The number of worker processes is usually limited to the number of available processors, while the number of threads used to solve the problem is independent of the number of processes. Threads are placed in a work queue and are removed and executed by worker processes. Upon completion of a thread, workers return to the queue for more work and continue in this manner until the queue is empty and all threads have been completed.

This work queue approach to parallel programming as been extended and generalized. The result is a number of user-level thread systems [Cooper1987] [Doeppner Jr.1987] [Bershad1988]. If the number of user-level threads exceeds the number of processors in the system, dynamic scheduling methods can be used in conjunction with user-level threads to adjust the number of processors allocated to each application during their execution [Tucker1989] [Zahorjan1990]. In the HURRICANE system used in conducting the experiments in this dissertation, this is accomplished by not permitting the worker process to continue (by descheduling it). (This is done in coordination with the thread package when the worker process has completed one thread and is about to dequeue and execute the next thread.)

Simulation and experimental evaluations of static and dynamic scheduling algorithms conclude that, in a UMA multiprocessor, dynamic scheduling is preferable to static scheduling [Zahorjan1990] [McCann1993]. Since the cost of reallocating processors in NUMA systems is likely higher than for UMA systems, it is important to consider how dynamic scheduling policies will perform in NUMA environments. The simulations conducted by Zahorjan and McCann [Zahorjan1990] indicate that, in UMA systems, dynamic scheduling is preferable to static

scheduling, even when reallocation overheads are quite large. As well, recent experimental work by Wu [Wu1993] on a NUMA multiprocessor lends support for the use of dynamic scheduling. Therefore, it is likely that some degree of processor reallocation can be used in NUMA systems. However, before a fair comparison can be made, we first need to consider what constitutes good scheduling policies in a NUMA environment and develop effective static and dynamic scheduling algorithms that can be used for comparison. Although we do not conduct a comparison, static and dynamic scheduling in NUMA environments are considered in Chapters 5 and 6 of this dissertation.

2.2.2. Time Sharing versus Space Sharing

A number of studies have examined different techniques for multiprogramming parallel applications in shared-memory multiprocessors. These techniques fall under two general categories, time-sharing policies and space-sharing policies. The difference between these two schemes lies in how the processors are shared among the many executing parallel applications. In a *time-sharing* scheme processors are shared over time by executing different applications on the same processor during different time intervals. One example of a time-sharing policy is *co-scheduling* [Ousterhout1982], which is designed to ensure that groups of cooperating processes are assigned to processors at the same time, in order to reduce delays incurred when exchanging messages. A *space-sharing*, or *processor-partitioning*, scheme divides processors among applications so that each application executes on a portion of the processors. Tucker and Gupta advocate matching the number of executing processes with the number of processors, ensuring that each processor executes only one process at a time, thus avoiding unnecessary context switch overheads, improving cache hit rates, and reducing mean response time [Tucker1989]. A number of simulation and experimental studies agree that, under a wide variety of circumstances, space-sharing is preferred to time-sharing [Tucker1989] [Gupta1991] [Crovella1991] [McCann1993]. Much of our work is, therefore, based on space-sharing rather than time-sharing processors.

2.3. Characterization of Parallel Programs

One of the main reasons for using a multiprocessor is to increase the speed of execution of an application over that of a serial machine by performing portions of the computation in parallel. Serial programs running in a uniprocessor environment are relatively well understood and characterized. Parallel programming and multiprocessor systems, on the other hand, are relatively new and as a result, are not as well understood. Earlier research focused more on understanding and characterizing the execution and performance of parallel applications in uniprogrammed environments [Amdahl1967] [Kumar1988]. However, more recent studies have considered how simple characterizations of a parallel application’s execution and performance

can be used to make scheduling decisions in a multiprogrammed environment [Majumdar1988] [Eager1989] [Sevcik1989]. We now present some commonly used measures of parallel application execution and performance and discuss how these measures are being characterized and used in making scheduling decisions.

2.3.1. Parallelism Profile

A *parallelism profile* plots the number of processors utilized by an application as a function of time (see Figure 2.1). It can be determined by executing an application with a sufficiently large number of processors and either keeping track of or periodically sampling the number of busy processors. Although a substantial amount of information about an application's execution can be obtained from the parallelism profile (e.g., the execution time, the maximum parallelism, the fraction of sequential computation), it would be difficult to use when making scheduling decisions because of the large amount of data used to represent the application's execution. For this reason, other measures of an application's execution are more widely used.

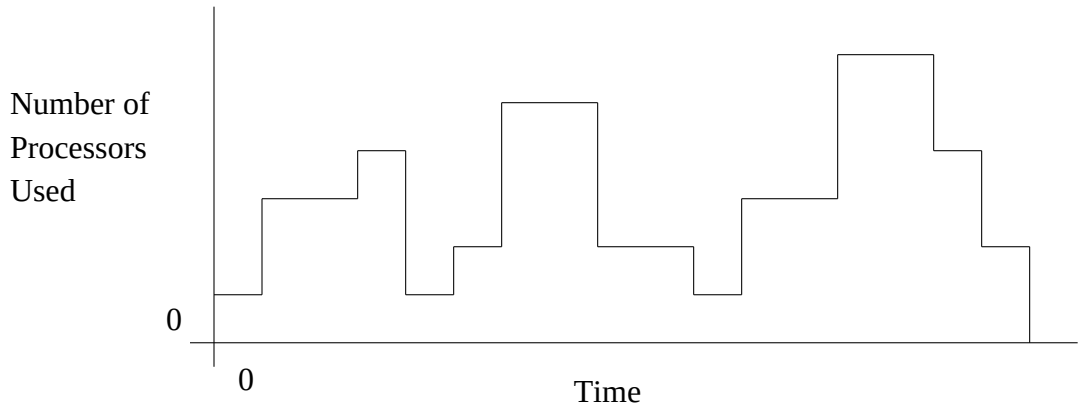


Figure 2.1: Parallelism Profile

2.3.2. Speedup and Efficiency

It is often a programmer's goal to write a parallel application such that when executing on P processors it runs P times faster than when executing the best known serial algorithm on one processor. A common measure of this performance is called *speedup*, which is defined to be the ratio of execution time attained using one processor, $T(1)$, to the execution time using P processors, $T(P)$.

$$S(P) = \frac{T(1)}{T(P)}$$

Although the theoretical limit to speedup on P processors is P (assuming that super-linear speedup is not possible), achieving that limit is not feasible for most applications. This is often due to factors such as: contention for the interconnection network, communication overhead, locking of shared resources, synchronization costs, and inherent limits in the parallelism of the problem. That is, the efficiency with which parallel applications execute often decreases as the number of processors is increased. Figure 2.2 shows speedup curves for some applications used in later experiments. (These applications are described in the sections in which they are used.) Note that as the number of processors is increased, the gain in performance per additional processor often decreases. In fact, in some cases adding too many processors can actually increase the execution time of the application.

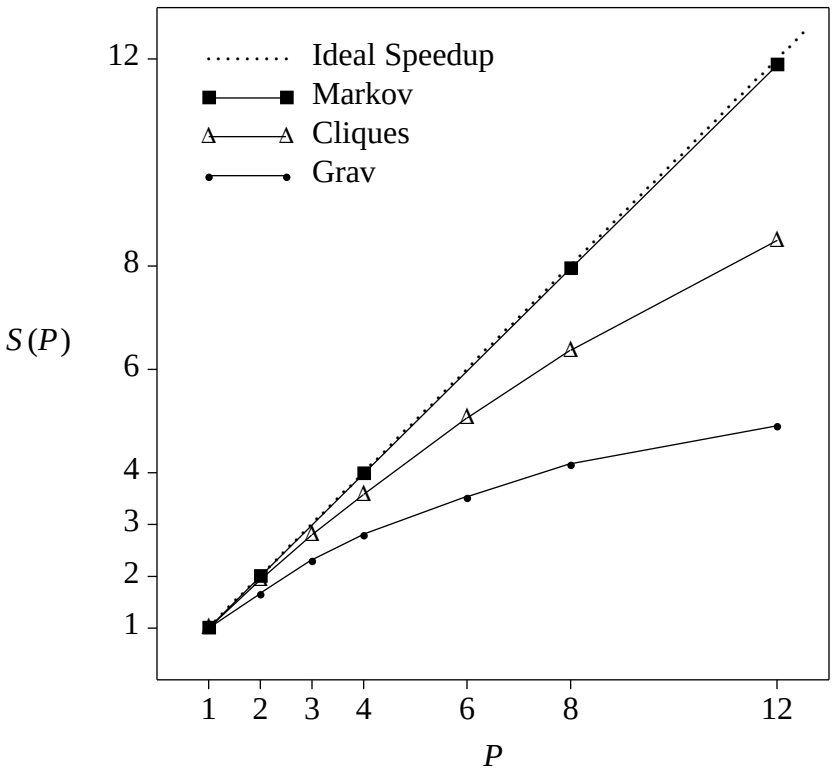


Figure 2.2: Speedup versus the number of processors used

The performance gained by adding a number of processors to an application may not be large enough to justify their use for that application. In fact, those processors might be used more effectively by other jobs. *Efficiency*, $E(P)$, is another measure of an application's performance. The efficiency of an application is related to its speedup as shown in the following equation:

$$E(P) = \frac{S(P)}{P} \quad .$$

Efficiency can be thought of as the mean *effective* processor utilization when P processors are allocated to the job [Eager1989].

Where speedup is a measure of the ‘‘benefit’’ of using some number of processors for the parallel execution of a job, the efficiency is a measure of the ‘‘cost’’ of using those processors. In a multiprogrammed environment, processors can be used to benefit one job at the expense of other jobs in the system. Because the increase in communication and synchronization costs associated with the use of additional processors will eventually outweigh the benefits of decreased execution time, it is useful to examine a graph of the execution time versus the efficiency, called the *execution time–efficiency profile* [Eager1989]. Figure 2.3 shows an example of an execution time–efficiency profile.

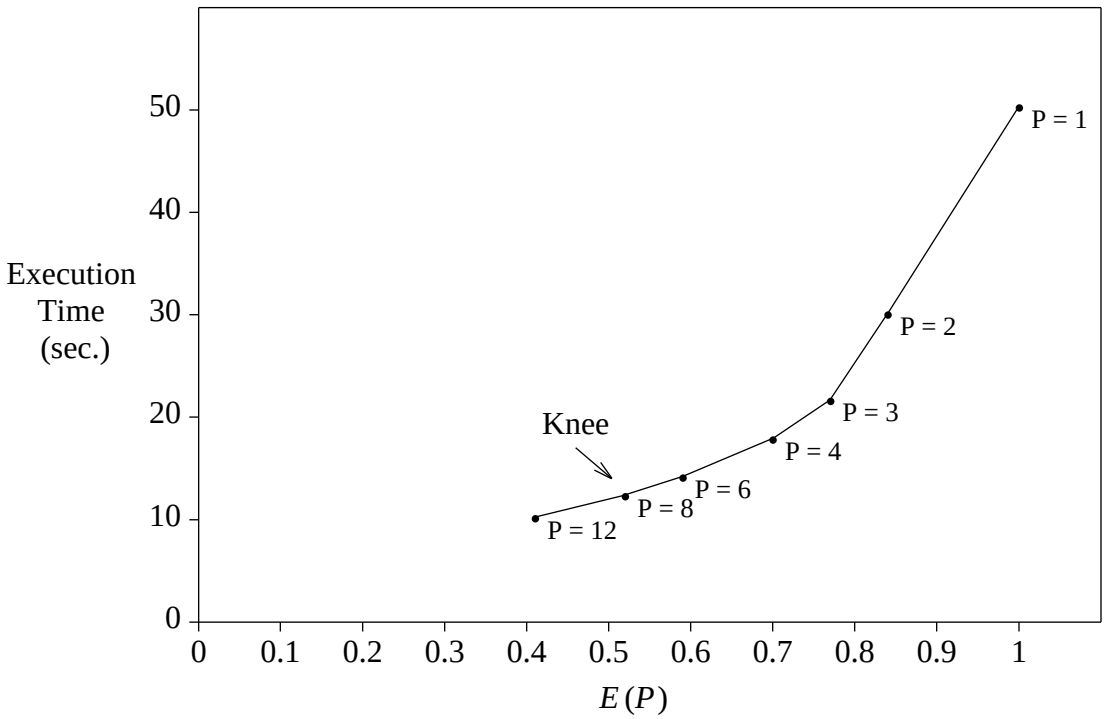


Figure 2.3: Execution time–efficiency profile, for the application Grav

This graph can be thought of as a means of analyzing the cost-benefit tradeoffs of allocating more processors to an application. The point at which the ratio of efficiency to execution time, $E(P)/T(P)$, ($T(P)$ is the execution time when using P processors) is maximized is called the knee of the execution time–efficiency profile and may be useful in determining effective processor allocations in multiprogrammed environments. Eager, Zahorjan and Lazowska also investigate how, under work-conserving scheduling disciplines, the average parallelism can be used to obtain bounds on the speedup and efficiency of an application [Eager1989]. A *work-*

conserving discipline is one that never leaves a processor idle when there is a task that is eligible for execution. *Average parallelism* is defined as the average number of processors that are busy during the execution of the application, given an unbounded number of available processors [Gurd1985]. Equivalently, average parallelism is the speedup of an application given an unlimited number of processors. Average parallelism provides, in some sense, information about the number of processors the job can utilize. It is designed to capture, in a single number, some of the information contained in a parallelism profile.

Although reducing the information contained within a parallelism profile to a single number may be attractive from the standpoint of employing the information when making scheduling decisions, it is important to note that measures such as speedup, efficiency, and average parallelism capture only information about a program's average behaviour (averaged over the execution of the entire program). It is, unfortunately, the different phases of program execution, as seen in an execution profile, that may be most meaningful when making scheduling decisions in a multiprogrammed environment, especially if dynamic scheduling is to be exploited. Evidence in support of this claim can found by noting the phases observed in the execution profile of parallel programs [Carlson1992] and the performance improvements obtained as a result of dynamically adjusting processor allocations with changes in the parallelism of the applications [McCann1993].

Some of the problems with using a single parameter characterization of parallelism are detailed by Marinescu and Rice [Marinescu1990]. Sevcik considers the use of additional information about an application's parallelism in making scheduling decisions [Sevcik1989]. He found that policies that consider this additional information, such as the minimum, maximum, and variation in parallelism, improve mean response time over methods that only consider average parallelism.

Recent studies by Zahorjan and McCann [Zahorjan1990] and McCann, Vaswani, and Zahorjan [McCann1993] show that simple information about applications (the current degree of parallelism) can be used to make more informed decisions about the number of processors to allocate to each application. They reallocate processors among jobs in response to changes in the parallelism of the jobs; in a sense attempting to utilize processors more effectively. This dynamic allocation policy improves mean response time when compared with repartitioning processors strictly upon job arrivals and departures [Tucker1989] [Gupta1991]. The advantage of the dynamic scheduling policy is that it may adjust to an application's demands for processors over time.

2.3.3. Work to be Executed

Although speedup and efficiency are important and widely used measures of a parallel application's performance, arguably the most important measure of a parallel program's performance is its execution time. The expected execution time is especially important when making scheduling decisions. A scheduler in a multiprogrammed environment, in attempting to minimize the mean response time over all applications, should consider not only the efficiency with which an application executes but also the amount of work to be executed. The amount of work, the efficiency with which that work can be executed, and the number of processors the scheduler allocates to each application will determine their execution time.

A number of studies have considered estimating the amount of work an application executes and using these estimates in making scheduling decisions. The idea behind these policies is to attempt to execute the shortest jobs first. Majumdar, Eager, and Bunt propose a policy called Smallest Number of Processes First (SNPF), a preemptive version (*SNPF), and a method called Smallest Cumulative Demand First (SCDF) that is used for comparison [Majumdar1988]. They perform a series of simulation studies and conclude that SCDF performs well when compared with FCFS and RR policies. They also found that if the total amount of work a job executes is correlated with the number of parallel processes it executes, *SNPF performs well compared with FCFS and RR. Leutenegger and Vernon also perform a simulation study of a number of scheduling policies. They conclude that policies that allocate processing power equally among all jobs (i.e., RR-based) perform best [Leutenegger1990]. Their simulation results also show that SCDF performs well, although because of reservations about the feasibility of its implementation, they draw different conclusions than Majumdar, Eager, and Bunt.

Our work in Chapter 3 more closely examines scheduling policies that allocate processing power equally among all jobs. We perform an analytic study to determine if mean response times can be improved by knowing information about the work a job is executing and determining bounds on the performance improvements. In order to perform this analytic comparison, a number of restrictive assumptions are made. Therefore, in Chapter 4, we investigate what kind of improvements can be expected under more realistic conditions and develop and evaluate new techniques for trying to distinguish short jobs from long jobs in order to complete shorter jobs first.

2.4. Scheduling in NUMA Multiprocessors

Two of the main reasons that new scheduling techniques are required for scheduling NUMA multiprocessors are:

1) A continually growing number of processors must be considered. This leads to new problems in coping with the scalability of the system.
2) The time to access memory is not uniform. Therefore, the execution time of an application can depend upon which processors it is assigned to execute on and where its data is located in memory.

Therefore, we describe some existing NUMA architectures while concentrating on those factors that will influence scheduling decisions the most. Then we briefly describe some techniques for designing and implementing software that will scale with the hardware (especially operating system software). This is followed by a discussion of previous research that is related to scheduling in NUMA systems in that it has concentrated on the problem of localizing data references.

2.4.1. Large-Scale Shared-Memory Multiprocessors

Three examples of existing large-scale shared-memory multiprocessors are the KSR1, from Kendall Square Research [Burkhardt1992], DASH, developed at Stanford University [Lenoski1992], and Hector, developed at the University of Toronto [Vranesic1991]. Each of these systems incorporates a hierarchical design to build larger systems by using small-scale multiprocessor components as building blocks. In Hector and DASH the base component is essentially a bus-based multiprocessor containing a small number of processors. (They are called stations and clusters, respectively.) In the KSR1, the base component, called Ring:0, is a unidirectional ring connecting up to 32 processors. Both the KSR1 and Hector use a ring to connect base components together to form larger systems. In the KSR1, this ring is called Ring:1 and in Hector it is called the station (or local) ring. These designs add another level to the hierarchy by connecting collections of rings together with another ring. The DASH system uses a mesh interconnection network to connect base components together to form a two-level structure.

The processing modules in the KSR1 and Hector, besides containing a processor and associated cache, also contain local processor memory, which provides short access times to some memory locations and reduces contention for the base component interconnection network. In the KSR1, the local memory is managed as a second level cache and consistency is

maintained in hardware. In DASH each cluster is essentially a Silicon Graphics multiprocessor. This design is different from the KSR1 and Hector in that memory within the cluster is attached to the bus rather than processors, thus forming a UMA multiprocessor at the base component level. The processor used in the KSR1 is a 20 MHz RISC processor developed by Kendall Square Research. DASH uses the 33 MHz MIPS R3000 processor while Hector uses the 16.67 MHz Motorola MC88100.

The Hector design does not provide hardware cache coherence. However, cache coherence is enforced in software by the HURRICANE operating system's memory manager. On the other hand, the DASH system implements cache coherence in hardware using a directory based scheme [Lenoski1990]. The KSR-1 design uses an interesting ''all-cache'' system in which all of memory is treated as a large slow cache. A limited broadcast method is used to locate memory blocks as they are needed. Coherence is maintained in hardware using a hierarchical broadcast and filter technique in which active ring components use directories to determine whether to accept or ignore packets.

Table 2.1 shows some memory latency times in processor cycles for each of these systems. The times for the KSR1 are in 50 nano-second cycles and are the times required to read one 128 byte cache line [Dunigan1992]. DASH and Hector have 30 and 60 nano-second cycle times respectively and the latencies shown in Table 2.1 are for loading one 16 byte cache line [Lenoski1992] [Stumm1993]. This table is not shown to compare these systems but to illustrate two of the key issues related to the use of shared-memory NUMA multiprocessors:

1) The time to access remote memory can be significant.
2) The time to access remote memory depends on the distance to the location being accessed (the number of levels of the hierarchy that must be traversed).

It is, therefore, quite natural to hypothesize that placing the parallel processes of an application, which typically share data, close to each other in order to reduce communication costs will be beneficial for their efficient execution. The degree to which the execution time of an application will benefit from a localized placement depends on the number, frequency, and latency of remote communication, as well as the effectiveness of the latency hiding techniques used (if they are used).

Latency hiding is an active area of research in which techniques are designed to hide communication latencies by overlapping communication with computation (rather than reducing the cost and frequency of communication). Different approaches include: violating the

System and CPU	Memory Level	Memory Location	Processor Cycles	Approx. System Size
KSR1	1	Local Memory	18	1
	2	Ring 0	126	32
	3	Ring 1	600	32–1088
DASH	1	Secondary Cache	15	1
	2	Local Bus Memory	29	4
	3	Remote Cluster Memory	132	16–64
Hector	1	Local Memory	19	1
	2	On Station Memory	29	4
	3	On Ring Memory	37	16–32
	4	Off Local Ring Memory	46	64–256

Table 2.1: Memory reference hierarchies and latencies of some NUMA multiprocessors

sequential consistency model and using weaker models of consistency to permit the pipelining and buffering of memory accesses [Gharachorloo1990] [Adve1990] [Dubois1990], pre-fetching [Gornish1990], multi-threaded processors [Smith1981] [Alverson1990] [Agarwal1990], and dynamically scheduling processors (by using micro-parallelism within a thread) [Gharachorloo1992]. If future systems are capable of effectively hiding memory access latencies, the main advantage offered by a localized placement of applications would be in the expected reduction in the use of and contention for the interconnection network (aside from first reducing the latencies that need to be hidden).

2.4.2. Software Scalability

The extent to which the processing power of large-scale shared-memory multiprocessors can be exploited will depend partially on the design and implementation of software structuring techniques that scale with the hardware. One promising approach is to use the concept of Hierarchical Clustering as a way to structure operating systems for scalability [Unrau1992] [Unrau1993]. This approach uses a small-scale symmetric multiprocessing operating system for a small number of processors as the base unit of structuring, called a cluster. In systems with roughly 64 to 256 processors, clusters are replicated such that each cluster manages a unique group of neighbouring processing modules. For larger systems clusters are grouped together hierarchically with looser coupling in higher and higher levels in the hierarchy. Besides being a means for achieving scalability, clusters also localize memory accesses, which is the key to good performance in NUMA systems [Unrau1992]. Similar structuring methods have been proposed for scheduling subsystems in scalable architectures [Feitelson1990] [Feitelson1990a] [Ahmad1991].

2.4.3. Localizing Data Accesses

As processor technology continues to improve at a faster rate than memory or interconnection network technology, the relative increase in communication costs in multiprocessors has become a topic of growing importance. A number of recent studies consider the importance of memory access costs when making scheduling decisions in shared-memory multiprocessors. One of the goals of this research is to reduce the number of cache misses or remote memory references by co-locating lightweight user-level threads and kernel processes with the data being accessed, thus reducing the time spent loading data into the local cache or memory.

Using an analytic model of a time-sliced, central ready-queue, scheduling environment and experimental evaluation on a UNIX based multiprocessor, Squillante and Lazowska [Squillante1990] [Squillante1993] argue and demonstrate that applications can build considerable cache context, or footprints [Thiebaut1987]. Recognizing that it may be more efficient to execute a process on a processor that already contains relevant data in that processor's cache, they design and examine techniques that consider the affinity that a process has for a processor. They observe that the execution time of applications which release a processor because of quantum expiration, preemption, or I/O, can be significantly reduced by using processor-cache affinity information to reschedule processes onto processors on which they have previously run.

Subsequent studies have arrived at different conclusions. Gupta, Tucker, and Urushibara [Gupta1991] also consider the importance of cache-affinity techniques but in a space-shared multiprocessor environment. They simulate a number of scheduling techniques and, using processor utilization as a performance metric, conclude that improvements due to processor-cache affinity are quite small, reducing mean processor utilization by only 3%. (They do not report on how mean response time is affected.) Vaswani and Zahorjan [Vaswani1991] draw similar conclusions in their experimental study of the importance of cache affinity. They use an analytic model to show that even with faster processors and larger caches, the benefits due to cache affinity will be minimal.

The apparent difference between the conclusions drawn in these studies is largely due to the difference in the scheduling policies used to multiprogram applications. While Squillante and Lazowska use time-sharing, the study by Gupta, Tucker, and Urushibara and the study by Vaswani and Zahorjan both use space-sharing. The time-sharing policy employs a small reallocation interval (quantum). This results in relatively frequent context switches and ensures that processes do not run long enough to interfere with each other significantly. Vaswani and

Zahorjan found that with space-sharing the frequency of context switches and cache reloading could be reduced and that intervening applications often ran long enough to significantly disrupt the cache context of the previous process, thus greatly reducing the benefits of processor-cache affinity.

While cache affinity studies investigate the benefits of reusing cached data when executing more than one application on the same processor (across applications), related work at the University of Rochester concentrates on the benefits of reusing cached data when executing lightweight threads of the same application on the same processor (within applications) [Markatos1992] [Markatos1992a] [Markatos1993]. The Rochester work also extends the notion of locality management to include one more level in the memory hierarchy by considering systems that may have local-cache, local-memory, and remote-memory, such as the BBN TC2000. Markatos [Markatos1993] first demonstrates that fine-grain parallel programs, because of the overhead required to load data into the local cache or memory, typically perform much worse than coarse-grain implementations even though the cost of thread management is negligible. This motivates the need for techniques that consider locality when scheduling lightweight threads within an application. Then Markatos develops a technique called memory-conscious scheduling which, when used with fine-grain applications, yields execution times that are comparable to coarse-grained implementations.

Markatos and LeBlanc [Markatos1992a] consider the conflicting requirements for load balancing and locality management when scheduling lightweight threads of an application. They conclude that of these two important considerations, locality management should be the primary factor influencing the assignment of threads to processors. The importance of locality management is also explored in their work on loop scheduling [Markatos1992]. They demonstrate how traditional loop scheduling techniques incur significant performance penalties on modern shared-memory multiprocessors. Then they propose and compare new loop scheduling algorithms that consider the requirements of load balancing, minimizing synchronization, and co-locating loop iterations with the data being referenced. These new algorithms are shown to improve performance by up to 60% in some cases.

Our work in Chapters 5 and 6 is complementary to processor-cache affinity and lightweight thread scheduling techniques for improving locality of data references. While these previous studies investigate the importance of scheduling techniques for reducing the number of non-local memory accesses by co-locating processes with the data being accessed, our work investigates the importance of scheduling techniques for reducing the cost of required non-local memory

accesses in environments where processes and data cannot be co-located. We have conducted a simulation study [Zhou1991] which indicates that placement is an important aspect of scheduling in large-scale, NUMA multiprocessors. This research has also provided motivation for our experimental approach.

In Chapters 5 and 6, we experimentally investigate the problem of scheduling parallel processes of an application that concurrently access shared data in an environment in which there is no *a priori* knowledge of sharing or communication patterns. The complexity of the problem is increased by the architectural trend to cluster processors and memory elements and to connect clusters together in a hierarchical fashion in order to build larger systems. This results in systems with a number of levels in the memory hierarchy and memory access latencies that vary with the number of levels of the hierarchy that must be traversed. Therefore, the placement problem becomes one of placing processes of an application onto processors such that the costs of required accesses to shared data are minimized. To our knowledge, this environment has not been considered in previous studies.

2.5. The Experimental Environment

The experiments presented in this dissertation were conducted using a prototype of a scalable shared-memory NUMA multiprocessor called Hector, developed at the University of Toronto [Vranesic1991]. Each processing module in the Hector prototype contains a 16.67 MHz Motorola MC88100 CPU, a 16 Kbyte instruction cache, a 16 Kbyte data cache, and 4 Mbytes of globally addressable memory. The hierarchical design used in Hector connects a number of processing modules with a bus to comprise a station. Several stations are connected with a bit-parallel slotted ring and rings can be further connected using a hierarchy of rings to easily support up to 256 processors. The prototype used consists of 4 stations, each containing 4 processor modules, for a total of 16 processors and 64 Mbytes of globally addressable memory (see Figure 2.4). This design has relatively mild NUMA characteristics.

There is no hardware support for cache coherence in Hector. Cache coherence is enforced in software by the HURRICANE operating system's memory manager at a 4 Kbyte page level of granularity, by permitting only unshared and read-shared pages to be cacheable [Unrau1992] [Unrau1993]. HURRICANE also supports page migration and replication but these features were disabled when conducting our experiments. This was done to ensure that observed differences in execution times were due solely to changes in scheduling algorithms. We expect that enabling these features would only mildly influence our quantitative results and would in no way change our qualitative results.

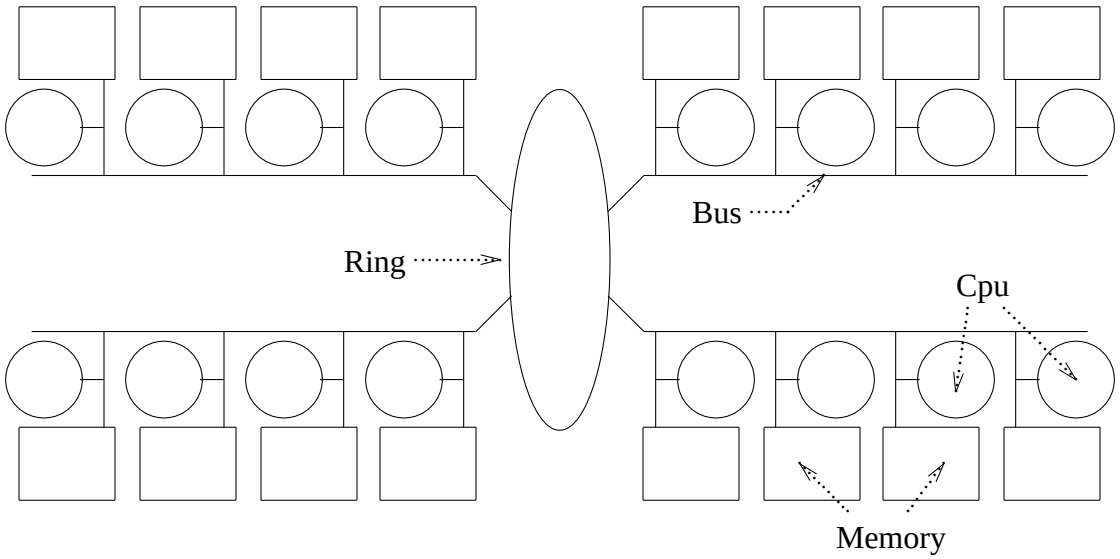


Figure 2.4: Prototype 16 processor Hector multiprocessor

Unlike many of the UNIX operating systems currently in use on a number of multiprocessor systems, HURRICANE uses lightweight kernel processes. The lightweight processes comprising each parallel application are placed on processors in the system by a system scheduler. This is accomplished by having each HURRICANE process creation first contact a user-level scheduler to determine which processor to execute on. The scheduler is designed for a dynamic multi-purpose, multiprogrammed environment so it is assumed that the number of processors desired by an application is not known *a priori*. As a result, processor requests and placement decisions occur one at a time, at the time of process creation. Applications request a number of processors by using a loop to create one process at a time. Space-sharing is enforced by the scheduling server, which also controls the number of processes and therefore, the number of processors the application is permitted to use. This is accomplished by not permitting the application to create more processes than the scheduling server has determined should be allocated to the application (i.e., the process creation fails). Applications were linked with a special library that directs creation calls to the scheduler and notifies the scheduler whenever a process is finished executing. Figure 2.5 shows the relationship between the thread library, scheduling server, and operating system kernel.

User-level threads are scheduled onto operating system processes by the thread package. The scheduling server is responsible for some subset (possibly all) of the processors, with the server being replicated in larger systems. For the purposes of our experiments the scheduling

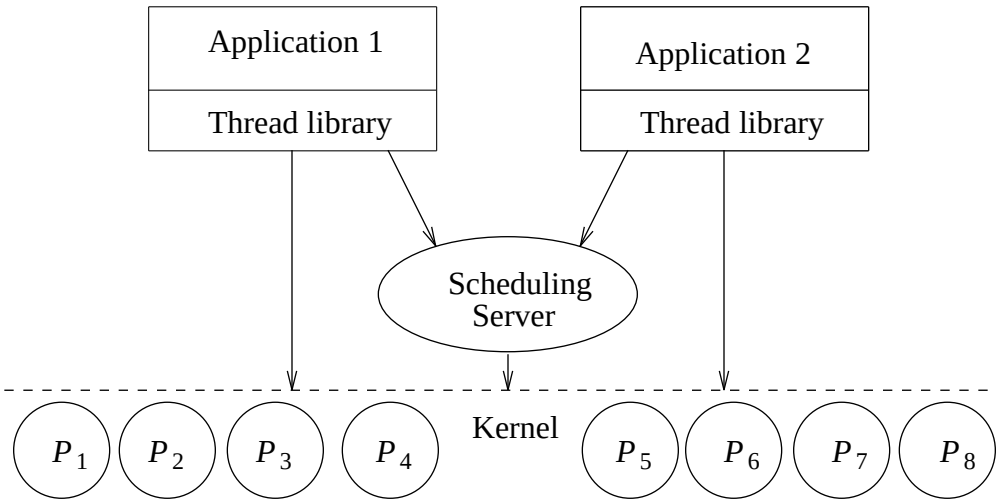


Figure 2.5: Structure of scheduling subsystem

server and all other system processes execute on a subset of processors separate from those used to execute the parallel applications. The scheduling server makes decisions about how many kernel processes to allocate to each application and where each of those processes is executed. The kernel is responsible only for dispatching processes assigned to its local ready queue. There is one ready queue per processor in the HURRICANE kernel and because the scheduling server implements space-sharing for the subset of processors that are used to execute parallel applications, there is typically only one process assigned to each of those processors.

2.6. Applications Used in Experiments

Because the current Hector system is a prototype and because much of the work being conducted on the system consists of operating systems research and performance evaluation, there is not a large body of regular users. As a result, the applications collected for experimentation consist mainly of programs that were written either to evaluate this type of research or as part of a course project on parallel programming. Consequently, some of the applications used are really kernels of what would be considered real parallel applications.

We use two different types of implementations for our experiments, each serving different purposes. The first set of implementations use medium-grained parallelism (e.g., each thread performs computation on a single row of a matrix). They use a locally developed user-level thread package designed specifically for Hector and HURRICANE which, in cooperation with the scheduling server, provide the ability to dynamically adjust which processors are assigned to

each application. The second set of implementations use kernel-level lightweight processes and coarse-grained parallelism. These programs request and are statically allocated a user specified number of processors. The amount of computation performed by each process is determined by simply dividing the data set evenly among the specified number of processors. The medium-grained set of implementations are predominantly used in experiments in Chapters 3 and 4 and will be explained in further detail when they are used. Some of those programs are medium-grained thread-based implementations of the same applications used in Chapters 5 and 6. We now describe the implementation of those applications.

In each of the coarse-grained implementations the main process creates a number of children which act as slaves. Since each process of the parallel program is allocated to a separate processor and we do not want processors to be unnecessarily idle, some programs were modified so that the master process not only controls and synchronizes the children but also performs its share of the computation, rather than simply waiting for the children to perform the computation.

These programs are of the data parallel or single program multiple data (SPMD) class of programs, which means that each process executes the same computational kernel on different portions of the data space. The data access patterns of each application are different, so the importance of the placement of the parallel processes should vary with each application. Since the placement of each parallel process of an application is important relative to the data that is being accessed, the HURRICANE operating system permits the application writer to roughly control where data will be located by specifying the policy to be used when requesting memory. In the implementations used, most of the shared data is allocated to memory according to a first-hit policy. That is, data will be physically located in the memory of the processor module that first touches the page containing that data. Some implementations specify a round-robin policy for some of the shared data so that frequent accesses to the data by many processors do not create hot spots and the remote memory access costs are reduced when executing on all 16 processors. Even though we do not use all of the processors in some of our experiments, we have not modified the memory allocation policies used by the programs. (Currently there is no policy for allocating memory on a round-robin basis from the subset of processors assigned to the application.)

The applications are listed in Table 2.2 along with the problem size, the precision used, the number of lines of C source code, and the speedup measured using four processors of one station, S(4). The speedup values shown were computed by comparing the execution times of the parallel application using one and four processors (since a serial version was not available for all applications and we are concerned with the scheduling of parallel applications). The

number of source code lines may be slightly high due to the large number of timing, tracing, and debugging calls used when tuning the applications. More detailed descriptions of each application are contained in the next section.

Name	Application / Problem Size	Precision	Lines of C	S(4)
FFT	2D Fast Fourier transform data size = 256x256	Single	1300	2.9
HOUGH	Hough transformation size=192x192, density=90%	Double	600	3.4
MM	Matrix multiplication data size = 192x192	Double	500	3.4
NEURAL	Neural network backpropagation 3 layers of 511 units, 4 iterations	Single	1100	3.8
PDE	Partial differential equation solver using successive over-relaxation data size = 96x96	Double	700	3.7
SIMPLEX	Simplex Method for Linear Programming 256 constraints, 512 variables	Double	1000	2.4

Table 2.2: Summary of the coarse-grained applications used

The size of the system used is relatively small (16 processors), and in order to evaluate different application placements, each application is executed using four processors. (Four processors were also chosen because some implementations constrained the number of processors used to a power of two or to a number that divides evenly by the size of the data set used.) Although the size of the data sets may appear to be small, they were chosen for a number of reasons:

1) They should execute on four processors in a reasonable amount of time since multiple executions of each application are used to compute means and confidence intervals.

2) The size of the data cache on each processor is relatively small (16 Kbytes). Consequently, cache misses and memory accesses will occur, even with a relatively small sized problem.

3) The amount of memory currently configured per processor is relatively small (4 Mbytes). If problem sizes are too large, data structures that are designed to be allocated to the local processor (by using the first-hit allocation policy) may have to be allocated to a different processor, resulting in remote memory references where the application programmer had not intended. That is, once all of the physical memory of the local processor has been allocated, the memory manager will allocate memory from a neighbouring but remote module (rather than evicting pages from local memory).

The seemingly poor speedup of some applications is the result of the small data sets used to perform these experiments, since most of the implementations were designed to be used with larger data sets on more processors (i.e., the parallelism is relatively coarse-grained).

We now provide a more detailed description of each of the applications and how they are implemented.

FFT computes the forward and reverse transforms of a two-dimensional array of data [Cooley1965] by performing an *O(log N)* one-dimensional transform along each row of the array and then along each of the columns, where *N* is the number of data points (256x256 in this case). The FFT is used extensively for signal processing, image processing, and solving differential equations. Parallelization is achieved by having each process compute a portion (a number of rows or columns) of the one-dimensional transforms. The computation would normally be dominated by sine and cosine operations, so this implementation improves performance by pre-computing and storing sine and cosine values in a lookup table. All major shared data structures are allocated to memory on a first-hit basis with the exception of the sine and cosine tables, which are allocated on a round-robin basis.

HOUGH implements a commonly used image processing and computer vision method for detecting lines and curves in digitized images [Duda1972]. The main data structures used are two matrices, one for the binary input image and another for the resulting accumulator array. Each process examines its predetermined portion of the image, performs the required transformations, and stores the results in a local data structure. This implementation then sequentially adds the locally computed results to the global result. The final summation phase is sequential because the results are being updated to a global matrix containing the final result. As was the case for FFT, values for sine and cosine are pre-computed and stored in lookup tables, which are allocated to memory on a round-robin basis. In both of these applications the tables are allocated in a round-robin fashion to avoid the creation of hot spots. (The extra cost of accessing remote memory is relatively small.) The input data image is also allocated to memory using a round-robin policy while the remaining shared data structures are allocated using a first-hit policy.

MM computes the matrix equation $C = AB$ [Strang1980]. In the parallel version of this application, each of the processes is responsible for computing an equal sized strip of the rows of the matrix *C*. To calculate the results for a strip of *C*, each process must access all of the elements of *C* contained in that strip, all of the elements in the same strip of *A*, and all of the elements of the entire *B* matrix. (Each matrix is stored in row-major order.) Therefore, the *A*

and *C* matrices are allocated from memory on a first-hit basis, providing for local memory accesses, except at the boundaries of the regions being computed by each process (since these may be located on the same physical page). The entire *B* matrix is accessed by all processes and is therefore a good candidate for distribution throughout the system, when all of the processors in the system are used (or for replication). However, since some of the other applications distribute shared data throughout the system, we used the first-hit policy for allocating memory for matrix *B*. Because the main (parent) process initializes *B*, it will be allocated in the memory associated with the first processor allocated to the application. Although this is not the best way to implement this application, it should help to emphasize the importance of the location of processes relative to the data being accessed since this forces all but one of the processes to access matrix *B* remotely.

NEURAL implements a neural network backpropagation learning algorithm [Rumelhardt1986]. It trains a general network topology with a specified number of layers and number of units to be processed per layer. Each layer contains the same number of units and every unit is connected to each unit in the adjacent layer. In order to ensure that execution times are somewhat deterministic, the number of iterations used in training the network is specified as an input parameter and training continues until the specified number of iterations have been completed. The algorithm operates in parallel by using matrices to represent the problem and reducing the calculation to a series of matrix operations. Once again processes operate in parallel on different row-wise partitions of the matrices. The training patterns and desired outputs are obtained by mapping files to memory using a first-hit policy. All other shared-data structures are also allocated on a first-hit basis.

PDE is a partial differential equation solver that implements a successive over-relaxation method for solving Laplace's equation for a two-dimensional data set [Press1988]. It starts with a set of initial or boundary conditions and iterates over the data space until the solution converges to a steady state. (The computation is done chaotically.) As a result the execution can be fairly non-deterministic and multiple executions can result in more variation than for some of the other applications (e.g., one execution of the application has processes executing an average of 524 iterations while a second execution requires processes to execute an average of 519 iterations). The particular implementation used finds the temperature distribution in an infinitely long square rod whose edges are held at a fixed value [Unrau1993]. Each process iterates over equal sized row-wise strips of the element array, which are allocated using a first-hit policy so accesses will be to local memory except along common edges where the data is accessed by processes in adjacent strips.

SIMPLEX is an implementation of an optimization method for solving linear programming problems [Dantzig1951] [Ravindran1987]. A linear program involves finding a solution which minimizes or maximizes the value of a criterion function subject to a set of constraints specified by linear equations or linear inequalities. The simplex algorithm is a popular heuristic method which performs very well in practice and is the method of choice for solving linear programming problems. The main data structure, sometimes called a tableau, is a matrix containing the coefficients of the system of equations specified by the linear programming problem being solved. Each process is responsible for several consecutive rows of the tableau and only performs read and write operations on its own portion of the tableau.

2.7. Summary

In this chapter we have described different techniques for characterizing parallel applications and a number of policies for scheduling in multiprogrammed multiprocessors. From recent research in UMA multiprocessors, general consensus on a number of conclusions appears to be emerging:

1) Space sharing is preferable to time sharing in multiprocessors because space sharing permits processing power to be divided among jobs without incurring the overheads due to preemption (as is the case with time sharing).

2) Two-level scheduling (i.e., the use of user-level threads) offers distinct performance advantages because it can be used to match the number of executing kernel threads with the number of available processors to avoid unnecessary oblivious preemptions. However, uncoordinated processor reallocations — those in which the kernel reallocates processors without interacting with the application — can lead to poor performance when compared with coordinated processor reallocations.

3) Dynamic scheduling policies that change the number of processors allocated to an application as a result of changes in workload (e.g., the number of jobs or their parallelism) are preferred over static policies.

4) The cost of memory accesses is increasing relative to cache accesses and techniques are required to ensure that the number of memory accesses is reduced.

The existence of larger multiprocessors is likely to provide more opportunities to multiprogram parallel applications since applications may not be capable of utilizing all of the processors. This increases the importance of techniques for effectively sharing processors. In particular, we are concerned with the problem of how many processors to allocate to each

application, given that space sharing is the preferred method of multiprogramming. We call this the allocation problem, which is the topic of study in Chapters 3 and 4. Specifically, we are interested in understanding why techniques that partition processors equally among applications have performed well in practice [Tucker1989] [Leutenegger1990] [Leutenegger1991], what is required if new techniques are to improve the performance of existing techniques, and what kind of performance improvements can be expected from new techniques. After all, it may not be worth implementing and using new techniques if they will only marginally improve on the performance of existing techniques.

As also described in this chapter, a number of recent studies have demonstrated the importance of data reference locality and have examined scheduling methods for reducing the number of costly references to memory. This is usually done by placing the execution unit (the process or user-level thread) near the data being accessed, as is done in the processor-affinity, user-level thread, and loop scheduling techniques. The existence of large scalable NUMA multiprocessors, however, presents new challenges in scheduling. Since parallel applications are likely to require access to shared data, the location of parallel processes of the application relative to that data may influence the execution time of the application. The placement of processes is examined in Chapters 5 and 6.

Chapter 3

Processor Partitioning

3.1. Introduction

This chapter examines the fundamental problem of determining the number of processors to allocate to each application and examines methods for making these decisions dynamically and adaptively. The goal is to obtain a better understanding of existing scheduling policies and to motivate new techniques that incorporate information about application characteristics.

We first establish several simplifying assumptions, then analytically compare currently popular techniques with optimal techniques. Some of the simplifying assumptions are gradually relaxed in order to examine a more realistic version of and practical solutions to the problem of multiprogrammed scheduling in multiprocessors.

These results all apply directly to shared-memory systems. As well, in the case when processor repartitioning is not permitted, the results can be applied to message-passing parallel systems (i.e., systems that do not support shared-memory). The results of this chapter motivate the work in Chapter 4 which examines practical techniques for obtaining and using execution time estimates to reduce mean response time.

3.2. Motivation

A number of recent studies propose using parallel application characteristics to determine the number of processors to allocate to each application [Majumdar1988] [Majumdar1988a] [Sevcik1989] [Ghosal1991] [Leutenegger1991] [Sevcik1994]. Unfortunately many of these characteristics are either difficult to obtain or require too much data to be of practical value and as a result have not found their way into multiprocessor system implementations. Consequently, processor allocation decisions are often made in the absence of knowledge of application characteristics. Without such information all applications are essentially treated equally. Hence, the single property that is often reported as being a desirable property of a multiprogrammed multiprocessor scheduler is that it should space-share the processors equally among all applications [Tucker1989] [Leutenegger1990] [Zahorjan1990]. In this Chapter we examine the policy decision to equipartition processors by proposing and examining alternative strategies that partition processors unequally among jobs in order to reduce mean response time when compared with equipartitioning strategies.

Recent work by McCann, Vaswani, and Zahorjan [McCann1993] shows that simple information about applications can be used to make more informed decisions about the number of processors to allocate to each application. They reallocate processors among jobs in response to changes in the parallelism of the jobs. Each job frequently advertises, to the scheduler, the number of additional processors it could use. Additionally, it notifies the scheduler when it has a processor that can be reallocated to another application. The scheduler may then reallocate the processor to another application that is able to use the additional processor. This dynamic allocation policy improves mean response time when compared with repartitioning processors strictly upon job arrivals and departures [Tucker1989] [Gupta1991]. Note, however, that under the ‘‘dynamic allocation’’ policy, if all applications have sufficient parallelism, each application is treated equally and is given an equal portion of processors.

A central thesis of this dissertation is that processor allocation schemes should consider relevant job characteristics to improve mean response time. Most notably we are interested in making more informed processor allocation decisions by considering the amount of work to be done by each application. Note that these schemes are likely to allocate an unequal number of processors to each application. This chapter concentrates on an analysis of the benefits that can be obtained from knowing the amount of work an application performs. Chapter 4 considers practical techniques for obtaining and applying such knowledge.

3.3. Simplifying Assumptions

In order to perform analytic comparisons of various policies, we begin with restricted models of the system and parallel applications. These simplifications are made to gain insight into the general problem of how to schedule and concurrently execute multiple parallel applications. Some of these simplifying assumptions are gradually relaxed and new policies derived and compared. (Note that the policies we consider are work-conserving.)

A3.1: The number of jobs being executed, N, is less than the number of processors, P.

A3.2: The N jobs arrive simultaneously at time zero and there are no new arrivals.

A3.3: Job J_i has W_i units of work to execute and the work is infinitely divisible. That is, the granularity of the computation can be made arbitrarily small.

A3.4: The number of cpus allocated to job J_i, denoted by p_i, may be fractional and processors that are allocated fractionally to more than one application time-share the applications in appropriate portions with negligible overhead.

A3.5: All processors in the system are equivalent. That is, the effects of cache contexts and memory latencies are ignored.

Two of the most important characteristics affecting a parallel application's execution time are the amount of work to be executed and the efficiency with which the work can be executed. In order to simultaneously execute multiple parallel applications effectively, processor allocation decisions must utilize knowledge of these characteristics. A number of studies have described various techniques for determining processor allocations based on different approximations of an application's efficiency [Majumdar1988a] [Eager1989] [Ghosal1991] [Carlson1992] [Rosti1994] [Sevcik1994]. In this chapter we are interested in determining how much benefit can be obtained by knowing and applying knowledge of how much work each application executes. Therefore, we also assume that the following information is available for each application:

A3.6: The amount of work being executed by each job J_i is known and is W_i.

A3.7: All applications execute with perfect efficiency. That is, they achieve perfect speedup.

These assumptions are made in order to make the problem more tractable, to determine if significant performance improvements can be obtained by using knowledge of the work an application executes and to gain insight into the problem of partitioning processors among applications. In Chapter 4 we relax assumptions A3.6 and A3.7 and consider practical, implementable approaches for obtaining such information and techniques for making processor allocation decisions based on the methods introduced and investigated in this chapter.

3.4. Scheduling without Reallocations (Static Partitioning)

We begin by considering a simplified processor allocation problem. If N perfectly efficient applications, $J_1, J_2, J_3, \ldots, J_N$, are to be simultaneously executed on P processors, and an application is not able to acquire additional processors once it has been activated, how many processors, p_i, should be allocated to each application, J_i, in order to minimize the mean response time of the applications?

That is, two additional assumptions are used in this section (but are later relaxed):

A3.8: N jobs, $J_1, J_2, J_3, \ldots, J_N$, are started simultaneously and executed to completion.

A3.9: Processor repartitioning is not permitted.

Reasons for considering this problem are:

- It provides insight into how to allocate processors to N jobs that are to be executed simultaneously.
- The costs associated with reallocating processors may be prohibitive. Certain overheads are associated with the reallocation of processors. If these overheads outweigh the benefits, reallocation should be avoided. Examples of such overheads include: context switching, resultant cache misses, and migration costs (or supporting reallocation in systems without shared-memory).

We first consider and compare possible techniques for addressing the stated problem. Later we will study a more general form of the scheduling problem and examine how relaxing the last two assumptions (A3.8 and A3.9) might affect processor allocation decisions.

3.4.1. Equal Partitions (EQUI)

One popular approach to allocating processors is to treat all applications equally and assign them each an equal portion of processors [Ghosal1991] [Rosti1994]. This method is sometimes referred to as an equipartition policy because it gives each application an equal sized partition of processors. (We call it EQUI.) If we let p_i represent the number of processors that job J_i is allocated, then $\sum_{i=1}^{N} p_i = P$.

EQUI: $$p_i = \frac{P}{N}$$

This policy is fair in one sense because each application is allocated the same number of processors. However, it has the drawback that if the applications execute different amounts of work, processors unnecessarily become idle since processors assigned to jobs that finish first remain idle until all jobs are completed. In order to improve the mean response time, this idling of processors must be reduced or eliminated. This is done by considering differences between applications. Given the current set of assumptions the only possible difference between applications is the amount of work to be executed, W_i.

We now examine how *a priori* knowledge of W_i might affect processor allocation decisions and how much performance improvement might be obtained compared with an equipartition policy (EQUI).

3.4.2. Proportional Partitions (PROP)

Intuitively, allocating a fraction of processors that is proportional to the amount of work being executed by each application should reduce the idling of processors. We call such a policy PROP because of its proportional allocation of processors.

PROP: $$p_i = \frac{P\ W_i}{\sum_{j=1}^{N} W_j}$$

This policy is of interest because:

1) All jobs start and complete at the same time and, therefore, execute with response times equal to the mean response time. [1] This policy is fair in one sense because all jobs have the same execution time.

2) Given W_i for $i = 1, 2, \ldots, N$, it is easy to calculate the number of processors to allocate to each application.

3) Intuitively, if the jobs do not execute equal amounts of work, then allocating a greater portion of processors to larger jobs might improve the mean response time over equally partitioning the processors.

Prasanna, Agarwal, and Musicus [Prasanna1994] use a heuristic called ‘‘Greedy’’, at compile time, to distribute processors among a set of nodes derived from a macro dataflow graph of a program. This ‘‘Greedy’’ approach involves assigning processors in proportion to the time required to execute each node and is equivalent to PROP. (Their method is, however, designed to minimize the time at which all nodes complete execution rather than minimizing the mean response time.) Since all of the nodes in the set being executed are required to complete before the next set can begin execution, the proportional assignment of processors minimizes the completion time of each set of nodes.

1

$$R_i = \frac{W_i}{p_i} = \frac{W_i \sum_{j=1}^{N} W_j}{P\ W_i} = \frac{\sum_{j=1}^{N} W_j}{P}$$

3.4.3. Comparing PROP with EQUI

The number of processors allocated to each application can be drastically different under the equipartition (EQUI) and proportional partition (PROP) policies. In order to compare the relative performance of the two policies, we now examine how the difference in allocation of processors affects the mean response times of applications executed under each policy.

Let $R_i^{(E)}$ be the response time of application J_i when scheduled using the equipartition policy and $\bar{R}^{(E)}$ be the mean response time of all N applications. The response time of an application is determined by the amount of work it is performing, W_i, and the number of processors it is allocated, p_i. Therefore, $R_i^{(E)} = \frac{W_i}{p_i}$ and under an equipartition strategy $p_i = \frac{P}{N}$.

So, $R_i^{(E)} = \frac{N\ W_i}{P}$ and $\bar{R}^{(E)} = \frac{1}{N}\sum_{i=1}^{N} R_i^{(E)} = \frac{1}{N}\sum_{i=1}^{N} \frac{N\ W_i}{P} = \frac{\sum_{i=1}^{N} W_i}{P}$.

Now let $R_i^{(P)}$ be the response time of job J_i when scheduled using the proportional partition policy and $\bar{R}^{(P)}$ be the mean response time of all N jobs. Again response time is determined by the amount of work, W_i, and the number of processors allocated, p_i. Therefore, $R_i^{(P)} = \frac{W_i}{p_i}$ and under the proportional partitioning policy $p_i = \frac{P\ W_i}{\sum_{j=1}^{N} W_j}$.

Hence, $R_i^{(P)} = \frac{W_i \sum_{j=1}^{N} W_j}{P\ W_i} = \frac{\sum_{j=1}^{N} W_j}{P}$ and the mean response time is:

$$\bar{R}^{(P)} = \frac{1}{N}\sum_{i=1}^{N} R_i^{(P)} = \frac{1}{N}\sum_{i=1}^{N} \frac{\sum_{j=1}^{N} W_j}{P} = \frac{\sum_{j=1}^{N} W_j}{P} .$$

Therefore:

$$\bar{R}^{(P)} = \frac{\sum_{i=1}^{N} W_i}{P} = \bar{R}^{(E)} .$$

From this analysis, we see that although the allocation of processors to applications can be very different between the equipartition and proportional partition policies, in fact their mean response times are identical.

3.4.4. Optimal Partitions (ROOT)

Our goal in partitioning processors among the applications is, however, to minimize the mean response time over all applications. Towards this goal we now derive a partitioning scheme that optimally partitions the processors, given the current set of assumptions (A3.1 through A3.9).

Since all jobs are started at time zero and execute with perfect efficiency, $R_i = \frac{W_i}{p_i}$ and $\bar{R} = \frac{1}{N} \sum_{i=1}^{N} \frac{W_i}{p_i}$. The problem thus becomes one of determining an assignment for each p_i, $i = 1, 2, \ldots, N$, such that $\bar{R} = \frac{1}{N} \sum_{i=1}^{N} \frac{W_i}{p_i}$ is minimized and $\sum_{i=1}^{N} p_i = P$.

This problem can be solved by applying the method of Lagrange multipliers to obtain the following formula for determining how many processors to allocate to each application:

ROOT: $$p_i = \frac{P \sqrt{W_i}}{\sum_{j=1}^{N} \sqrt{W_j}}$$

Theorem 3.1 : Under the previously stated assumptions, A3.1 through A3.9, this allocation policy (ROOT) yields an optimal partitioning of P processors among N simultaneously executing applications.

Proof : See the Appendix for a proof of this theorem. ■

Sevcik [Sevcik1994] has also considered the amount of basic work that an application executes, along with a number of factors that ‘‘slowdown’’ an application’s execution and has obtained the same result. These slowdown factors include the overhead required to achieve parallel processing, the imbalance with which work is distributed across the available processors, and communication overhead.

Before comparing this new policy (ROOT) with the equipartition policy (EQUI) we first illustrate the differences among the three policies discussed to this point by using a simple example. Consider three jobs J_1, J_2 and J_3 executing on 60 processors. The amount of work being executed by each application is $W_1 = 100$, $W_2 = 400$, and $W_3 = 1000$, respectively. Table 3.1 shows the proportion and the number of processors allocated to each application, the response time of each job, and the mean response time obtained using each of the three policies.

Policy	Job	W_i	p_i	R_i	$\bar{R}$
EQUI	J_1	100	20.00	5.00	25.00
	J_2	400	20.00	20.00	
	J_3	1000	20.00	50.00	
PROP	J_1	100	4.00	25.00	25.00
	J_2	400	16.00	25.00	
	J_3	1000	40.00	25.00	
ROOT	J_1	100	9.74	10.27	21.10
	J_2	400	19.47	20.54	
	J_3	1000	30.79	32.48	

Table 3.1: Differences between EQUI, PROP and ROOT partitioning schemes: an example

Figure 3.1 provides a graphical representation of the differences between the equipartition, proportional and optimal partition strategies by showing the different number of processors allocated to each application, the response times of each application, and the mean response time of all applications. Each rectangle in the diagram represents the execution of a job. The width is proportional to the number of processors allocated to the job while the length is proportional to the execution time. Note that because of the assumptions that work divides infinitely and can be executed with perfect efficiency, the area of the rectangle depicts the work executed.

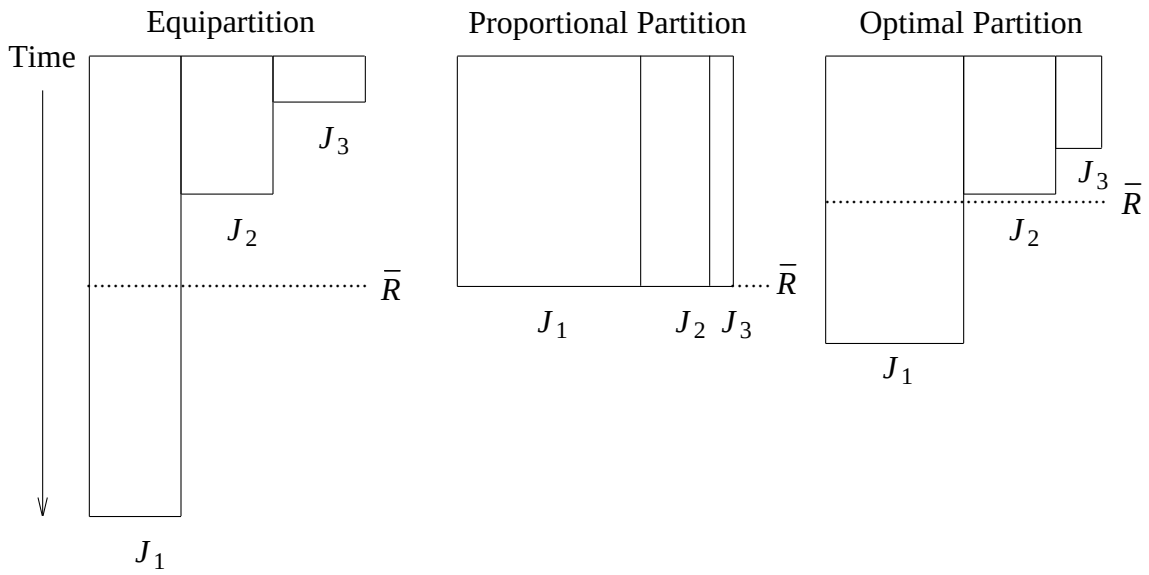


Figure 3.1: Comparison of EQUI, PROP and ROOT partitioning techniques

3.4.5. Comparing ROOT with EQUI

We now consider the potential benefits obtained by knowing the amount of work, W_i, and using the ROOT policy, by comparing it with EQUI. (Since EQUI and PROP yield identical mean response times, there is no need to compare with PROP.) This is done by considering the mean response time ratio, $\bar{R}^{(R/E)} = \frac{\bar{R}^{(R)}}{\bar{R}^{(E)}}$, where $\bar{R}^{(R)}$ and $\bar{R}^{(E)}$ are the mean response times obtained using ROOT and EQUI, respectively.

First consider the response time, $R_i^{(R)}$, of job J_i, when executed using ROOT.

$$R_i^{(R)} = \frac{W_i}{p_i} \quad \text{and} \quad p_i = \frac{P\sqrt{W_i}}{\sum_{j=1}^{N}\sqrt{W_j}} \quad \text{so} \quad R_i^{(R)} = \frac{W_i \sum_{j=1}^{N}\sqrt{W_j}}{P\sqrt{W_i}} = \frac{\sqrt{W_i}\sum_{j=1}^{N}\sqrt{W_j}}{P}.$$

$$\text{So} \quad \bar{R}^{(R)} = \frac{1}{N}\sum_{i=1}^{N}\frac{\sqrt{W_i}}{P}\sum_{j=1}^{N}\sqrt{W_j} = \frac{1}{NP}\left(\sum_{i=1}^{N}\sqrt{W_i}\right)^2. \quad \text{Since } \bar{R}^{(E)} = \frac{\sum_{k=1}^{N} W_k}{P},$$

$$\bar{R}^{(R/E)} = \frac{\bar{R}^{(R)}}{\bar{R}^{(E)}} = \frac{1}{NP}\,\frac{\left(\sum_{i=1}^{N}\sqrt{W_i}\right)^2}{\frac{\sum_{k=1}^{N} W_k}{P}} = \frac{\left(\sum_{i=1}^{N}\sqrt{W_i}\right)^2}{N\sum_{k=1}^{N} W_k}.$$

The reason for deriving the mean response time ratio, $\bar{R}^{(R/E)}$, is to consider how close to and far from optimal EQUI can be. We are, therefore, interested in upper and lower bounds on $\bar{R}^{(R/E)}$.

Theorem 3.2 : $\frac{1}{N} \le \bar{R}^{(R/E)} \le 1$ or $\frac{1}{N}\bar{R}^{(E)} \le \bar{R}^{(R)} \le \bar{R}^{(E)}$.

Proof : For a proof of this theorem refer to the Appendix. ■

These bounds are achievable and in fact, the equipartition (EQUI) and optimal (ROOT) policies yield identical processor allocations (i.e., $\bar{R}^{(R/E)} = 1$) when $W_1 = W_2 = W_3 = , \ldots , = W_N$. However, because reallocating processors is not permitted when a job completes execution, the equipartition method can yield a mean response time N times

worse than the ‘‘optimal’’ technique (thus achieving the other bound). This can be seen intuitively by noting that the mean response time of the equipartition method will be poorest when there is one large job, J_1, and $N-1$ arbitrarily small jobs, $J_2, J_3, \ldots, J_N$.

Figure 3.2 illustrates this example and shows the differences in processor allocations and execution times between these two policies. The $N-1$ infinitesimally small jobs will complete immediately and the mean response time will be dominated by the time to complete the execution of J_1. The response time of J_1 and thus the mean response time could be reduced by allocating it more processors and allocating the other jobs fewer processors. As the work being done by jobs $J_2, J_3, \ldots, J_N$ approaches zero, the improvement in the response time of J_1 approaches N. Therefore, under the given assumptions, the equipartition method could be up to N times worse than the optimal partition.

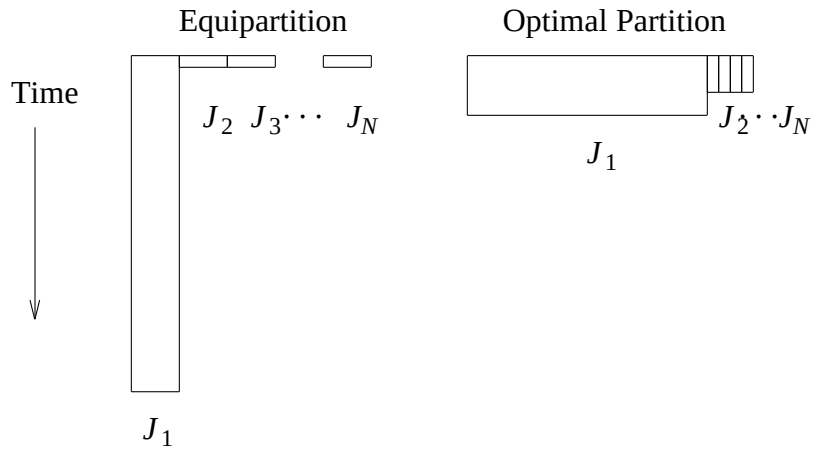


Figure 3.2: Static Equal Partition versus Optimal Partition

This section has shown that knowing and using the amount of work when making processor allocation decisions can improve mean response time substantially. Under the given assumptions, allocating processors in proportion to the square root of the amount of work yields optimal performance. It is also worth noting that if the assumption that all jobs must be started simultaneously is relaxed, a Least Work First (LWF) policy is optimal. In the next section we compare the LWF policy with a policy that dynamically repartitions processors equally among all jobs.

3.5. Scheduling with Reallocations (Dynamic Partitioning)

Some potential problems with the version of the problem stated in the previous section are the following.

- Processors are wasted if they can not be reallocated when an application finishes executing. In some environments repartitioning may be possible and the cost of doing so may not be prohibitive.
- Performance may be improved by holding some applications for later execution. That is, it might be preferable not to start executing *N* applications at once.

Therefore, in this section we allow the execution of jobs to be delayed and processors to be reallocated when a job finishes execution (i.e., we relax assumptions A3.8 and A3.9). In order to keep the analysis simple, however, we assume that repartitioning is instantaneous and free.

A3.10: Processor repartitions occur instantaneously and without cost.

We now compare a popular method for determining processor allocations that does not require any information about applications (which we call dynamic equipartition), with a method that is optimal if the amount of work each job executes is known *a priori*. This is done to gain insight into how close to or far from optimal the equipartition technique might be and to determine how much performance might be improved by knowing the amount of work being executed by each job.

3.5.1. Dynamic Equipartition (DYN-EQUI)

Practical scheduling policies that have recently received favourable attention in the literature dynamically repartition processors among applications [Tucker1989] [Leutenegger1990] [Zahorjan1990] [McCann1993]. One technique, proposed by Tucker and Gupta, limits the total number of processes in the system to be equal to the number of processors and space-shares processors among applications [Tucker1989]. The scheduling policy employed by Tucker and Gupta is called equipartition because it allocates an equal number of processors to each application (if they have enough parallelism) and reallocates processors upon job arrivals and departures. This policy is easy to implement as it requires no information besides the number of jobs and processors in the system. A technique called dynamic scheduling, proposed by Zahorjan and McCann [Zahorjan1990], improves upon equipartition by recognizing that the parallelism of some applications changes during execution and by dynamically adjusting processor allocations with changes in job parallelism. In a sense they attempt to utilize processors more effectively. Dynamic scheduling differs from equipartition in that processor reallocations can occur as a result of changes in job parallelism rather than just at job arrivals

and departures. Although dynamic scheduling is able to distinguish between different applications to some degree it does not take into account the amount of work being executed by applications or the extent to which the processors allocated to each application are effectively utilized.

Under our current assumptions (A3.1 to A3.7 plus A3.10) all jobs execute with perfect efficiency and are thus capable of utilizing any number of processors. Therefore, for our immediate purposes, these two techniques are equivalent and we refer to them as dynamic equipartition (DYN-EQUI). The term ‘‘dynamic’’ is used to indicate that processors can be repartitioning dynamically rather than being partitioned in a static fashion, as was the case in the previous section.

DYN-EQUI treats all jobs equally by giving an equal portion of processors to all jobs and reallocating processors whenever a job arrives or departs. Because we assume that there are no new arrivals, repartitioning takes place only when jobs complete. Let $n(t)$ be the number of applications executing at time t and $p_i(t)$ be the number of processors allocated to job J_i, then this policy can be expressed as follows.

DYN-EQUI: $$p_i(t) = \frac{P}{n(t)}$$

3.5.2. Least Work First (LWF)

If the amount of work or simply a ranking of the relative amounts of work each job executes is known, then executing jobs with the Least amount of Work First (LWF) is optimal [Sevcik1994].

LWF:

$$p_i = \begin{cases} P & \text{for some } i \mid W_i = min(W_j), \quad \text{over all } j \text{ that have not been executed} \\ 0 & \text{for all other jobs} \qquad\qquad\qquad\qquad \text{(ties are broken arbitrarily)} \end{cases}$$

Sevcik also points out that if there is a Poisson stream of arrivals of new applications and preemption is free, then a Least Remaining Work First (LRWF) policy is optimal. Under a LRWF policy, if a job arrives and the amount of work it requires is less than the amount of work left for the executing application to complete, then the executing application is preempted and all P processors are used to execute the new arrival.

3.5.3. Comparing LWF with DYN-EQUI

Figure 3.3 provides an example of how the two scheduling policies would partition processors differently when executing three jobs, J_1, J_2, J_3, $W_1 > W_2 > W_3$. The equipartition method begins by sharing the processors equally among the three applications. Once job J_3 completes at time $R_3^{(D)}$, the processors are divided between the two remaining jobs until job J_2 completes at time $R_2^{(D)}$, after which point only job J_1 remains and is allocated all P processors until its execution is completed at time $R_1^{(D)}$.

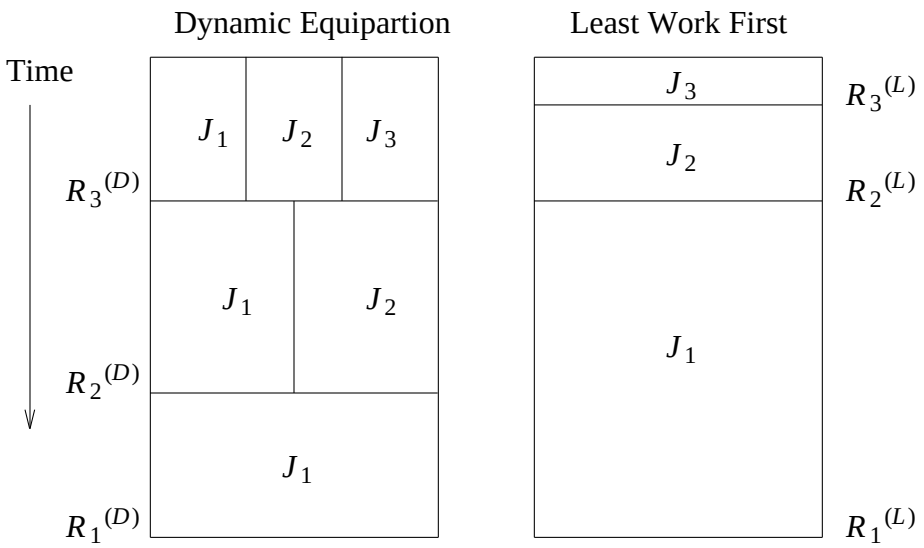


Figure 3.3: Dynamic Equipartition versus Least Work First

The Least Work First policy (LWF) ranks each application by the amount of work to be executed. Then, all P processors are allocated to each application in turn beginning with the application executing the least amount of work. Completion times of jobs J_3, J_2, and J_1 are denoted by $R_3^{(L)}$, $R_2^{(L)}$, and $R_1^{(L)}$, respectively.

As was done in the previous section, we now analytically compare DYN-EQUI, which treats all applications equally but does not require knowledge of the work an application executes, with LWF, which requires and uses knowledge of the amount of work to obtain an optimal schedule (LWF). We perform this comparison in order to:

- Determine how much room there might be to improve upon the equipartition technique.
- Compare static and dynamic processor allocation techniques.

Without loss of generality, we label the applications $J_1, J_2, \ldots, J_N$ such that $W_1 \geq W_2 \geq \cdots \geq W_N$. $R_i^{(D)}$ represents the response time of J_i under the DYN-EQUI scheduling policy and $R_i^{(L)}$ is the response time of J_i under the LWF scheduling policy. The mean response time when the jobs are scheduled using the DYN-EQUI policy is denoted, $\bar{R}^{(D)}$, and $\bar{R}^{(L)}$ denotes the mean response time of the same applications when executed with the LWF policy.

$$\bar{R}^{(D)} = \frac{R_1^{(D)} + R_2^{(D)} + \cdots + R_N^{(D)}}{N} \qquad \bar{R}^{(L)} = \frac{R_1^{(L)} + R_2^{(L)} + \cdots + R_N^{(L)}}{N}$$

First consider the application response times when scheduled using the LWF policy. Because jobs have been labelled such that $W_1 \geq W_2 \geq \cdots \geq W_N$, J_N is the job executing the least work. It is, therefore, executed first and is allocated all P processors. Hence, its response time is:

$$R_N^{(L)} = \frac{W_N}{P} .$$

The response times of the remaining jobs can be expressed in terms of the response times of the previously completed jobs. In turn, each remaining job is allocated all P processors. Therefore, the response time of each job is comprised of $\frac{W_i}{P}$ time units to execute and a wait period determined by the response time of the previously completed application. Consequently:

$$R_{N-1}^{(L)} = R_N^{(L)} + \frac{W_{N-1}}{P} ,$$

$$R_i^{(L)} = R_{i+1}^{(L)} + \frac{W_i}{P} , \text{ and}$$

$$R_1^{(L)} = R_2^{(L)} + \frac{W_1}{P} .$$

Using $\bar{R}^{(L)} = \frac{1}{N} \left(R_1^{(L)} + R_2^{(L)} + \cdots + R_N^{(L)} \right)$ and substituting recursively for each $R_i^{(L)}$ we obtain:

$$\bar{R}^{(L)} = \frac{1}{P\,N} \left(W_1 + 2\,W_2 + 3\,W_3 + 4\,W_4 + \cdots + N\,W_N \right) , \text{ or}$$

$$\bar{R}^{(L)} = \frac{1}{P\,N} \sum_{i=1}^{N} i\, W_i \,. \tag{3.1}$$

Note that alternate formulations of $\bar{R}^{(L)}$ are possible but we make use of equation (3.1) later.

Now consider the response time of each application when scheduled using the dynamic equipartition policy. Once again J_N is the job executing the least amount of work and will, therefore, be the first to complete. Because J_N is allocated $\frac{P}{N}$ processors (as are each of the other applications), its response time is $R_N{}^{(D)} = \frac{N\,W_N}{P}$.

The response time of each remaining job can again be expressed in terms of the response time of the previously completed job. Figure 3.4 shows an example of a number of jobs and their corresponding response times. The response time of J_{N-1} is $R_{N-1}{}^{(D)}$, which is equal to the response time of J_N, $R_N{}^{(D)}$, plus the time required to complete the work remaining to be executed by J_{N-1}. (Since a portion of the work is executed during the execution of J_N.) The amount of work remaining is $(\,W_{N-1} - W_N\,)$ because the work that J_{N-1} was able to execute during the execution of J_N, is the same as J_N was able to execute, and is equal to W_N. The number of time units required to execute the remaining work is determined by the number of processors assigned. After J_N completes, $N-1$ jobs remain in the system. Therefore, J_{N-1} will finish executing with $\frac{P}{N-1}$ processors. Consequently, the number of time units required to execute the remainder of the work for J_{N-1} will be:

$$\frac{W_{N-1} - W_N}{\frac{P}{N-1}} \,.$$

Therefore,

$$R_{N-1}{}^{(D)} = R_N{}^{(D)} + \frac{N-1}{P}\left(\, W_{N-1} - W_N \right), \quad \text{in general}$$

$$R_i{}^{(D)} = R_{i+1}{}^{(D)} + \frac{i}{P}\left(\, W_i - W_{i+1} \right), \quad \text{and}$$

$$R_1{}^{(D)} = R_2{}^{(D)} + \frac{1}{P}\left(\, W_1 - W_2 \right).$$

Now using $\bar{R}^{(D)} = \frac{1}{N}\,(\,R_1{}^{(D)} + R_2{}^{(D)} + \cdots + R_N{}^{(D)}\,)$ and substituting recursively for each $R_i{}^{(D)}$ we obtain:

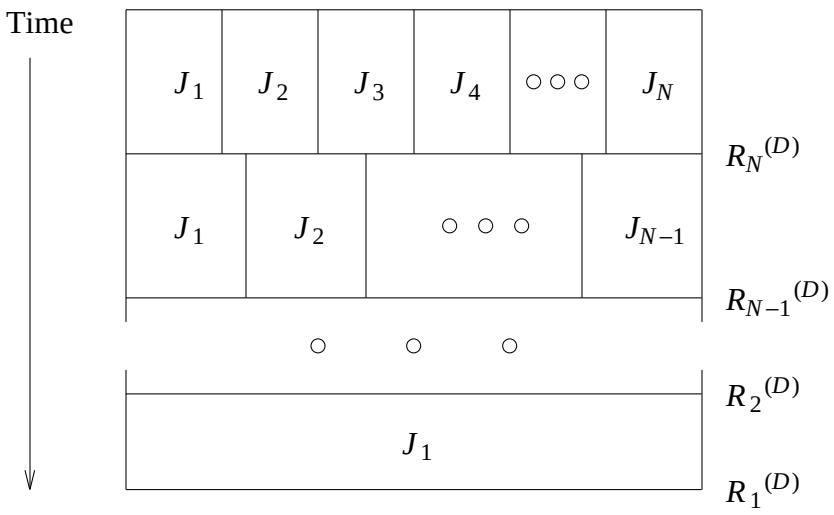


Figure 3.4: Computing Response Times for Dynamic Equipartition

$$\bar{R}^{(D)} = \frac{1}{P\,N}\left[\, W_1 + 3\,W_2 + 5\,W_3 + 7\,W_4 + \cdots + (2N-1)\,W_N \right] , \text{ or}$$

$$\bar{R}^{(D)} = \frac{1}{P\,N}\sum_{i=1}^{N} (\,2i-1\,)\,W_i \,. \tag{3.2}$$

In order to determine how far from optimal the dynamic equipartition method is we consider the mean response time ratio,

$$\bar{R}^{(L/D)} = \frac{\bar{R}^{(L)}}{\bar{R}^{(D)}} \,. \tag{3.3}$$

By substituting equations (3.1) and (3.2) into equation (3.3) we have:

$$\bar{R}^{(L/D)} = \frac{\frac{1}{P\,N}\sum_{i=1}^{N} i\,W_i}{\frac{1}{P\,N}\sum_{i=1}^{N} (\,2i-1\,)\,W_i} = \frac{\sum_{i=1}^{N} i\,W_i}{\sum_{i=1}^{N} (\,2i-1\,)\,W_i} \,. \tag{3.4}$$

Our interest is in the range of values that can be assumed by $\bar{R}^{(L/D)}$. Therefore, we determine bounds on $\bar{R}^{(L/D)}$ and examine the conditions under which these bounds are attained.

Theorem 3.3 : $\frac{N+1}{2\,N} \le \bar{R}^{(L/D)} \le 1$ or $\frac{N+1}{2\,N}\,\bar{R}^{(D)} \le \bar{R}^{(L)} \le \bar{R}^{(D)}$.

Proof : (This proof is presented here rather in the Appendix because we will make use of some of these equations in later discussions.)

$$\frac{1}{\bar{R}^{(L/D)}} = \frac{\sum_{i=1}^{N} (2i-1)\, W_i}{\sum_{i=1}^{N} i\, W_i} = \frac{2\sum_{i=1}^{N} i\, W_i}{\sum_{i=1}^{N} i\, W_i} - \frac{\sum_{i=1}^{N} W_i}{\sum_{i=1}^{N} i\, W_i} = 2 - \frac{\sum_{i=1}^{N} W_i}{\sum_{i=1}^{N} i\, W_i}.$$

Now let $W = \sum_{i=1}^{N} W_i$ (i.e., the total work) and let $E[I] = \sum_{i=1}^{N} i \left(\frac{W_i}{W} \right)$. Therefore,

$$\bar{R}^{(L/D)} = \frac{1}{2 - \frac{1}{E[I]}}. \tag{3.5}$$

$\frac{W_i}{W}$ can be thought of as the fraction of work that W_i contributes to the total amount of work to be executed, W. Therefore, $0 \le \frac{W_i}{W} \le 1$ and $\sum_{i=1}^{N} \frac{W_i}{W} = 1$. So the values of $\frac{W_i}{W}$ can be thought of as a discrete probability distribution and $E[I] = \sum_{i=1}^{N} i \left(\frac{W_i}{W} \right)$ is simply the definition for the expected value or mathematical expectation of that distribution [Mendenhall1981]. Recall that we have labelled the jobs such that $W_1 \ge W_2 \ge \cdots \ge W_N$. $E[I]$ is minimized and the bound approaches 1 when all but one of the jobs have essentially no work to execute $W_2 = W_3 = W_4 = \cdots = W_N \approx 0$ and the work of job J_1 is, therefore, $W_1 \approx W$. $E[I]$ is maximized when $W_1 = W_2 = W_3 = \cdots = W_N$ and in this case $E[I] = \frac{N+1}{2}$. Therefore, $1 \le E[I] \le \frac{N+1}{2}$ and $\frac{N+1}{2N} \le \bar{R}^{(L/D)} \le 1$, thus proving the theorem and showing that the bounds are achievable.

■

It is interesting to note that these results are the same as recent competitive analysis results for round-robin scheduling in uniprocessor systems [Motwani1993]. This is not surprising since both the round-robin policy for uniprocessors and dynamic equipartition policies for multiprocessors share processing power equally among all applications. Note that under our assumptions a round-robin job policy as described by Leutenegger and Vernon [Leutenegger1990] that has arbitrarily small quanta and does not incur any overhead when switching between jobs is equivalent to the dynamic equipartition policy.

An important question is, under what circumstances can these extremes be realized. As explained in the proof, the mean response time obtained with DYN-EQUI will approach optimal only when the work to be executed by all but one of the jobs is arbitrarily small. However, such a distribution is unlikely to occur. Therefore, DYN-EQUI will not be optimal for ‘‘interesting’’ distributions of W_i. As also explained in the proof, one circumstance under which DYN-EQUI performs most poorly relative to optimal occurs when the work being done by each application is equal ($W_1 = W_2 = W_3 = \cdots = W_N$). In this case, since $\lim_{N\to\infty} \frac{N+1}{2N} = \frac{1}{2}$, $\bar{R}^{(L/D)}$ asymptotically approaches $\frac{1}{2}$ as the number of jobs, N, approaches infinity. Table 3.2 shows how quickly the value of $\bar{R}^{(L/D)}$ grows with N in the case when all jobs execute the same amount of work. It demonstrates that significant benefits may be obtained by using the amount of work each job executes, even when there are not a large number of jobs in the system (although in under admittedly restrictive assumptions).

N	$\frac{N+1}{2N} = \bar{R}^{(L/D)}$	
1	1	= 1.00
2	3 / 4	= 0.75
3	2 / 3	= 0.67
4	5 / 8	= 0.62
10	11 / 20	= 0.55
∞	1 / 2	= 0.50

Table 3.2: How far Dynamic Equipartition can be from Optimal

We have just discussed two contrived and extreme distributions of W_i that demonstrate circumstances under which the equipartition technique yields mean response times that are closest to and farthest from optimal. We now consider how more realistic distributions of work, W_i, will affect the difference between mean response times obtained using the dynamic equipartition and optimal policies.

Once again consider equation (3.5): $\bar{R}^{(L/D)} = \frac{1}{2 - \frac{1}{E[I]}}$. For any distribution for which $\lim_{N\to\infty} E[I] = \infty$ we have $\bar{R}^{(L/D)} = \frac{1}{2}$. This is true for all distributions with non zero amounts of work since the value of $\sum_{i=1}^{N} i \left\lfloor \frac{W_i}{W} \right\rfloor$ can only grow as jobs are added. (Recall that jobs are

labelled such that $W_1 \geq W_2 \geq W_3 \geq \cdots \geq W_N$.) Therefore, for ‘‘interesting’’ distributions of work, W_i, as N approaches infinity the mean response time of DYN-EQUI asymptotically approaches twice optimal. Note, however, that the rates at which the bound is approached varies with different distributions of W_i. These results emphasize the dangers in performing comparisons using only a small number of jobs since the difference in mean response times obtained when using DYN-EQUI and LWF increases with the number of jobs.

3.5.4. Substantiating the Analysis

The previous sections have demonstrated analytically that knowing and using the service demands of applications may dramatically improve performance over techniques that do not use such information (such as equipartition). The potential problems with applying these theoretical results in a more realistic environment are:

1) We may not have *a priori* knowledge of the amount of work to be executed by each application (Assumption A3.6).

2) Applications rarely execute with perfect efficiency (Assumption A3.7).

3) Repartitioning processors can not be done for free (Assumption A3.10). In fact, the overheads may outweigh the gains.

For a moment let us continue to use Assumptions A3.6 and A3.7, but relax the assumption that processor reallocations can be done instantaneously and for free (assumption A3.10). We perform a simple experiment, using the Hector multiprocessor, in which we compare the mean response time of two scheduling policies executing an identical collection of applications. The purpose of this experiment is to:

- Determine if the overheads are likely to outweigh the benefits.
- Validate the model used in obtaining the analytic results.

We vary the number of simultaneously started applications from 1 to 4 and compare the mean response times obtained when using the Least Work First (LWF) policy and the Dynamic Equipartition Policy (DYN-EQUI). The case when the differences between the response times is greatest is examined; that is, when $W_1 = W_2 = \cdots = W_N$. This is done by executing the same application on the same data a number of times. A simple application called Markov is used, which computes steady state probabilities for a specified queueing system by recursively enumerating all possible states the system might be in and then finding stationary probabilities for each state. Markov was chosen because it conforms to the assumptions we have made regarding the execution of applications. That is, the characteristics of the application are:

1) It achieves close to perfect speedup on the system used.

2) The work being done, although not infinitely divisible, has been divided into many relatively fine-grained threads.

3) Location effects are negligible. Therefore, only the number of processors allocated to each application affects the overall performance and not which processors the application executes on.

The allocation policies have been modified to handle an integral number of processors by adjusting each p_i to the nearest integer such that $p_1 + p_2 + \cdots + p_N = P$. However, the number of processors used in the experiment, $P = 12$, can be evenly divided by the number of jobs executed: 1, 2, 3, and 4.

Table 3.3 shows the mean response times obtained when using two scheduling policies, LWF and DYN-EQUI, to execute different numbers of copies of the same application on the Hector multiprocessor. The same application is used to maximize the difference between the policies and to try to achieve the bound. The columns of the table show the number of copies of the application executed simultaneously, N, the mean response times using the two policies, along with the ratio of the measured mean response times $\frac{\text{LWF}}{\text{DYN-EQUI}}$ and the analytic ratio $\frac{N+1}{2N}$. These results show that for a simple application that roughly follows the given assumptions, the mean response time can be improved dramatically if the relative amounts of work to be executed by each application are known *a priori*. It also supports the relevance of the analytic model and the results obtained in the previous section since the measured results follow the theoretical results very closely.

	Mean Response Time (sec.)		Ratio = $\overline{R}^{(L/D)}$	
N	LWF	DYN-EQUI	$\frac{\text{LWF}}{\text{DYN-EQUI}}$	$\frac{N+1}{2N}$
1	21.56	21.56	1.00	1.00
2	31.61	40.87	0.77	0.75
3	41.25	60.01	0.69	0.67
4	51.60	79.76	0.65	0.62

Table 3.3: Comparing Measured with Analytic Results

3.6. Considering New Arrivals

If jobs execute with perfect efficiency, preemption overhead is negligible, and if there is a Poisson stream of new arrivals, a policy that preemptively allocates all processors to the job with the Least Remaining Work First (LRWF) is optimal [Sevcik1994]. We further extended this result to include arbitrary arrivals. First, any jobs arriving at time zero are executed according to the optimal Least Work First policy [Sevcik1994]. Next, consider the time of any new arrival, t, and let $W_i(t)$ be the amount of work remaining to be executed by job J_i at time t. We need to determine an optimal scheduling for jobs, from time t. However, this is can be thought of as the jobs arriving simultaneously at time t, with $W_i(t)$ units of work to execute (rather than W_i). Therefore, an optimal scheduling of jobs from time t, schedules jobs in a least work first fashion according to the remaining work for each job, $W_i(t)$. This is a preemptive least remaining work first policy.

In order to further our understanding of the dynamic equipartition policy and the benefits that can be obtained by knowing and using knowledge of the amount of work each job executes, we again examine the difference between the mean response time obtained using an optimal scheduling of jobs and that obtained using the dynamic equipartition policy. Our interest in considering this problem is to compare these results with the bounds obtained for the case when all jobs arrive simultaneously. If, in the case of new arrivals, the mean response time obtained using DYN-EQUI is guaranteed to be sufficiently close to the optimal mean response time, one might consider using the DYN-EQUI policy.

Once again we compare the mean response time obtained using the optimal policy (LRWF), $\overline{R}^{(LR)}$, with that obtained using DYN-EQUI, $\overline{R}^{(D)}$, by considering the mean response time ratio $\overline{R}^{(LR/D)} = \frac{\overline{R}^{(LR)}}{\overline{R}^{(D)}}$. In this case we construct a workload under which the mean response time ratio is not bounded by some fixed value but decreases with the number of jobs executed. This workload was derived from the workload used to obtain the equivalent result for round-robin scheduling in uniprocessor systems [Motwani1993]. Job sizes and arrival times are chosen such that under LRWF all jobs (except one) execute immediately upon arrival and complete at the instant of the arrival of the next job. However, when scheduled using DYN-EQUI all jobs complete at the same time, maximizing the response time of all jobs. Although this workload is contrived and quite extreme, the point is to demonstrate that the mean response time ratio no longer has a constant guaranteed bound, as was the case when jobs arrived simultaneously, and that blindly using DYN-EQUI can result in mean response times that are very far from optimal.

Let a_i be defined to be the arrival time of job J_i. Consider a workload in which two jobs, J_1 and J_2, arrive at time 0 and execute the same amount of work. That is, $W_1 = W_2$ and $a_1 = a_2 = 0$. Under a LRWF policy, since both J_1 and J_2 arrive at time 0 and execute the same amount of work, we arbitrarily choose to execute job J_2 first. Other jobs that arrive will have less work to execute than J_1, so J_1 will not be activated until all of the other jobs have completed. We choose the arrival time of job J_3 to be equal to the time at which J_2 finishes. That is, $a_3 = \frac{W_2}{P} = \frac{W_1}{P}$. The work job J_3 executes is $W_3 = \frac{W_1}{2}$.

In general: $W_i = \frac{W_1}{i-1} \quad 3 \le i \le N$.

The arrival time of each new job is chosen to correspond to the completion time of the previous job. Therefore,

$$a_i = a_{i-1} + \frac{W_{i-1}}{P}$$

$$= a_{i-1} + \frac{W_1}{(i-2)\,P} \quad 3 \le i \le N .$$

Now consider the execution of this workload using the LRWF scheduling policy. Let $R_i^{(LR)}$ be the response time of job J_i using LRWF. $R_2^{(LR)} = \frac{W_2}{P} = \frac{W_1}{P} = a_3$. Since job J_3 arrives at time a_3 with $W_3 = \frac{W_1}{2}$ units of work to execute, it is activated immediately. Its response time is $R_3^{(LR)} = \frac{W_3}{P} = \frac{W_1}{2\,P}$. This is also equal to the period of time between a_3 and a_4, which means that J_4 executes with all P processors immediately upon arrival. In general,

$$R_i^{(LR)} = \frac{W_1}{(i-1)\,P} \quad 2 \le i \le N .$$

If $W = \sum_{i=1}^{N} W_i$ is the total amount of work executed, then all work is completed at time $\frac{W}{P}$. Because J_1 is not activated until all other jobs have completed, it finishes last with response time $R_1^{(LR)} = \frac{W}{P}$.

Therefore, the mean response time, $\bar{R}^{(LR)}$, is:

$$\bar{R}^{(LR)} = \frac{1}{N}\left(R_1{}^{(LR)} + \sum_{i=2}^{N} R_i{}^{(LR)} \right) = \frac{1}{N}\left(\frac{W}{P} + \sum_{i=2}^{N} \frac{W_1}{(i-1)\,P} \right)$$

$$= \frac{1}{N}\left(\frac{W}{P} + \frac{W_1}{P} \sum_{i=1}^{N-1} \frac{1}{i} \right)$$

Note that, $\sum_{k=1}^{N} \frac{1}{k}$ is the *k*-th harmonic number. We represent it as H_k, thus,

$$\bar{R}^{(LR)} = \frac{1}{N}\left(\frac{W_1 + W_1\, H_{N-1}}{P} + \frac{W_1\, H_{N-1}}{P} \right)$$

$$= \frac{1}{N}\left(\frac{W_1\left(2\, H_{N-1} + 1 \right)}{P} \right) .$$

Under the DYN-EQUI scheduling policy, jobs J_1 and J_2 are started at time 0, with each being allocated $\frac{P}{2}$ processors. When job J_3 arrives at time $a_3 = \frac{W_1}{P}$, job J_1 will have executed for $\frac{W_1}{P}$ units of time while using $\frac{P}{2}$ processors, thus executing $\frac{W_1}{2}$ units of work. Therefore, $\frac{W_1}{2}$ units of work remain to be executed by job J_1 at time a_3. The same is true of job J_2. The work job J_3 executes is $W_3 = \frac{W_1}{2}$, which is equal to the amount of work remaining to be executed by jobs J_1 and J_2 at time a_3. Once job J_3 arrives, each job executes using $\frac{P}{3}$ processors until the arrival of the next job, at time a_4. This period of time between a_3 and a_4 is $a_4 - a_3 = \frac{W_1}{2\,P}$. Therefore, during that time period each job executes $\frac{W_1}{6}$ units of work, leaving each job with $\frac{W_1}{3}$ units of work left to execute, which is the amount of work to be executed by the newly arriving job J_4. In general, when each new job arrives, all jobs in the system (including the new arrival) have the same amount of work left to execute. Therefore, under DYN-EQUI, all jobs finish executing at time $\frac{W}{P}$. If $R_i{}^{(D)}$ is the response time of job J_i using DYN-EQUI, then

$$R_i{}^{(D)} = \frac{W}{P} - a_i \ .$$

Therefore, the mean response time, $\bar{R}^{(D)}$, is:

$$\bar{R}^{(D)} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{W}{P} - a_i \right) = \frac{1}{N} \left(\frac{N\,W}{P} - \sum_{i=1}^{N} a_i \right) .$$

Since $a_1 = a_2 = 0$,

$$\bar{R}^{(D)} = \frac{1}{N} \left(\frac{N\,W}{P} - \sum_{i=3}^{N} \sum_{i=1}^{N-2} \frac{W_1}{i\,P} \right) = \frac{1}{N} \left(\frac{N\,W}{P} - \frac{W_1}{P} \sum_{i=1}^{N-2} \frac{1}{i^2} \right) .$$

Using H_k as the k-th harmonic number and rewriting W in terms of W_1, we have:

$$W = W_1 + \sum_{i=2}^{N} W_i = W_1 + \sum_{i=1}^{N-1} \frac{W_1}{i} = W_1 + W_1\,H_{N-1} .$$

Therefore,

$$\begin{aligned} \bar{R}^{(D)} &= \frac{1}{N} \left(\frac{N\,W}{P} - \frac{W_1}{P} \sum_{i=1}^{N-2} \frac{1}{i^2} \right) \\ &= \frac{1}{N} \left(\frac{N\,W_1 + N\,W_1\,H_{N-1}}{P} - \frac{W_1}{P} \sum_{i=1}^{N-2} \frac{1}{i^2} \right) \\ &= \frac{1}{N} \left(\frac{N\,W_1 \left(H_{N-1} + 1 \right)}{P} - \frac{W_1}{P} \sum_{i=1}^{N-2} \frac{1}{i^2} \right) . \end{aligned}$$

Now consider the mean response time ratio:

$$\bar{R}^{(LR/D)} = \frac{\bar{R}^{(LR)}}{\bar{R}^{(D)}} = \frac{W_1 \left(2\,H_{N-1} + 1 \right)}{N\,W_1 \left(H_{N-1} + 1 \right) - W_1 \sum_{i=1}^{N-2} \frac{1}{i^2}}$$

$$= \frac{2\,H_{N-1} + 1}{N \left(H_{N-1} + 1 \right) - \sum_{i=1}^{N-2} \frac{1}{i^2}} .$$

As N approaches infinity the mean response time ratio, $\bar{R}^{(LR/D)}$, asymptotically approaches zero. Therefore, the DYN-EQUI policy can yield mean response times that are arbitrarily far from optimal (when scheduling a sufficiently large number of jobs). Table 3.4 shows how the mean response time ratio, $\bar{R}^{(LR/D)}$, changes with the number of jobs, N.

N	$\overline{R}^{(LR/D)}$	$1 / \overline{R}^{(LR/D)}$
1	1.0000000	1.000
2	0.7500000	1.333
3	0.6153846	1.625
4	0.5283019	1.893
5	0.4661654	2.145
10	0.3050046	3.279
100	0.0556122	17.982
1000	0.0179587	125.649
10000	0.0010283	972.480
100000	0.0001259	7943.166
1000000	0.0000149	67147.325

Table 3.4: How far Dynamic Equipartition might be from Optimal with arrivals

Although, under the assumption of simultaneous arrivals, the mean response time of DYN-EQUI is guaranteed to be within a factor of two of optimal, a similar bound does not hold when the assumption is relaxed.

We have constructed a workload that demonstrates the extent to which a policy that utilizes information about the work jobs have left to execute can improve mean response time over a policy that does not. However, further research is required to determine the mean response time ratio that could be expected for more realistic distributions of arrival times, a_i, and work, W_i.

3.7. Relaxing Perfect Efficiency

We believe that the bounds on the mean response time ratio, obtained under the assumption that all jobs execute with perfect efficiency, are also valid in the case when the assumption of perfect efficiency is relaxed, if we assume that under the dynamic equipartition policy jobs are never allocated more than $p_i^{\max}$ processors. We define $p_i^{\max}$ to be the largest number of processors that should ever be allocated to job J_i (because the addition of even one more processor will increase its execution time). Informally, the reasons for this belief are that an optimal allocation of processors will not allocate all processors to each job as is the case when applications execute with perfect efficiency. The difference between DYN-EQUI and an optimal partitioning is maximized when the optimal partitioning does not space-share processors (but instead executes them preemptively one at a time) and each application is able to effectively utilize all of the processors. This hypothesis is tested in Chapter 4.

For applications that do not execute with perfect efficiency, an optimal scheduling will space-share processors. The proportion of processors allocated to each job will be influenced by the efficiency with which the work can be executed (which may change over time) and the time required to execute that work. In Chapter 4 we examine a range of space-sharing policies that allocate processors to jobs according to some function of the amount of work each job executes. These policies include variants of the ROOT and PROP policies presented in this chapter.

3.8. Summary

The results derived in this chapter indicate some important points concerning the allocation of processors to jobs if the amount of work for each job is known and can be executed with perfect efficiency:

- If executing multiple jobs simultaneously, without processor reallocations, processors should be partitioned in proportion to the square root of the amount of work executed by each job. That is:

$$p_i = \frac{P\ \sqrt{W_i}}{\sum_{j=1}^{N} \sqrt{W_j}}$$

- This technique can be used to obtain mean response times that are as much as N times better than currently popular techniques that allocate an equal portion of processors to each job.

- If it is possible to control the number of jobs executing and processor reallocations are permitted, Least Work First (LWF) can improve mean response time over dynamic equipartition (DYN-EQUI) by as much as a factor of two.

- If job arrivals are considered the mean response time of DYN-EQUI, relative to optimal, is not bounded by a constant as in the case of simultaneous arrivals. In fact, the improvements obtained in mean response time by using LRWF rather than DYN-EQUI grow with the number of jobs.

The results in this chapter indicate that knowledge of an application's expected remaining work can be used to partition processors unequally to substantially improve performance over equally partitioning processors. This motivates the need for practical techniques for estimating an application's expected remaining work and for policies that effectively use these estimates. These topics are explored in the following chapter.

Chapter 4

Practical Partitioning Strategies

4.1. Introduction

In the previous chapter we have shown that scheduling policies that take advantage of knowledge about an application's remaining work can significantly improve performance over policies that do not, specifically policies that partition processors equally among all applications. Unfortunately, *a priori* knowledge regarding the amount of work being executed by an application is often not available. Consequently, in this chapter we propose and evaluate two heuristics for estimating an application's expected remaining work and experimentally evaluate the performance of scheduling policies that use these heuristics. Because the goal is to reduce mean response time relative to the dynamic equipartition strategy, we first try to gain some insight into how much the mean response time of dynamic equipartition can be improved under some simple but more realistic workloads.

4.2. Reevaluating Dynamic Equipartition

In Chapter 3 we explored bounds on the performance of the equipartition scheduling policies. This involved constructing and examining workloads for which dynamic equipartition exhibits worst case behavior relative to an optimal policy. However, such artificial workloads are unlikely to occur in a real environment. In order to better understand how dynamic equipartition is likely to perform under workloads that are not pessimal, we begin by conducting experiments using a few simplistic workloads. Because we are interested in comparing dynamic equipartition and optimal strategies in an experimental setting, the workloads are designed so that they are closer to what might be observed in a system than the pessimal workloads considered in Chapter 3, but are subject to the constraint that an optimal (or close to optimal) schedule can be identified and executed.

4.2.1. Considering New Arrivals

We consider a simple case where a number of jobs arrive at different times but execute an equal amount of work. Under these circumstances LRWF is optimal. This workload is chosen because a least remaining work first scheduling is equivalent to a first come first serve (FCFS) scheduling and FCFS can easily be implemented. We conduct a series of experiments in which we continue to increase the inter-arrival times and examine the difference in mean response

times between LRWF and DYN-EQUI. For each of the experiments we start one job, by executing Markov, wait for some period of time determined by the inter-arrival time and start the next job (another copy of Markov that will execute the same amount of work). The inter-arrival time for each experiment is fixed and we examine how increasing the number of jobs affects the mean response time ratio. Then we increase the inter-arrival time and perform the same experiment. When executed in isolation on a 12-processor Hector system each application takes approximately 20 seconds to execute and achieves a speedup of 11.89. The first experiment is performed by using an inter-arrival time of zero seconds (i.e., simultaneous arrivals). The second, third, and fourth experiments use inter-arrival times of 5, 10, and 15 seconds, respectively. The inter-arrival times, the number of jobs executed, the mean response times of the two policies and the ratio of their response times are shown in Table 4.1. The last column contains the value $\frac{N+1}{2N}$, which is the ratio of the optimal mean response time over the mean response time of DYN-EQUI when N jobs arrive simultaneously.

Inter-arrival time	N	Mean Response Time (sec.)		Ratio	
		LRWF	DYN-EQUI	$\frac{\text{LRWF}}{\text{DYN-EQUI}}$	$\frac{N+1}{2N}$
0	1	20.13	20.21	1.00	1.00
	2	30.48	39.82	0.77	0.75
	3	41.36	60.54	0.68	0.67
	4	51.81	78.54	0.66	0.62
5	1	20.13	20.21	1.00	1.00
	2	27.86	35.78	0.78	0.75
	3	35.65	50.32	0.71	0.67
	4	43.41	64.52	0.67	0.62
10	1	20.13	20.21	1.00	1.00
	2	25.38	31.43	0.81	0.75
	3	30.64	40.85	0.75	0.76
	4	35.82	49.85	0.72	0.62
15	1	20.13	20.21	1.00	1.00
	2	22.91	27.22	0.84	0.75
	3	25.91	30.98	0.84	0.76
	4	28.36	36.05	0.79	0.62

Table 4.1: Executing a simple workload with increasing inter-arrival times

We see that as the inter-arrival time is increased the mean response time of both policies is reduced. However, as the inter-arrival time increases the mean response time of DYN-EQUI decreases at a faster rate than LRWF, thus decreasing the gap between DYN-EQUI and the optimal policy. In fact, if the inter-arrival time is large enough, a new job will not arrive until

the previous job has completed execution and the DYN-EQUI policy will yield optimal mean response times. These results are not surprising since from our analysis in Chapter 3 we know that DYN-EQUI performs poorly (when compared with the optimal strategy) when it is forced to execute a large number of jobs at the same time, thus allocating a relatively small number of processors to each application.

Although these experiments are very limited in scope, we believe that they will be helpful when examining and interpreting the results in following sections. They illustrate that although DYN-EQUI may perform very poorly relative to an optimal strategy for workloads designed to exercise the extreme behaviour of the two policies, the differences for other workloads can be smaller. Because the system we are using has a relatively small number of processors, the number of jobs in the system at any one time will be relatively small. As we have seen in Chapter 3, this will also limit the degree to which mean response time can be improved.

4.2.2. Considering Imperfect Efficiency

If applications do not execute with perfect efficiency, scheduling policies that use only knowledge of the amount of work an application executes (without considering the efficiency with which the work is executed) may not yield optimal schedules. Unfortunately, if applications do not execute with perfect efficiency, it is not known how to optimally allocate processors without considering a combinatorial number of possibilities [Sevcik1994].

We conduct a simple experiment that is designed to show how the performance of a technique that considers only the amount of remaining work a job has to execute will be affected when applications are not able to execute the work with perfect efficiency. Recall that if all applications execute with perfect efficiency, the mean response time of DYN-EQUI will be farthest from optimal when all jobs arrive simultaneously and execute the same amount of work. Therefore, we consider the case when jobs arrive simultaneously, execute the same amount of work, but do not execute with perfect efficiency. The application used in this experiment, called Clique, finds all cliques of a fixed size k in a given graph. For these experiments, we look for all cliques of size 200 in a random graph containing 1000 nodes and 350,000 edges. Executing with this data set on a 12-processor Hector system this application achieves a speedup of 8.5. The execution times, speedup and efficiency obtained when executed using different numbers of processors are shown in Table 4.2.

Table 4.3 shows the results of starting $N = 1, 2, 3, \text{and } 4$ copies of this application simultaneously. The second column, labelled LWF, contains the mean response time obtained when executing the application using a LWF policy and the third column, DYN-EQUI shows the

Processors	1	2	3	4	6	8	12
Time	83.79	43.33	29.95	23.38	16.57	13.15	9.86
Speedup	1.00	1.93	2.80	3.58	5.06	6.37	8.50
Efficiency	1.00	0.97	0.93	0.90	0.84	0.80	0.71

Table 4.2: Execution time, speedup, and efficiency for Clique

mean response time obtained using the DYN-EQUI policy. The fourth column contains the ratio between the mean response times obtained with LWF and DYN-EQUI. The fifth column contains the analytic bound for when jobs execute with perfect efficiency and the sixth column shows the difference between the experimental mean response time ratio and the analytic bound.

	Mean Response Time (sec.)		Ratio		
N	LWF	DYN-EQUI	$\frac{\text{LWF}}{\text{DYN-EQUI}}$	$\frac{N+1}{2N}$	Diff
1	9.78	9.76	1.00	1.00	0.00
2	16.53	18.34	0.90	0.75	0.15
3	23.23	26.21	0.89	0.67	0.22
4	29.86	32.85	0.91	0.62	0.28

Table 4.3: A simple workload of jobs that do not execute with perfect efficiency

These experiments demonstrate that because the LWF scheduling is not optimal, the difference between the mean response time of LWF and that of DYN-EQUI is not as large as when jobs execute with perfect efficiency. This is because the application is not able to fully utilize all of the processors that it is allocated using the LWF policy. Nonetheless, performance is improved over simply using the DYN-EQUI policy.

The efficiency with which the application is executed affects the response time of the application under both scheduling policies. However, as the number of jobs is increased, the DYN-EQUI policy allocates fewer processors to each application and the application is able to utilize a smaller number of processors with greater efficiency than it can a larger number of processors (as seen in Table 4.2).

We next conduct the same experiment using an application that is less efficient than Clique. This application, called Grav, implements the Barnes-Hut clustering algorithm [Barnes1986] for simulating the gravitational interaction of objects (stars or particles) over time. This particular simulation uses 1000 stars and executes for one unit (iteration) of simulated time. When executed on our Hector multiprocessor it obtains a speedup of 4.9 using 12 processors. This is an admittedly poor implementation of the Barnes-Hut algorithm on this system, but it was

intentionally chosen to show how the LWF policy is affected, relative to the DYN-EQUI policy, when applications execute with poor efficiency. Table 4.4 shows the execution times, speedup, and efficiency when executed using 1, 2, 3, 4, 6, 8, and 12 processors.

Processors	1	2	3	4	6	8	12
Time	50.35	30.14	21.72	17.93	14.21	12.04	10.23
Speedup	1.00	1.67	2.32	2.81	3.54	4.18	4.92
Efficiency	1.00	0.84	0.77	0.70	0.59	0.52	0.41

Table 4.4: Execution time, speedup, and efficiency for Grav

Table 4.5 shows the results of using the LWF and DYN-EQUI policies to execute a number of copies of this application. These results show that the efficiency of this application is low enough that DYN-EQUI yields better mean response times than LWF (i.e., the mean response time ratio in this case is greater than 1.0). As the number of jobs in the system is increased the mean response time ratio increases because DYN-EQUI allocates fewer processors to each application but they are able to use them with greater efficiency. These experiments show that using only information about the amount of work applications execute can improve mean response time, even when applications do not execute with perfect efficiency. However, when the efficiency of the applications is too low, it is preferable to execute more than one application at the same time (i.e., to space-share the processors) as is done under the DYN-EQUI policy.

	Mean Response Time (sec.)		Ratio		
N	LWF	DYN-EQUI	$\frac{\text{LWF}}{\text{DYN-EQUI}}$	$\frac{N+1}{2N}$	Diff
1	10.15	10.23	0.99	1.00	-0.01
2	16.10	14.98	1.07	0.75	0.32
3	21.68	20.16	1.08	0.67	0.41
4	27.34	24.83	1.10	0.62	0.48

Table 4.5: A simple workload of jobs that execute with poor efficiency

In the future, we hope to acquire estimates of an application's efficiency. One simple estimate could be obtained by using average sampled processor utilization as produced, for example, by some systems when displaying a program's executing profile. (Such parallel program execution profiles have been produced by a number of researchers [Owicki1989] [Carlson1992].) Note that if the processors assigned to an application are fully utilized but the processors are not being used effectively (e.g., executing a spin-lock), then these estimates provide no additional information. However, if the processors are not being fully utilized, this

under-utilization will be detected and this information may be used to reallocate some of the processors.

However, before such estimates can be applied, further research is required in order to determine how these estimates might be used. Sevcik has recently proposed the use of an execution time function for modeling the execution of parallel applications [Sevcik1994]. Since most significant scientific applications are executed over and over, the four parameters of the function are estimates based on observations made during previous executions of the applications. (This method may be sensitive to the data being used in the computation.) Wu has shown that this technique can be implemented and that the function closely matches measured execution time functions for a number of programs [Wu1993]. We would hope to develop heuristics that only utilize information available to the system during the execution of the application and do not require the application to be executed and characterized *a priori*.

4.3. Estimating Expected Remaining Work

The optimal processor allocation policy, Least Remaining Work First, does not require detailed information about the amount of work remaining to be executed by each application. Instead, a simple ranking of jobs by the amount of remaining work is sufficient to minimize mean response time. Matloff analytically demonstrates that predictive information regarding the remaining work of jobs need not be strongly correlated with actual values in order to improve the mean response time of applications executing in a uniprocessor environment [Matloff1989]. We, therefore, hypothesize that mean response time can be improved if the scheduler can differentiate between jobs with different remaining execution times (for example, classify jobs as small, medium or large) rather than treating all jobs equally.

A number of possibilities exist for obtaining execution time estimates. They range from requiring that the user know and specify the execution time when the application is submitted, to having the system predict the service demands of applications. Four possible schemes for estimating execution times are:

1) user supplied execution time estimates;

2) system maintained logs for estimating execution times;

3) using the run-time system, or user-level thread package, to keep track of the execution time of each thread and estimating the expected remaining execution time based on the number of threads left to execute and the average execution time of each thread; and

4) estimating remaining execution behaviour based on the past execution of the application. In this case the scheduler keeps track of the execution time accumulated by an application and uses that as an estimate of the expected remaining work. This estimate does not require knowing how many threads are left to execute nor the execution time of the individual threads and has been shown to be accurate in uniprocessor systems [Leland1986].

The first two methods depend on information obtained during previous executions of the application or ideally during the previous executions of the entire workload. The last two schemes do not depend on any knowledge of past execution and are designed to be used in environments in which the workload cannot be easily predicted based on previous executions (or in situations in which estimates can not be provided for all applications). These techniques attempt to gain information about applications during their execution and to use that information to predict future execution.

We briefly describe the first two methods and discuss their advantages and disadvantages. Then, we focus our attention on last two schemes, describe how they can be implemented, and experimentally evaluate their performance.

4.3.1. Using User Supplied Estimates

The method of relying on users to provide estimates of the work an application will execute could work very well in production environments, since after the first execution of the set of applications sufficient information is known to improve scheduling on subsequent runs. (There are environments in which the applications being run are well understood by the users.) If users cooperate with the system, this technique can substantially improve system performance over treating all applications equally. (Some evidence of this has been shown in the previous section which compared DYN-EQUI and LRWF policies.) While theoretically possible, this approach has some practical drawbacks:

- Users may not be able to accurately estimate the execution time of their programs or to even rank them relative to other jobs.
- A greedy user can intentionally supply false information to receive favourable treatment. (However, pricing penalties can eliminate this problem by motivating users to be honest in their estimates.)

4.3.2. Using System Maintained Logs

A method that does not require user-supplied information automatically associates an execution time estimate with each application. In this approach, the system maintains records of previous executions of applications and uses these records to provide service demand estimates to the scheduler on subsequent runs. Application executions could be simply identified by name. One potential drawback of this approach is that the amount of work being executed can change dramatically with changes in the input data and execution parameters used. Naturally, improvements could be made under these circumstances by maintaining additional information such as the data being used, program parameters, and even the user executing the program (since users can use applications in substantially different ways). In a production environment in which the same applications are being executed with similar data on a regular basis, this technique could work very well, because previous executions of the application would serve as accurate estimators of current and future execution requirements. However, the problem remains that the estimates could be dramatically wrong.

The performance of these two techniques will depend upon how closely the workload being scheduled adheres to the timings obtained during previous executions of the workload (since it is not likely that a user will be able to accurately predict execution times without executing the application at least once). Given *a priori* estimates of the total execution times for all applications the system could either produce a batch schedule of job executions or dynamically track the work executed by each application to estimate the remaining work, which could be used with a Least Remaining Work First (LRWF) scheduling policy. These techniques are likely to work very well in production environments in which the same workload is executed regularly. As seen in Chapter 3 and the first section of this chapter, using estimates of the expected remaining work can yield significant improvements in mean response time.

The next two schemes do not depend on information obtained from previous executions of the applications. These techniques attempt to gain information about applications during their execution and to use that information to estimate the expected remaining work. The expected remaining work estimates are then used by the processor allocation policy to dynamically repartition processors.

4.3.3. Using the Run-Time System

We propose and experimentally investigate a method in which the run-time system (in our case the user-level thread package) obtains estimates of an application's expected remaining work and coordinates with the scheduler to improve processor partitions. Note that this technique is not meant to be a definitive approach to multiprocessor scheduling as it can not be applied easily and effectively to all applications. (Limitations will be discussed after further

describing user-level threads). Instead it is meant as a means for studying potential performance gains as well as investigating its use on and benefits to certain types of applications. (Later it will also be used as a basis for comparing other techniques.)

Traditionally multiprocessor applications are divided into *P* units of computation, where *P* is the number of processors in the system. Obvious drawbacks of this technique are that not all problems divide naturally into *P* units of computation, and when they do, the division of the problem may not result in balanced execution times on all *P* processors. Moreover, dedicating *P* processors to one application that spends a large portion of its computation time using only one processor is not an effective way to utilize processors. To alleviate some of these problems important trends have emerged with respect to the parallel programming of multiprocessors. Among them is the use of and system support for user-level threads [Doeppner Jr.1987] [Bershad1988] [Marsh1991] [Anderson1991].

The use of user-level threads offers a number of advantages:

- **Load Balancing** - The more units of computation there are and the smaller each unit is, the better the opportunity for load balancing (assuming overheads are not prohibitive).
- **Dynamic Partitioning** - The processor a thread has been executing on may be reallocated to another application. This is most effectively done by coordinating with the application (e.g., ensuring that it is not holding any locks).
- **Reduced Overhead** - User-level threads are implemented outside of the kernel and as a result incur less overhead for creation, manipulation, context switching and destruction than kernel processes.
- **Improved Scheduling** - The execution of applications is also improved by moving the responsibility for scheduling user-level threads outside of the kernel. Because the kernel has no information regarding the execution of threads of an application the application or run-time system is better equipped to make scheduling decisions.

Upon creation, threads are placed in a queue for future execution (or a note is made of how many threads need to be created if the creation is done in a lazy fashion [Brecht1990] [Mohr1991]). When a processor becomes available a thread is removed from the queue and executed. If the application initially creates all of the threads, and the run-time system monitors the execution time of each thread, then once the execution of the first thread is completed an estimate of the expected remaining work can be obtained by simply multiplying the execution time of the completed thread by the number of remaining threads. (The estimates will be most

accurate when all threads execute the same amount of work, which will not always be the case.) In fact, this estimate of the expected remaining work can be continually updated by maintaining an average of the thread execution times and multiplying the average by the number of threads that have not completed execution (those in the queue and those currently executing). (As we will see in the next section, these assumptions are quite reasonable for some applications.)

Using this technique the run-time system will be able to provide accurate expected remaining work estimates for applications that execute with:

1) only one phase of thread creation,

2) close to perfect efficiency, and

3) small variation in thread execution times.

Note that expected remaining work estimates can be obtained if these criteria are not met. However, the estimates may be very inaccurate and, in fact, may be completely misleading. This is because the execution time of threads that have been executed are used to estimate the execution time of remaining threads and, therefore, the expected remaining work of the application. If the application uses more than one phase of thread creation, the execution times of the later phases will not be factored into the estimates. For example, the first phase of an application may involve executing 100 threads, each of which executes for 1 second, while the second phase might execute 1000 threads, each of which executes for 2 seconds. Estimates of the expected remaining work calculated during the first phase would be grossly inaccurate (100 seconds), when in fact the application executes a total of 2100 seconds of work. The argument could be made that the estimate, although initially bad, may improve once the second phase begins. However, the problem is compounded when several phases of thread creation occur which is common in many parallel applications.

If the application does not execute with perfect efficiency, thread execution times will not accurately reflect the amount of work being executed. However, they may reflect the expected execution time of future threads if all threads encounter approximately the same degradation from overheads. Inaccurate estimates can also be obtained, especially during the early phases of an application's execution, if there is a large variation in thread execution times. As a simple example, consider an application in which each thread performs operations on one row of a lower triangular matrix. All threads will perform operations on a different number of columns, resulting in longer execution times for those threads performing operations on larger numbers of columns. Although the estimates will become more accurate over time (as the average thread execution time increases), the estimates will not accurately reflect the expected remaining work.

We presently set aside the limitations of this technique and consider only those applications for which accurate estimates can be obtained. This is done in the interest of exploring possibilities for improving scheduling decisions and reducing mean response time. (The limitations will be addressed later.)

Assuming that the run-time system is able to accurately estimate the expected remaining work, a scheduling policy that preemptively executes the applications with the least remaining work first (LRWF) can be used. We call this policy RT-LEWF (Run-Time system, Least Expected Work First). Note that this policy is close to optimal if criteria for obtaining accurate estimates are met, since it assumes that applications execute with perfect efficiency.

4.3.3.1. Applicability

The application Markov, used in Chapter 3, approximately follows the current set of assumptions. Therefore, it is reasonable to assume that the run-time system will be able to accurately estimate the amount of work remaining. Other applications (or application kernels) exist that also follow these assumptions and as a result are candidates for use with a technique that coordinates the run-time system estimates of remaining work with a LRWF processor allocation policy. A number of these applications have been implemented and executed on Hector. Table 4.6 shows relevant statistics gathered from the execution of these applications. The statistics are: the input data or parameters used; the number of threads, t; the minimum, maximum, mean and coefficient of variation (i.e., standard deviation divided by the mean) of thread execution times; as well as the total time of all threads executed. Note that the estimate used to predict remaining work is the number of threads remaining to be executed multiplied by the average execution time of those threads that have already finished executing. All times shown are in seconds.

Appl	inputs	t	min	max	mean	cov	total
Markov	$k=4, d=4, c=12$	35	0.88	0.96	0.92	0.02	32.2
Markov	$k=4, d=4, c=14$	84	0.88	0.98	0.94	0.02	78.8
Markov	$k=4, d=4, c=16$	120	1.81	1.98	1.91	0.02	222.5
Markov	$k=3, d=7, c=10$	462	0.68	0.81	0.74	0.03	342.5
Markov	$k=3, d=6, c=12$	252	1.92	2.19	2.06	0.03	517.9
MM	128x128	128	0.05	0.07	0.05	0.08	6.4
MM	256x256	256	0.18	0.23	0.19	0.05	48.4
MM	512x512	512	0.73	0.92	0.75	0.03	383.1
Clique	k=200	127	0.10	2.34	0.68	0.71	87.0
Clique	k=150	127	0.47	14.52	4.08	0.67	518.4
Clique	k=129	127	1.27	22.30	7.69	0.62	976.9

Table 4.6: Accuracy of Run-Time System in Predicting Remaining Work (times in seconds)

The column titled input, describes the different inputs used for the various runs of each application. For the Markov application *k* represents the number of customer classes, *d* the number of devices, and *c* the total number of customers. For the matrix multiply application, MM, the input determines the size of the *n x n* matrices being multiplied. The Clique application finds all cliques of size *k* in the specified graph. In this case the graph is a random graph containing 1000 nodes and 350,000 edges. The measurements were all taken while exclusively using 12 of the 16 processors in the Hector system. (Four of the processors were reserved for the execution of system processes, to avoid interference.)

Table 4.6 shows a number of different applications and data sets for which the thread execution times have a small coefficient of variation and which execute with close to perfect efficiency. (Clique is an exception in both cases.) Therefore, our technique for estimating the expected remaining work should be fairly accurate for this set of applications and, when used with a least remaining work first scheduling policy, should improve response time significantly over the equipartition policy. We now test this hypothesis.

4.3.3.2. Experimental Results

The first experiment we conducted is designed to determine how quickly and accurately the run-time system is able to estimate the expected remaining work. In this experiment we start *N* jobs simultaneously. Each job executes the same amount of work and all jobs execute with perfect efficiency. In order to ensure that all jobs execute the same amount of work, we actually use a number of copies of the same application executing with the same data set. This is the same experiment that was conducted in Section 3.5.4. The mean response time obtained using RT-LEWF should ideally be the same as that obtained using LWF. This workload is designed so that we can compare the mean response time obtained using the expected remaining work estimates with a known and implementable optimal (or nearly optimal) schedule. In this case LWF is optimal and since all jobs execute the same amount of work and arrive simultaneously a FCFS scheduler can be used to obtain a LWF scheduling.

Because we are executing in a dynamic environment where the number of threads that an application creates (and therefore the number of processes the thread system creates) is not known *a priori*, RT-LEWF initially allocates free processors to applications on a first-come-first-served basis until all processors are allocated. Each job is capable of using all processors and continues to request processors until they are all allocated. Each application is allocated at least one processor. After the first thread of an application is executed an estimate of the remaining work is known. Before the first thread is completed the estimate of that job's remaining work is infinity. (This may seem extreme but it prevents it from obtaining more

processors until an estimate of its execution time is obtained.) Once all P processors have been allocated, each of the N applications is dynamically adjusted so that the application estimated to have the least remaining work will be allocated $P-(N-1)$ processors. All other applications are allocated one processor each because one processor is required so that estimates of remaining execution time can be updated. After each thread finishes executing, the process executing that thread notifies the scheduler that it is willing to yield its processor, at the same time providing an updated estimate of the expected remaining work. The granularity of the threads is not large, so the scheduler receives estimates from the run-time system fairly quickly and frequently. If more than one application has the same expected remaining work, the one that has been in the system the longest is given preference.

Table 4.7 contains the results of the experiment using $N = 1, 2, 3$ and 4 jobs. The first column shows the number of jobs executed (again using the Markov application). The next columns contain the mean response times obtained using DYN-EQUI, LWF and RT-LEWF policies respectively. The right half of Table 4.7 contains various mean response time ratios. The first column shows the theoretical value, the second the experimental value obtained by comparing LWF and DYN-EQUI and the third column compares RT-LEWF and DYN-EQUI. Ideally, RT-LEWF will yield results that are as good as LWF.

	Mean Response Time (sec.)			Mean Response Time Ratio		
N	DYN-EQUI	LWF	RT-LEWF	$\frac{N+1}{2N}$	$\frac{\text{LWF}}{\text{DYN-EQUI}}$	$\frac{\text{RT-LEWF}}{\text{DYN-EQUI}}$
1	21.56	21.56	21.56	1.00	1.00	1.00
2	40.65	31.40	31.98	0.75	0.77	0.79
3	60.01	41.25	45.35	0.67	0.69	0.76
4	79.78	51.60	58.90	0.63	0.65	0.74

Table 4.7: Measured Results for RT-LEWF

The results show that when executing only one job RT-LEWF and LWF produce the same response time. This demonstrates that the overheads involved in determining remaining work estimates and making them available to the scheduler are relatively small for one job. (Even though only one job is executing, RT-LEWF policy still provides expected remaining work estimates to the scheduler.) The results also show that for two jobs, the mean response time is only slightly higher when using RT-LEWF than when using LWF. The main reason for the difference is that RT-LEWF allocates at least one processor to each job for the lifetime of the job while LWF does not. Therefore, the number of processors allocated to the job with the least remaining work is reduced by the number of other jobs that have not completed execution. This also explains why the difference between RT-LEWF and LWF increases when the number of

jobs executing is increased. The difference would be smaller if a larger number of processors were used because the relative difference in the number of processors allocated to jobs would be smaller.

In order to illustrate how the mean response time is affected by allocating at least one processor to each job, Table 4.8 shows the results obtained using a modified LWF policy (MLWF). The LWF policy is modified so that each job is allocated at least one processor. Because the FCFS policy was used to emulate LWF for this workload (i.e., when the jobs are the same size), we implement MLWF by using a first-come-first-served policy (FCFS) in which each application is allocated at least one processor. Since all jobs arrive at approximately the same time and all jobs are the same size, this yields a least work first scheduling of the jobs while ensuring that each job is allocated at least one processor. The mean response times obtained using MLWF are nearly identical to those obtained using RT-LEWF (with differences due to experimental error). The run-time system is able to quickly rank the different simultaneously arriving jobs and to schedule them accordingly. (Further evidence of this will be provided.)

Jobs	MLWF	RT-LEWF	$\frac{\text{MLWF}}{\text{RT-LEWF}}$
1	20.11	20.16	1.00
2	31.67	32.35	0.98
3	44.92	45.23	0.99
4	57.17	56.74	1.01

Table 4.8: Comparing Modified LWF and RT-LEWF

In the following series of experiments, we demonstrate that RT-LEWF, which determines expected remaining work estimates during execution, can be used with a number of different applications (besides Markov) and that it performs almost as well as the close to optimal policy, LWF, for which *a priori* knowledge of the work being executed is required. Table 4.9 shows the results of simultaneously starting three copies of the same job while using the DYN-EQUI, RT-LEWF, and MLWF policies. The MLWF policy is included for comparison sake. Each workload was executed five times and mean response times and 90% confidence intervals are shown for each of the policies. The mean response time of each policy is compared with that of DYN-EQUI by dividing the response time by that obtained using DYN-EQUI. This mean response time ratio is shown in the column labelled ‘‘Ratio’’.

The RT-LEWF policy yields response times that are comparable to the MLWF policy and that improve mean response time over DYN-EQUI. Note that RT-LEWF improves mean response time by 26% for the Markov and MM applications. However, the improvements

Appl	input	Jobs	mean	90% CI	Policy	Ratio
Markov	k=4,d=4,c=16	3	59.86	0.19	DYN-EQUI	1.00
Markov	k=4,d=4,c=16	3	44.42	0.40	RT-LEWF	0.74
Markov	k=4,d=4,c=16	3	44.49	0.26	MLWF	0.74
MM	512x512	3	105.85	1.17	DYN-EQUI	1.00
MM	512x512	3	78.27	0.17	RT-LEWF	0.74
MM	512x512	3	81.62	0.12	MLWF	0.77
Clique	k=200	3	26.32	0.60	DYN-EQUI	1.00
Clique	k=200	3	22.96	0.11	RT-LEWF	0.87
Clique	k=200	3	23.13	0.27	MLWF	0.88

Table 4.9: Using RT-LEWF with different applications

obtained for the Clique application, 13%, are not as large. This is because the Clique application does not execute as efficiently as the Markov and MM applications (as demonstrated earlier in this chapter). Using these input values and executing on 12 processors, the Markov application achieves a speedup of 11.9, MM achieves 11.4, while the Clique application attains a speedup of 8.5. The improvements obtained are not as large for the Clique application because $P-(N-1)$ processors are being assigned to each application one at a time and they are not able to use the processors efficiently. That is, the benefits of giving an application a larger portion of the processors are diminished.

The applications used in each of these experiments are identical, therefore, the ability to identify the application with the least expected remaining work has not been clearly demonstrated (since RT-LEWF could have done equally well by choosing any job at random). The applications were also assumed to have arrived simultaneously, therefore, the ability of this technique to adapt to changes in the degree of multiprogramming has not been demonstrated. We now consider these two points by conducting experiments using jobs of various sizes with different arrival times. Once again we compare the performance of RT-LEWF and DYN-EQUI and when possible compare RT-LEWF with that of a close to optimal scheduling of the jobs.

In this experiment, the arrival of jobs is ordered by the amount of work they execute, with larger jobs arriving before smaller jobs. This is done in order to demonstrate that RT-LEWF yields mean response times that are relatively close to optimal. The arrival times are chosen so that the amount of work remaining to be executed for each application is greater than the amount of work to be executed by new arrivals. Hence, a nearly optimal schedule, for this sequence of jobs and arrivals, allocates all of the processors to each application in the order Last Come First Served (LCFS), which can be easily implemented and used as a basis for comparison. Because RT-LEWF allocates at least one processor to each application, we use a modified version of

LCFS (MLCFS) that also ensures that each application is allocated at least one processor, thus permitting us to easily determine how well RT-LEWF performs.

The application used again is Markov, except this time different data sets are used so that each job executes a different amount of work. Table 4.10 shows the arrival time, number of threads, total work and execution time using 12 processors (both in seconds) for each of the four jobs.

Job	Arrival Time	Threads	Work	Time
1	0	252	518.7	45.7
2	5	462	342.5	32.2
3	10	120	229.5	21.1
4	15	35	32.2	4.3

Table 4.10: Workload with different job sizes and different arrival times

Table 4.11 shows the mean response time for the DYN-EQUI, RT-LEWF, and MLCFS policies as well as the ratio comparing these mean response times with the mean response time of DYN-EQUI. The results show that RT-LEWF yields mean response times that are very close to those obtained with MLCFS, which is nearly optimal for this sequence of jobs. (Assuming that at least one processor is allocated to each job and they execute with perfect efficiency.)

Policy	Mean Response Time	Ratio
DYN-EQUI	64.17	1.00
RT-LEWF	49.06	0.76
MLCFS	48.83	0.76

Table 4.11: Comparing DYN-EQUI and RT-LEWF

The series of experiments in this section demonstrates that the run-time system is able to determine estimates of the expected remaining work and to coordinate with the scheduler to change processor allocations according to these estimates to reduce mean response time when compared with DYN-EQUI. In fact, if the applications execute with perfect efficiency (or sufficiently close), RT-LEWF schedules jobs close to optimally (with the exception of allocating at least one processor to each application).

4.3.3.3. Sources of Inaccuracy

Up to this point we have considered applications that mainly follow the assumptions required in order to accurately estimate an application's expected remaining work. Recall that it was assumed that the application has only one phase of thread creation, executes with perfect efficiency, and has only a small variation in the execution time of its threads. The ability to make good processor allocation decisions will depend on the accuracy of these estimates.

Note that this technique can not be used with all applications. However, the scope of applications to which this technique can be applied may be larger than those that strictly adhere to the stated assumptions. For example, consider an application for which the execution times of threads has a large variance. (Each thread may perform some operation on one row of a triangular matrix.) A simple technique for obtaining better expected remaining work estimates might execute threads out of the order in which they were created, or use sampling techniques in order to try to obtain a more realistic estimate of the mean thread execution time as quickly as possible. (Unfortunately, such a technique is likely to adversely affect locality.) Possibilities do exist for improving expected remaining work estimates when there is a large variance in thread execution times, the applications do not execute with perfect efficiency, or there are multiple phases of thread creation and execution. However, rather than discussing a number of possible improvements, we instead propose a more general technique that does not depend on the run-time system.

4.3.4. Using Past Execution

As just discussed, the run-time system is not capable of estimating the remaining work for all applications. However, we have shown both theoretically and experimentally that the benefits from having and using estimates of an application's expected remaining work can be substantial. For this reason, we now propose an alternative strategy for estimating an application's expected remaining work that can be employed with any application.

If we compare multiprocessor scheduling to uniprocessor scheduling, we see that in both cases the optimal scheduling policy requires running the application with the least remaining work first (assuming perfect efficiency). In both cases the optimal scheduling policy requires *a priori* knowledge of the amount of remaining work for each application. For practical purposes, however, such information may be unobtainable. Without knowledge of the remaining work, a possible approach is to treat every application identically. This results in a round-robin time sharing scheme in the uniprocessor case and an equipartition approach in a multiprocessor environment (since it is preferable to space-share processors among jobs rather than to time-share them) [Tucker1989] [Leutenegger1990] [Zahorjan1990] [Gupta1991] [Crovella1991].

One approach that has been successful in uniprocessor scheduling and improves performance over techniques that simply treat all applications equally (e.g., round-robin), is to estimate a job's remaining execution time based on past behavior [Coffman1973]. In a study in which they analyze the behaviour of over nine million UNIX processes, Leland and Ott found that at any point in time (after a three seconds), there is an almost perfectly linear correlation between the age of UNIX processes and the amount of cpu time they will require [Leland1986]. (Age is defined as the amount of cpu time a process has currently received.)

Performance is improved over strictly round-robin scheduling by giving preference to short jobs. This is accomplished by using a multilevel feedback queue and by reducing the priority of larger cpu-bound jobs, thereby reducing the number of quanta they receive [Corbato1962] [Coffman1968]. Under a workload in which the service demand has a large coefficient of variation, this technique yields a lower mean response time than a simple round-robin method (i.e., treating all jobs equally) [Coffman1968] [Coffman1973] [Bunt1976].

We apply this idea in a multiprocessor environment by adjusting the number of processors allocated to each job according to their expected remaining execution time. The processing power allocated to the jobs that are likely to be long running jobs is reduced in favour of giving more processors to shorter jobs, since finishing shorter jobs sooner will reduce mean response time. We now investigate a simple method for estimating a parallel application's expected remaining work based on its past execution.

4.3.4.1. Cumulative Execution Time

A uniprocessor scheduler that uses a multilevel feedback queue is in effect predicting an application's expected remaining execution time using the accumulated execution time of that application. That is, jobs that have been executing for a long time are expected to be large jobs. Therefore, in a multiprocessor system, a natural method for predicting an application's expected remaining work (and its execution time) is to sum the execution times of all of the application's processes.

We combine the technique of estimating expected remaining work by using the cumulative execution time of the application with a least expected work first policy and call this policy ACC-LEWF (ACCumulated execution time Least Expected Work First). The problem with this approach is that because processor allocations change over time, the cumulative execution time rankings of two or more applications may ‘‘ping-pong’’ among the applications.

For example, consider two applications J_1 and J_2. If J_1 has a larger cumulative execution time than J_2, processors may be taken away from J_1 and allocated to J_2 in favour of completing the seemingly smaller application quicker. However, once the processors have been reallocated to J_2, the rate at which J_2 accumulates execution time is greater than J_1. As a result J_2's cumulative execution time may surpass J_1's, at which point processors may be taken away from J_2 and allocated to J_1. This movement of processors between two or more jobs could continue indefinitely, resulting in a type of thrashing in which an excessive number of context switches takes place and jobs are unable to make effective use of the processors before having to give them up. In this case, the reallocation of processors is over-aggressive. As we will see, this thrashing can be controlled. We will also describe (in the next section) a simple method for a more controlled reallocation of processors.

This thrashing phenomenon is illustrated in Figure 4.1. The experiment was run by simultaneously starting three copies of the same application (Markov with the input parameters $k = 4$, $d = 4$, and, $c = 16$). The numbers in parentheses are line numbers used to explain the figure. The characters ''a'', ''b'', and ''c'' represent the three applications and which processors in the system they are executing on. Dashed lines represent unused processors. Each line represents a snapshot of processor allocations and is accompanied by a summary of how many processors are allocated to each application. Snapshots are taken at 3 second intervals. Although each application arrives at approximately the same time, application ''a'' starts slightly ahead of the others and creates as many processes as possible (line 2). Once all of the applications have started executing, application ''c'' is estimated to have the least expected remaining work and is allocated as many processors as possible (line 3). While executing some of that work ''c'' accumulates more execution time, until it eventually has the greatest expected remaining execution time. At this point ''b'' is estimated to have the least work (line 4). However, ''b'' quickly accumulates processor execution time and ''a'' becomes the job estimated to have least amount of remaining work (line 5). This switching of allocations continues until the applications have finished execution. Note that if thrashing becomes frequent, scheduling reduces to a round robin time-sharing policy among jobs, which has been shown to be inferior to space-sharing processors because of the unnecessary overheads incurred when context switching all processors to a new application after each quantum [Leutenegger1990] [Gupta1991] [Crovella1991] [McCann1993].

Table 4.12 shows the mean response time obtained using DYN-EQUI and compares it with ACC-LEWF, when executing the workload just described. MLWF is also included as a basis for comparison. The results were obtained by executing the experiment five times and computing the mean and 90 percent confidence intervals. Note that even though processor allocations change fairly frequently the mean response time is the same as when DYN-EQUI is used. This

```
( 1)  ---- ---- ----                  (12)  bbbb bbbc abbb   1a 10b  1c
( 2)  aaaa baaa aaaa  11a  1b         (13)  aaaa baac aaaa  10a  1b  1c
( 3)  cccc bccc accc   1a  1b 10c     (14)  aaaa baac aaaa  10a  1b  1c
( 4)  bbbb bbbc abbb   1a 10b  1c     (15)  baaa bbac aaaa   8a  3b  1c
( 5)  aaaa baac aaaa  10a  1b  1c     (16)  bbbb bbbc abbb   1a 10b  1c
( 6)  aaaa baac aaaa  10a  1b  1c     (17)  cccc bccc accc   1a  1b 10c
( 7)  aaaa baac aaaa  10a  1b  1c     (18)  cccc bccc accc   1a  1b 10c
( 8)  bbbb bbbc abbb   1a 10b  1c     (19)  bbbb bbbc abbb   1a 10b  1c
( 9)  cccc bccc accc   1a  1b 10c     (20)  bbbb bbbc abbb   1a 10b  1c
(10)  cccc bccc accc   1a  1b 10c     (21)  aaaa baac aaaa  10a  1b  1c
(11)  cccc bccc accc   1a  1b 10c     (22)  ---- ---- ----
```

Figure 4.1: Execution using the ACC-LEWF policy

is because jobs were essentially time-shared in a round-robin fashion resulting in a roughly equal allocation of processing power. (Context switching overhead and cache context are also negligible.) The speed with which processors can be reallocated is mainly determined by the granularity of the threads being executed. This is because reallocation involves coordination between the scheduling server and application (thread system) which occurs only upon the completion of a thread, and because the scheduling server executes on processors separate from those being used to execute the applications.

Policy	Mean	90% CI	Ratio
DYN-EQUI	59.86	0.19	1.00
ACC-LEWF	60.61	0.25	1.01
MLWF	44.49	0.26	0.74

Table 4.12: Comparing DYN-EQUI, ACC-LEWF, and MLWF (times are in seconds)

It is possible to eliminate excessive thrashing by simply maintaining a time-out period during which processor reallocation is not allowed, thus ensuring that once processors are reallocated applications will be given the opportunity to make use of them for a while. This essentially results in a round-robin time-sharing strategy with a large quantum. (McCann, Vaswani, and Zahorjan mention a similar problem with permitting applications to accumulate credits for using fewer processors than their equal share, in order to use a larger share of processors at a later time [McCann1993]. Their solution involves using the rate of accumulation rather than accumulation totals to prevent one application from being able to dominate during periods of contention. A similar scheme is also likely to work here.) Table 4.13 shows the results of repeating the previous experiment while increasing the length of the time-out period.

Note that the mean response time improves as the time-out period increases and in fact eventually approaches the mean response time obtained with MLWF. This is only because the time-out period is large enough that applications are essentially executed one at a time in a first come first served fashion. Since each application is the same size this is equivalent to MLWF.

Policy	Time-out	Mean	90% CI	Ratio
DYN-EQUI	—	59.86	0.19	1.00
ACC-LEWF	0.25	60.61	0.25	1.01
ACC-LEWF	2.50	59.91	0.52	1.00
ACC-LEWF	7.50	58.66	0.35	0.98
ACC-LEWF	15.00	57.89	0.03	0.97
ACC-LEWF	25.00	44.32	0.24	0.74
MLWF	—	44.49	0.26	0.74

Table 4.13: Changing time-out periods, times are in seconds

4.3.4.2. Generalizing the Partitioning Policies

In this section we present a generalized form for obtaining different processor allocation policies. Specific instances of this generalization include the EQUI, PROP, and ROOT policies presented in Chapter 3. This generalization has the positive property that, in a dynamic environment, it will permit us to bias against jobs that are expected to continue running for a long time without incurring the overhead penalties observed when using the ACC-LEWF method (from the previous section).

Generalized Allocation: $$p_i = \frac{P\ W_i^{\alpha}}{\sum_{j=1}^{N} W_j^{\alpha}}$$

It is easy to see that when $\alpha = 1$ we have the PROP policy and when $\alpha = 0.5$ we have the ROOT policy. It is also easy to show that the specific policy obtained when $\alpha = 0$ is EQUI.

EQUI: $$p_i = \frac{P\ W_i^{0}}{\sum_{j=1}^{N} W_j^{0}} = \frac{P}{N}$$

Therefore, when $\alpha = 0$ the policy obtained allocates an equal number of processors to all jobs in the system, independent of the amount of work each job executes. However, when larger values of α are used the resulting policies allocate larger portions of processors to jobs that execute more work. (This was desirable when jobs began execution at the same time and

processors could not be reallocated.) In the extreme (for a large enough value of α), the resulting policy will behave in a preemptive Most Remaining Work First fashion.

However, in a dynamic environment we are more likely to want to bias against larger jobs, especially if the accumulated execution time is being used to estimate the expected remaining work. The generalized form for determining processor allocations can be used with negative values of α to produce an allocation policy that reduces the number of processors allocated to long running jobs. For a large enough negative value of α, the resulting policy will behave in a preemptive Least Expected Remaining Work First fashion (least accumulated work first, if accumulated work is being used as the estimate). A policy that results from choosing a negative value of α also has the favourable property that as the job's accumulated execution time increases, the number of processors allocated to that job decreases.

Figure 4.2 shows the spectrum of allocation policies defined by various values of α, as well as where the EQUI, ROOT, and PROP policies fall in this spectrum. We also further illustrate how α controls the number of processors allocated to each job with a simple example. Consider five jobs, J_1, J_2, J_3, J_4, and J_5. Table 4.14 indicates the amount of work, W_i, that each of these jobs executes.

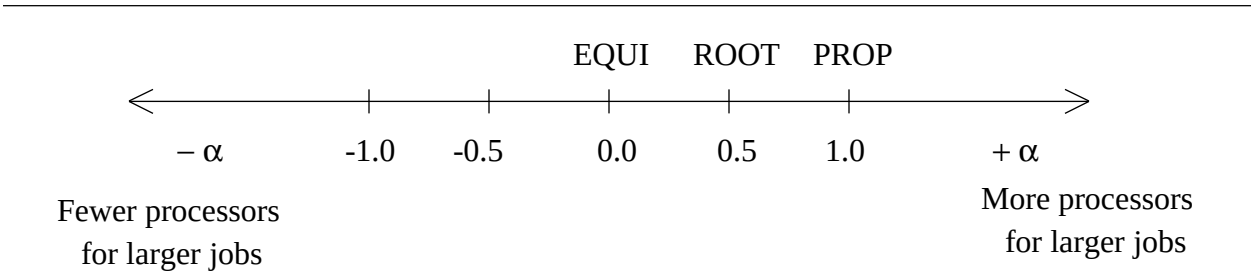


Figure 4.2: Generalization of the processor allocation policies

W_1	W_2	W_3	W_4	W_5
10.00	20.00	40.00	80.00	500.00

Table 4.14: The work executed by the five example jobs

The number of processors allocated to each job will depend on the particular value of α chosen and the number of processors in the system. In this example we assume that the number of processors is $P = 100$. In Table 4.15, the first column contains a number of different values of α, while the remaining columns show the number of processors, p_i, that would be allocated to

each job using the allocation policy that results from the specified value of α. Note that with $\alpha = 3$, the largest job, J_5, would be allocated all 100 processors. As the value of α decreases, the portion of processors it is allocated shrinks. If $\alpha = 0$, job J_5 and the other jobs will each be allocated 20 processors. However, if $\alpha = -2$ or $\alpha = -3$, job J_5 would be allocated zero processors.

α	p_1	p_2	p_3	p_4	p_5
3.00	0	0	0	0	100
2.00	0	0	1	2	97
1.00	2	3	6	12	77
0.50	7	10	14	20	49
0.33	10	13	17	21	39
0.00	20	20	20	20	20
-0.33	31	25	20	16	8
-0.50	37	26	19	13	5
-1.00	53	26	13	7	1
-2.00	75	19	5	1	0
-3.00	88	11	1	0	0

Table 4.15: Number of processors allocated to each of the jobs from Table 4.14, P = 100

By examining the smallest job, J_1, we see the contrast in the number of processors it is allocated. When $\alpha = 3$, it is allocated zero processors. However, as α becomes smaller, J_1 is allocated a greater portion of the processors and when $\alpha = -3$ it is allocated 88 processors.

If it is known that jobs are capable of effectively utilizing all of the processors, it is not desirable to space-share processors. Instead, it is preferable to execute jobs one at a time. However, if the jobs are not capable of effectively utilizing all processors, space-sharing is preferred and the size of the partition allocated to each job should be determined by considering the amount of work the job is expected to execute, in conjunction with its ability to use the processors effectively. This generalization of processor allocation strategies can be thought of as providing a continuum between preemptive (one job at a time), run-to-completion policies and space-sharing policies. At the two extremes (large positive and negative values of α) processors are not space-shared, while smaller values of α provide more and more equal sharing of processors among all jobs. The appropriate choice of α depends on the character of the expected workload, with $\alpha = 0$ being safe, if suitable workload information is not available.

4.4. Summary

In this Chapter we have shown the following:

- The difference between the mean response time obtained using DYN-EQUI and that obtained using an optimal policy is not likely to be as large in practice as the extreme cases demonstrated in Chapter 3.
- Coordination between the run-time system and the scheduler can be used to improve mean response time for applications for which the run-time system is able to estimate an application's expected remaining work.
- Accurate knowledge of application characteristics is not required to improve processor allocations and simple expected remaining work estimates can be derived during the execution of an application.
- The technique of obtaining and applying expected remaining work estimates can be used to reduce mean response time over a technique that allocates an equal portion of processors to each application.
- The generalized form of the processor allocation policies, when combined with a negative value of α, has the desirable property of reducing the processing power allocated to longer running jobs. We expect that an appropriate negative value of α would result in a policy that could be used effectively when executing workloads with a large coefficient of variation in the amount of work executed.

Chapter 5

Application Placement

5.1. Introduction

The main concern with and motivation for designing and developing large-scale multiprocessors is the continued quest for increased performance. As a result, large-scale shared-memory architecture designs strive for scalability. It is likely that all such multiprocessors will have non-uniform memory access times because failure to optimize memory accesses to at least some memory locations will result in memory accesses that are uniformly slow. It is, therefore, imperative that software designers and implementors recognize the differences in memory access times and design software components for scalability.

One of the central propositions of this dissertation is that processor scheduling decisions must consider not only how many processors, but which processors to allocate to an application. We refer to the problem of assigning parallel processes of an application to processors as *application placement*. In this chapter we explore the importance of application placement by measuring the execution time of several parallel applications using different application placements on a shared-memory NUMA multiprocessor. The results of this chapter motivate our work in Chapter 6, where we describe a new scheduling technique designed specifically to improve location decisions and improve scalability.

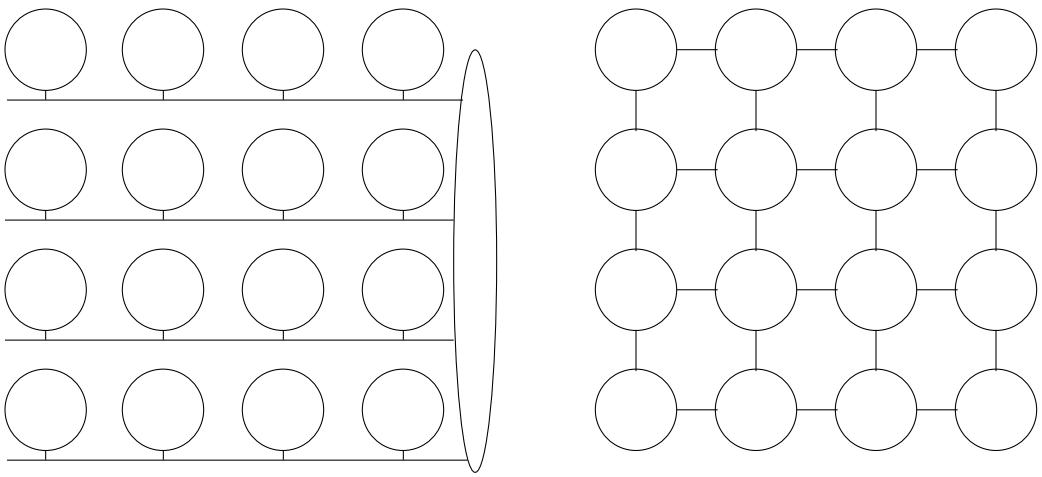
5.2. Application Placement for Localization

In this section we describe how a scheduler might choose a ‘‘localized’’ subset of processors on which to execute an application. Fortunately, most scalable shared-memory architectures adhere to a hierarchical design and, as a result, determining a ‘‘localized’’ subset of processors is not difficult. Since we are assuming a dynamic scheduling environment in which there is no *a priori* knowledge of the number of processes an application will create, processes of an application must be placed one at a time. That is, the problem is different from, and perhaps more difficult than, knowing that an application will be executing with a specified number of processors, p_i, and determining the most localized set of p_i processors. (If it were available, information regarding how many processors an application is capable of using could be used to improve placement decisions.)

From any one processor, remote memory accesses can have successively higher and higher costs as the distance from the requesting processor increases. These costs can be thought of as forming a hierarchy where the access time from a given processor to any memory module within the same level is the same. If, from each processor, we define M_l to be the time to access memory at level l in the hierarchy and $l = 1, 2, 3, \ldots, L$, then:

$$M_1 < M_2 < \cdots < M_l < \cdots < M_L.$$

A currently popular method of building large-scale, shared-memory multiprocessors is to connect processors in a hierarchical fashion, as is shown in Figure 5.1a. For example, this is the type of interconnection scheme used in the Hector system. Figure 5.1b is an example of an alternative interconnection scheme that is not a strictly hierarchical design. (In each diagram a circle represents a processor/memory pair.) This is the interconnection scheme used in the Alewife multiprocessor currently being built at MIT [Agarwal1991]. This non-toroidal, two-dimensional mesh is presented in order to describe how a localized placement of processes could be done in such a system.



(a) (b)

Figure 5.1: Memory access hierarchy in two multiprocessors designs

Since our goal is to preserve the memory reference locality inherent in each application, we consider how to place the processes of an application in a fashion that is localized for the particular architecture. Once the first processor on which the application will begin execution

has been chosen, additional processors are selected such that communication latencies among the set of selected processors is minimized. (How the first processor is selected will be discussed in Chapter 6.)

To preserve an application's locality, in a hierarchical system such as the one in Figure 5.1a, we start by choosing all processors within the same cluster as the first processor selected. Once all of the processors in that cluster have been allocated, the next processor (the fifth selected in this example) can be chosen from any one of the three remaining clusters. The sixth, seventh, and eighth processors, however, should be taken from the same cluster as the processor that was selected fifth. A localized placement would continue in this fashion until all of the processors have been allocated.

In a mesh-based system, such as the one in Figure 5.1b, a localized placement attempts to allocate processors so that the subset of processors selected at any point in time is as close to a rectangle or square as possible, and such that the addition of a minimal number of processors would complete a square. This placement assumes that extra latency is incurred for each hop in the interconnection network. Note however, that a two-dimensional mesh may also be constructed using a number of buses, thus making access times in one dimension independent of the number of processors. This design, however, limits the size of the system because of the limits on the number of processors that can be connected to each bus. (A localized placement in this case would use a similar approach to that used for the hierarchical system, except that the choice of each new cluster, or bus, would be based on proximity to those processors already selected.) Although the DASH architecture is built using a mesh to connect several buses, and it does not consist of processor/memory pairs, memory access times adhere to a hierarchical structure similar to that shown in Figure 5.1a.

In a multiprogrammed environment choosing a localized set of processors for each job may not be as straightforward as just described. This problem is further examined in Chapter 6.

5.3. Impacts of Placement on Performance

In order to examine the importance of localization in shared-memory NUMA multiprocessors, we conduct a series of experiments using the Hector multiprocessor and six parallel applications (or application kernels) each executing on four processors. These applications are described in detail in Chapter 2. The main purpose of these experiments is to assess the importance of application placement; that is, the importance of localization. The experiments are conducted by running each application in isolation under different placement strategies. The execution times of the localized placement are then compared with the non-localized placements.

The Hector system used to conduct the experiments is configured with 16 processors. We dedicate one station (the four processors in Station 0) to the execution of system processes and the workload generator to prevent these processes from interfering with the applications being measured, thus ensuring that differences in execution times are due solely to different placements. Therefore, only stations 1, 2, and 3 are used to execute the applications being tested. Figure 5.2 illustrates a localized and a non-localized placement of four processes of an application and the notation used to represent these placements. In the localized placement, the four dashes ‘‘––––’’ above Stations 0, 1, and 2 indicate that the four processors in each of these stations are not used, while the numbers ‘‘1234’’ above Station 3 indicate where each of the four processes of the application is placed. The first process (1) being the main (parent) process of the program and the remaining three (2, 3, and 4) being child processes. The placement is localized because all four processes of the application execute within one station and the notation is ‘‘–––– –––– –––– 1234’’. The non-localized placement spreads the processes across the twelve processors being considered. (As mentioned previously, Station 0 is not used, in order to to avoid interference with system and workload generating processes which are restricted to that station.) The first process executes on Station 3, the second and fourth on Station 2, and the third on Station 1 and the notation is ‘‘–––– ––– 3 ––42 ––– 1’’.

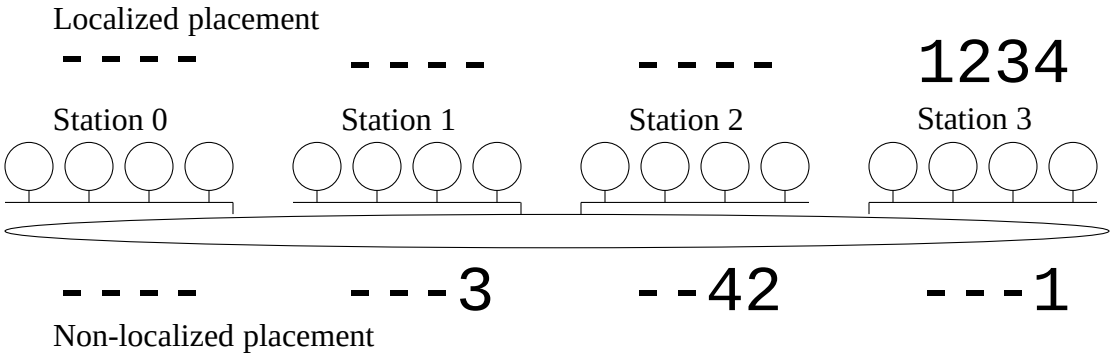


Figure 5.2: Localized and non-localized placements

Figure 5.3 shows the normalized mean execution times of the six applications when executed using the localized and non-localized placements. This graph along with the detailed results in Table 5.1 show that localized application placement does improve the execution time of some of the applications examined (although the differences are very small).

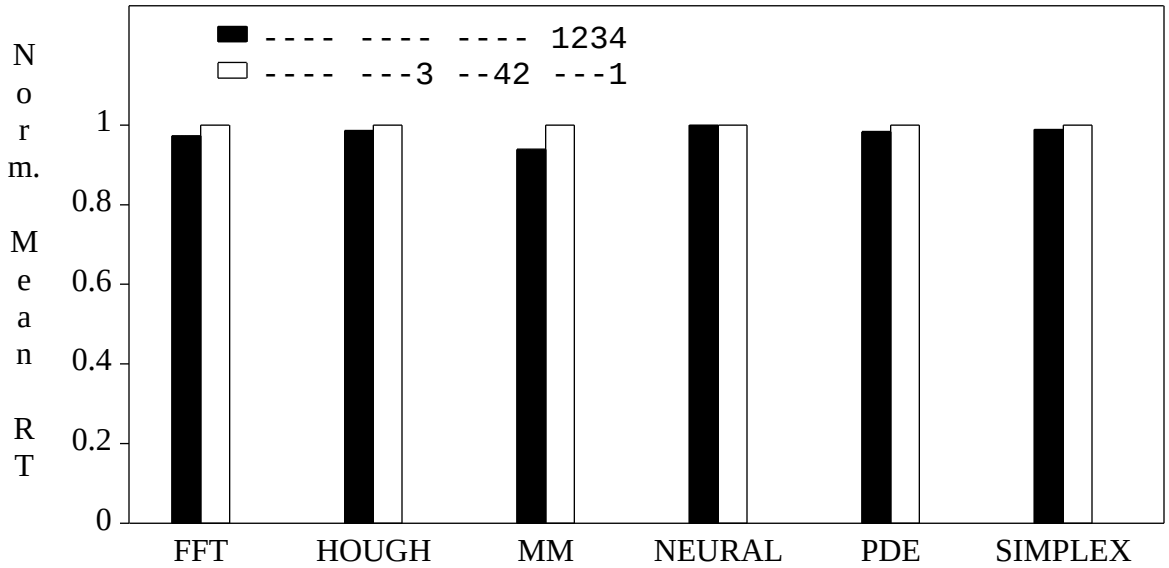


Figure 5.3: Normalized execution times using localized and non-localized placements

Appl	Localized		Non-Localized		% Impr
	Mean	90% CI	Mean	90% CI	
FFT	4.84	0.00	4.71	0.00	2.7
HOUGH	5.01	0.00	4.94	0.01	1.4
MM	5.25	0.01	4.93	0.01	6.1
NEURAL	4.83	0.01	4.83	0.00	0.0
PDE	5.45	0.00	5.36	0.01	1.7
SIMPLEX	19.27	0.06	19.06	0.07	1.1

Table 5.1: Mean execution times, in seconds, using localized and non-localized placements

The table was constructed by executing the applications eight times for each placement. The table contains the mean execution time (Mean) and 90 percent confidence interval (90% CI) for each of the placements, as well as the improvements obtained by using the localized placement (% Impr). This is given as the percentage by which the mean execution time was improved by using the localized placement rather than the non-localized placement, expressed as a percentage of the mean execution time of the non-localized placement. Times are measured in seconds.

Notice that improvements obtained here are not large. This is due to relatively small-size and mild NUMA structure of the prototype 16 processor Hector system. We hypothesize that under a heavy multiprogrammed workload, contention for shared-resources such as the interconnection network (ring) would be reduced under a localized placement, resulting in even

greater benefits. However, experimental results indicate that under the multiprogrammed workloads tested, contention for the ring is not significant. We therefore conclude that in this small-scale mildly NUMA environment, application placement does not greatly affect the execution time of these parallel applications.

The following sections consider larger systems, systems with different architectures, and future multiprocessors, by studying how the importance of localization is affected by increases in memory access latencies.

5.3.1. Increasing Memory Latencies

The NUMAness of a system can be thought of as the degree to which memory access latencies are affected by the distance between the requesting processor and the desired memory location. It is determined by:

- The differences in memory access times between the levels in the memory access hierarchy.
- The number of processors that can be accessed at each level.
- The number of levels.

In order to study the effects of changes in the NUMAness of the system, Hector features a set of switches, called delay switches, that add additional delays to off-station memory requests. The range of settings possible are: 0, 1, 2, 4, 8, 16, 32, and 64 cycles. Every packet destined for a memory module not located on the same station is held up at the requesting processor for the selected number of cycles. The delay switches are used to emulate and gain insight into the performance of:

1) Larger systems — since increases in the size of the system will result in increased memory latencies.
2) Systems of different designs — because some systems have larger memory latencies due to the complexity of the interconnection network or hardware cache coherence techniques.
3) Future systems — because processor speeds continue to increase at a faster rate than memory and interconnection networks.

Table 5.2 shows latencies for local, on-station, and off-station (or ring) memory accesses in units of 60 nano-second cycles. Off-station requests, or those requiring the use of the ring are shown for 0, 4, 8, 16, 32, and 64 cycle delays. The values shown for cache line reads are

pessimistic values in the sense that the true values depend on the relative positions of the source and destination stations. The values shown represent worst case relative positioning. This is because, even though the system is symmetric, asymmetry is introduced because cache line reads consist of one request packet and two reply packets (in order to return the entire 16 byte cache line). Note that the delay switches are not in effect during local or on-station requests. Detailed descriptions of the Hector memory system are available elsewhere [Vranesic1991] [Gamsa1992] [Stumm1993].

	Delay	32bit load	32bit store	cache load	cache writeback
local	-	10	10	19	19
station	-	19	9	29	62
ring	0	23	17	37	42
	4	31	21	49	58
	8	39	25	61	74
	16	55	33	85	106
	32	87	49	133	170
	64	151	81	229	298

Table 5.2: Memory reference times, in processor cycles, on a 16 processor Hector system

To provide insight into the importance of localization on a slightly larger system and in other shared-memory multiprocessors, we set the delay switches to 16 and conduct the same localized versus non-localized placement experiment. The results of this experiment are shown in Figure 5.4. Note that with a delay of 16 cycles, the 16 processor Hector system requires 55 cycles to load one (32 bit) word from remote memory while DASH can require up to 132 cycles per (32 bit) word (assuming there is no contention in both cases). If we account for the differences in cycle times these latencies are 3300 and 3960 nano-seconds, respectively. This illustrates that the 16 cycle delay does not result in unrealistic latencies.

The experimental results of Figure 5.4 show substantial improvements in execution times as a result of localization. The matrix multiplication application (MM) is improved by more than 50%, while the fast fourier transformation (FFT) is improved by more than 25%, the partial differential equation solver (PDE) more than 20%, and the Hough transformation (HOUGH) by more than 15%. Only the neural network application is not significantly improved. This is because it contains an unusually large number of system calls. (A more detailed explanation is provided in a subsequent section.)

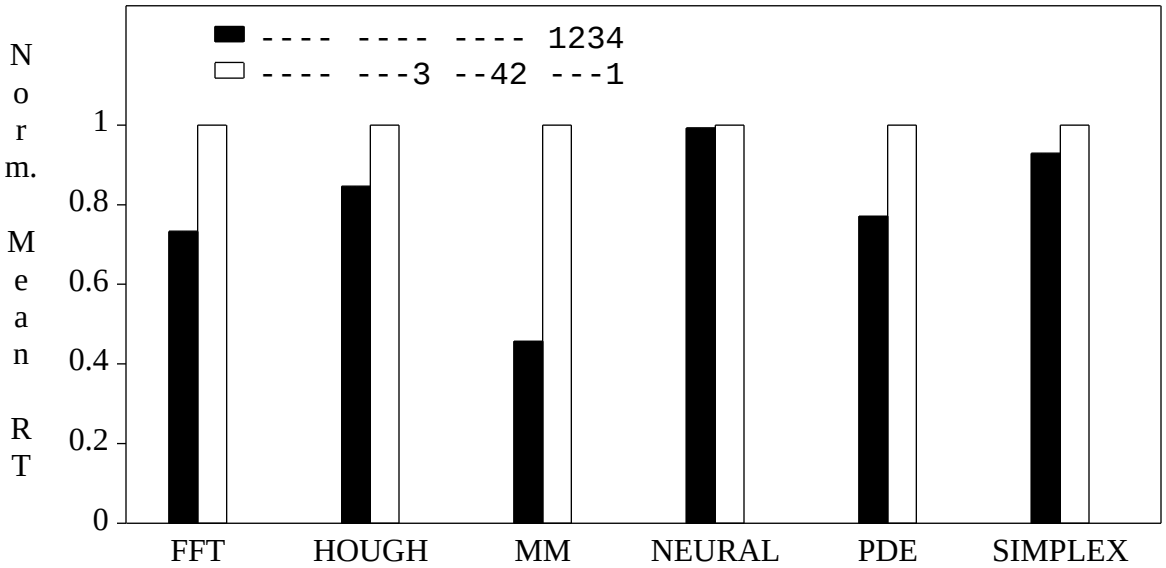


Figure 5.4: Normalized response times using localized and non-localized placements, delay = 16

5.3.2. NUMAness

Figure 5.5 illustrates the effects that the NUMAness of the system has on the execution of these applications under non-localized and localized placements. The graphs show the normalized execution times of each application obtained with delay settings of 0, 4, 8, 16, 32, and 64. The delay setting is shown just below the pair of bars representing the localized and non-localized execution times.

These graphs demonstrate that, as the latencies in the system increase, the performance benefits achieved through localization increase for all applications except NEURAL. If the communication and memory references within an application are completely localized and a localized placement is used, then the execution times should not be affected when the latencies are increased. The results show this to be true for MM and PDE. Note however, that this is not the case for FFT, HOUGH, SIMPLEX, and especially NEURAL.

There are a number of reasons why, under a localized placement, some of these applications are more affected by increased latencies than others.

1) The data being accessed is not entirely localized. That is, some of the data is located on memory associated with processors outside of those the application is executing on. This is true, for example, of FFT. To improve the performance of the calculations, a table of sine and cosine values is pre-computed and stored for later use. For each of these tables, the memory allocation scheme used assigns pages to physical memory on a round-robin basis.

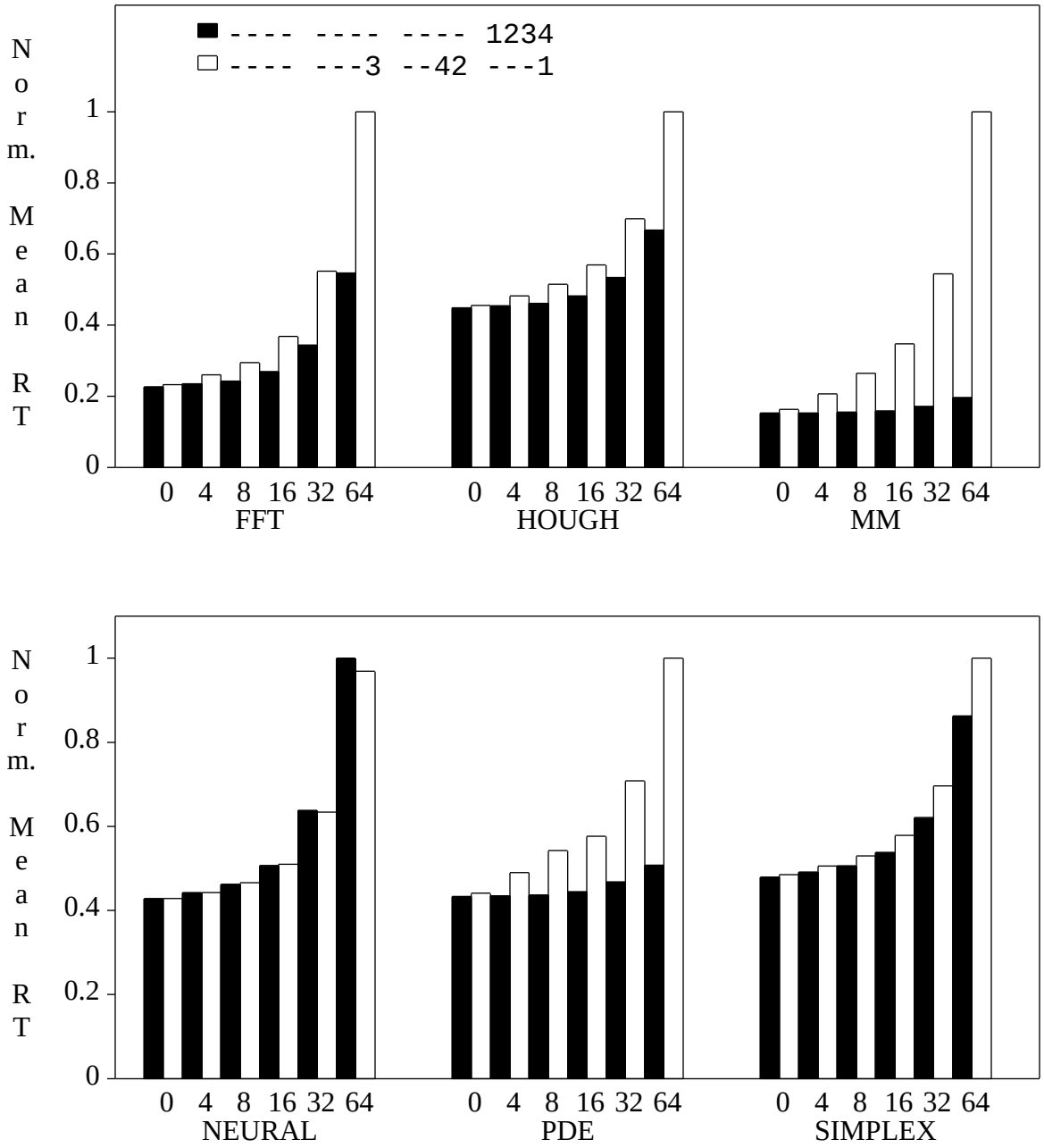


Figure 5.5: The importance of localization using different degrees of NUMAness

In this case, because 64 bit double precision variables are used with a problem size of 256, two 4 Kbyte pages will be allocated for each table and references to these tables may require remote memory accesses. The round-robin allocation of these tables was done by the original author in order to achieve good performance by reducing hot spots when using a large number of processors. For the same reasons HOUGH also uses pre-computed lookup tables for sine and cosine values which, along with the input image, are allocated using a round-robin page allocation policy. A more localized solution, such as replicating

the sine and cosine tables, was not required however, because of the mild NUMA characteristics of the system the application was originally written for.

2) There are a relatively high number of system calls.

   a) Some system calls are performed by communicating, via message passing, with a server process that is executing on Station 0, thus incurring delays because of remote communication. This is reflected in increases in execution time as the delays increase because communication with Station 0 requires using the interconnection ring which means incurring the delays. This is the case with the SIMPLEX application.

   b) Some system calls are handled by a server process that is migrated to the processor of the calling process (handoff-scheduling). The server may then access system data structures, many of which have been allocated on Station 0, thus requiring off-station memory requests which increase execution times as latencies are increased. The application NEURAL performs a large number of such system calls, which is the reason that the performance is not improved by using a localized placement. In fact, localization actually degrades performance in this case because the algorithm used executes synchronously, with each of the processes requiring access to shared resources at the same time. The non-localized placement decreases the degree to which the processes are synchronized and decreases the contention for shared resources, thus slightly improving the execution times. (In this case most of these system calls are done in order to reset the state of the software cache consistency mechanisms.)

These examples illustrate the need to consider locality during all phases of program creation and execution (especially on the part of the application programmer and operating system).

5.3.3. System Size

Another way to view the importance of application placement is to consider increases in system size and the different possible application placements, given a fixed number of required processors. For example, if an application requires four processors and it is executed on a system with four processors, a localization strategy is not required since any placement is localized. The potential benefits of localization increase in an eight processor system but are not as large as the benefits that can be obtained in much larger systems. That is, if the number of processors allocated to an application is fixed and different sized systems are considered, the importance of localization increases with the size of the system. This is illustrated in Figure 5.6.

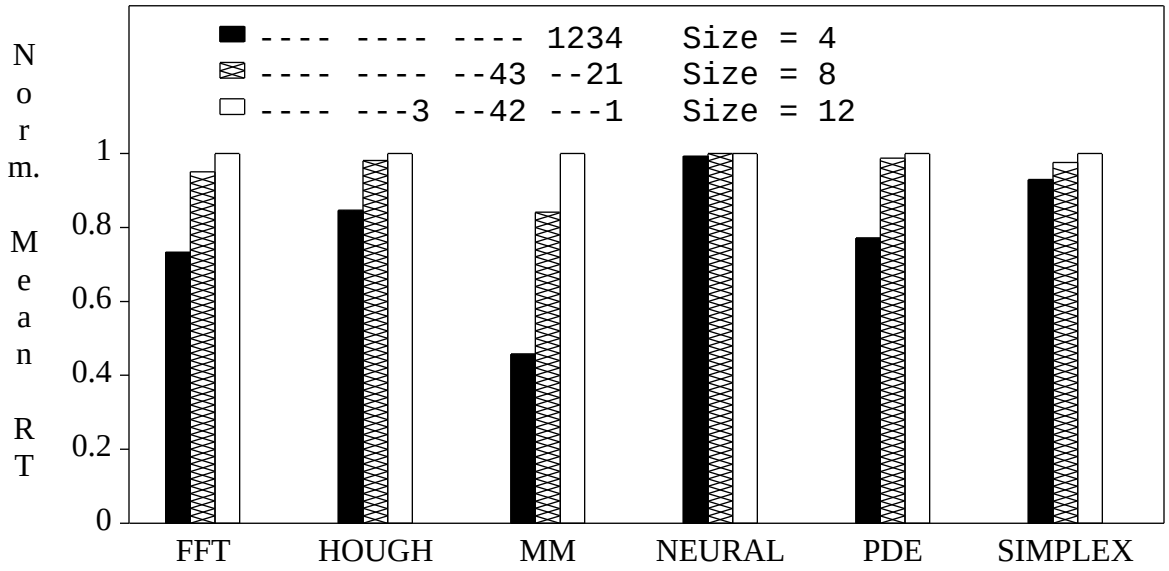


Figure 5.6: The importance of localization with varying system sizes, delay = 16

The different placements used correspond to considering localized versus non-localized placements in systems of 4, 8, and 12 processors. The localized placement, ‘‘−−−− −−−− −−−− 1234’’ is the same for each system size, while the placement ‘‘−−−− −−−− −−43 −−21’’ represents a non-localized placement in a system of eight processors because only the eight processors of Stations 2 and 3 are considered and the placement ‘‘−−−− −−−3 −−42 −−−1’’ represents a non-localized placement in a system of 12 processors. Therefore, each of the bars of the graphs in Figure 5.6 represent a non-localized placement in systems of size 4, 8, and 12 (with four processors, localized and non-localized placements of four processes are identical) and should each be compared with the localized placement ‘‘−−−− −−−− −−−− 1234’’ to determine the improvements possible due to localization for a system of that size. We expect that interference from other applications would increase the performance differences in larger systems.

The results of these experiments demonstrate that, for all applications except NEURAL, the benefits obtained from using a localized placement increase as the size of the system is increased, thus demonstrating the need for and increased importance of localization in larger and larger systems. Note also that the prototype system being used is relatively small and as a result the performance of each placement is also affected by the number of processors being used by the application (four). This can be seen by the small difference between the non-localized placements of four processes on two stations (eight processors) and three stations (12 processors). We expect that if we could continue to increase the size of the system, the difference in performance between the non-localized placements would also continue to increase.

5.3.4. Placement of Processes within the Application

All of the applications used in these experiments consist of a parent (main) process and three children. In each application, the parent process creates the children, notifies them of the functions they are to perform along with the sub-section of the data the functions are to be performed on, and controls the synchronization. The parent and its children each perform the same amount of computation on different subsets of the problem. However, because the parent process is created first, it may be responsible for initializing some of the data causing that data to be located on the same processor/memory pair as the parent process. This may be as innocuous as a few variables; for example, the number of processes used and the size of the problem. But if these variables are not cached and are referenced often, the cumulative cost of the remote memory accesses can increase execution times. Figure 5.7 shows the results of an experiment that was conducted in order to study how the execution time of each application is affected by the location of the child processes relative to the parent. This study can be thought of as addressing the following question:

- Once a localized subset of processors has been chosen for an application's execution, is the execution time affected by the location of its parallel processes within that subset of processors?

Intuitively this will depend on the symmetry, communication, and remote memory access patterns of the application.

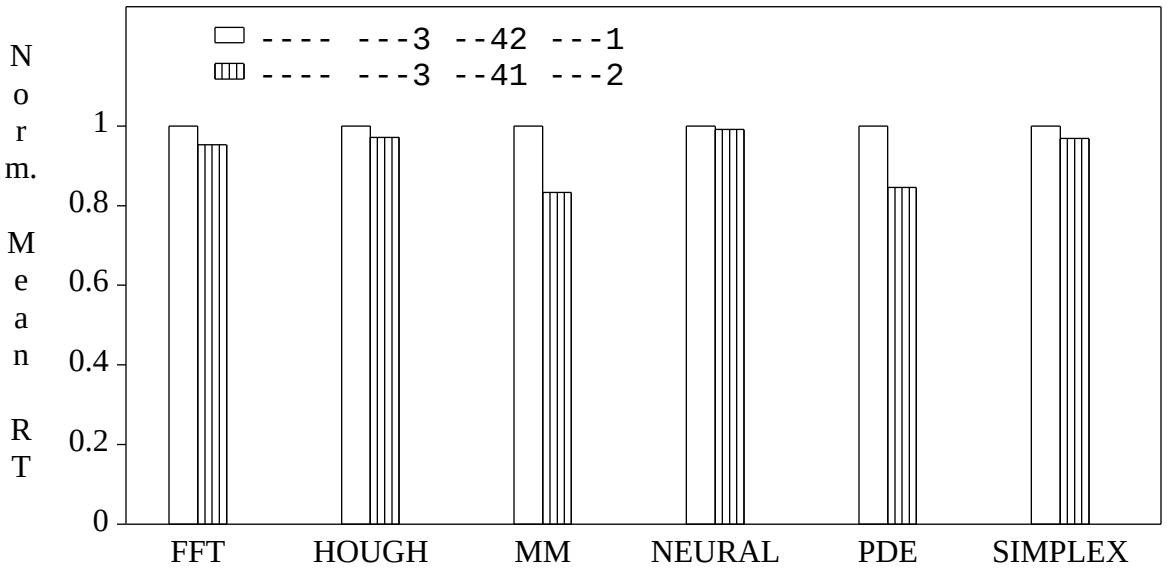


Figure 5.7: Importance of the placement of children relative to the parent process.

The experiment performed considers two non-localized placements, one in which none of the child processes are placed in the same station as the parent ‘‘−−−− −−−3 −−42 −−−1’’ and one in which one of the child processes is placed in the same station as the parent ‘‘−−−− −−−3 −−41 −−−2’’. We see in Figure 5.7 that the performance of each application is affected by the placement of the child processes relative to the parent since exactly the same subset of processors is used in each case and the only difference is that the location of processes 1 (the parent) and 2 (the first child) have been switched. A delay setting of 16 cycles was used. The results show that in two cases, MM and PDE, the execution times differ by 15%. These results are significant enough to indicate that the location of parallel processes of an application within a subset of localized processors is important for some applications and that the child processes should be placed as close to the main process as possible. It is expected, however, that these differences could be reduced by carefully rewriting the applications so that the children less frequently access data located near the parent process.

We suspect that some of the observed differences in execution times are caused by memory contention. Such differences are unlikely to be as large in other architectures, especially if memory is not as contented for as it can be in Hector. Therefore, the importance of the placement of the parent, relative to the other processes, may be reduced in other systems (particularly in the KSR-1, which uses an all-cache memory design).

5.4. Summary

In this chapter we have demonstrated that in large-scale NUMA multiprocessors, preserving the locality of applications by placing processes close to the data they are accessing and close to each other is essential to achieving good performance. In particular the experiments conducted in this chapter have shown:

- As expected, in small-scale mildly NUMA multiprocessors, placement decisions have only a minor influence on the execution time of parallel applications.

- In large-scale systems application placement that considers the architectural grouping of processor and memory modules inherent in NUMA multiprocessors is imperative and improves performance significantly.

- The importance of placement decisions increases with the size and NUMAness of the system and will continue to increase as the gap between processor speeds and memory access times continues to widen.

- In Hector, the placement of the children relative to the parent (main) process affects application performance significantly. Specifically, frequently referenced data is often located on or near the processor that the parent is placed on. Thus placing children as close as possible to the parent reduces execution time.

- Although placement decisions made by the scheduler significantly influence the execution time of parallel applications, localization must also be considered by the application programmer, the memory manager, and other operating system services.

These results motivate the work in Chapter 6, in which a new scheduling strategy for enforcing locality is proposed and evaluated.

Chapter 6

Processor Pool-Based Scheduling

6.1. Introduction and Motivation

The experiments conducted in Chapter 5 motivate the need for new scheduling techniques that consider the placement of an application when making scheduling decisions. In this chapter we propose and experimentally demonstrate the performance benefits scheduling algorithms based on the concept of processor pools.

Processor pools are intended to specifically address the following requirements of scheduling techniques for large-scale NUMA multiprocessors:

Localization

Processes of the same application need to be placed close to each other in order to minimize overhead due to remote communication. That is, the locality of the application must be preserved.

Isolation

When possible, different applications should be placed in different portions of the system in order to reduce contention for shared resources.

Adaptability

The system should be able to adapt to varied and changing demands. A large-scale multiprocessor should support the execution of a single highly parallel application that is capable of utilizing all of the processors as well as a number of applications each executing on a small number of processors.

Scalability

A pervasive requirement of all software designed for scalable architectures is that the software also scales.

6.2. Processor Pools

A *processor pool* is a software construct for organizing and managing a large number of processors by dividing them into groups called pools. Since the goal of localization is to place processes of the same application in a manner in which the costs of memory references are minimized, it implies that the architectural clusters inherent in NUMA multiprocessors must be considered when forming pools. The locality of applications is preserved by choosing pools to match the clusters of the system and executing the parallel processes of an application within a single pool (and thus a cluster), unless there are performance advantages for it to span multiple pools. Isolation is enforced by allocating different applications to different pools, thus executing applications within separate sub-systems and keeping unnecessary traffic off of higher levels of the interconnection network. Note, that it is possible for several applications to share one pool.

In very large systems (with 100 or more processors), processor pools can be grouped together to form ‘‘pools of pools’’. These ‘‘pools of pools’’ are chosen and managed in the same way as the original smaller pools except that they are constructed and managed in a hierarchical fashion. This hierarchical structuring technique is called Hierarchical Symmetric Multiprocessing and has been used by Unrau, Stumm, and Krieger to structure operating systems for scalability [Unrau1992] [Unrau1993].

In contrast to hardware groups or clusters of processors, processor pools are used as an operating system construct for scheduling applications and are applicable even in systems with no apparent processor clusters (even though the scalability of such systems is limited). Because each pool of processors may be managed independently, the likelihood of bottlenecks inherent in traditional single ready-queue systems is reduced. The notion of grouping processors to enhance scalability has also been proposed by other researchers [Black1990] [Feitelson1990] [Feitelson1990a] [Ahmad1991]. Feitelson and Rudolph’s distributed hierarchical technique is designed to also gang-schedule and load balance multiple applications in large multiprocessor systems [Feitelson1990] [Feitelson1990a]. Their evaluation of this technique does not take into account NUMA multiprocessors. They do point out that this technique could be used in NUMA systems. However, they do not describe how to map their tree structured distributed hierarchy onto NUMA architectures, although in a symmetric tree structured architecture the mapping is direct and should preserve locality. One advantage offered by processor pools is that they are explicitly designed to preserve the locality of parallel applications in a fashion that is not tied to a particular architecture. Furthermore, they are also designed to isolate the execution of multiple applications from one another. The combination of these two properties is intended to reduce the cost of remote references to shared data and to reduce the likelihood of contention for the interconnection network.

Although the concept of a processor pool is based on the large-scale and NUMA characteristics of a system and is not tied to any particular architecture, in actually implementing processor pools on a specific system, its NUMA characteristics should be identified and fully exploited. In most architectures, there are clusters of processors that are good candidates for pools. For example, in a large-scale KSR1 system [Burkhardt1992] containing a number of RING:0 subsystems connected together with a RING:1 at the higher level, pools may be formed by grouping together the processors in each of the RING:0's. In the case of the University of Toronto's Hector system [Vranesic1991] and Stanford's DASH [Lenoski1992], pools may be formed by grouping the processors of an individual station or cluster. In the MIT Alewife multiprocessor [Agarwal1991], pools might be chosen to consist of the four processors forming the smallest component of the mesh interconnect or a slightly larger set of nearest neighbour processors.

Since the concept of processor pools is proposed to help simplify the placement problem, processors are grouped together so that the main location decision to be made is which pool to place the process in. The decision of which processor within the pool to use, the *in-pool* scheduling decision, is one that can be made by another level of software that handles scheduling within the pool. For our purposes we consider processors within a pool to be indistinguishable, thus simplifying the task of in-pool scheduling. Therefore, we focus our attention on the problem of determining which of the pools to place a process in.

In this chapter we examine two central issues related to processor pools: How are they formed and how are they used?

1) How are processor pools formed? What influences which processors should belong to which pool? How closely should pools fit the architecture of the system?

2) How are processor pools used? That is, once we know more about how pools should be formed, we require policies for using them.

6.3. Assumptions

Before discussing the issues involved in forming and using processor pools, we first outline some of the assumptions about the environment we are considering and the restrictions imposed by the system being used.

- The parallelism of the application and the number of processors it is allocated are not known *a priori*.

Note that if the parallelism and number of processors allocated to an application are known *a priori* and remain fixed, then the problem of assigning applications to pools is similar to the bin packing problem [Garey1979].

The environment being considered is a dynamic one in which applications create processes as they are needed. The scheduling server executes outside of the kernel's address space. Therefore, when an application wishes to create a process and thus gain access to another processor, the library call to create a process first contacts the scheduling server. The scheduling server determines if the calling application should be allocated another processor and if so which processor it should be allocated. If the scheduler decides to allocate another processor to the application, the system call is then passed on to the kernel, which creates the process on the processor specified by the scheduler. When a process finishes executing or is killed by an exception, the scheduler is notified and updates its internal state. The scheduler, therefore, sees requests for processors one at a time and assigns them to applications until all processors have been allocated, at which point requests to create additional processes fail. All of our applications and workloads have been written to execute in this fashion.

- Process migration is not permitted.

We simplify the larger problem of scheduling in a NUMA environment by not considering process migration. This is done for a number of reasons:

1) We reduce the parameter space of the problem by eliminating the possibility of process migration, thus permitting us to study effective methods for application placement.
2) In the experimental environment we use, the cost associated with migrating a process is quite high because a simple software cache consistency scheme is used. The migration of a process would require either many uncached data references or cache flushes.
3) The cost of process migration can vary greatly from system to system and depends on factors such as memory access latencies and the methods used to maintain cache consistency.
4) We are not aware of any existing studies that conclusively demonstrate that the benefits due to migration are significant.

6.4. Forming Processor Pools

The question of how to choose the groupings of processors that form processor pools is one that is influenced by two main factors:

1) How many processors should belong to each pool?

2) What is the relationship between the system architecture and processor pools? That is, which processors should belong to which pool?

The details of specific policies for assigning processors to pools are considered in section 6.5. For now we assign newly arriving jobs to the pool with the largest number of available processors. Other processes of the job are placed within the same pool if possible. If there are no available processors in that pool then some other pool with the largest number of available processors is chosen. This algorithm was devised using the results of the experiments in Chapter 5 and is designed to isolate the execution of different jobs and to allow them ‘‘room to grow’’.

We now conduct a series of experiments designed to further explore the influences on the choice of processor pools. Although we do not exclude the possibility of choosing processor pools of different sizes, this work only considers pools of equal sizes. The goal here is to gain insight into the forming of pools, the design of policies for their use, and the benefits of processor pool-based scheduling, leaving further exploration of their use to future research.

6.4.1. Experimental Setting

The workload is comprised of five of the parallel application kernels described in Chapter 2 and used for the experiments in Chapter 5: FFT, HOUGH, MM, NEURAL, and PDE. The problem sizes were chosen so that each application used executes with approximately the same execution time using four processors. This was done to ensure that placement decisions must be made by the scheduler throughout the lifetime of the workload rather than just at the beginning of its execution. The SIMPLEX application was not used in most of the experiments in this chapter because its execution time on four processors using a reasonable sized data set was long enough to significantly increase the execution time of the entire workload, making multiple executions in order to attain confidence intervals difficult. (Smaller data sets were not large enough to be executed in parallel effectively.)

The workload consists of a number of ‘‘streams’’ of parallel jobs. A stream is formed by repeatedly executing the five applications, one after another. Since each stream is implemented using a shell script, there are unpredictable but small delays between the completion of one application and the start of the next. The delays are small enough that they do not significantly

affect the results. Each stream contains a different ordering of the five applications and all streams are started at the same time. The number of streams is adjusted to determine the multiprogramming level. The number of streams used is determined by dividing the number of processors (12) by the parallelism of the applications. In the first experiment, each stream consists of 15 repetitions of the five applications for a total of 75 jobs per stream. The applications are each allocated four processors, so three streams are used and the entire workload consists of 225 jobs.

6.4.2. Determining Processor Pool Sizes

The first experiment is designed to determine if processor pool-based scheduling improves performance and, if it does, to examine appropriate pool sizes. This is done by varying the pool size while executing the same workload. Using 12 processors we examine: 1 pool of 12, 2 pools of 6, 3 pools of 4, and 6 pools of 2 processors. We also consider 12 pools each containing one processor. Note that one pool of size 12 is comparable to not using processor pools and is equivalent to using a central ready queue from which idle processors grab processes. (This is implemented by placing processes randomly within the pool.) Because no grouping of pools is done to form ‘‘pools of pools’’, 12 pools of one processor is also equivalent to not using pools (with the exception that the overheads of managing 12 pools, although not significant, are present). (Although applications are permitted to span more than one pool and multiple jobs may execute within a single pool, our implementation of pool-based scheduling avoids these situations whenever possible.)

Pools are chosen to correspond to the hardware stations in Hector. This means that when pools of size two are used, each of the three stations used contains two pools, and when pools of size four are used, they exactly correspond to hardware stations. When two pools of size six are used, Pool 1 contains four processors from Station 1 and two from Station 2, while Pool 2 contains four processors from Station 3 and two from Station 2.

Because the system is relatively small and is only mildly NUMA, and because we are interested in how increases in NUMAness affect the results, we also run each set of experiments with delay settings of 0, 8, and 16 cycles. The results of these experiments can be seen in Figure 6.1. Each line in a graph plots the mean response time versus the pool size. Graphs are shown for each of the different applications with the graph labelled ‘‘COMBINED’’ representing the mean response times over all jobs. The vertical bars represent 90 percent confidence intervals.

We first note that, as expected in a small system that is only mildly NUMA (represented by the graphs labelled Delay=0), the mean response times are not significantly improved by using processor pools. However, as the NUMAness of the system increases, the performance improvements due to pool-based scheduling increase and are substantial when using a delay of

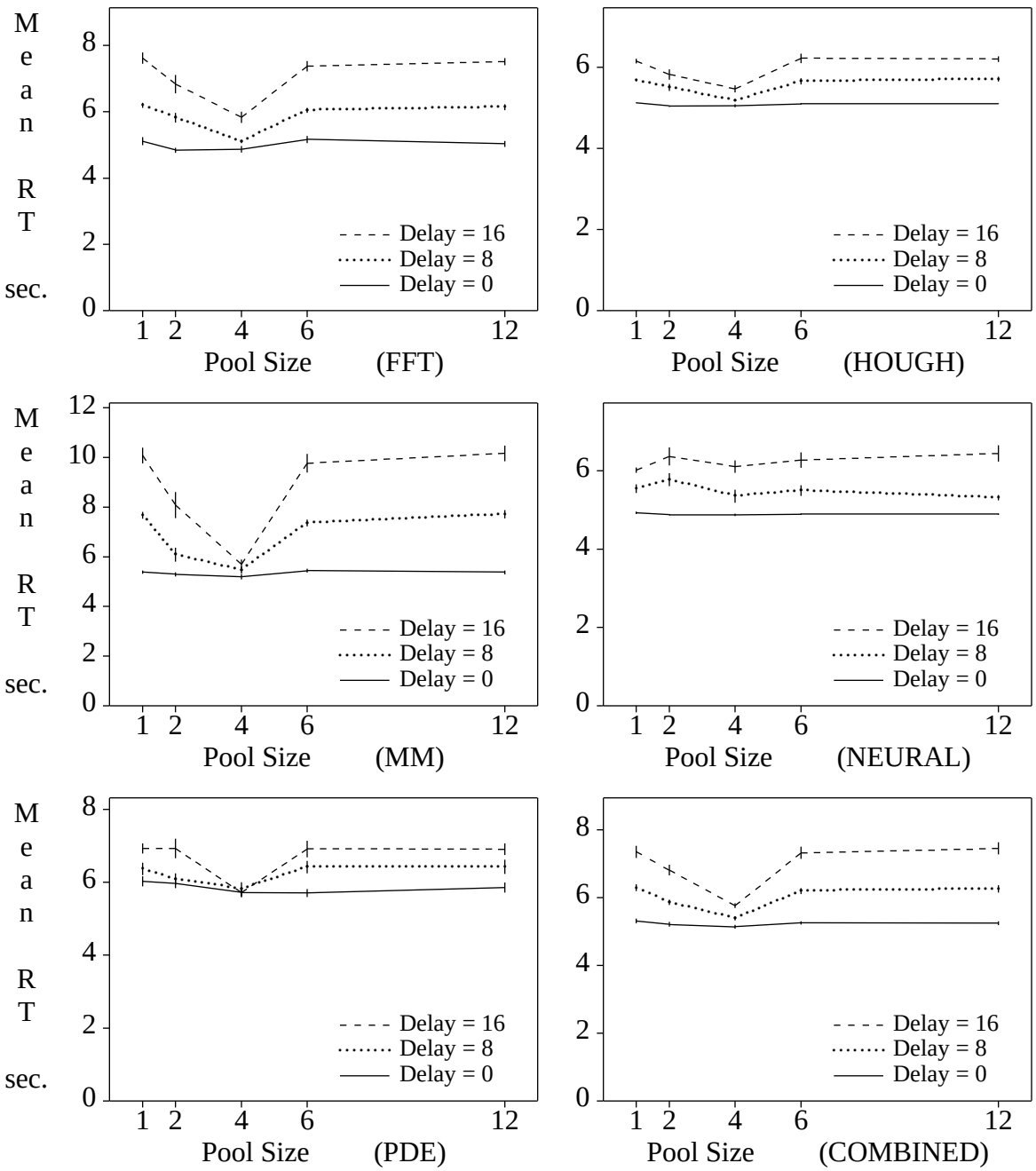


Figure 6.1: Effects of NUMAness when using pool-based scheduling

16 cycles. This can be seen by comparing the execution times of the applications using a pool of size of 12 (the no pool case), with those using other pool sizes. The closer the pool size is to 4 the better the performance. (We explore the influences that the architecture and the parallelism of the applications have on the preferred pool size is subsequent sections.) The exception is the NEURAL application which, as described in Chapter 5, suffers from an excessive number of system calls which overshadow the locality in the application.

Although two pools of six processors may not seem appropriate for the current workload, it is included for completeness, since it will play a central role in a future experiment. It also permits us to determine if a small enforcement of localization improves performance. The results show that even though there is a trend toward improved performance when using two pools compared with using no pools, those improvements are not large enough to be considered significant.

The degree to which performance is improved varies from application to application and depends on the number and frequency of remote memory references. However, the mean response time over all jobs is improved by using pools, as shown in the graph labelled ‘‘COMBINED’’ in Figure 6.1.

The graphs in Figure 6.1 also show that for this set of experiments a pool size of four yields the best performance. However:

1) The parallelism of each application is four.

2) Each station in the Hector system contains four processors.

Consequently, we next explore the importance of these two factors; application parallelism and system architecture.

6.4.3. Application Influences on Pool Size

In order to examine the importance of application parallelism in determining an appropriate pool size, we now vary the parallelism of the applications and perform the same experiments conducted in the previous section. A delay of 16 cycles is used and the number of streams is adjusted with the parallelism of the applications (when possible keeping all processors busy). Figure 6.2 shows the results obtained when executing each application with two, four, and eight processes. In the case when the application parallelism is two, six streams are used, each consisting of 10 repetitions, for a total of 300 jobs. When the application parallelism is four, three streams are used, each consisting of 15 repetitions, for a total of 225 jobs. In the case when the application parallelism is eight, one stream with 25 repetitions is used for a total of 125 jobs. (In this case applications are not multiprogrammed because we can not space-share two applications each using eight processors on 12 processors.) The three lines plotted in each graph represent the mean response times of the applications obtained with application parallelism of two, four, and eight, versus different pool sizes. The vertical bars at each data point represent 90 percent confidence intervals.

We first observe that when eight processes are used for each application, performance is not significantly affected by the pool size. This is because the placement of eight processes within a 12-processor system does not afford as much room for localization as applications which use a

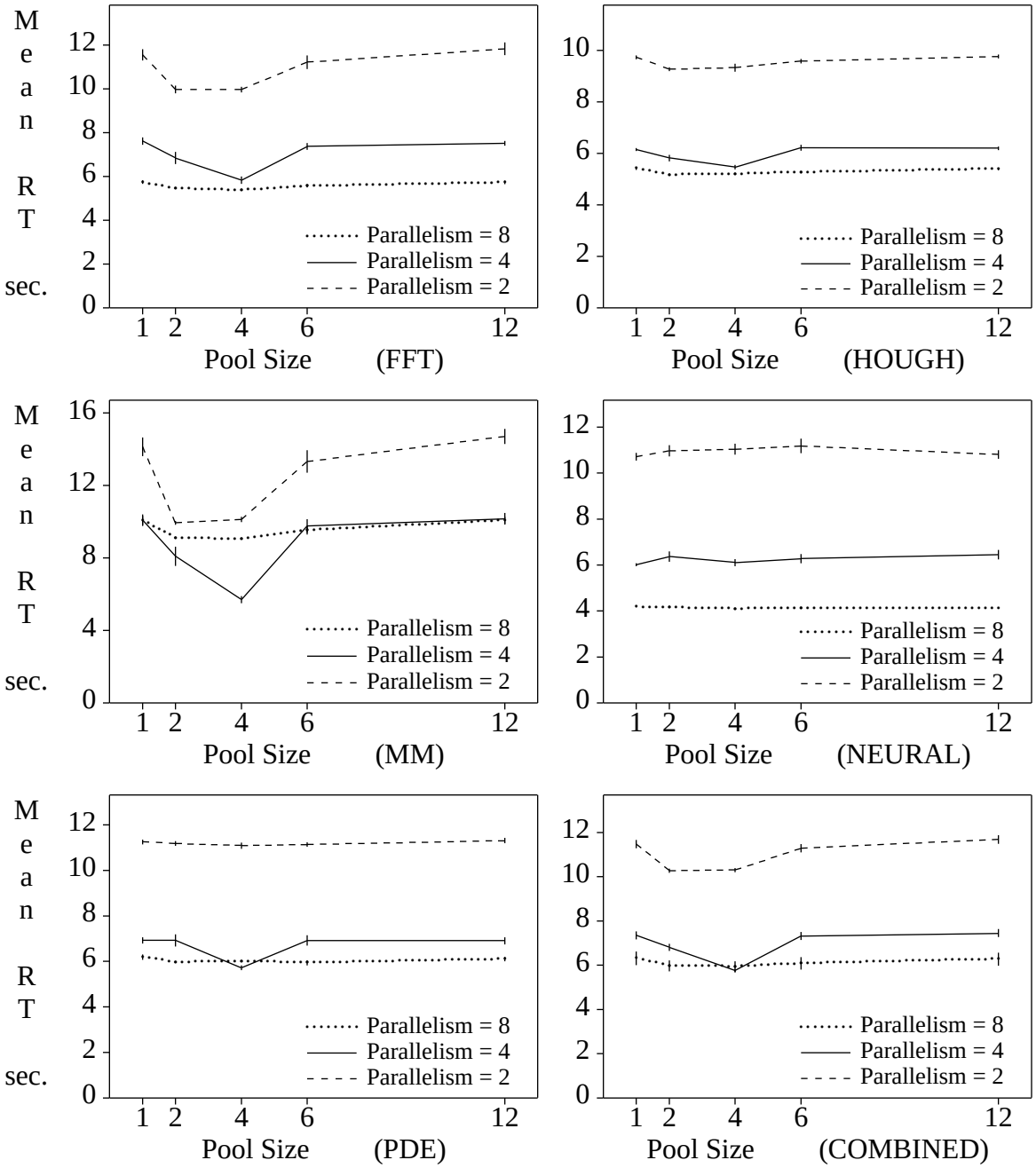


Figure 6.2: Various pool sizes with application parallelism of 2, 4 and 8, delay = 16

smaller number of processors. Next, we observe that when applications are allocated two processors, pools of size two and four yield the best performance, again with the exception of the NEURAL application. When the applications each require two processors, there is no significant difference in performance between using pools of size two or four because in either case each application is able to execute within one hardware station. Finally, we observe that when the application parallelism is four, the mean response time is minimized when pools of size four are used. These results all suggest that the appropriate choice of pool size might be

related to the parallelism of the jobs. This relationship is explored after we first point out that there is an interdependence between the number of processors an application should be allocated (the allocation problem) and which of the available processors an application should execute on (the placement problem).

An interesting outcome of the experiments shown in Figure 6.2 is that for some applications, most notably MM, increasing the number of processors the application uses does not necessarily improve response time. For example, this can be seen in the MM case by observing that the mean response time obtained using eight processors is equal to or higher than the mean response time obtained using four processors, no matter what pool size is used. These graphs demonstrate that an application's execution time can be dramatically affected by the NUMA environment and that in some cases a localized execution using fewer processors will outperform a necessarily less localized execution using more processors. Thus the decision regarding the number of processors an application is allocated should not be made in isolation. In fact, the allocation of processors should be determined in conjunction with the knowledge about which processors are available, thus greatly increasing the complexity of the problem.

One approach to this problem would be to determine the number of processors to allocate to each application based on which processors are available. Good solutions to this problem would require knowledge about the efficiency of execution of each application, when executed on different subsets of the available processors. For example, a family of execution profiles or speedup curves in which the application is executed on all possible subsets of processors (or unique subsets if the system is symmetric) could be used to make improved allocation and placement decisions. However, the effort required to collect such information and the amount of data available as a result is substantially greater than in a UMA environment and is likely prohibitive.

One simplified approach to this problem is to limit the number of pools that each job is permitted to span. If the available processors are situated in a small number of pools the application may be allocated and execute on that localized set of processors. However, if the available processors are spread throughout a number of pools, the number of processors allocated to the job is limited and the application will be executed on a smaller number of processors. This approach could work reasonably well in large systems executing a workload with a relatively high degree of multiprogramming, comprised of jobs that are not capable of executing efficiently on a large number of processors, because the benefits of giving any one job a large portion of the processors is limited. Unfortunately, one of the drawbacks of this approach is that applications would all be treated equally, and as we've pointed out previously, applications do not all have the same characteristics.

If more information about each application could be made available (or ideally, determined during execution) the approach of restricting the number of pools an application is permitted to span could be done on a per application basis. For example, some measure of the ‘‘importance of locality’’ for the efficient execution of each application could be used to determine the number of pools that each application is permitted to span.

Because of the complexity of this interplay between the number of processors allocated and which processors are available, the relationship between the allocation problem and the placement problem is demonstrated and possible approaches to dealing with the interdependence of these problems are presented. However, a more detailed investigation of this problem is left as a topic of future research.

6.4.4. Architectural Influences on Pool Size

While the experiments shown in Figure 6.2 suggest that there is a relationship between pool size and application parallelism, these experiments do not fully explore the relationship between pool size and the system architecture. To determine the strength of the connection between pool size and system architecture, we conduct another experiment in which each application executes using six processors. In this experiment the HOUGH, MM and SIMPLEX applications were used. The other applications (FFT, NEURAL, and PDE) are not used because some are written to execute using a number of processors that is a power of two, and others a number of processors that evenly divides the size of the data set. In these experiments, we use two streams, each of which executes the three applications 15 times, for 45 jobs per stream and a total of 90 jobs.

The graphs in Figure 6.3 plot the mean response times for each of the applications versus different pool sizes. The mean response times over all of the applications is shown in the graph labelled ‘‘COMBINED’’. The number above each of the bars gives the percentage improvement when compared with one pool of size 12. A negative value indicates that the mean response time was increased. The main data points of interest in these experiments are the pools of size four, because this matches the size of a Hector station, and pools of size six, because this matches the application parallelism. For the HOUGH and SIMPLEX applications, we observe slight differences in mean response times when pools of size four and six are used. (The differences are not statistically significant.) A pronounced difference is observed for the MM application. This is somewhat surprising since exactly the same set of processors is assigned to each application in each case. The differences in mean response times is, however, due to the placement of processes within the pools.

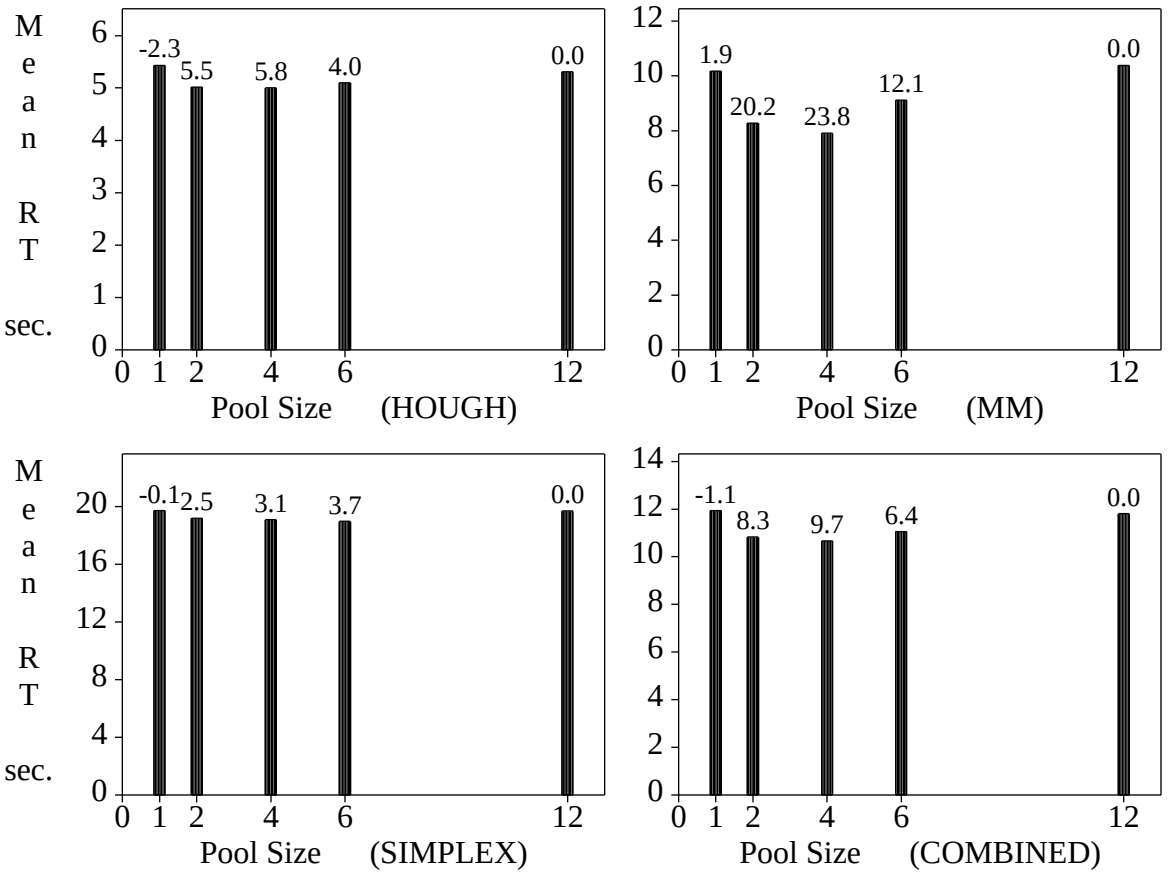


Figure 6.3: Various pool sizes with application parallelism of 6, delay = 16

First, we briefly review the pool placement policy in order to understand why the placements are different. Then we explain why the resulting execution times are different. The first process of each job is placed in the pool with the largest number of available processors. Subsequent processes of that job are placed in the same pool as the first process until all the processors in the pool are used. If more processors are required, the next pool with the most available processors is chosen. One of the goals in the design of processor pools is to form groups of processors that can be managed easily and uniformly within the pool. We, therefore, place processes randomly within pools and point out that if placement within processor pools affects performance significantly the pools have not been chosen to appropriately reflect the architecture.

Figure 6.4 illustrates a possible scenario in which the response times of the same applications would differ using the pool sizes four and six. Figure 6.4a shows an example placement of two applications, *A* and *B*, when using pools of size four. In this case the first four processes of application *A* are placed randomly on Station 1 and the remaining processes of application *A* are placed on Station 2. The first four processes of application *B* are placed

randomly on Station 3, since that is the pool with the largest number of available processors, and the remaining two processes are placed on Station 2.

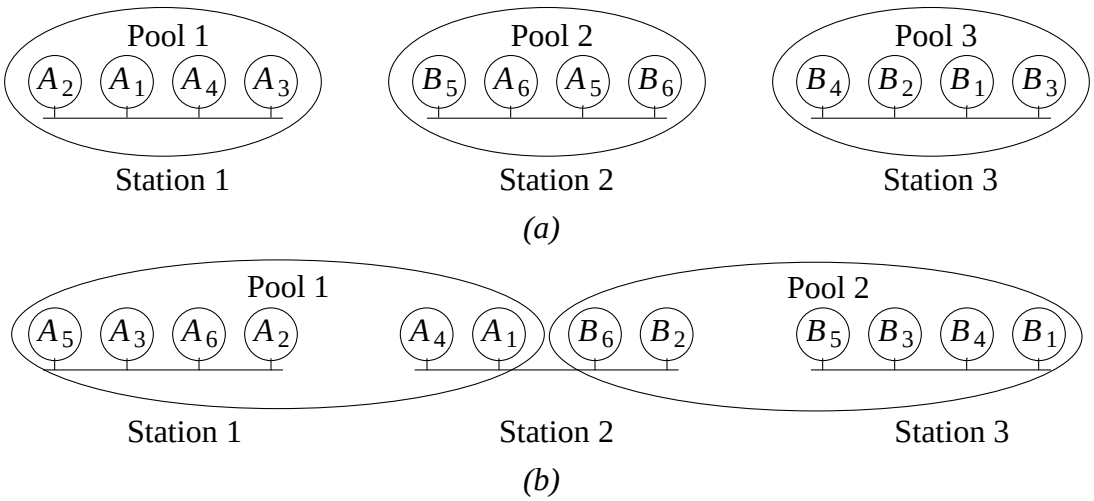


Figure 6.4: Individual process placement using processor pools of size 4 and 6

Figure 6.4b shows an example placement when pools of size six are used. In this case each application fits entirely within a single pool. Application *A* is placed and executed in Pool 1 and application *B* is placed and executed in Pool 2. In Chapter 5 we observed that if the first process of an application (the parent) is located on a station that is different from the rest of the processes of the application, the response time can be affected significantly. Note that in the case when pools of size four are used, as many children as possible will be placed in the same pool as the parent. However, our experiments will soon demonstrate that the same is not true when the pool size is six.

Because processes are placed randomly within the pool and because these pools ‘‘span’’ more than one hardware station, if the first process of an application is placed on Station 2, the response time may be higher than if the first process were placed on one of the other stations. We further illustrate this effect by conducting the same experiment using pools of size six but ensuring that the first process of each application is not placed on Station 2. In this experiment, we place processors within each pool as shown in Figure 6.5. The mean response times and 90% confidence intervals for each application of this experiment are shown in Table 6.1 under the columns labelled ‘‘Pool Size 6 (ordered)’’. Table 6.1 also contains the results obtained for pools of size four and six when using random placement within each of the pools.

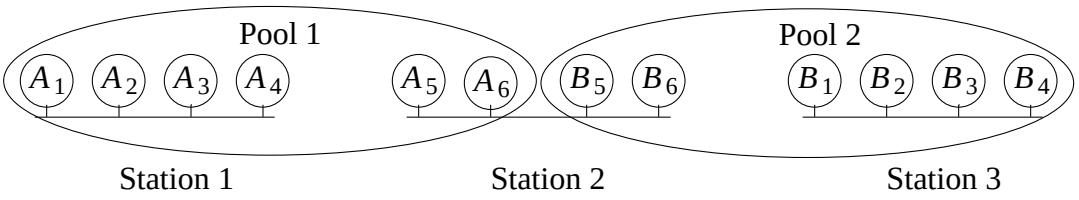


Figure 6.5: Ordered placement of processes within pools of size six

These measurements show that when pools of size six are used, ensuring that the first process of the application is not placed in Station 2 lowers the mean response time of MM significantly and in fact results in mean response times comparable to using pools of size four. The execution times of the SIMPLEX and HOUGH applications are not changed much under any of these placement strategies. This is because, as observed in Chapter 5, the location of the first process of these applications does not appear to have as strong an influence on their execution times as some of the other applications (e.g., MM).

Appl	Pool Size 4		Pool Size 6 (ordered)		Pool Size 6	
	Mean	90% CI	Mean	90% CI	Mean	90% CI
HOUGH	5.00	0.01	4.96	0.01	5.10	0.06
MM	7.91	0.02	7.97	0.02	9.12	0.36
SIMPLEX	19.09	0.05	19.05	0.09	18.97	0.11
COMBINED	10.67	1.06	10.66	1.06	11.06	1.02

Table 6.1: Comparing pools of size four with an ordered placement within pools of size six

The results of this experiment show that there is a difference between using three pools of size four and two pools of size six when allocating six processors to each application. Three pools of size four yield better performance, indicating that in this case it may be more important to choose pool sizes to reflect the architecture of the system than the parallelism of the applications. The experiments throughout this chapter have demonstrated that the selection of pools and their sizes should consider both the architecture of the system and the parallelism of the applications being executed. However, matching pools to the architecture is likely to be relatively straightforward while, in general, a workload will consist of a number of applications with different (and possibly) changing degrees of parallelism, making it difficult to match pool sizes with application parallelism.

6.5. Using Processor Pools

One motivation for processor pool-based scheduling is to ease placement decisions by reducing the number and types of considerations required to make good placement decisions. This is accomplished by making placement decisions that consider pools rather than individual processors when scheduling parallel applications. An important aspect of pool-based scheduling is the strategy used for making placement decisions. However, before examining different strategies, we first outline the types of placement decisions that are made during the lifetime of an application and briefly point out how these decisions may influence placement strategies:

Initial Placement

Before an application begins execution it must be assigned to a processor. The decision of where to place the first process of an application is an important one that can influence not only the placement of the remaining processes of the application but also the placement of other applications.

Expansion

Once a parallel application begins execution, it will, at some point, create and execute a number of processes. We call this creation of new processes *expansion*. How to properly place these processes is a key consideration in preserving the locality of an application. As a result, it is essential to consider where the existing processes of the application are located.

Repartitioning with Pools

A change in the number of processors allocated to each application may require a repartitioning of the processors. An important and difficult problem is how to repartition the processors while maintaining the locality of the executing applications.

We now examine each of these decision points more carefully, present algorithms for making these decisions and, when possible, evaluate their performance.

6.5.1. Initial Placement

The main considerations for making an initial placement decision are:

1) Give the new application as much room as possible for the future creation of processes. That is, provide as much room for expansion as possible.

2) Try to isolate the execution of each application to the extent possible. That is, try to reduce the possibility of interfering with the execution of other applications by placing each application in its own portion of the system.

The problem of placing applications into pools has similarities to the problem of allocating memory in non-paged systems. An especially notable similarity is the desire to avoid fragmentation, since fragmenting processes of an application across different pools will hurt localization. Because of these similarities, we briefly consider a number of possible strategies for initial placement adapted from well known placement policies for non-paged memory systems [Deitel1984] [Silberschatz1991].

First-Fit

Pools are listed in a predetermined order by simply numbering each pool. The first process of an application is then placed in the first pool with an available processor.

Best-Fit

The first process of an application is placed in a pool with the smallest, non-zero number of available processors.

Worst-Fit

The first process of an application is placed in a pool with the largest number of available processors.

Of these techniques the Best-Fit and the First-Fit policies do not isolate applications from each other and may not provide room for the expansion of applications within a pool. For example, if three applications arrive in an empty system, all three are initially placed in the same pool, thus leaving little room for the localization of subsequent processes. (Recall that the number of processors an application will use is not known *a priori*.) However, the Worst-Fit policy would place each of these three applications into different pools, thus permitting each to execute in their own portion of the system.

A comparison of the First-Fit and Worst-Fit policies is shown in Figure 6.6. A workload of three streams is used. Each stream consists of 15 repetitions of five applications for a total of 75 jobs per stream and a grand total of 225 jobs. Pools of size four are chosen to correspond to the hardware stations and a delay of 16 cycles is used to emulate systems that are more NUMA than our small, mildly NUMA prototype. The normalized mean response times of each of the five applications and the overall mean response time (COMBINED) are shown. The mean response times obtained using the Worst-Fit policy are normalized with respect to the mean response times obtained using the First-Fit policy.

As expected the Worst-Fit policy performs significantly better than the First-Fit policy and in fact it improves response times by 20% or more for three of the five applications. By examining the execution traces obtained when using the First-Fit policy (shown in Figure 6.7), we observe that the different applications are not always placed within one pool (and therefore

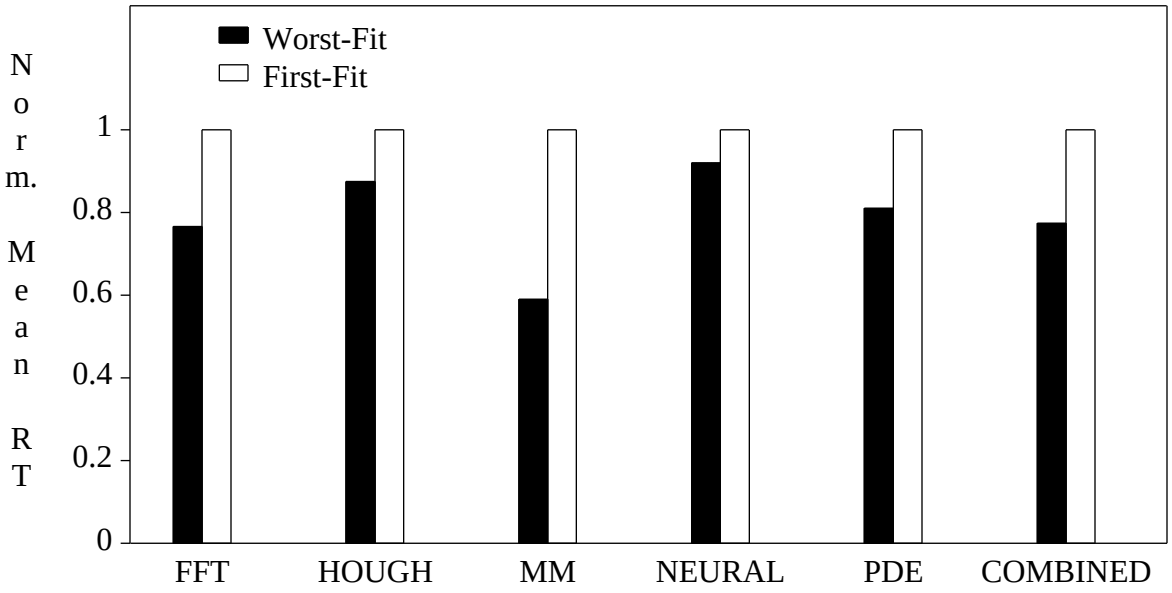


Figure 6.6: Comparing Worst-Fit with First-Fit placement strategies, pool size = 4, delay = 16

one station) and that sometimes the applications actually do execute within separate pools. In the trace in Figure 6.7, lines 12 through 17 show a period of execution where each of the applications is executing within a separate station. Each application is therefore localized and isolated from the others. Lines 23 and 24 show an example of how all three applications can each have one process executing in the same station (’a’, ’b’ and ’z’ each have a processes on the middle station). Although placements using the First-Fit policy are not always ‘‘bad placements’’, the mean response time is significantly improved by using the Worst-Fit policy.

```
(1)  ---- ---- ----    (10)   mooo mnmm nnno    (19)   xwxw yyyy wxxw
(2)  ---- ---- f---    (11)   -ooo p--- ---o    (20)   xwxw yyyy wxxw
(3)  ghhh ghgg ffff    (12)   ---- pppp q---    (21)   xwxw ---- wx--
(4)  ghhh ghgg ffff    (13)   rrrr pppp qqqq    (22)   abab zbab zzaz
(5)  ghhh ghgg iiii    (14)   rrrr ssss q---    (23)   abab zbab zzaz
(6)  k--- jjjj iiii    (15)   uuuu ssss tttt    (24)   abab -bab cca-
(7)  kkkk jjjj l---    (16)   uuuu ---- tttt    (25)   dede cede ccdc
(8)  kkk- ---- llll    (17)   uuuu vvvv tttt    (26)   dede cede ccdc
(9)  m--- mnmm nnno    (18)   xwxw vvvv wxxw
```

Figure 6.7: Sample execution trace, over time, using First-Fit initial placement policy

Other possible initial placement strategies are numerous. For example, the first process might be placed only in pools that are empty, and new applications would wait for a pool to become empty before being permitted to begin execution. Another method is a Worst-Fit policy based on the number of applications executing in a pool rather than the number of processes

executing in the pool. That is, a count of the number of jobs executing within each pool is maintained and rather than assigning new jobs to the pool containing the fewest processes, they would be assigned to the pool containing the fewest jobs. This policy may be more suited to isolating applications and providing room for expansion under certain types of workloads. However, the best policy will depend on a number of factors such as the size and NUMAness of the system, the likelihood of contention for shared resources such as the interconnection network, the applications and their parallelism and locality characteristics, the number of processors allocated to each application and the number of applications being simultaneously executed. As a result, the exploration of optimal initial placement strategies is left as a topic for future research. However, the Worst-Fit policy, based on assigning new jobs to the pool with the largest number of available processors, seems to incorporate the desirable properties of a good placement policy, such as isolating applications from each other while also providing room for their expansion.

6.5.2. Expansion

Processor pool-based scheduling strategies for supporting application expansion are relatively straightforward:

1) Place new processes as close to existing processes of the application as possible. This is accomplished by placing new processes in pools that are already occupied by the application. By doing so, processes are placed close to the shared data being accessed.

2) If there are no available processors in the pools already occupied by the application, choose new pools so there is as much room for future expansions as possible and interference with other applications is minimized. Since these requirements are the same as for initial placement, the same algorithms can be applied in both cases.

6.5.3. Repartitioning

Dynamic scheduling techniques repartition processors among applications with changes in the number of executing applications and changes in application parallelism. If the overheads incurred because of repartitioning do not outweigh the benefits obtained from improving processor utilization, dynamic scheduling will reduce the mean response time when compared with static scheduling. Recent research has shown that, in UMA multiprocessors, dynamic scheduling significantly reduced mean response time when compared with static scheduling [Zahorjan1990] [McCann1993]. The initial simulation studies conducted by Zahorjan and McCann [Zahorjan1990] also conclude that even for relatively large overheads (in UMA multiprocessors) dynamic scheduling policies produced mean response times lower than static policies.

An important question is, do dynamic scheduling policies reduce mean response time when compared with static scheduling policies in NUMA multiprocessors. Wu compares static and dynamic scheduling algorithms using policies that do not consider the placement of processes [Wu1993]. He notes that a drawback of the static approach is that new arrivals may not be started until long after their arrival. He addresses this shortcoming with a policy called Immediate Start Static that starts new arrivals almost immediately after arrival. This is done by taking a processor away from one of the executing applications (i.e., the policy is dynamic). He performs experiments using four workloads and finds that the Immediate Start Static approach outperforms the static approach by 15-40%.

Instead of performing a similar comparison, we briefly describe some of the issues involved in repartitioning processors in NUMA multiprocessors using processor pool-based scheduling. The goal, when performing dynamic scheduling in a NUMA environment, is to be able to repartition processors, while keeping the execution of each application as localized as possible.

Processors may be repartitioned under two different situations:

1) An application relinquishes a processor (e.g., when an application has no more work to perform on that processor).
2) The scheduler determines that one application should be allocated more processors at the expense of another application.

The second situation, described above, can be viewed as taking one processor away from an application, *A*, and giving that processor to another application, *B*. Note that at any instant of time such a decision will only involve two applications and will only modify the number of processors allocated to each application by one. The situation described in the first case can be regarded as a subset of the second case because a decision does not have to be made regarding which application to take a processor away from; only which application to allocate the free processor to.

We assume that the allocation mechanism within the scheduler determines how many processors each application should be allocated. The scheduler knows how many processors each application should be allocated but implements modifications to processor allocations by selecting a pair of applications, one of which will be yielding a processor, application *A*, the other of which will be allocated the yielded processor, application *B*. Therefore, larger changes in partitions are done as a series of pairwise changes. The placement decision then involves determining which of the processors to take away from application *A* in order to give that

processor to application *B*. (As pointed out previously the allocation and placement decisions should ultimately be made cooperatively.)

We further illustrate the problems associated with processor repartitioning in NUMA multiprocessors using the two examples illustrated in Figure 6.8a and 6.8b. Each figure shows an example repartitioning of processors over time. For simplicity assume that the repartitioning policy being used is the dynamic equipartition policy and that the reallocation decisions are made elsewhere. (We are therefore concerned only with placement decisions regarding the reallocation.) Applications *X* and *Y* are executing and are each allocated six processors (lines 1-2). If a new application, *Z*, arrives (at line 3), it needs to be allocated an initial processor. If it creates parallel processes, it will then be allocated more processors. So again the placement of the new application can be thought of in terms of two types of problems: initial placement and expansion.

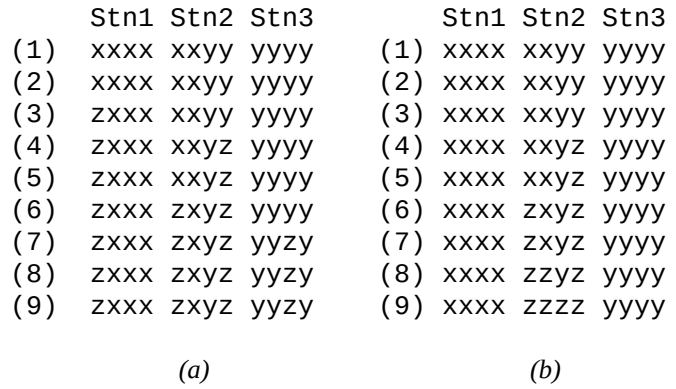


Figure 6.8: Dynamic repartitioning in NUMA multiprocessors

A delicate balance must be maintained between taking a processor from an application, *X* or *Y* and maintaining a localized set of processors for each of the applications *X*, *Y*, and *Z*. To be effective, repartitioning should be accomplished with minimal overhead and delay. However, processor reallocations should also be coordinated between the scheduler and the application, in order to avoid the pitfalls associated with uncoordinated reallocations; for example, descheduling threads that are holding locks [Zahorjan1991]. Therefore, obtaining a specific processor from an application may not occur instantaneously and in fact may not be possible at all, since there is no guarantee that the thread or threads executing on the specified processor will be able to relinquish it. As a result, techniques for repartitioning processors in NUMA systems involve a trade off between quickly repartitioning but obtaining a non-localized placement or delaying repartitions in the hope that a more localized processor will become available.

(Scheduler activations could be used to force the reallocation of processors [Anderson1991], however, the scheduler activations are likely to adversely affect locality.)

The example in Figure 6.8a shows how processors might be reallocated if the first available processor is used (case 1), while Figure 6.8b shows an example of how each reallocation might occur if the scheduler waits for a processor that results in a more localized placement for all of the applications (case 2). In the case depicted in Figure 6.8b, an initial placement of application *Z* that allows for further expansion (if necessary) while preserving the localized placements of applications *X* and *Y* places the first process of application *Z* in Station 2 (Stn2). Therefore, it takes longer for application *Z* to begin execution in case 2 than in case 1, since case 1 simply takes the first processor that becomes available.

As seen in these examples, obtaining each new processor in the second case is likely to take longer than obtaining each new processor in the first case. These delays will increase the response time of application *Z*, but will decrease the response times of applications *X* and *Y*. However, these changes in the response times depend on the costs of performing the reallocations and the relative benefits obtained from either obtaining a localized or non-localized placement of the remaining applications. The problem is further complicated if, as we have seen earlier in this chapter and in Chapter 5, the first process of the application requires and is given special consideration.

We have outlined some of the issues that must be addressed in order to effectively implement dynamic scheduling methods in NUMA multiprocessors. The main complication is the desire to localize application placements.

One simple approach is to limit the number of pools that each application is executing on (i.e., the number of pools spanned by each application). The extent to which the limit is enforced controls the extent to which localization is enforced. This approach was used in simulation studies by Zhou and Brecht, but under slightly different operating conditions for processor pools [Zhou1991].

6.6. Summary

In this chapter we have proposed algorithms for scheduling in NUMA multiprocessors based on the concept of processor pools. A processor pool is a software construct for organizing and managing processors by dividing them into groups called pools. The main reasons for using processor pools are to preserve the locality of an application's execution and to isolate the execution of multiple applications from each other. The locality of applications is preserved by executing them within a pool when possible, but permitting them to span pools if it is beneficial to their execution. Isolation is enforced by executing multiple applications in separate pools (to

the extent possible). This reduces execution times by reducing the cost of remote memory accesses. We also expect that processor pools reduce contention for the interconnection network, although we were not able to observe this on our small-scale, mildly NUMA multiprocessor. (Reducing the distance required to obtain remote memory references should reduce the use of the interconnection network.) It is expected that the scalability of the system will also be enhanced because processors within a pool can be treated equally.

We have conducted a series of experiments that explore desirable attributes of processor pool-based scheduling. In particular, we have found:

- Pool-based scheduling is an effective method for localizing application execution and reducing mean response time.
- Optimal pool size is a function of the parallelism of the applications and the system architecture. However, we believe that it is more important to choose pools to reflect the architectural clusters in the system than the parallelism of the applications. (Especially since the parallelism of an application may not be known and may change during execution.)
- The strategies of placing new applications in a pool with the largest potential for in-pool growth and of isolating applications from each other seem to be desirable properties of algorithms for using pools. The Worst-Fit policy incorporates both of these properties.

Chapter 7

Conclusions

7.1. Introduction

The goals of this dissertation are to gain insight into the factors involved in designing and implementing scheduling policies for NUMA multiprocessors and to demonstrate that application and architectural characteristics can be obtained and used to make improved scheduling decisions. In pursuit of these goals we analytically and experimentally evaluate scheduling techniques that reduce mean response time over existing methods. The main factors and issues we consider are:

- **Work:** The total amount of work to be executed by an application.
- **Efficiency:** The efficiency with which processors can be effectively utilized by an application.
- **Allocation:** How many processors should be allocated to each application?
- **Placement:** Which processors should be allocated to each application?

The exploration of these factors and their interdependence has led to the contributions outlined in the following section.

7.2. Summary of Contributions

Throughout this thesis we demonstrate how knowledge of application and architectural characteristics can be obtained and applied when making scheduling decisions. We also demonstrate that using these characteristics, or estimates of them, can reduce the mean response time of parallel applications when compared with existing scheduling techniques.

One method traditionally used for determining the number of processors to allocate to each application simply gives each application an equal portion of processors. This equipartition strategy is popular because it is easy to implement and it does not require any information about the characteristics of each application's execution. In order to gain insight into the problem of determining processor allocations, we examine the performance of the equipartition policy when applied to static and dynamic scheduling policies and analytically compare its mean response time with optimal policies under assumptions of perfect speedup.

We prove, as has Sevcik [Sevcik1994], that if jobs arrive and must be activated simultaneously, the optimal processor partitioning allocates processors in proportion to the square root of the amount of work each application executes. We also show that this technique can significantly improve mean response time when compared with a strategy that allocates processors equally.

In the case of dynamic scheduling, we show that if jobs arrive simultaneously and are scheduled using a least work first policy (which is optimal) rather than the dynamic equipartition policy, the mean response time will be reduced by at most $\frac{N+1}{2N}$, where N is the number of jobs. We also establish that for any distribution of work in which the total amount of work grows with the number of jobs, the mean response time of the optimal policy asymptotically approaches one half the mean response time of the dynamic equipartition policy as the number of jobs approaches infinity. Then we construct a simple workload that considers new job arrivals, but for which the ratio of the mean response time obtained by using the dynamic equipartition policy over that obtained using the optimal strategy asymptotically approaches zero as the number of jobs approaches infinity. Since a dynamic equipartition policy shares processing power equally among all jobs, it is not surprising that these results are analogous to results comparing round-robin and optimal strategies on uniprocessor systems.

Having demonstrated the potential for improving mean response time by using knowledge of an application's remaining work, we introduce a simple technique for estimating an application's expected remaining work. An estimate of an application's expected remaining work is provided by the run-time system to the scheduler by multiplying the number of remaining threads by the average execution time per thread. Experiments conducted using this technique show that for some applications the estimates are very accurate and when combined with a least expected work first scheduling policy, improve mean response time significantly when compared with the dynamic equipartition policy.

Then we propose a more general alternative method for estimating an application's expected remaining work. This technique is similar in approach to a popular method used in uniprocessor scheduling in that it estimates a job's expected remaining work based on its accumulated execution time. We then present a generalization of the processor allocation policies developed and used in Chapter 3. This generalization has the desirable property that it can be used to produce allocation policies that reduce the number of processors allocated to jobs that are expected to continue executing for a long time.

In considering the placement of parallel applications, we argue that existing scheduling techniques, which have been designed for UMA multiprocessors, are inadequate for the emerging class of NUMA multiprocessors because they fail to consider the extended memory hierarchy and the differences in memory access times from different processors. That is, they treat all processors equally (with the exception of those methods that consider cache context). Using a representative NUMA multiprocessor, we experimentally demonstrate the importance of parallel application placement by comparing application placements that consider the costs of remote memory accesses with placements that do not. Our results show that placement decisions in large-scale NUMA multiprocessors are critical and that localization that considers the architectural clusters found in most NUMA architectures is essential to good performance. We also demonstrate that the importance of localization increases with the size of the system and the cost of remote memory accesses.

Having motivated the need for and the increasing importance of new techniques for scheduling in NUMA multiprocessors, we propose, implement, and experimentally evaluate a technique called processor pool-based scheduling. A processor pool is a software construct for organizing and managing a large number of processors by dividing them into groups called pools. Our experiments with this technique demonstrate the importance of choosing processor pools to reflect the architecture of the system. They also show that performance can be significantly improved when compared with techniques that treat all processors equally. We propose the use of a WORST-FIT initial placement strategy when assigning applications to pools. This policy not only localizes application placement when possible but also isolates the execution of different applications, permitting room for expansion of applications and decreasing contention for shared resources (e.g., the interconnection network).

The contributions of this dissertation will ultimately depend on the extent to which the insights and techniques discussed are adopted and used in other systems. Processor pool-based scheduling is a technique for coping with a very real problem in current large-scale NUMA systems. It is expected that this technique could have an immediate impact since it could be easily used to improve locality and reduce execution times in other existing architectures, such as the KSR1 and DASH.

The adaptive partitioning techniques, motivated in Chapter 3 and described in Chapter 4, are not expected to be directly applied to production multiprocessors. Their use depends on the ability to dynamically reallocate processors during the execution of an application. At present, user-level threads are the commonly used mechanism for supporting the dynamic reallocation of

processors. Unfortunately, many vendors of large-scale systems have yet to incorporate user-level threads into their systems and as a result employ relatively simplistic, static scheduling policies (presumably because they view their systems as being used in mainly uniprogrammed environments). However, as the size of such systems continues to grow, it is likely that they will be used in multiprogrammed environments where the systems are shared by a large number of users, increasing the need for and importance of dynamic and adaptive scheduling strategies. In the meantime, we expect that our results will prove beneficial to other researchers in the study, design, and implementation of scheduling techniques that consider application characteristics when making allocation decisions.

7.3. Future Work

Throughout this dissertation we have mentioned possible avenues of future research. We now briefly outline the ones that we feel are most promising.

A fair amount of insight into the processor allocation problem has been obtained by examining analytic bounds on the performance of equipartition policies. We expect that similar insights might be gained by analytically comparing static with dynamic scheduling policies. It would also be beneficial to consider bounds on the performance of equipartition strategies with known arrival distributions and work to be executed, in both static and dynamic scheduling environments. Additionally, including some notion of application efficiency should also prove useful, especially if existing techniques could be compared with optimal strategies.

The analysis of techniques that utilize application characteristics are of little benefit if they can not be applied in practical situations. To this end, further research into methods for distinguishing between different applications and allocating processors accordingly should prove beneficial, especially if they combine some notion of the amount of work an application has left to execute and the efficiency with which it is likely to be executed. Recent work by Sevcik explores this problem [Sevcik1994].

While examining the importance of application placement in NUMA multiprocessors, the results of our experiments emphasized the importance of localization considerations not only by the operating system scheduler but by all portions of the system. These results stress the need for cooperation among the application writer, compiler, run-time system, memory manager, and scheduler, in order to enhance the localization of data accesses. An obvious need for coordination in our system is between the scheduler and the memory manager. One of the main goals of the scheduler is to place processes close to the data they are accessing. Memory

management techniques for NUMA multiprocessors have a similar but opposite view of this approach to localization, in that they try to place data close to the processors that are accessing it. As a simple example of the need for coordination, consider a scheduler that migrates a process to execute on a processor that is closer to data that is being frequently accessed, while simultaneously the memory manager decides to replicate or migrate the page of data being referenced. Both the scheduler and memory manager make decisions that should reduce the time to access data being frequently accessed. However, the net result may produce memory access times that are the same (while incurring the overhead of moving the process and moving or copying the data). There is an important interplay between these two important facets of the system that must be considered if the costs of remote memory accesses are to be reduced to the greatest extent possible. In fact, the interplay among the compiler, thread scheduler, operating system scheduler, and memory manager should all be considered when trying to reduce memory access costs.

To our knowledge processor pool-based scheduling is the first and only scheduling method that is designed specifically to be used in shared-memory NUMA multiprocessors. As a result, it is really a first attempt at a difficult and interesting problem. Further work is required in developing dynamic repartitioning methods that maintain the locality of the application and in comparing static and dynamic policies in a NUMA environment. Wu recently compared static and dynamic policies. Unfortunately, the policies used make no attempt to preserve the locality of the applications [Wu1993]. However, these results do indicate that the dynamic policies can offer performance advantages over static policies, even in a NUMA environment.

In order to first gain a better understanding of the factors involved in making processor allocation and application placement decisions, we have examined these problems in isolation. As pointed out in Chapter 6, these two decisions are actually related in that the number of processors allocated to an application may depend on which processors are available. This relationship, although making the problem more difficult, must be understood in order to make truly effective scheduling decisions in NUMA multiprocessors.

Any research examining and characterizing applications and workloads found in production multiprocessor systems would be extremely beneficial to many aspects of multiprocessor systems research.

Appendix

Theorem 3.1:

Under assumptions A3.1 through A3.9 of Chapter 3, the ROOT allocation policy $p_i = \frac{P\sqrt{W_i}}{\sum_{j=1}^{N}\sqrt{W_j}}$ yields an optimal partitioning of P processors among N simultaneously executing applications.

Proof:

The problem being solved can be stated as follows: given that the response time of job J_i is $R_i = \frac{W_i}{p_i}$ when assigned p_i processors and the processors are to be partitioned among the N jobs such that $\sum_{i=1}^{N} p_i = P$, determine the number of processors to assign to each job, p_i, such that mean response time $\bar{R} = \frac{1}{N}\sum_{i=1}^{N}\frac{W_i}{p_i}$ is minimized.

We use the method of Lagrange multipliers † [Lang1979] and let $\underline{p}$ denote the vector of real values, $p_1, p_2, \ldots, p_N$. To compute the extremal values of a function $F(\underline{p})$, subject to the constraint $G(\underline{p}) = K$, form the function $U = F(\underline{p}) + c\,G(\underline{p})$ where c is a constant to be determined. Then solve the system $\frac{\partial u}{\partial p_i} = 0$ together with $G(\underline{p}) = K$. The extremal values of F will be among the solutions $(\underline{p}, c)$ to these equations. We now apply this method of Lagrange multipliers to the problem:

$$F(\underline{p}) = \frac{1}{N}\sum_{i=1}^{N}\frac{W_i}{p_i}, \qquad G(\underline{p}) = \sum_{i=1}^{N} p_i = P, \qquad U = \frac{1}{N}\sum_{i=1}^{N}\frac{W_i}{p_i} + c\sum_{i=1}^{N} p_i$$

† This approach was suggested by Tom Fairgrieve.

Consequently: $\begin{cases} \dfrac{\partial u}{\partial p_i} = -\dfrac{W_i}{{p_i}^2} + c = 0 \qquad i=1,2,...,N \\ \displaystyle\sum_{i=1}^{N} p_i = P \end{cases}$

Therefore, for all i, $c = \dfrac{W_i}{{p_i}^2}$, so, $\dfrac{W_i}{{p_i}^2} = \dfrac{W_1}{{p_1}^2}$ and $p_i = p_1 \dfrac{\sqrt{W_i}}{\sqrt{W_1}}$.

Hence, $\displaystyle\sum_{i=1}^{N} p_i = p_1 \frac{\displaystyle\sum_{i=1}^{N} \sqrt{W_i}}{\sqrt{W_1}} = P$ and $p_1 = \dfrac{\sqrt{W_1}}{\displaystyle\sum_{i=1}^{N} \sqrt{W_i}}$.

Therefore, $p_i = \dfrac{P\sqrt{W_i}}{\displaystyle\sum_{j=1}^{N} \sqrt{W_j}}$, which proves the theorem.

■

Theorem 3.2:

$$\frac{1}{N} \le \bar{R}^{(R/E)} \le 1, \quad \text{where } \bar{R}^{(R/E)} = \frac{\left[\displaystyle\sum_{i=1}^{N} \sqrt{W_i}\right]^2}{N \displaystyle\sum_{i=1}^{N} W_i} \quad \text{and } W_i \ge 0 \ \ \forall i .$$

Proof:

Let $W = \displaystyle\sum_{i=1}^{N} W_i$ (i.e., W is the total work), so, $W > 0$. We, therefore, wish to find the maximum and minimum (extrema) of the function $\bar{R}^{(R/E)} = \dfrac{\left[\displaystyle\sum_{i=1}^{N} \sqrt{W_i}\right]^2}{N\,W} = f(\underline{W})$, where $\underline{W} = (W_1, W_2, W_3, \ldots, W_N)$. ‡

Now change variables so that W_i is scaled by the total work W. Define $x_i = \dfrac{W_i}{W}$. Since $W_i \ge 0$ and $W > 0$, $x_i \ge 0$ and $\| \underline{x} \|_1 = |x_1| + |x_2| + |x_3| + \cdots + |x_N| = 1$. Now finding extrema of $f(\underline{W})$ is equivalent to finding extrema of:

‡ This approach was suggested by Tom Fairgrieve.

$$g(\underline{x}) = \frac{\left[\sum_{i=1}^{N} \sqrt{x_i\, W}\right]^2}{N\, W} = \frac{1}{N}\left[\sum_{i=1}^{N} \sqrt{x_i}\right]^2 \text{ subject to } \|\underline{x}\|_1 = 1 \text{ and } x_i \geq 0.$$

Since $x_i \geq 0$, it is convenient to transform the variables via the transformation ${y_i}^2 = x_i$, $y_i > 0$, so that $\sqrt{{y_i}^2} = |y_i| = y_i$. Since $\|\underline{x}\|_1 = 1$ and $x_i \geq 0$, it follows that $\|\underline{y}\|_2{}^2 = {y_1}^2 + {y_2}^2 + {y_3}^2 + \cdots + {y_N}^2 = 1$ and $\|\underline{y}\|_2 = 1$. Then, finding extrema of $g(\underline{x})$ (and $f(\underline{W})$) is equivalent to finding extrema of:

$$h(\underline{y}) = \frac{1}{N}\left[\sum_{i=1}^{N} \sqrt{{y_i}^2}\right]^2 = \frac{1}{N}\left[\sum_{i=1}^{N} |y_i|\right]^2 = \frac{1}{N}\|\underline{y}\|_1{}^2$$

subject to $\|\underline{y}\|_2 = 1$ and $y_i \geq 0$.

By a well-known property of norms [Golub1989], for all vectors $\underline{z} \in \mathbb{R}^n$, $\|\underline{z}\|_2 \leq \|\underline{z}\|_1 \leq \sqrt{n}\, \|\underline{z}\|_2$. Therefore, $\|\underline{y}\|_2{}^2 \leq \|\underline{y}\|_1{}^2 \leq N\, \|\underline{y}\|_2{}^2$ or $1 \leq \|\underline{y}\|_1{}^2 \leq N$, since $\|\underline{y}\|_2 = 1$. Therefore, $\frac{1}{N} \leq \frac{1}{N}\|\underline{y}\|_1{}^2 < 1$ and $\frac{1}{N} \leq h(\underline{y}) \leq 1$.

Since $\bar{R}^{(R/E)} = f(\underline{W}) = g(\underline{x}) = h(\underline{y})$, $\frac{1}{N} \leq \bar{R}^{(R/E)} \leq 1$, thus proving the theorem. ■

Furthermore, these bounds are tight for $h(\underline{y}) = 1$ when $y_i = \frac{1}{\sqrt{N}}\ \forall\, i$ and for $h(\underline{y}) = \frac{1}{N}$ when $y_i = 1$ for some value of i and when $y_j = 0$ for $j \neq i$. $y_i = \frac{1}{\sqrt{N}}\ \forall\, i$ corresponds to $x_i = \frac{1}{N}\ \forall\, i$ and $W_i = W_j\ \forall\, i, j$. That is, all jobs execute the same work and $\bar{R}^{(R/E)} = 1$. $y_i = 1$ for some value of i and $y_j = 0$ for $j \neq i$ corresponds to $x_i = 1$ and $W_i = W$ for some i, and $x_j = 0$, $W_j = 0$ for all $j \neq i$. That is, only one job has work to execute and $\bar{R}^{(R/E)} = \frac{1}{N}$.

Bibliography

[Adve1990]

S. Adve and M. Hill, ‘‘Weak Ordering - a New Defintion’’, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 2-14, New York, NY, 1990.

[Agarwal1990]

A. Agarwal, B. Lim, J. Kubiatowicz, and D. Kranz, ‘‘APRIL: A Processor Architecture for Multiprocessing’’, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 104-114, New York, NY, May, 1990.

[Agarwal1991]

A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B. Lim, G. Maa, and D. Nussbaum, ‘‘The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor’’, **Scalable Shared Memory Multiprocessors**, ed. M. Dubois and S. S. Thakkar, Kluwer Academic Publishers, Norwell, Massachusetts, pp. 239-261, 1991.

[Ahmad1991]

I. Ahmad and A. Ghafoor, ‘‘Semi-Distributed Load Balancing for Massively Parallel Multicomputer Systems’’, *IEEE Transactions on Software Engineering*, *Vol*. 17, *No*. 10, pp. 987-1004, October, 1991.

[Alverson1990]

R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, ‘‘The Tera Computer System’’, *Proceedings of the 1990 International Conference on Supercomputing*, pp. 1-6, 1990.

[Amdahl1967]

G. M. Amdahl, ‘‘Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities’’, *Proceedings of AFIPS*, *Vol*. 30, pp. 483-485, 1967.

[Anderson1991]

T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, ‘‘Scheduler Activations: Effective Kernel Support for User-Level Management of Parallelism’’, *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 95-109, Pacific Grove, CA, October 1991.

[Barnes1986]

J. Barnes and P. Hut, ‘‘A Hierarchical $O(n \log n)$ Force-Calculation Algorithm’’, *Nature*, *Vol*. 324, *No*. 4, pp. 446-449, December, 1986.

[Bershad1988]

B. N. Bershad, E. D. Lazowska, and H. M. Levy, ‘‘PRESTO: A System for Object-oriented Parallel Programming’’, *Software - Practice and Experience*, *Vol*. 18, *No*. 8, pp. 713-732, August, 1988.

[Black1990]
D. L. Black, ‘‘Scheduling Support for Concurrency and Parallelism in the Mach Operating System’’, *IEEE Computer*, pp. 35-43, May, 1990.

[Brecht1990]
T. B. Brecht, ‘‘Processor Scheduling in Large-Scale Shared-Memory Multiprocessors’’, *Proceedings of the Computer Measurement Group of Canada 1990*, pp. 110-117, June, 1990.

[Bunt1976]
R. B. Bunt, ‘‘Scheduling Techniques for Operating Systems’’, *IEEE Computer*, *Vol*. 9, *No*. 10, pp. 10-17, October, 1976.

[Burkhardt1992]
H. Burkhardt, S. Frank, B. Knobe, and J. Rothnie, ‘‘Overview of the KSR1 Computer System’’, Kendall Square Research, Boston, Technical Report KSR-TR-9202001, February, 1992.

[Carlson1992]
B. Carlson, T. Wagner, L. Dowdy, and P. Worley, ‘‘Speedup Properties of Phases in the Execution Profile of Distributed Parallel Programs’’, *Proceedings of the 6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp. 83-95, Edinburgh, Scotland, September, 1992.

[Coffman1968]
E. Coffman and L. Kleinrock, ‘‘Feedback Queueing Models for Time-Shared Systems’’, *Journal of the ACM*, *Vol*. 15, *No*. 4, pp. 549-576, October, 1968.

[Coffman1973]
E. G. Coffman and P. J. Denning, **Operating Systems Theory**, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1973.

[Cooley1965]
J. W. Cooley and J. W. Tukey, ‘‘An Algorithm for the Machine Calculation of Complex Fourier Series’’, *Mathematics of Computation*, *Vol*. 19, *No*. 90, pp. 297-301, 1965.

[Cooper1987]
E. C. Cooper and R. P. Draves, ‘‘C Threads’’, Department of Computer Science, CMU Technical-Report (draft), July 20, 1987.

[Corbato1962]
F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, ‘‘An Experimental Time-Sharing System’’, *Proceedings of the AFIPS Fall Joint Computer Conference*, pp. 335-344, 1962.

[Crovella1991]
M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos, ‘‘Multiprogramming on Multiprocessors’’, *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pp. 590-597, December, 1991.

[Dantzig1951]
G. B. Dantzig, ‘‘Maximization of a Linear Function of Variables Subject to Linear Inequalities’’, **Activity Analysis of Production and Allocation**, ed. T. C. Koopmans, John Wiley, New York, pp. 339-347, 1951.

[Deitel1984]
H. M. Deitel, **An Introduction to Operating Systems**, Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[Doeppner Jr.1987]
T. W. Doeppner Jr., ‘‘Threads - A System for the Support of Concurrent Programming’’, Technical Report CS-87-11, Department of Computer Science, Brown University, Providence RI, June 16, 1987.

[Dubois1990]
M. Dubois and C. Scheurich, ‘‘Memory Access Dependencies in Shared-Memory Multiprocessors’’, *IEEE Transactions on Software Engineering*, *Vol*. 16, *No*. 6, pp. 660-673, June, 1990.

[Duda1972]
R. O. Duda and P. E. Hart, ‘‘Use of Hough Transformation to Detect Lines and Curve in Pictures’’, *Communications of the ACM*, *Vol*. 15, *No*. 1, pp. 11-15, January, 1972.

[Dunigan1992]
T. H. Dunigan, ‘‘Kendall Square Multiprocessor: Early Experiences and Performance’’, Engineering and Mathematics Division, Oak Ridge National Laboratory, ORNL/TM-12065, March, 1992.

[Eager1989]
D. L. Eager, J. Zahorjan, and E. D. Lazowska, ‘‘Speedup Versus Efficiency in Parallel Systems’’, *IEEE Transactions on Computers*, *Vol*. 38, *No*. 3, pp. 408-423, March, 1989.

[Feitelson1990]
D. G. Feitelson and L. Rudolph, ‘‘Distributed Hierarchical Control for Parallel Processing’’, *IEEE Computer*, pp. 65-77, May, 1990.

[Feitelson1990a]
D. G. Feitelson and L. Rudolph, ‘‘Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control’’, *1990 International Conference on Parallel Processing*, pp. I1-I8, 1990.

[Gamsa1992]
B. Gamsa, **Region-Oriented Main Memory Management in Shared-Memory NUMA Multiprocessors**, M.Sc. Thesis, University of Toronto, Toronto, Ontario, September, 1992.

[Garey1979]
M. R. Garey and D. S. Johnson, **Computers and Intractability — A Guide to the Theory of NP-Completeness**, W. H. Freeman, New York, 1979.

[Gharachorloo1990]
K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, ‘‘Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors’’, *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 15-26, New York, NY, 1990.

[Gharachorloo1992]
K. Gharachorloo, A. Gupta, and J. Hennessy, ‘‘Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors’’, *The Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 22-33, Gold Coast, Australia, May, 1992.

[Ghosal1991]
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, pp. 443-453, May, 1991.

[Golub1989]
G. H. Golub and C. F. VanLoan, **Matrix Computations**, The Johns Hopkins University Press, Baltimore, MD, p. 54, 1989.

[Gornish1990]
E. H. Gornish, E. D. Granston, and A. V. Veidenbaum, ‘‘Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies’’, *Proceedings of the 1990 International Conference on Supercomputing*, pp. 354-368, 1990.

[Gupta1991]
A. Gupta, A. Tucker, and S. Urushibara, ‘‘The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications’’, *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 120-132, San Diego, CA, May, 1991.

[Gurd1985]
J. R. Gurd, C. C. Kirkham, and I. Watson, ‘‘The Manchester Prototype Dataflow Computer’’, *Communications of the ACM*, *Vol*. 28, *No*. 1, pp. 34-52, February, 1985.

[Junkin1989]
M. Junkin and D. Wortman, ‘‘Supervisors - An Approach to Controlling Concurrency’’, Computer Systems Research Institute, Report CSRI-235, University of Toronto, Toronto, Ontario, March, 1989.

[Kumar1988]
M. Kumar, ‘‘Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications’’, *IEEE Transactions on Computers*, *Vol*. 37, *No*. 9, pp. 1088-1098, September, 1988.

[Lang1979]
S. Lang, **Calculus of Several Variables**, Addison-Wesley Publishing Company, Reading, Massachusetts, p. 269, 1979.

[Leland1986]
W. E. Leland and T. J. Ott, ‘‘Load-balancing Heuristics and Process Behavior’’, *Proceedings of the 1986 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 54-69, Raleigh, NC, 1986.

[Lenoski1990]
D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, ‘‘The Directory-Based Cache Coherent Protocol for the DASH Multiprocessor’’, *The Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 146-160, New York, NY, May, 1990.

[Lenoski1992]
D. Lenoski, J. Laudon, T. Joe, D. Nakahari, L. Stevens, A. Gupta, and J. Hennessy, ‘‘The DASH Prototype: Implementation and Performance’’, *The Proceedings of the 19th Annual International Symposium on Computer Architecture*, pp. 92-103, May, 1992.

[Leutenegger1990]
S. T. Leutenegger and M. K. Vernon, ‘‘The Performance of Multiprogrammed Multiprocessor Scheduling Policies’’, *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 226-236, Boulder, CO, May, 1990.

[Leutenegger1991]
S. T. Leutenegger and R. D. Nelson, ‘‘Analysis of Spatial and Temporal Scheduling Policies for Semi-Static and Dynamic Multiprocessor Environments’’, IBM Research Report RC 17086 (No. 75594), August 1, 1991.

[Majumdar1988]
S. Majumdar, D. Eager, and R. B. Bunt, ‘‘Scheduling in Multiprogrammed Parallel Systems’’, *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 104-113, May, 1988.

[Majumdar1988a]
S. Majumdar, **Processor Scheduling in Multiprogrammed Parallel Systems**, Ph.D. Thesis, University of Saskatchewan, Saskatoon, Saskatchewan, April, 1988.

[Marinescu1990]
D. C. Marinescu and J. R. Rice, ‘‘On Single Parameter Characterization of Parallelism’’, *Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pp. 235-237, 1990.

[Markatos1992]
E. P. Markatos and T. J. LeBlanc, ‘‘Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors’’, *Proceedings of Supercomputing ’92*, pp. 104-113, Minneapolis, MN, November, 1992.

[Markatos1992a]
E. P. Markatos and T. J. LeBlanc, ‘‘Load Balancing vs. Locality Management in Shared-Memory Multiprocessors’’, *1992 International Conference on Parallel Processing*, pp. 258-267, August, 1992.

[Markatos1993]
E. P. Markatos, **Scheduling for Locality in Shared-Memory Multiprocessors**, Ph.D. Thesis, Department of Computer Science, University of Rochester, Rochester, New York, May, 1993.

[Marsh1991]
B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, ‘‘First-Class User-Level Threads’’, *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 110-121, Pacific Grove, CA, October 1991.

[Matloff1989]
N. S. Matloff, ‘‘On the Value of Predictive Information in a Scheduling Problem’’, *Performance Evaluation*, *Vol*. 10, *No*. 4, pp. 309-315, December, 1989.

[McCann1993]
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, pp. 146-178, May, 1993.

[Mendenhall1981]
W. Mendenhall, R. L. Scheaffer, and D. D. Wackerly, **Mathematical Statistics with Applications**, Duxbury Press, Boston, Massachusetts, p. 96, 1981.

[Mohr1991]
E. Mohr, D. Kranz, and R. Halstead, ‘‘Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs’’, *IEEE Transactions on Parallel and Distributed Systems*, *Vol*. 2, *No*. 3, pp. 264-280, July, 1991.

[Motwani1993]
R. Motwani, S. Phillips, and E. Torng, ‘‘Non-Clairvoyant Scheduling’’, *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 422-431, Austin, Texas, January, 1993.

[Ousterhout1982]
J. K. Ousterhout, ‘‘Scheduling Techniques for Concurrent Systems’’, *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pp. 22-30, October, 1982.

[Owicki1989]
S. Owicki, ‘‘Experience with the Firefly Multiprocessor Workstation’’, Digital Systems Research Center, Research Report 51, September 15, 1989.

[Prasanna1994]
G. N. Prasanna, A. Agarwal, and B. R. Musicus, ‘‘Hierarchical Compilation of Macro Dataflow Graphs for Multiprocessors with Local Memory’’, *IEEE Transactions on Parallel and Distributed Systems (to appear)*, 1994.

[Press1988]
W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Veterling, **Numerical Recipes in C: The Art of Scientific Computing**, University Press, Cambridge, 510 North Avenue, New Rochelle, NY, 1988.

[Ravindran1987]
A. Ravindran, D. T. Philips, and J. J. Solberg, **Operations Research - Principles and Practice, 2nd Edition**, John Wiley and Sons, 1987.

[Rosti1994]
E. Rosti, E. Smirni, L. Dowdy, G. Serazzi, and B. Carlson, ‘‘Robust partitioning policies of multiprocessor systems’’, *Performance Evaluation*, *Vol*. 9, *No*. 2-3, pp. 141-166, 1994.

[Rumelhardt1986]
D. E. Rumelhardt, J. L. McClelland, and the PDP Research Group, **Parallel Distributed Processing**, MIT Press, 1986.

[Sevcik1989]
K. C. Sevcik, ‘‘Characterizations of Parallelism in Applications and their use in Scheduling’’, *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 171-180, 1989.

[Sevcik1994]
K. C. Sevcik, ‘‘Application Scheduling and Processor Allocation in Multiprogrammed Multiprocessors’’, *Performance Evaluation, Vol*. 9, *No*. 2-3, pp. 107-140, 1994.

[Silberschatz1991]
A. Silberschatz, J. L. Peterson, and P. B. Galvin, **Operating System Concepts**, Addison-Wesley, Reading, Massachusetts, 1991.

[Smith1981]
B. Smith, ‘‘Architecture and Applications of the HEP Multiprocessor Computer’’, *Real-Time Signal Processing IV, Vol*. 298, pp. 241-248, 1981.

[Squillante1990]
M. S. Squillante, **Issues in Shared-Memory Multiprocessor Scheduling: A Performance Evaluation**, Ph.D. Thesis, Department of Computer Science and Engineering, University of Washington, Seattle, Washington, Technical Report 90-10-04, October, 1990.

[Squillante1993]
M. S. Squillante and E. D. Lazowska, ‘‘Using Processor Cache Affinity Information in Shared-Memory Multiprocessor Scheduling’’, *IEEE Transactions on Parallel and Distributed Systems, Vol*. 4, *No*. 2, pp. 131-143, February, 1993.

[Strang1980]
G. Strang, **Linear Algebra and Its Applications**, Academic Press, New York, 1980.

[Stumm1993]
M. Stumm, Z. Vranesic, R. White, R. Unrau, and K. Farkas, ‘‘Experiences with the Hector Multiprocessor’’, *Proceedings of the International Parallel Processing Symposium Parallel Processing Fair*, pp. 9-16, April, 1993.

[Thiebaut1987]
D. Thiebaut and H. S. Stone, ‘‘Footprints in the Cache’’, *ACM Transactions on Computer Systems, Vol*. 5, *No*. 4, pp. 305-329, November, 1987.

[Tucker1989]
A. Tucker and A. Gupta, ‘‘Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors’’, *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pp. 159-166, 1989.

[Unrau1992]
R. Unrau, M. Stumm, and O. Krieger, ‘‘Hierarchical Clustering: A Structure for Scalable Multiprocessor Operating System Design’’, *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 285-303, Seattle, WA, April, 1992.

[Unrau1993]
R. Unrau, **Scalable Memory Management through Hierarchical Symmetric Multiprocessing**, Ph.D. Thesis, University of Toronto, Toronto, Ontario, January, 1993.

[Vandevoorde1988]
M. T. Vandevoorde and E. S. Roberts, ‘‘WorkCrews: An Abstraction for Controlling Parallelism’’, *International Journal of Parallel Programming, Vol*. 17, *No*. 4, pp. 347-366, August, 1988.

[Vaswani1991]
R. Vaswani and J. Zahorjan, ‘‘The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors’’, *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 26-40, Pacific Grove, CA, October 1991.

[Vranesic1991]
Z. Vranesic, M. Stumm, D. Lewis, and R. White, ‘‘Hector: A Hierarchically Structured Shared-Memory Multiprocessor’’, *IEEE Computer, Vol*. 24, *No*. 1, pp. 72-79, January, 1991.

[Wu1993]
Chee-Shong Wu, **Processor Scheduling in Multiprogrammed Shared Memory NUMA Multiprocessors**, M.Sc. Thesis, University of Toronto, Toronto, Ontario, October, 1993.

[Zahorjan1990]
J. Zahorjan and C. McCann, ‘‘Processor Scheduling in Shared Memory Multiprocessors’’, *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 214-225, Boulder, CO, May, 1990.

[Zahorjan1991]
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, Vol*. 2, *No*. 2, pp. 180-198, April, 1991.

[Zhou1991]
S. Zhou and T. B. Brecht, ‘‘Processor Pool-Based Scheduling for Large-Scale NUMA Multiprocessors’’, *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 133-142, San Diego, CA, May, 1991.