# Competitive Analysis of Dynamic Multiprocessor Allocation Strategies

Nian Gu

Technical Report CS-95-02
May, 1995

Department of Computer Science
York University
North York, Ontario, Canada
M3J 1P3

# Competitive Analysis of Dynamic Multiprocessor Allocation Strategies

Nian Gu

Technical Report CS-95-02
May, 1995

Department of Computer Science
York University
North York, Ontario, Canada
M3J 1P3



This report is an adaptation of Nian Gu's M.Sc. thesis.

## Competitive Analysis of Dynamic Multiprocessor Allocation Strategies

Nian Gu

A thesis submitted in conformity with the requirements
for the Degree of Master of Science
Graduate Program of Computer Science
York University

### ABSTRACT

The subject of this thesis is to study the problem of dynamic processor allocation in parallel application scheduling. Processor allocation involves determining the number of processors to allocate to each of several simultaneously executing parallel applications and possibly dynamically adjusting the allocations during execution to improve overall system performance. We devise and analytically evaluate dynamic allocation policies that operate in environments in which full information about the jobs being executed is not known when making scheduling decisions.

We use competitive analysis to compare the performance of algorithms which do not know the arrival or execution time of jobs with the performance of the optimal algorithm which uses complete information about jobs. The result of such a comparison is called a competitive ratio. The competitive ratio is indicative of the utility of different allocation policies. Our study is carried out under two performance objectives: minimizing the makespan and minimizing the mean response time respectively.

Our results for minimizing the makespan consist of three parts. First we use competitive analysis to devise an allocation policy, *OptComp*, which yields the optimal competitive ratio for scheduling two parallel jobs. As well, we compare the dynamic equipartition (DEQ) policy with *OptComp* and find that the relative ratio of DEQ to *OptComp* is 1.175729. This implies that the competitive ratio of DEQ is very close to optimal. Secondly we extend the results of sequential job scheduling to parallel job scheduling. Our results cover the situations when the application parallelism varies during execution as well as when applications do not arrive at the system simultaneously. Lastly we consider the case when some applications may execute infinitely because of programming errors which lead to an infinite loop. Our results show that DEQ yields the optimal competitive ratio.

Our results for the problem of minimizing the mean response time assume that all applications arrive at the system simultaneously and that application parallelism does not change during execution. We show that in this case DEQ yields an optimal competitive ratio of $\left(2 - \frac{2}{N+1}\right)$.

## Acknowledgments

I am grateful to Dr. Xiaotie Deng and Dr. Tim Brecht for introducing me to this area of research and supervising the thesis. Their insightful suggestions and advice improve this thesis greatly. Their guidance, encouragement and understanding have sailed me through the rough times in this research.

I would like to extend my gratitude to Dr. Neal Madras and Dr. Anestis Toptsis for being on my examination committee and providing valuable comments regarding this thesis. Thanks to Kaushik Guha and Kenneth Man for proof-reading parts of the thesis. I am also grateful for the financial support from the Department of Computer Science which enables me to complete my degree.

I wish to thank my grandparents and my parents for their love and support through these years without which I would not have been able to pursue my goals. Thanks also to my friends Kaushik, Choi, Ben, Ken, Khuzaima, Roy, David and Qi for lots of fun and the inspiration which have made my stay at York happy and enjoyable. Special thanks to Maylee, Peter, Karen, Matthias, Rachel, Simon, Linda and Diana for providing me with the moral support, encouragement and entertainment during these two years.

I would also like to express my thankfulness to the Friday night Light Fellowship group and Inter-Varsity Fellowship group for providing me a place to take out the frustration. Thanks to the people from Saint Christopher's Anglican Church and the Toronto Chinese Disciples' Church for the continuous prayers.

# Table of Contents


## Chapter 1

## Introduction ........ 1

1.1. Motivation ........ 1

1.2. Goals ........ 2

1.3. Contributions ........ 2

1.4. Overview of the Thesis ........ 3

## Chapter 2

## Background ........ 4

2.1. Performance Objectives ........ 4

2.2. Scheduling Parallel Jobs on Multiprocessors ........ 5

2.2.1. Time-sharing versus Space-sharing ........ 5

2.2.2. Static versus Dynamic Scheduling ........ 5

2.3. Application Characteristics ........ 6

2.3.1. Parallelism Profile ........ 6

2.3.2. Speedup and Efficiency ........ 6

2.3.3. Work to Be Executed ........ 8

2.4. Competitive Analysis ........ 8

2.5. Dynamic Equipartition Policy ........ 9

2.6. Summary ........ 10

## Chapter 3

## The Job, Workload and System Models ........ 11

3.1. Job Model ..... 11

3.2. Workload Model ..... 11

3.3. System Model ..... 12

3.4. Summary ..... 12

# Chapter 4

# Minimizing the Makespan ..... 13

4.1. Optimal Scheduling with Complete Information ..... 14

4.2. Optimal Competitive Scheduling for Two Jobs on P Processors ..... 20

4.3. Scheduling N Jobs on P Processors ..... 23

4.3.1. Scheduling N Jobs with Single-phased Parallelism Profiles ..... 23

4.3.2. Scheduling N Jobs with Multiple Phases of Parallelism ..... 25

4.3.3. Scheduling with New Arrivals ..... 26

4.4. Scheduling with Erroneous Infinite Jobs ..... 26

4.4.1. All Jobs Arrive at the System Simultaneously ..... 27

4.4.2. Scheduling Infinite Jobs and with New Arrivals ..... 29

4.5. Summary ..... 30

# Chapter 5

# Minimizing the Mean Response Time ..... 31

5.1. Lower Bounds on Mean Response Time ..... 32

5.2. An Upper Bound for DEQ ..... 34

5.3. Summary ..... 40

# Chapter 6

# Conclusions and Future Research ..... 41

6.1. Conclusions ..... 41

6.2. Direction for Future Research ..... 43

**Appendix** .......... 44
**Bibliography** .......... 45

# Glossary

| | |
|---|---|
| $J_i$ | - job $i$ |
| $P$ | - total number of processors in the system |
| $P_i$ | - parallelism of job $i$ |
| $l_i$ | - execution time of job $i$ if executed on unlimited number of processors |
| $p_i$ | - number of processors allocated to job $i$ |
| $p$ | - equipartition value according to DEQ |
| $N$ | - total number of jobs in the system |
| $W_i$ | - work executed by job $i$ |
| $JS$ | - job set |
| $JS_{equi}$ | - job set in which the allocation to each job is equal to $\bar{p}$ |
| $k_e$ | - number of jobs in $JS_{equi}$ |
| $JS_{para}$ | - job set in which the allocation to each job is equal to $P_i$ |
| $k_p$ | - number of jobs in $JS_{para}$ |
| $JS'_{para}$ | - subset of $JS_{equi}$ in which the allocation to each job is increased to $P_i$ after processor reallocation |
| $k_p'$ | - number of jobs in $JS'_{para}$ |
| $JS'_{equi}$ | - a subset of $JS_{equi}$ in which the allocation to each job is equal to $\bar{p}$ before and after processor reallocation |
| $k_e'$ | - number of jobs in $JS'_{equi}$ |
| $M_A(JS)$ | - the value of the objective function for algorithm $A$ with job set $JS$ |
| $Opt(JS)$ | - the optimal value of the objective function with job set $JS$ |

| | |
|---|---|
| $\alpha$ | - used to devise the optimal policy for scheduling two jobs, $0 \le \alpha \le 1$ |
| $R_1$ | - competitive ratio for scheduling two jobs when $J_1$ completes execution first |
| $R_2$ | - competitive ratio for scheduling two jobs when $J_2$ completes execution first |
| $m$ | - parallelism of $J_1$ represented as a fraction of $P$ |
| $n$ | - parallelism of $J_2$ represented as a fraction of $P$ |
| $t_i$ | - time at which job $i$ arrives at the system |
| $\tau$ | - time span during which $\sum_{i=1}^{N} P_i \ge P$ |
| $\tau'$ | - time span during which $\sum_{i=1}^{N} P_i < P$ |
| $t^*$ | - time at which $\sum_{i=1}^{N} P_i < P$ for the first time |
| $r_i$ | - remaining execution time of job $i$ |
| $L_1$ | - maximum of $l_i$ over all jobs |
| $K$ | - number of infinite jobs in the system |
| $T_i$ | - execution time of job $i$ |
| $T_{equi}$ | - time span during which the allocation to job $i$ is equal to $\bar{p}$ |
| $T_{para}$ | - time span during which the allocation to job $i$ is equal to $P_i$ |
| $l_{equi}$ | - the time required to execute $\bar{p}\, T_{equi}$ units of work if job $i$ is allocated $P_i$ processors |
| $\beta_i$ | - response time of job $i$ |
| $f$ | - phase during which the allocation to job $i$ is equal to the job parallelism |
| $e$ | - phase during which the allocation to job $i$ is equal to the equipartition value |
| $A(JS)$ | - squashed area bound for job set $JS$ |

| | |
|---|---|
| $L(JS)$ | - sum of $l_i$ for all jobs in job set $JS$ |
| $FT_A(JS)$ | - flow time for job set $JS$ with algorithm $A$ |
| $C$ | - constant which is defined to be $2 - \frac{2}{N+1}$ |

# Chapter 1

# Introduction

Historically, increases in computing power were obtained by employing faster processors using high-speed semiconductor technology. Multiprocessor systems have emerged as an alternative means for reducing program execution time. Processor scheduling becomes critical to the performance of multiprocessor systems because an inappropriate scheduling decision can substantially degrade system performance.

Scheduling parallel applications on multiprocessors involves determining the number of applications to execute simultaneously (an activation decision) and the number of processors to allocate to each of these simultaneously executing applications (an allocation decision). A number of different policies have been proposed and evaluated for scheduling on multiprocessor systems [22][19][28][23][16][33][12][20][24][3][2]. As well, a number of different approaches are possible for comparing the performance of various algorithms [10][8][9][1][5]. We study dynamic scheduling policies for scheduling multiple parallel applications in shared-memory multiprocessors when the scheduler does not have full information about applications at the time of scheduling. The approach used is competitive analysis, which compares the performance of an algorithm without full information with the optimal algorithm operating with full information about jobs.

## 1.1. Motivation

Recent studies have demonstrated by both analysis and simulation that utilizing certain job characteristics may improve the mean response time of parallel applications [6][23][33][20][24][2]. However, in real systems it is unlikely that the scheduler will have access to all of this information. For example, information about the service demand of jobs may be used by the scheduler to reduce mean response time [18][19][24][2], however, such information may not be available to the scheduler. Therefore, we study the processor allocation problem assuming that the scheduler does not have complete information about incoming applications. To be specific, we assume that the scheduler has no information about a job's arrival and its execution time.

Competitive analysis is an approach to studying algorithms that operate in environments in which information about the jobs being executed is not known at the time of scheduling. The idea behind competitive analysis is to compare, over all possible inputs, the performance of an algorithm when full information about the input is unavailable with the performance of an optimal algorithm which operates with complete information.

A number of studies have applied competitive analysis to sequential job scheduling on multiprocessors in order to minimize the makespan [10][8][13][25]. In this thesis we examine this problem in the context of parallel applications. As well, a number of recent studies have utilized competitive analysis to study the problem of minimizing the mean response time for scheduling parallel applications in a static environment, where the number of processors allocated to each job does not change during execution [30][17][31][29]. In this thesis we study this problem in a dynamic environment in which the allocations to applications may be adjusted during execution in order to improve the overall system performance.

## 1.2. Goals

One of our goals is to use competitive analysis to study dynamic processor allocation policies under realistic assumptions of job characteristics and the workload. Another goal is to gain a better understanding of the dynamic equipartition policy and the significance of using certain job characteristics in scheduling by comparing the performance of the policies using limited available information about job characteristics with the optimal policy which uses full information.

## 1.3. Contributions

We study the dynamic processor allocation problem under the assumption that the scheduler does not know the job arrival and execution time. Our study is conducted under realistic job and workload models. We study dynamic processor allocation policies designed to minimize makespan and mean response time respectively. Our contributions are twofold:

(1) Minimizing the makespan: we use competitive analysis to devise an optimal policy for scheduling two jobs. This method may be of help for devising optimal competitive policies for generalized cases. We extend the results of sequential job scheduling to parallel job scheduling. Our analysis considers realistic situations when there are new arriving jobs and the job parallelism varies during execution.

(2) Minimizing the mean response time: we study the problem of minimizing the mean response time in a dynamic scheduling environment. We show that the dynamic equipartition policy, which allocates an equal fraction of processing power to each job, provided that the job has enough parallelism, has an optimal competitive ratio. This explains why the dynamic equipartition policy performs well under various workloads.

Our results show that when the information of arrival and execution time is not available at the time of scheduling, the policy which space-shares processors evenly among all jobs in the system has the optimal competitive ratio. This leads to a better understanding of the significance of utilizing the application execution time in scheduling parallel jobs.

### 1.4. Overview of the Thesis

The structure of the thesis is as follows: In Chapter 2 we present some of the relevant background and previous results for the problem under study. In Chapter 3 we introduce the job workload and system models upon which our research is based. In Chapter 4 we study the problem of minimizing the makespan. We first devise an optimal algorithm for scheduling two jobs, then we study the problem in the context of scheduling $N$ jobs. Lastly we consider the situation in which some applications may execute infinitely. In Chapter 5 we study the problem of minimizing the mean response time. We prove that, in the worst case, the mean response time for the dynamic equipartitioin (DEQ) policy is no more than twice the optimal. In Chapter 6 we summarize our research.

# Chapter 2

# Background

Two important aspects of parallel application scheduling in multiprocessors are: (a) how many applications are allowed to execute in the system at the same time and (b) how many processors should be allocated to each of them (including dynamically adjusting those allocations to improve overall system performance). In this chapter we briefly outline the issues in parallel application scheduling and the relevant research in literature.

## 2.1. Performance Objectives

Most of the research in multiprocessor scheduling considers one of two performance objectives: minimizing the makespan or minimizing the mean response time of applications. For a set of jobs, *makespan* is defined as the amount of time that elapses from the arrival of the first job to the departure (completion) of the last job. Minimizing makespan is important for scheduling jobs in a batch mode as well as in manufacturing. The problem of minimizing makespan has been treated extensively in the context of scheduling sequential jobs [10][8][26][1][13][25][7]. We shall extend the previous results obtained for scheduling sequential jobs to the scheduling of parallel jobs.

In an interactive environment minimizing mean response time is of greater interest than minimizing makespan. The *response time* of a job is the amount of time which elapses from its arrival until its completion. A number of recent analytic studies have been carried out in parallel job scheduling in order to minimize the mean response time [21][29][30][31][17]. However, all of these studies are conducted in a static environment, in which the processors are not preempted from an executing job until it has completed execution. We shall study this problem of minimizing the mean response time in a dynamic environment.

The problem of minimizing makespan is simpler than minimizing the mean response time, because minimizing makespan is only concerned with the completion time of the last job in the system while minimizing the mean response time requires considering the completion time of all jobs in the system.

### 2.2. Scheduling Parallel Jobs on Multiprocessors

As the number of processors available in multiprocessors continues to grow it is likely that many applications will not be able to utilize all of the processors in the system. The effective simultaneous execution of multiple parallel applications has become a research area of growing interest and importance. We briefly describe the well-accepted approaches to multiprogramming parallel applications, upon which our work is based.

#### 2.2.1. Time-sharing versus Space-sharing

Techniques for multiprogramming parallel applications in shared-memory multiprocessors 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.

Under a *time-sharing* scheme processors are shared over time by executing different applications on the same processor during different time intervals. Such a time interval is called a *time-slice.* Usually processes of the same application are executed on different processors during the same time slice. As a result, each processor incurs context switching overheads at every time-slice.

A *space-sharing* scheme partitions processors among jobs. That is, it allocates different portions of the system to different applications [28]. This eliminates the need to rotate processors from one job to another and avoids unnecessary context switching overheads. Studies have shown that space-sharing is preferable to time-sharing in small-scale shared-memory multiprocessors [28][12][20]. Therefore, our study is focused on space-sharing schemes.

#### 2.2.2. Static versus Dynamic Scheduling

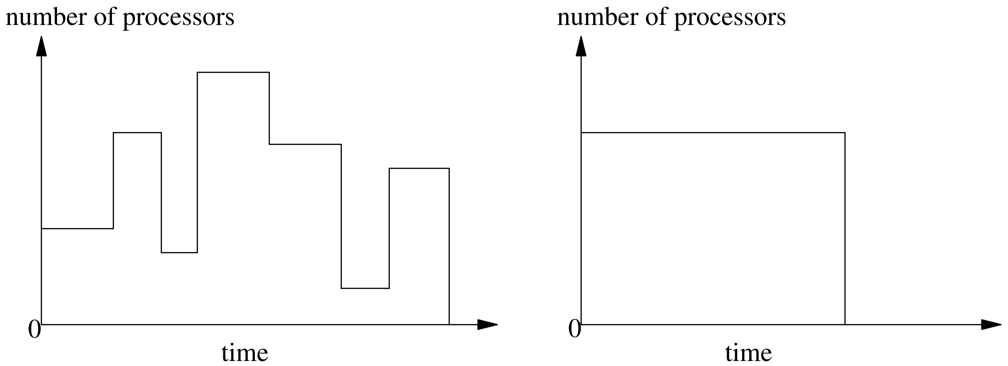
Scheduling algorithms can be classified as static or dynamic according to the frequency with which processor reallocation decisions are made and revised. A *static* algorithm assigns a certain number of processors to an application at the time it is activated and does not reallocate processors during its execution. A *dynamic* algorithm permits the allocation of processors among applications to be changed at any time [28][20]. Both simulation and experimental evaluation have concluded that dynamic scheduling is preferable to static scheduling in UMA multiprocessors, even when reallocation overheads are relatively large [33][20]. Experimental evaluations in a NUMA multiprocessor also arrive at a similar conclusion [32]. Therefore, our research is concentrated on dynamic scheduling policies.

## 2.3. Application Characteristics

Recent studies have demonstrated that in multiprogrammed environments certain application characteristics can be used to make scheduling decisions in order to improve mean response time [6][23][20] [2][11] Other studies have shown that some of these characteristics can be obtained and used in realistic environments [2][24]. The following is a brief description of the application characteristics that we believe to be potentially useful.

### 2.3.1. Parallelism Profile

A *parallelism profile* is defined as the number of processors an application is capable of using at any point in time during its execution [14]. Figure 2.1 shows two examples of parallelism profiles. If, during execution, the parallelism of an application varies with time, its parallelism profile is said to have multiple phases (as shown in Figure 2.1(a)). If the parallelism of a job does not change during execution, the parallelism profile is said to consist of a single phase (as shown in Figure 2.1(b)).




(a) parallelism profile with multiple phases (b) parallelism profile with a single phase

**Figure 2.1: Parallelism profile**

### 2.3.2. Speedup and Efficiency

Speedup and efficiency are two common performance measures of a parallel program. *Speedup* is defined to be the ratio of execution time attained using one processor to the execution time using $p$ processors. The *efficiency* of a job is defined as the mean effective processor utilization when $p$ processors are allocated to the job.

Define $T_j(p)$, where $p = 1, \ldots, P$, as the execution time of a parallel job, $J_j$, assuming a static allocation of $p$ processors. The speedup and efficiency can be derived as:

$$\textit{Speedup:}\quad S_j(p) = \frac{T_j(1)}{T_j(p)}$$

$$\textit{Efficiency:}\quad E_j(p) = \frac{S_j(p)}{p}$$

Speedup can be viewed as the benefit obtained from using some number of processors for the parallel execution of a job and the *efficiency* can be seen as the cost of using those processors. Due to the execution on multiple processors, threads of an application must communicate in order to synchronize and exchange information. The costs of communication and synchronization will increase as the number of processors in use increases and may eventually outweigh the decrease in execution time (benefit). Eager, et al. use an execution time - efficiency profile to illustrate the cost benefit tradeoff [6]. Figure 2.2 contains an example of such a profile. The vertical axis represents the execution time of a parallel job using different numbers of processors while the horizontal axis represents its efficiency. The *knee* of the profile is the point at which the ratio of efficiency to execution time $E(p)/T(p)$ is maximized. That is, the point at which highest possible benefit is attained at the lowest cost.

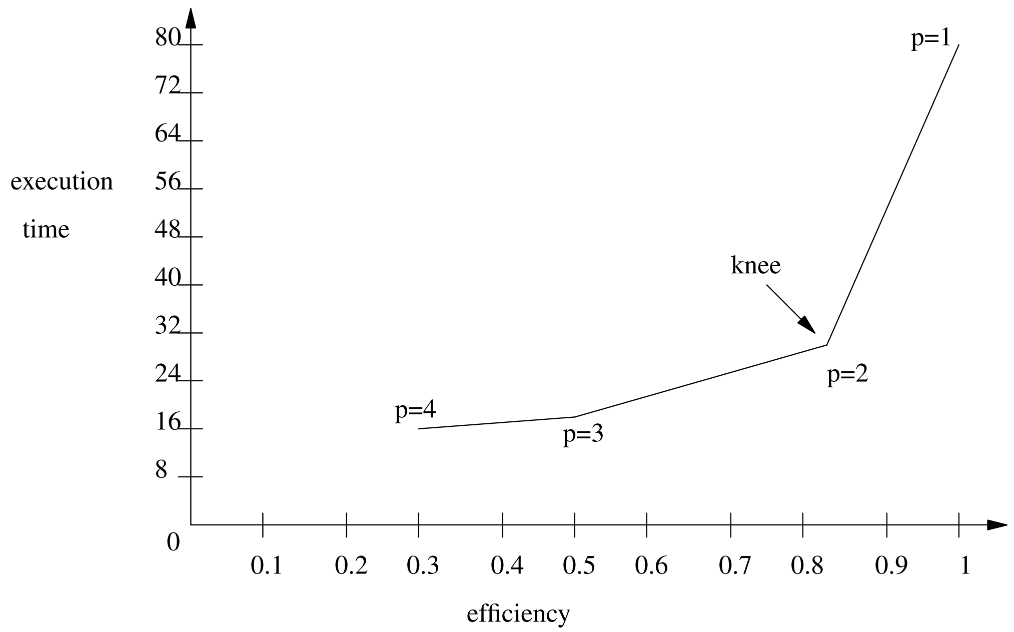



**Figure 2.2: An execution time - efficiency profile**

In our study, for applications whose parallelism profiles have only one phase, we assume that all applications execute with perfect efficiency. For applications whose parallelism profiles have multiple phases, we assume that they execute with perfect efficiency in each individual phase.

### 2.3.3. Work to Be Executed

The work to be executed by a job is defined to be the amount of basic computation performed by the job if it uses one processor. A number of allocation policies have been proposed to schedule jobs according to the increasing order of the amount of work. Majumdar, et al. propose and evaluate scheduling algorithms based on the total work and number of tasks in a parallel job [19]. They have proposed policies called Smallest Number of Processes First (SNPF) and Smallest Cumulative Demand First (SCDF) as well as preemptive version of these algorithms. Research by Sevcik proves that when there are no new arrivals and jobs execute with perfect efficiency, Least Work First (LWF) is optimal [24]. It has also been shown, if jobs execute with perfect efficiency, that Least Remaining Work First (LRWF) yields optimal mean response time when there are new arrivals and preemption is allowed during a job's execution [24][2]. Optimality of these policies does not hold when applications do not execute with perfect efficiency.

Studies have also proposed possible ways to obtain information about the work, efficiency and maximum number of processors a job can use. The study by Brecht has provided three possible ways to estimate expected remaining work [2]. It might be obtained from user supplied estimates, from past execution logs or by cooperating with the run-time system. We believe that in general purpose systems it is impractical to obtain an accurate estimate about job execution time. Therefore, in our research we assume that the scheduler has no information about job execution time at the time of scheduling.

## 2.4. Competitive Analysis

The competitive analysis of algorithms is a measurement of performance for algorithms operating with incomplete information, first introduced by Sleator and Tarjan [27] in the study of a system memory management problem. Applying competitive analysis to the study of scheduling algorithms, the *competitive ratio* of a scheduling algorithm refers to the worst case ratio of the outcome of a policy on a job set to the *optimal outcome* on the same job set over all possible job sets [21]. We denote the value of the objective function for a scheduling algorithm $A$ on a job set, $JS$, with $M_A(JS)$, and denote the value of the objective function for the optimal algorithm which has complete information about this job set with $Opt(JS)$. The competitive ratio, $R_A$, for algorithm $A$ is defined as $\max_{all\ JS}\left(\frac{M_A(JS)}{Opt(JS)}\right)$.

$R_A$ has a lower bound $f(n)$ if $R_A \geq f(n)$. This implies that the objective function for policy $A$ can not be smaller than $f(n) \cdot Opt(JS)$. $R_A$ has an upper bound $g(n)$ if $R_A \leq g(n)$, which implies that, in the worst case, the competitive ratio of policy $A$ can not be greater than $g(n)$. The algorithm $A$ is said to be $g(n) - competitive$ if its upper bound is $g(n)$ [21]. The algorithm is said to be competitive if there exists a constant $k$ for which it is $k - competitive$ [21]. Knowing the competitive ratio of a scheduling policy is indicative of the utility of such a policy.

However, competitive analysis has limitations. The competitive ratio does not distinguish the performance of one policy from another if the objective is minimizing makespan. We will show that any policy that does not leave idle processors will be $2 - competitive$. If the objective is minimizing mean response time, no policy can have a smaller lower bound than $N^{1/3}$ when there are new arriving jobs ($N$ is the total number of jobs in the system) [21].

### 2.5. Dynamic Equipartition Policy

Our analysis concentrates on one particular scheduling policy called Dynamic Equipartition (DEQ), which is reported to yield good mean response time in a number of environments [16][3] and is considered to possess the desired properties of a good scheduler [16][15]. The Equipartition (EQ) policy was first introduced to parallel job scheduling by Tucker and Gupta as a *process control* policy, which limits the total number of processes in the system to be equal to the number of processors and space-shares processors among applications [28]. The main idea behind this approach is to allocate an equal fraction of the processing power to each application if they have enough parallelism and to reallocate processors upon job arrivals and departures. Zahorjan and McCann propose a dynamic equipartition (DEQ) policy, which improves the equipartition (EQ) policy by dynamically adjusting processor allocations with changes in job parallelism [33]. It differs from the equipartition policy in that processor reallocations can occur during job execution rather than just at job arrivals and departures. Assume that all jobs execute with perfect efficiency and that all jobs arrive at the system simultaneously. When the scheduler does not know the job execution time, DEQ guarantees that the job with the least amount of work will finish execution first, by allocating an equal fraction of processing power to each job.

Assume that there are $N$ jobs in a system of $P$ processors. We denote the parallelism of each job, $J_i$, with $P_i$. Let $p_i$ represent the actual number of processors allocated to each job. DEQ is defined as follows: the initial allocation to all jobs is set to zero. Increment the number of processors allocated to each job by one. Any job whose allocation has reached its parallelism drops out. This process continues until either all $P$ processors have been allocated or there are no remaining jobs. In this way the equipartition value is recursively recomputed. As a result, if $P_i$ is less than the equipartition value $\bar{p}$, $J_i$ is allocated $P_i$ processors, otherwise, the number of processors allocated to $J_i$ is equal to $\bar{p}$.

We shall show that the competitive ratio of DEQ is very close to that of the optimal policy in terms of minimizing makespan as well as minimizing the mean response time.

### 2.6. Summary

In this chapter we have outlined the relevant research in literature. Studies in small-scale shared-memory multiprocessor systems have shown that space-sharing is preferable to time-sharing. Simulation and experimental results have demonstrated that dynamic scheduling is preferable to static scheduling in both UMA and NUMA systems. Our analysis will be focused on a simple dynamic space-sharing policy called Dynamic Equipartition (DEQ), which has been stated to possess desirable properties of a good scheduling algorithm [16][15]. DEQ requires only information of the number of jobs in the system and the current parallelism of applications.

Recent studies have shown that making use of certain application characteristics such as the amount of work that a job needs to execute will improve the mean response time. However, we believe that the scheduler is not likely to know job arrival and execution time *a priori*. Competitive analysis is an approach to studying algorithms that operate in environments in which full information about the jobs being executed is not available at the time of scheduling. By comparing the competitive ratio of different policies we can gain a better understanding of the utility of these policies.

# Chapter 3

# The Job, Workload and System Models

A number of studies have used application parallelism and execution time to characterize parallel jobs [29][30][31][17]. In studies by Turek, et al. [29][30][31][17] each parallel job can be allocated an arbitrary number of processors and the job execution time is a known function of the number of processors allocated to it. We use a similar job model in our analysis but our study is conducted in a dynamic environment in which processors may be reallocated among jobs during execution.

### 3.1. Job Model

We define the total number of jobs in the system to be $N$. We characterize a job, $J_i$, with $(P_i, l_i)$, where $1 \le i \le N$. $P_i$ is the instantaneous parallelism of the job, which denotes the number of processors a job is capable of using in its execution. $P_i$ may vary with time. Let $p_i$ be the number of processors actually allocated to each job $J_i$. The parameter $l_i$ represents the execution time of $J_i$ if $p_i$ is always equal to $P_i$. Note that $l_i$ is a theoretic bound on the job execution time. We define $P_i\ l_i$ to be the amount of work that $J_i$ executes, which is denoted by $W_i$. Note that $W_i$ is fixed.

A job can be allocated any number of processors that is no greater than its parallelism (i.e., $p_i \le P_i$). If a job may is allocated fewer processors than $P_i$ its execution time will be lengthened proportionally. This implies that the actual execution time of $J_i$ is at least $l_i$.

### 3.2. Workload Model

We study the problem of minimizing the makespan by first assuming that all jobs arrive at the system simultaneously. This assumption will be relaxed at a later point. In order to distinguish different work-conserving policies, we consider the case where some jobs may execute infinitely because of some error conditions inside the program which lead to an infinite loop.

For the problem of minimizing mean response time, we assume that there are no new job arrivals. We do not consider new arrivals because it has been proven that the lower bound on the competitive ratio for any policy that does not know job execution time is $N^{1/3}$.

### 3.3. System Model

We assume that the system consists of $P$ identical processors and that all jobs arrive at a central job queue. No processors are idle if the job queue contains one or more jobs. At the time of scheduling the scheduler has no information about a job's execution time or the variation of parallelism. That is, at any point in time the scheduler knows only the instantaneous parallelism of the jobs in the system and the number of jobs and processors in the system.

Processors may be preempted from one job and reallocated to another during their execution and we assume that the scheduling overhead and the preemption costs are negligible.

### 3.4. Summary

In this chapter we have described the job, workload and system models upon which our analysis is based. In the following chapters we utilize the models and assumptions described in this chapter to examine the problems of minimizing the makespan and the mean response time for parallel job scheduling. To summarize, the following assumptions will be used throughout this thesis:

**Assumption 3.1:** During each phase of the parallelism profile, jobs execute with perfect efficiency.

**Assumption 3.2:** Each job can be executed on any number of processors that is no more than its parallelism (i.e., $p_i \leq P_i$).

**Assumption 3.3:** The scheduling and preemption overheads are negligible.

**Assumption 3.4:** The scheduler has no *a priori* information about job arrival times.

**Assumption 3.5:** The scheduler has no *a priori* information about job execution times.

**Assumption 3.6:** The scheduler has no *a priori* information about a job's variation in parallelism.

# Chapter 4

# Minimizing the Makespan

In this chapter we use competitive analysis to study dynamic processor allocation policies for parallel jobs whose execution time is not known to the scheduler *a priori*. The objective in this chapter is to minimize the makespan. We assume that the preemption and reallocation of processors can take place during the execution of a job as well as at the arrival and departure of jobs and that the preemption cost is negligible. We begin by considering jobs with only one phase of parallelism (i.e., the parallelism does not change during execution) and devise an optimal competitive scheduling policy for scheduling two parallel jobs on $P$ processors. We then derive upper and lower bounds for scheduling $N$ parallel jobs on $P$ processors.

In this chapter, the *competitive ratio* of a scheduling algorithm refers to the worst case ratio of the makespan of a policy (without full information about applications) to the *optimal makespan* when the scheduler has complete information about the incoming jobs. That is, if $M_A(JS)$ denotes the makespan for a scheduling algorithm $A$ on job set, $JS$, and $Opt(JS)$ denotes the makespan for the optimal algorithm which has complete information about the jobs, the competitive ratio of algorithm $A$ is defined as $\max_{all\ \ JS}\left(\frac{M_A(JS)}{Opt(JS)}\right)$. The goal is to find an algorithm which leads to the minimum competitive ratio.

In Section 4.1 we obtain the minimum makespan $Opt(JS)$ for scheduling jobs on $P$ processors, assuming that complete information about job execution time is known to the scheduler. In Section 4.2 we use competitive analysis to devise a policy, *OptComp*, which yields the optimal competitive ratio for scheduling two parallel jobs when job execution time is *not known* to the scheduler at the time of scheduling. Note that the parallelism of these two jobs does not change during execution. As well, we compare the *OptComp* policy with DEQ. In Section 4.3 we study the problem of scheduling $N$ parallel jobs by combing a generalization of our analysis in Section 4.2 and the results for scheduling sequential jobs on multiprocessors by Graham [10], Hall and Shmoys [13] and Shmoys, et al.[25]. We prove that for scheduling $N$ single-phased parallel jobs on $P$ processors the upper and lower bounds are $2-\frac{1}{P}$. We then consider the cases where the parallelism profiles contain multiple phases and the case where there are new jobs arriving. We prove that the competitive ratio for scheduling $N$ parallel jobs with multiple phased parallelism profile is also $2 - \frac{1}{P}$. As well, we prove that the competitive ratio for scheduling $N$ parallel jobs (with single or multiple phased parallelism profiles) with new arrivals is also $2 - \frac{1}{P}$. In Section 4.4 we consider the case where there are $K$ jobs $(1 \le K < N)$ that may execute infinitely because of some error conditions within the application that lead to an infinite loop. In Section 4.5 we give a brief summary of this chapter.

### 4.1. Optimal Scheduling with Complete Information

In order to simplify the analysis, we first add the following restrictions which will be relaxed later in this chapter.

**Assumption 4.1:** All jobs have single-phased parallelism profiles.

**Assumption 4.2:** All jobs arrive at the system simultaneously.

Observe that the fewer idle processors a scheduler leaves, the lower the makespan. Therefore, a scheduling policy that minimizes makespan will not leave processors idle if there exists a job in the system capable of using it. (Such a policy is also called a *work-conserving* policy in literature [6].) For example, consider scheduling two jobs. Assume that both jobs arrive simultaneously and that a scheduler, $A$, activates two jobs consecutively. Figure 4.1 shows that the makespan for scheduler $A$ is $l_1 + l_2$. The number of processors allocated to $J_1$ and $J_2$ are denoted by $p_1$ and $p_2$ respectively. Since both $P_1$ and $P_2$ are less than $P$, the allocations to $J_1$ and $J_2$ are equal to their parallelism (i.e., $p_1 = P_1$ and $p_2 = P_2$). As a result, when job $J_i$, whose parallelism is $P_i$, is executing, $P - P_i$ processors are idle.

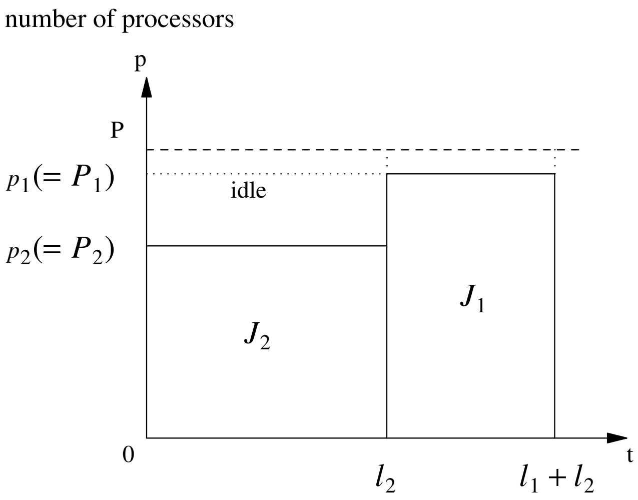



**Figure 4.1: Activating two jobs one by one**

Suppose that there is another scheduler $B$, which allocates $P - P_2$ processors to $J_1$ from time zero to $l_2$ instead of leaving those $P - P_2$ processors idle. $J_1$ will execute $(P - P_2)l_2$ units of work in this period of time. As a result, the makespan for scheduler $B$ is less than $l_1 + l_2$. In Figure 4.2 $P - P_2$ processors are allocated to $J_1$ from time zero to $l_2$. Therefore, when $J_2$ finishes execution at time $l_2$, the remaining amount of work that $J_1$ needs to execute is $P_1 l_1 - (P - P_2)l_2$, which can be completed on $P_1$ processors. Hence the makespan for scheduler $B$ is much less than the makespan for scheduler $A$.

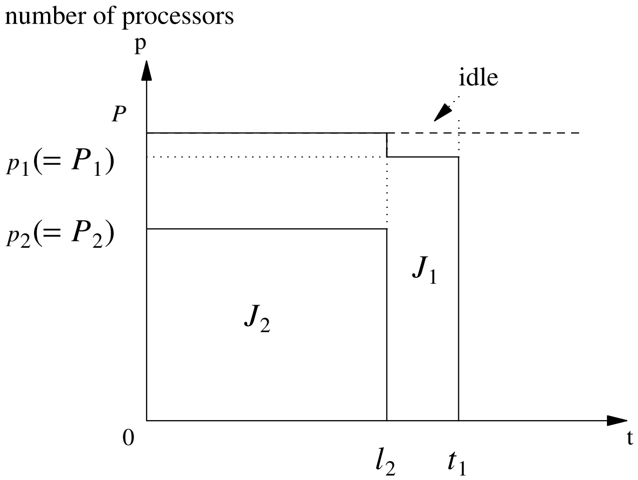



**Figure 4.2: Activating two jobs at the same time**

If we call the regions marked *idle* in Figure 4.1 and Figure 4.2 *holes*, this example illustrates that the smaller the holes are, the lower the makespan produced by the scheduler. This is also true for $N$ jobs. Therefore, our objective in this chapter is to reduce the number of unnecessarily idle processors.

In order to carry out the competitive analysis, we first need to obtain the optimal makespan, $Opt(JS)$, when the scheduler has the information about the parallelism and the execution time of all incoming jobs. We start by considering the case where the job parallelism does not change during execution. Assume that $P_1$ is the parallelism of job $J_1$, and $P_2$ is the parallelism of job $J_2$. Let $l_1$ represent the time $J_1$ needs to complete its execution if it is allocated $P_1$ processors. Similarly $l_2$ is the time required for $J_2$ to execute if allocated $P_2$ processors. There are $P$ processors in the system. The total amount of work to be executed by $J_1$ and $J_2$ can be denoted by $W_1 + W_2$ which is equal to $P_1 l_1 + P_2 l_2$. Let $p_i$ be the number of processors that are actually allocated to $J_i$.

**Theorem 4.1 :** The optimal makespan for scheduling two parallel jobs on $P$ processors is

$$Opt(JS) = \max\left(l_1, l_2, \frac{W_1 + W_2}{P}\right) = \max\left(l_1, l_2, \frac{P_1 l_1 + P_2 l_2}{P}\right).$$

**Proof :** Processor allocation is trivial when $P_1 + P_2 \le P$ : allocate $P_i$ processors to $J_i$. The theorem holds in this case. Therefore, we only need to prove the theorem when $P_1 + P_2 > P$. Without loss of generality we assume that $P_1 \ge P_2$. We consider three possible relations among $P_1, P_2$ and $P$: (1) $P > P_1 \ge P_2$, (2) $P_1 \ge P > P_2$ and (3) $P_1 \ge P_2 \ge P$. Note that there are two extremes for initial allocations:

Least Parallelism First (LPF)

> Allocate $min(P, P_2)$ processors to $J_2$ and allocate the remaining processors to $J_1$. When one job finishes execution, allocate $min(P, P_i)$ processors to the remaining job ($P_i$ is the parallelism of the job still executing).

Most Parallelism First (MPF)

> Allocate $min(P, P_1)$ processors to $J_1$ and allocate the remaining processors to $J_2$. When one job finishes execution, allocate $min(P, P_i)$ processors to the remaining job ($P_i$ is the parallelism of the job still executing).

LPF and MPF deal with two extreme situations and therefore, the initial allocation of processors using any other policy will lie between these two extremes. For a special case where $l_1 = l_2$, the following policy will not leave any processors idle before both of the jobs have finished execution.

Dynamic Proportional Partition (DPP)

The number of processors allocated to each job is proportional to the job parallelism. That is, allocate $\frac{P_1 P}{P_1 + P_2}$ processors to $J_1$, and allocate $\frac{P_2 P}{P_1 + P_2}$ processors to $J_2$. After one job has finished execution, allocate $min(P, P_i)$ processors to the remaining job.

We now prove that the theorem holds.

(1) $P > P_1 \geq P_2$

Obviously, neither $J_1$ nor $J_2$ can use all $P$ processors. The processor allocation varies with the relation between $l_1$ and $l_2$. There are three possible relations between $l_1$ and $l_2$.

(1a) $l_1 = l_2$

Since $P_1 + P_2 > P$, the processors allocated to at least one of the jobs will be less than its parallelism. DPP yields the optimal makespan, since it keeps all processors busy and both jobs finish execution at the same time. The makespan for DPP is $\frac{P_1 l_1 + P_2 l_2}{P}$.

(1b) $l_1 < l_2$

In order to reduce the number of idle processors during the execution of both jobs, the initial allocation of $J_2$ should be $P_2$ and therefore, $J_1$ should be $P - P_2$. In other words, the processors should be initially allocated according to LPF until the jobs reach a point at which the remaining execution time of both jobs are the same. Once they reach that point, the remainder of the execution is identical to case (1a). Therefore, at that point the processors should be allocated according to DPP so that both jobs finish simultaneously. The completion time is $\frac{P_1 l_1 + P_2 l_2}{P}$ if there is work remained after using LPF. Otherwise, $J_1$ will finish execution before $J_2$ and the makespan will be $l_2$.

(1c) $l_1 > l_2$

This case is symmetric to (1b). The initial allocation of processors is done according to MPF. When the remaining execution time of both jobs is equal, the processors should be reallocated according to DPP.

(2) $P_1 \geq P > P_2$

In this case LPF yields the optimal makespan regardless of the values of $l_1$ and $l_2$ because LPF keeps all processors busy until both jobs have finished execution.

Therefore, the makespan is $\dfrac{P_1 l_1 + P_2 l_2}{P}$.

(3) $P_1 \geq P_2 \geq P$

In this case any work-conserving policy yields the optimal makespan since either $J_1$ or $J_2$ by itself can utilize all $P$ processors. In other words, there are no idle processors until both jobs have finished execution. The makespan is $\dfrac{P_1 l_1 + P_2 l_2}{P}$.

The above analysis shows that $\max\left(l_1, l_2, \dfrac{P_1 l_1 + P_2 l_2}{P}\right)$ is an upper bound on the makespan. It is not difficult to see that $\max\left(l_1, l_2, \dfrac{P_1 l_1 + P_2 l_2}{P}\right)$ is also a lower bound on the makespan. Therefore, the optimal makespan for scheduling two jobs is $\max\left(l_1, l_2, \dfrac{P_1 l_1 + P_2 l_2}{P}\right)$.

□

**Theorem 4.2 :** The optimal makespan for scheduling $N$ parallel jobs on $P$ processors is

$$Opt(JS) = \max\left(l_1, l_2, \cdots l_N, \frac{\sum_{i=1}^{N} P_i l_i}{P}\right).$$

**Proof :** When $\sum_{i=1}^{N} P_i \leq P$, the number of processors allocated to each job is equal to its parallelism and therefore each of the job will finish execution within $l_i$ units of time. The makespan in this case is $\max(l_1, l_2, \ldots, l_N)$. (Notice that in this case $\dfrac{\sum_{i=1}^{N} P_i\, l_i}{P} \leq \max(l_1, l_2, \ldots, l_N)$ because not all $P$ processors can be busy throughout $\max(l_1, l_2, \ldots, l_N)$ units of time.)

When $\sum_{i=1}^{N} P_i > P$, the total amount of work to execute of all $N$ jobs is $\sum_{i=1}^{N} P_i\, l_i$. Note that the optimal policy will keep all processors busy, provided that there is work to execute. Therefore, the time required to complete $\sum_{i=1}^{N} P_i\, l_i$ units of work is at least $\dfrac{\sum_{i=1}^{N} P_i\, l_i}{P}$. That is, the optimal makespan, $Opt(JS)$, is at least $\dfrac{\sum_{i=1}^{N} P_i\, l_i}{P}$. Note that in this case the allocation to $J_i$ cannot be equal to its parallelism and therefore its execution time must be more than $l_i$, which implies that $\frac{\sum_{i=1}^{N} P_i\ l_i}{P} > \max(l_1, l_2, \ldots, l_N)$.

Therefore, the optimal makespan for scheduling $N$ parallel jobs on $P$ processors is $Opt(JS) \geq \max\left(l_1, l_2, \cdots l_N, \frac{\sum_{i=1}^{N} P_i l_i}{P}\right)$.

Now we prove that $Opt(JS) \leq \max\left(l_1, l_2, \cdots l_N, \frac{\sum_{i=1}^{N} P_i l_i}{P}\right)$. We combine the previous analysis in this section with the optimal algorithm for scheduling sequential jobs on multiprocessors by Shmoys, et al. [25] to construct the following optimal algorithm for scheduling $N$ parallel jobs on $P$ processors. Without loss of generality we assume that $l_1 \leq l_2 \leq \cdots \leq l_N$. First, we allocate processors to jobs according to the following recursive rules:

(a) (Base case): $l_1 = l_2 = \cdots = l_N$. Allocate processors to jobs in proportion to their parallelism.

(b) (Recursive case): $l_m < l_{m+1} = l_{m+2} = \cdots = l_N$. If $\sum_{i=m+1}^{N} P_i < P$, allocate $P_i$ processors to $J_i$, where $m + 1 \leq i \leq N$. Then recursively allocate the remaining $P - \sum_{i=m+1}^{N} P_i$ processors to the remaining jobs. Otherwise, allocate $\frac{P_i\ P}{\sum_{i=m+1}^{N} P_i}$ to $J_i$, $m + 1 \leq i \leq N$.

This process continues until one of the following cases occurs:

(a) All applications have finished execution.

(b) One job finishes execution, after which the processors are reallocated according to the above rules.

(c) Some job $J_i$ reaches a point at which its remaining execution time becomes equal to that of one or more other jobs (which were not previously the same as $J_i$). We then reallocate the processors using the above recursive rules.

This algorithm for scheduling $N$ jobs guarantees that

$$Opt(JS) \le \max\left(l_1, l_2, \cdots l_N, \frac{\sum_{i=1}^{N} P_i l_i}{P}\right).$$ Therefore, the theorem holds. □

## 4.2. Optimal Competitive Scheduling for Two Jobs on P Processors

In the previous section, we have derived the optimal makespan, $Opt(JS)$, for the problem of scheduling two as well as $N$ parallel jobs when the scheduler has *a priori* information about job execution times. In this section we devise an optimal competitive policy, *OptComp*, for scheduling two parallel jobs when the scheduler does not know job execution times at the time of scheduling. Note that *OptComp* is optimal in the sense that it yields the best possible competitive ratio in the worst case over all possible job sets.

Without loss of generality we assume that $P_1 \ge P_2$. We first examine LPF and MPF policies for the number of idle processors in the worst case. The worst case of MPF occurs when $J_1$ finishes execution before $J_2$ does. After this point, $P - P_2$ processors will be idle. Similarly, the worst case of LPF is when $J_2$ finishes execution first, after which $P - P_1$ processors will be idle. Since there are more idle processors in the worst case using MPF than using LPF, the competitive ratio for MPF is larger than that of LPF. Therefore, if we consider the number of processors allocated to $J_1$ (or $J_2$) to be a continuum with the allocations of MPF and LPF being the two extremes, the policy that yields the minimum competitive ratio will be closer to the LPF end than to MPF end.

DPP can be viewed as a combination of LPF and MPF. Its worst case occurs when $J_1$ finishes execution first since there will be more idle processors than the case when $J_2$ finishes first. The policy with the optimal competitive ratio will be a combination of LPF and DPP in the form of $\alpha\text{LPF} + (1-\alpha)\text{DPP}$. Let $p_1$ be the actual number of processors allocated to $J_1$ and $p_2$ be the actual number of processors allocated to $J_2$. Then $p_1 = \alpha(P - P_2) + (1-\alpha)\dfrac{P_1 P}{P_1 + P_2}$ and $p_2 = \alpha P_2 + (1-\alpha)\dfrac{P_2 P}{P_1 + P_2}$. The policy that yields the minimum competitive ratio should yield the same competitive ratio no matter which job finishes execution first. We can thus determine $\alpha$ and the optimal policy accordingly.

Define $M_\alpha(JS)$ as the makespan of the combined policy, *Policy($\alpha$)*, with parameter $\alpha$ on a job set $JS$. Its competitive ratio is $\max_{JS} \dfrac{M_\alpha(JS)}{Opt(JS)}$. If $J_1$ finishes first using Policy($\alpha$), the optimal allocation policy will be LPF. Therefore, $Opt(JS) = l_2$. As well, $l_1$ and $l_2$ satisfy the following relation: $l_2 = \dfrac{P_1 l_1}{P - P_2}$. In this case we denote the competitive ratio with $R_1(P_1, P_2, \alpha)$, $0 \le \alpha \le 1$. Similarly, $Opt(JS) = l_1$ if $J_2$ finishes first, where $l_1 = \dfrac{P_2 l_2}{P - P_1}$. We denote the competitive ratio in this case with $R_2(P_1, P_2, \alpha), 0 \le \alpha \le 1$. When $R_1(P_1, P_2, \alpha) = R_2(P_1, P_2, \alpha)$, we can obtain a function of $\alpha(P_1, P_2)$ which yields the smallest competitive ratio.

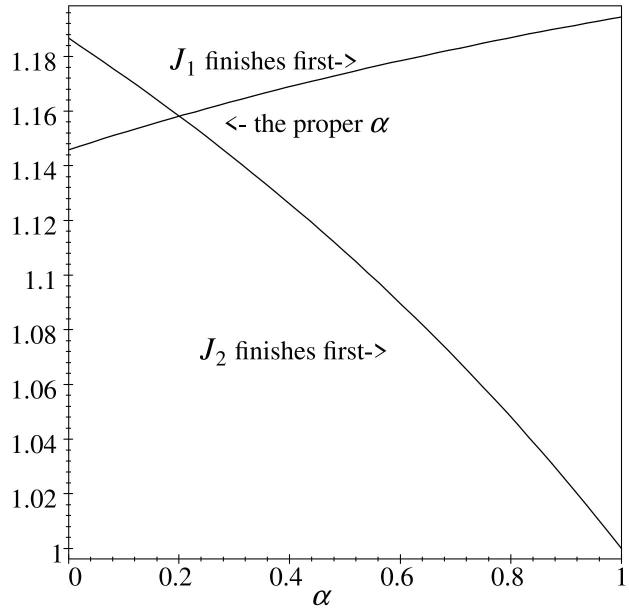



**Figure 4.3: Proper $\alpha$ when $P_1 = 0.75P$ and $P_2 = 0.6P$.**

We have shown in the previous section that the allocation is simple when either $P_1$ or $P_2$ is greater than $P$. Therefore, we only consider the case where $P_1 \le P$, $P_2 \le P$ and $P_1 + P_2 > P$. Note that both $P_1$ and $P_2$ can be represented in terms of $P$. We let $P_1 = mP$, and $P_2 = nP$, $0 \le m \le 1$ and $0 \le n \le 1$. Then $R_1$ and $R_2$ can be represented as functions of $m, n$ and $\alpha$. By setting $R_1 = R_2$, we obtain the proper $\alpha$ which is represented in terms of $m$ and $n$, $\alpha(m, n)$, and thus obtain the minimum competitive ratio. (The solution is presented in the Appendix.) Figure 4.3 illustrates how a proper $\alpha$ value is selected when $m = 0.75$ and $n = 0.6$. The two curves in Figure 4.3 are $R_1$ and $R_2$ respectively. The proper value of $\alpha$ corresponds to the point at which $R_1$ intersects with $R_2$. In this case, $\alpha$ is approximately 0.2.

We substitute the term $\alpha$ in $R_1$ for the proper $\alpha(m, n)$ and obtain $R_1$ in terms of $m$ and $n$. Figure 4.4 illustrates the competitive ratio of this optimal competitive policy, $R_1$ (note that $R_1 = R_2$), as a function of $m$ and $n$. Observe that in Figure 4.4 $m$ ranges from

0.5 to 1. The reason is that in order to satisfy both $P_1 + P_2 > P$ and $P \geq P_1 \geq P_2$, $P_1$ must be larger than $\frac{P}{2}$. That is, $m \geq 0.5$.

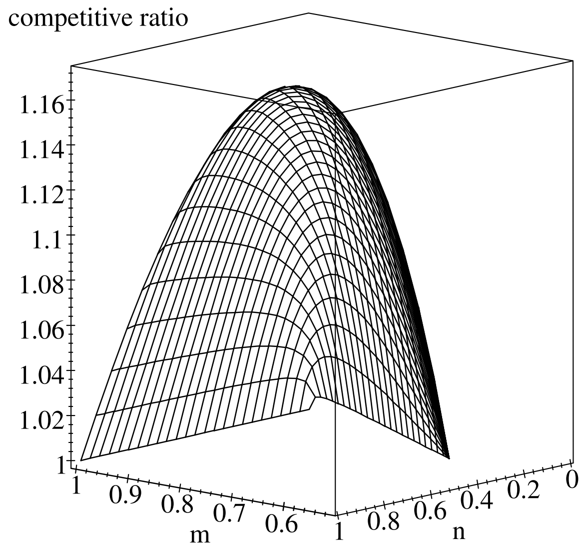



**Figure 4.4: Competitive ratio of the optimal policy**

We can also observe that, across all possible $m$ and $n$ where $0 \leq m, n \leq 1$, $R_1$ reaches the maximum of $4 - 2\sqrt{2}$ (approximately 1.175729) when $m = n = \frac{\sqrt{2}}{2}$. This means that, in the worst case, the makespan for Policy ($\alpha$) is within a constant of 1.175729 times the optimal (i.e., the competitive ratio of Policy($\alpha$) is 1.175729). As well, in the process of deriving this policy we have shown that 1.175729 is the best possible competitive ratio among all possible policies. That is, Policy($\alpha$) has the optimal competitive ratio. Therefore, we call it *OptComp*. Note that *OptComp* is different from *Opt*($JS$). *Opt*($JS$) refers to the policy which has the optimal makespan when it has information about the execution time of the jobs being executed, while *OptComp* refers to the policy which has the optimal competitive ratio when it does not know the job execution time at the time of scheduling.

Now we compare *OptComp* with the dynamic equipartition (DEQ) policy. The purpose of this comparison is to find out the difference between DEQ and *OptComp* for scheduling two parallel jobs when the job execution time is not known to the scheduler *a priori*. We find that the competitive ratio for DEQ is the same as that for *OptComp*. This indicates that DEQ produces the best possible competitive ratio for scheduling two jobs.

### 4.3. Scheduling N Jobs on P Processors

In Section 4.2 we have devised an optimal competitive policy for scheduling two parallel jobs on $P$ processors when the job execution time is not known to the scheduler *a priori*. In this section, we study the problem of scheduling $N$ jobs on $P$ processors in the same environment.

#### 4.3.1. Scheduling N Jobs with Single-phased Parallelism Profiles

Let $P_i$ denote the parallelism of $J_i$ and let $l_i$ denote the time it takes to complete execution if it is allocated $P_i$ processors. We first assume that all N jobs arrive simultaneously, and there are no new arrivals. These assumptions will be relaxed later. We also assume that $\sum_{i=1}^{N} P_i > P$. (It would be trivial to schedule N jobs if $\sum_{i=1}^{N} P_i \le P$.) According to Theorem 4.2 the optimal makespan is:

$$Opt(JS) = \max\left( l_1, l_2, \cdots, l_N, \frac{\sum_{i=1}^{N} P_i l_i}{P} \right)$$

Note that we still have the restrictions that all jobs arrive at the system simultaneously and all jobs have single-phased parallelism profiles. According to the results for sequential job scheduling on multiprocessors by Graham [10], Hall and Shmoys [13] and Shmoys, et al.[25] we have the following theorem.

**Theorem 4.3 :** The competitive ratio of any work-conserving policy for scheduling $N$ jobs with single-phased parallelism profiles is $2 - \frac{1}{P}$.

**Proof :** Suppose that all jobs arrive at the system at time $t_0$. For any work-conserving algorithm $A$ there will be a point in time $t^*$ at which the sum of the parallelism of the remaining jobs in the system is less than $P$ for the first time. Assume that the last job finishes execution at time $t_N$. The execution of these $N$ jobs can be divided into two phases. The first phase is from time $t_0$ to $t^*$, which is the unshadowed region shown in Figure 4.5. Assume that the length of this period is $\tau$. The second phase is from time $t^*$ to $t_N$, which is the shadowed region in Figure 4.5. Suppose that the length of this time span is $\tau'$. Therefore, the makespan (for any work-conserving algorithm), $M_A(JS)$ is $\tau + \tau'$. Let $W_i = P_i l_i$ and let the $W'_i$ be the amount of work executed by $J_i$ at time $t^*$. Then $W'_i$ is no more than $W_i$, $1 \le i \le N$. Therefore, we have:

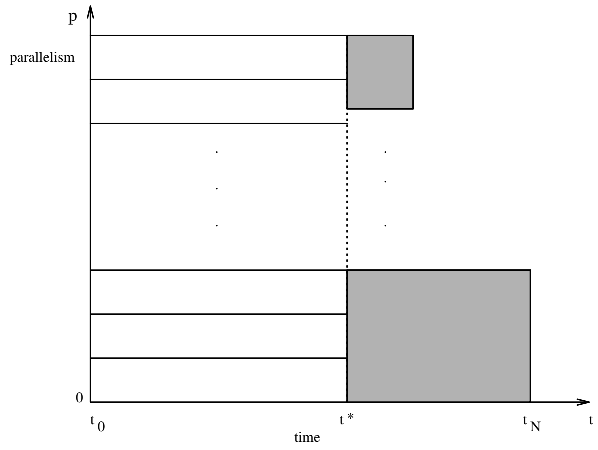



**Figure 4.5: The execution of jobs is divided into $\tau$ and $\tau^{'}$**

$$\tau = \frac{\sum_{i=1}^{N} W_i^{'}}{P} \le \frac{\sum_{i=1}^{N} P_i l_i}{P}.$$

After time $t^*$, the number of processors allocated to each remaining job will be equal to its parallelism, since $\sum_{i=0}^{N} P_i \le P$. Denote the remaining execution time of each jobs after $t^*$ with $r_i$

$$\tau^{'} = max(r_1, r_2, \cdots, r_N) \le max(l_1, l_2, \cdots, l_N)$$

Therefore,

$$M_A(JS) = \tau + \tau^{'} = \frac{\sum_{i=1}^{N} W_i^{'}}{P} + max(r_1, r_2, \cdots, r_N) \le \frac{\sum_{i=1}^{N} W_i}{P} + max(l_1, l_2, \cdots, l_N)$$

Let $L = max(r_1, r_2, \cdots, r_N)$. Note that $Opt(JS) \ge \frac{\sum_{i=1}^{N} P_i\ l_i}{P}$ and $\sum_{i=1}^{N} P_i l_i \ge L + \sum_{i=1}^{N} W_i^{'}$. We have the following relation:

$$Opt(JS) \ge y\,\frac{L + \sum_{i=1}^{N} W_i^{'}}{P} + xL, \ \ where \quad x + y = 1.$$

$$\ge y\,\frac{\sum_{i=1}^{N} W_i^{'}}{P} + (x + \frac{y}{P})L$$

Since $\tau + \tau^{'} = \frac{\sum_{i=1}^{N} W_i^{'}}{P} + L$, we let $y = \frac{1}{2 - \frac{1}{P}}$ so that we can simplify the expression of

$\frac{M_A(JS)}{Opt(JS)}$. Therefore, we have an upper bound on the competitive ratio :

$$\frac{M_A(JS)}{Opt(JS)} = \frac{\tau + \tau'}{Opt(JS)} \leq \frac{L + \frac{\sum_{i=1}^{N} W'_i}{P}}{y\left(L + \frac{\sum_{i=1}^{N} W'_i}{P}\right)} = 2 - \frac{1}{P}.$$

Therefore, an upper bound on the competitive ratio for scheduling $N$ jobs on $P$ processors is $2 - \frac{1}{P}$.

Shmoys, et al. [25] prove that a lower bound on competitive ratio for scheduling $N$ sequential jobs on multiprocessors is $2 - \frac{1}{P}$. Since scheduling sequential jobs on multiprocessors can be viewed as scheduling parallel jobs all of which have parallelism of one, a lower bound on competitive ratio for scheduling $N$ jobs on $P$ processors is also $2 - \frac{1}{P}$. Since both the upper and lower bounds are $2 - \frac{1}{P}$, the competitive ratio for scheduling $N$ parallel jobs on $P$ processors is $2 - \frac{1}{P}$. Therefore, we prove that the competitive ratio of any work-conserving policy for scheduling $N$ parallel jobs on $P$ processors is $2 - \frac{1}{P}$. □

### 4.3.2. Scheduling N Jobs with Multiple Phases of Parallelism

We have studied the problem of scheduling $N$ parallel jobs, whose parallelism does not change during execution. In this section, we relax Assumption 4.1 and consider jobs that execute with multiple phases of parallelism. The analysis here is similar to that used to prove the upper bound on the competitive ratio for scheduling parallel jobs in Section 4.3.1.

Without loss of generality let $J_i$ be the last completed job. We can divide the total execution time into two phases: phase 1 is when the number of processors allocated to $J_i$ is equal to $P_i$. We denote the length of this time period with $\tau$. Phase 2 occurs when the number of processors allocated to $J_i$ is less than $P_i$. We denote the length of this time period with $\tau'$. If we let the total amount of work executed by both jobs during $\tau$ be $W'$, the makespan is $\frac{W'}{P} + \tau'$. From this we obtain the same competitive ratio $\left(2 - \frac{1}{P}\right)$ as in the previous subsection.

#### 4.3.3. Scheduling with New Arrivals

Note that so far we assume that all jobs arrive at the system simultaneously. We now relax Assumption 4.2 and assume that there are new arriving jobs and the scheduler does not have any *a priori* information about the job arrival times. Shmoys, et al. prove that the competitive ratio of scheduling sequential jobs on multiprocessors with new arrivals is $2-\frac{1}{P}$ [25]. Let $J_i$ be the last job to finish execution. Again, in our analysis, we divide the total execution time into two phases: phase 1 occurs while the number of processors assigned to $J_i$ is equal to $P_i$. Phase 2 takes place while the number of processors assigned to $J_i$ is less than $P_i$. Combining the above analysis with the result by Shmoys, et al. [25], we obtain the same competitive ratio $\left(2-\frac{1}{P}\right)$ for scheduling parallel jobs on multiprocessors with new arrivals.

### 4.4. Scheduling with Erroneous Infinite Jobs

We have shown, in the previous section, that any work-conserving policy is $\left(2-\frac{1}{P}\right)-competitive$. That is, the competitive ratio does not distinguish one work-conserving policy from another in minimizing the makespan. Therefore, we introduce the factor of robustness in order to make a distinction. Notice that the previous result is based on the assumption that all the jobs will finish execution within a finite time period. However, it is possible that a job exists that may execute infinitely because of an error condition in the program.

In this section we show that DEQ is robust in the sense that it is $(K+1)-competitive$ in the presence of $K$ infinitely executing jobs while other work-conserving policy such as dynamic proportional policy may have a much larger competitive ratio. A finite job here refers to the job that will finish execution within a finite number of time units. Note that the makespan in this chapter refers to the time which elapses from the arrival of the first job to the departure of the last finite job in a job set. Let $M_A(JS)$ be the completion time of the last finished finite job for the allocation algorithm $A$. Let $Opt(JS)$ denote the optimal completion time when the scheduler has complete information about all jobs. (Note that the optimal scheduler will not execute any infinite job.) The competitive ratio is then defined as $\max_{all\ JS} \frac{M_A(JS)}{Opt(JS)}$.

### 4.4.1. All Jobs Arrive at the System Simultaneously

Assume that there are $K$ infinite jobs in the system. The scheduler does not have any information to distinguish infinite jobs from finite jobs.

**Theorem 4.4 :** If there are no new arrivals, a *lower* bound on the competitive ratio of *any* scheduling policy when there are $K$ infinite jobs in the system is $K + 1$.

**Proof :** Since we need to prove a lower bound, we assume that all of the jobs have parallelism $P$. Note that this can be viewed as scheduling $N$ sequential jobs on a uniprocessor system because each individual job can make use of all processors in the system. Let $N = K + 1$ and assume that there is only one finite job which finishes execution after $t$ units of time. Let $M_A(JS)$ represent the makespan of a scheduler when there is no information available about which $K$ jobs are infinite and let $Opt(JS)$ represent the optimal makespan when the scheduler can distinguish the $K$ infinite job. The makespan $M_A(JS)$ for any scheduling algorithm is at least $(K + 1)t$. Therefore, the competitive ratio $R_A$ is

$$R_A = \frac{M_A(JS)}{Opt(JS)}$$

$$\geq \frac{(K+1)t}{t} = K + 1. \qquad \square$$

In the following section, we prove that an *upper* bound on the competitive ratio for the dynamic equipartition (DEQ) policy is also $K + 1$.

**Theorem 4.5 :** If there are no new arrivals, DEQ has an upper bound of $K + 1$ on its competitive ratio. ($K$ denotes the number of infinite jobs in the system).

**Proof :** Let $J_i$ be the finite job that finishes execution last. Denote its parallelism with $P_i$ and denote its completion time with $T_i$. Since $l_i$ is its execution time when it is allocated $P_i$ processors, obviously $T_i \geq l_i$. $T_i$ is equivalent to the sum of two parts: $T_{equi} + T_{para}$, where $T_{equi}$ represents the period of time during which the number of processors allocated to $J_i$ is equal to the equipartition value $\bar{p}$ and $T_{para}$ represents the time span that the number of processors allocated to $J_i$ is equal to $P_i$. We also divide $l_i$ into two parts: $l_{equi}$ and $l_{para}$, where $l_{para} = T_{para}$, and $l_{equi}$ is the time required for $J_i$ to execute $\bar{p}\ T_{equi}$ units of work if $J_i$ is allocated $P_i$ processors. There are two possibilities:

(a) $T_{para} = 0$.

In this case the number of processors allocated to $J_i$ is equal to the equipartition value $\bar{p}$. Because there is at least one finite job the number of processors allocated to finite jobs at any point in time is at least $\dfrac{P}{K+1}$. Therefore, the completion time of $J_i$ is at most $\dfrac{\dfrac{\sum_{j=1}^{N} P_j l_j}{P}}{K+1}$. Note that $Opt(JS) = \dfrac{\sum_{j=1}^{N} P_j l_j}{P}$ in this case. Therefore, the competitive ratio $R_{DEQ} = \dfrac{T_i}{Opt(JS)} \le K+1$ .

(b) $T_{para} \ne 0$.

In this case the number of processors allocated to $J_i$ is first equal to $\bar{p}$ (which may change after reallocation) and then increased to $P_i$ after another finite job finishes execution. The execution time of $J_i$ is equal to $T_{equi} + T_{para}$. During time $T_{equi}$, no infinite job is allocated more processors than $J_i$ (according to the DEQ policy). The amount of work $J_i$ has executed is $\bar{p}\, T_{equi}$. Therefore, the amount of work infinite jobs execute during that time is no more than $K \cdot (\bar{p} T_{equi})$. Let $W_{finite}$ denote the total amount of work executed by finite jobs. Then the total amount of work executed (including the work infinite jobs have executed) is no more than $K \cdot (\bar{p} T_{equi}) + W_{finite}$, which is equal to $K \cdot (P_i l_{equi}) + W_{finite}$. Therefore, the following relation holds:

$$T_i = T_{para} + T_{equi} \le l_{para} + \frac{K \cdot (P_i\ l_{equi}) + W_{finite}}{P}$$

Note that $Opt(JS)$ satisfies the following two relations: $Opt(JS) \ge l_{para} + l_{equi}$, and $Opt(JS) \ge \dfrac{W_{finite}}{P}$. Therefore, $T_i$ is no more than $(K+1) \cdot Opt(JS)$. That is, an upper bound on the competitive ratio for DEQ is $K+1$. □

We have shown that an upper bound on the competitive ratio of DEQ is the same as the lower bound for any work-conserving policy. It implies that DEQ produces the best possible competitive ratio in this situation. We now show that the competitive ratio for Dynamic Proportional Partition (DPP) policy may be much larger than the upper bound for DEQ. That is, we can find at least one case that the competitive ratio for DPP is larger than $K+1$ ($K$ is the number of infinite jobs).

Suppose that $N = K+1$ ($K$ jobs are erroneous). All of the $K$ erroneous jobs have the same parallelism of $xP$ ($x > 0$). The finite job has a parallelism of $P$. The optimal scheduling policy is to allocate all $P$ processors to the finite job, $J_i$, until it finishes execution. The optimal makespan, $Opt(JS)$, is therefore, $l_i$. However, DPP will allocate $\dfrac{P \cdot P}{K\ xP + P}$ processors to the finite job and allocate $\dfrac{(xP) \cdot P}{K\ Px + P}$ processors to each of the erroneous jobs. As a result, the finite job takes $(Kx+1)\ l_i$ units of time to execute. The competitive ratio is then $Kx + 1$, which can be much larger than that of DEQ. Since $x$ is chosen arbitrarily, dynamic proportional partition does not guarantee a fixed competitive ratio.

### 4.4.2. Scheduling Infinite Jobs and with New Arrivals

In this section we relax Assumption 4.2. We show that the upper and lower bounds obtained in Section 4.4.1 hold when there are new arrivals. We assume that there are $K$ infinite jobs in the system. The scheduler is unable to distinguish infinite jobs from finite jobs.

**Theorem 4.6 :** In a multiprocessor system with new arrivals and $K$ infinite jobs, DEQ is $(K+1) - competitive$.

**Proof :** Note that we are trying to obtain an upper bound on DEQ and that makespan only considers the completion time of the last finite job. Suppose that $J_i$ is the last finite job to finish execution according to the DEQ policy. Let the arrival time of $J_i$ be $t_0$. Again, we divide the execution of $J_i$ into two parts: (a) $T_{para}$, which is the period of time that the number of processors allocated to $J_i$ is equal to $P_i$ and (b) $T_{equi}$, which is the period of time that the allocation of $J_i$ is equal to the equipartition value $\bar{p}$. Let the amount of work $J_i$ executes during $T_{equi}$ be $W_{equi}$. Since there are at most $K$ infinite jobs, the total amount of work executed by the infinite jobs is no more than $K \cdot W_{equi}$ during $T_{equi}$. Let $W_{finite}$ be the total amount of work that finite jobs have executed during $T_{equi}$. The completion time of $J_i$ is no more than $t_0 + T_{para} + \frac{KW_{equi} + W_{finite}}{P}$. That is,

$$T_i \leq t_0 + T_{para} + \frac{K \cdot W_{equi} + W_{finite}}{P} = t_0 + T_{para} + \frac{W_{equi}}{P} + (K-1) \cdot \frac{W_{equi}}{P} + \frac{W_{finite}}{P}$$

Note that for $Opt(JS)$ the following relations hold:

$$Opt(JS) \geq t_0 + T_{para} + \frac{W_{equi}}{P}$$

$$Opt(JS) \geq \frac{W_{finite}}{P}$$

Therefore, $\frac{T_i}{Opt(JS)} \leq K + 1$. That is, DEQ is $(K+1) - competitive$. □

The result in this section shows that the makespan of DEQ with $K$ infinite jobs and new arrivals is still optimal (i.e., $K + 1$). This demonstrates the robustness of DEQ.

### 4.5. Summary

In this chapter we address the problem of minimizing the makespan for scheduling $N$ parallel jobs on multiprocessors. We use competitive analysis to devise a policy which yields the optimal competitive ratio for scheduling two parallel jobs on multiprocessors. This method may be of help for the analysis in generalized situations. We compare the competitive ratio of DEQ with this optimal competitive policy *OptComp* and find that the competitive ratios of DEQ and *OptComp* are the same. Moreover, as we have demonstrated in this chapter, the allocation algorithm of *OptComp* is much more complicated than DEQ. Therefore, for scheduling two parallel applications without information about the execution time DEQ is a good policy.

Our analysis of scheduling $N$ parallel applications considers the cases where there are new job arriving as well as the parallelism of applications varies during execution. Our results show that any work-conserving policy is $\left(2 - \frac{1}{P}\right) - competitive$. That is, any policy which does not leave unnecessarily idle processors yields the optimal makespan.

In order to distinguish different work-conserving policies, we consider the robustness of policies by introducing $K$, which denotes the number of erroneous jobs that may execute infinitely due to programming error. In this case we have shown that DEQ is $(K + 1) - competitive$. That is, the makespan of DEQ is no more than $(K + 1) \cdot Opt(JS)$ in the worst case. Note that $(K + 1)$ is the best possible competitive ratio in this case. We have also demonstrated that the dynamic proportional partition (DPP) policy, which is also a work-conserving policy, its competitive ratio may be much larger than $K + 1$. Our analysis shows that DEQ is robust in the presence of infinitely executing jobs.

We conclude from our analysis that the dynamic equipartition (DEQ) policy is a simple and robust policy for scheduling parallel applications in order to minimize the makespan. In Chapter 5, we show that DEQ is also a good policy for minimizing the mean response time.

# Chapter 5

# Minimizing the Mean Response Time

In this chapter we study the problem of minimizing the mean response time. We focus on the dynamic equipartition (DEQ) policy for scheduling parallel jobs when job execution time is not known *a priori*. DEQ has been widely studied and is considered by some researchers to possess the desirable properties of a good scheduler [16][15]. Our goal here is to understand why DEQ performs well in various environments.

Motwani, et al. have proved that for scheduling $N$ sequential jobs on a uniprocessor without new arrivals a lower bound on the competitive ratio is $\left(2 - \frac{2}{N+1}\right)$. That is, no scheduling algorithm can yield a mean response time that is smaller than $\left(2 - \frac{2}{N+1}\right)$ times the optimal when the scheduler does not have information about the job execution time. This lower bound also applies to the parallel job scheduling on multiprocessors because scheduling sequential jobs on a uniprocessor can be viewed as scheduling parallel jobs on $P$ processors with the parallelism of each of the jobs being equal to $P$. We now prove that an upper bound on the competitive ratio for DEQ is $\left(2 - \frac{2}{N+1}\right)$. That is, the mean response time that DEQ produces in the worst case is no more than $\left(2 - \frac{2}{N+1}\right)$ times the optimal. This result implies that DEQ yields the optimal competitive ratio for scheduling without information about the job execution time.

We do not consider new arrivals, because Motwani, et al. [21] have proved that no scheduling algorithm which does not have *a priori* information about job execution and arrival times can achieve a competitive ratio that is smaller than $N^{1/3}$. We assume that the parallelism profiles of all jobs have only one phase. Although we do not have results pertaining to jobs with multiple phases, we hope that our results here will provide insight into the problem. Therefore, in addition to the assumptions used in Chapter 3, we have the following restrictions:

**Assumption 5.1:** All jobs arrive at the system simultaneously.

**Assumption 5.2:** All jobs have single-phased parallelism profiles.

If the total number of jobs in the system is $N$ and the response time of job $J_i$ is $\beta_i$, where $1 \leq i \leq N$, the mean response time is defined as $\frac{1}{N}\sum_{i=1}^{N}\beta_i$. (Note that the response time of a job refers to the time that elapses from a job's arrival until its departure from the system.) Observe that minimizing $\frac{1}{N}\sum_{i=1}^{N}\beta_i$ is equivalent to minimizing $\sum_{i=1}^{N}\beta_i$, which is also called the *flow time* in literature [4]. Therefore, throughout this chapter our objective function is minimizing the flow time. Suppose that there is a job set: $JS = \{(P_1, l_1), (P_2, l_2), \cdots, (P_N, l_N)\}$, in the system, the competitive ratio for a scheduling policy $S$ in this chapter is defined as the ratio of worst case ratio of the flow time to the minimum flow time on the same job set. That is, if $FT_S(JS)$ denotes the flow time for the policy $S$ and $Opt(JS)$ denotes the optimal flow time, the competitive ratio is: $\max_{all\ JS}\left(\frac{FT_S(JS)}{Opt(JS)}\right)$. However, Turek, et al. have reported [31] that the problem of finding an optimal scheduling algorithm (i.e., with minimum mean response time) is NP-hard in the strong sense. Therefore, we first obtain a lower bound on the optimal scheduling algorithm. Then we compare the mean response time of DEQ with that lower bound and obtain an upper bound on the mean response time for DEQ.

In Section 5.1 we first obtain a lower bound on the optimal flow time. In Section 5.2 we prove that DEQ is $\left(2 - \frac{2}{N+1}\right)-$ *competitive*, when the scheduler can only make use of instantaneous job parallelism to make scheduling decisions. In Section 5.3 we briefly summarize this chapter.

## 5.1. Lower Bounds on Mean Response Time

Note that the execution of a parallel job, $J_i$, can be characterized by $P_i$ and $l_i$, which can be viewed as a rectangle with $P_i$ as its height and $l_i$ as its length. If we let $W_i$ denote the amount of work that $J_i$ executes, which is equal to $P_i \cdot l_i$, $W_i$ corresponds to the area of the rectangle that is associated with $J_i$. Our main result is based on the *squashed area bound*, which is introduced for parallel job scheduling in a static environment [29][30][31]. Denote the total number of processors in the system with $P$. Suppose that a job set, $JS = \{(P_1, l_1), (P_2, l_2), \cdots, (P_N, l_N)\}$, is arranged in increasing order of the amount of work so that $W_i \leq W_{i+1}$, where $1 \leq i < N$. The squashed area bound is defined as [31]:

$$A(JS) = \frac{1}{P}\sum_{i=1}^{N}(N-i+1)P_i l_i \tag{5.1}$$

Let $\sum_{i=1}^{N} l_i = L(JS)$, which is equivalent to the *height bound*, $H_{JS}$, by Turek, et al. [29][30][31]. By setting the parallelism of all jobs to $P$, a new job set can be created from the original job set, $JS$, as $JS_{squash} = \left\{\left(P, \frac{P_1 l_1}{P}\right), \left(P, \frac{P_2 l_2}{P}\right), \ldots, \left(P, \frac{P_N l_N}{P}\right)\right\}$. This is called *squashed area construction* [29]. The squashed area construction does not change $W_i$, $1 \le i \le N$, since it does not change the area associated with each job. Because the original job set is arranged in order of increasing work, the new job set after the squashed area construction is arranged in order of increasing execution time. Figure 5.1 illustrates the squashed area bound after squashed area construction. The vertical axis represents the parallelism of jobs. The horizontal axis represents the job execution time. After the squashed area construction, all jobs have the same parallelism $P$. That is, any job is capable of making use of all of the processors in the system during its execution. In Figure 5.1 the length of the shadowed region is the execution time of the jobs, and the length of the unshadowed region is the time each job spends waiting to be activated. Turek, et al. prove that the optimal flow time for this job set, $Opt(JS)$, is no less than the squashed area bound [29]. That is,

$$A(JS) = A(JS_{squash}) \le Opt(JS). \tag{5.2}$$

They also prove that $Opt(JS) \ge \max(A(JS), L(JS))$.

In order to present our result, we first prove that the sum of the squashed area bound and the height bound is also a lower bound on the optimal flow time. The execution of a job $J_i = (P_i, l_i)$ can be divided into two phases : $J_{i1}$ and $J_{i2}$. We let $J_{i1} = (P_i, l_{i1})$ and $J_{i2} = (P_i, l_{i2})$, so that $l_{i1} + l_{i2} = l_i$, $1 \le i \le N$. Let $t_{i1}$ denote the time at which $J_i$ has executed $P_i\ l_{i1}$ units of work using an optimal policy $Opt(JS)$. Let $t_{i2}$ denote the time at which $J_i$ has completed execution using $Opt(JS)$. Denote the number of processors actually allocated to $J_i$ with $p_i$.

**Lemma 5.1** :

$$Opt(JS) \ge A(JS_{i1}) + L(JS_{i2}). \tag{5.3}$$

**Proof :** During the period of time from $t_{i1}$ to $t_{i2}$, $p_i \le P_i$. Therefore, $t_{i2} - t_{i1} \ge l_{i2}$, $1 \le i \le N$. The optimal flow time is:

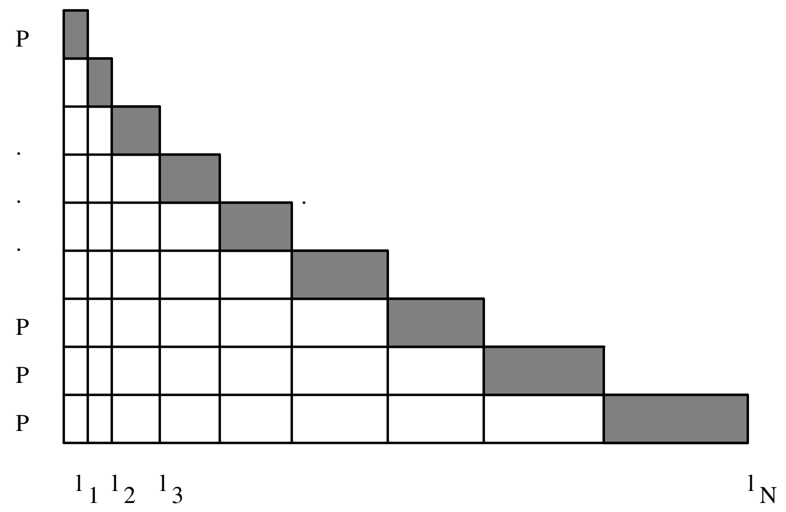


**Figure 5.1: The squashed area bound**

$$Opt(JS) = \sum_{i=1}^{N} t_{i2} \;\geq\; \sum_{i=1}^{N} t_{i1} + \sum_{i=1}^{N} l_{i2}$$

According to (5.2),

$$Opt(JS) \geq A(JS_{i1}) + L(JS_{i2}).$$

Therefore, $A(JS_{i1}) + L(JS_{i2})$ is a lower bound on the optimal flow time. □

Now that we have obtained a lower bound on the optimal flow time we will compare the flow time for DEQ with this lower bound and obtain an upper bound on the competitive ratio for DEQ.

## 5.2. An Upper Bound for DEQ

In this section, we prove that the flow time for the Dynamic Equipartition policy (DEQ) is within a factor of $\left(2 - \frac{2}{N+1}\right)$ of the optimal, where $N$ is the total number of jobs in the system. We assume that $\sum_{i=1}^{N} P_i > P$. (The allocation is trivial if $\sum_{i=1}^{N} P_i \leq P$: allocate $P_i$ processors to each job. The flow time in this case is $\sum_{i=1}^{N} l_i$, which is equivalent to $L(JS)$.)

Using DEQ each job will be allocated an equal fraction of processing power, provided that it has enough parallelism. Therefore, $p_i$ must be either equal to the job parallelism, $P_i$, or equal to the equipartition value, $\bar{p}$, which is recursively recomputed according to the definition presented in Section 2.5. As a result, we divide all of the jobs in the system into two subsets :

(1) $JS_{para}$, in which $p_i$ is equal to the job parallelism (i.e., $p_i = P_i$). Denote the number of jobs that belong to $JS_{para}$ with $k_p$.

(2) $JS_{equi}$, in which each job is allocated $\bar{p}$ processors (i.e., $p_i = \bar{p} < P_i$). Denote the number of jobs that belong to $JS_{equi}$ with $k_e$.

We use the following observations to derive an upperbound for DEQ:

(a) The parallelism of any job that belongs to $JS_{para}$, $P_i$, is less than $\bar{p}$. That is, $(\forall J_i \in JS_{para}) \quad p_i = P_i < \bar{p}$.

(b) The number of processors allocated to any job that belongs to $JS_{equi}$ is at least $\frac{P}{N}$. That is,

$$(\forall J_i \in JS_{equi}) \quad p_i = \bar{p} \geq \frac{P}{N} . \tag{5.4}$$

(c) The sum of the processors allocated to all the jobs that belong to $JS_{equi}$ and the processors allocated to all the jobs that belong to $JS_{para}$ is $P$. That is,

$$\sum_{i \in JS_{para}} P_i + k_e \bar{p} \leq P.$$

(d) If $p_i = P_i$ for $1 \leq i \leq N$, $p_i$ will not change until $J_i$ completes execution. If $p_i = \bar{p}$, $p_i$ may increase up to $P_i$ at a later point of time.

Without loss of generality, the execution of any job, $J_i$ $(1 \leq i \leq N)$, can be divided into two phases :

(1) Phase *e*: when it is allocated a equal fraction of processing power according to DEQ. Denote the time span that $J_i$ executes in this state with $T_{iequi}$.

(2) Phase *f*: when it is allocated $P_i$ processors. Denote the time span that $J_i$ executes in this state with $T_{ipara}$.

Note that either $T_{iequi}$ or $T_{ipara}$ can be zero but not both. Lemma 5.1 is equivalent to $Opt(JS) \geq A(JS(e)) + L(JS(f))$.

Denote the flow time of DEQ for job set $JS$ with $FT_{DEQ}(JS)$. We now prove that $FT_{DEQ}(JS) \leq \left(2 - \frac{2}{N+1}\right) \cdot A(JS(e)) + \left(2 - \frac{2}{N+1}\right) \cdot L(JS(f))$. Combining this result with Lemma 5.1, we show that $FT_{DEQ}(JS) \leq \left(2 - \frac{2}{N+1}\right) \cdot Opt(JS)$.

**Theorem 5.1 :** Let $A(JS(e)) = \frac{1}{P}\sum_{i=1}^{N}(N-i+1)P_i T_{iequi}$ and $L(JS(f)) = \sum_{i=1}^{N} T_{ipara}$.

$$FT_{DEQ}(JS) \le \left(2 - \frac{2}{N+1}\right)\cdot A(JS(e)) + \left(2 - \frac{2}{N+1}\right)\cdot L(JS(f)). \tag{5.6}$$

**Proof :** We use mathematical induction in terms of the number of jobs, $N$, to prove this theorem. Let $k$ be the number of jobs, where $1 \le k \le N$.

(a) Base Case ($k = 1$):
If $P_1 \le P$, the scheduler will allocated $P_i$ processors to this job. Therefore, $FT_{DEQ}(JS) = l_1$. Since in this case $A(JS(e)) = 0$ and $L(JS(f)) = l_1$, the inequality (5.6) holds.
If $P_1 > P$, all $P$ processors will be allocated to this job. Therefore, $FT_{DEQ}(JS) = \frac{P_1 l_1}{P}$. Since $A(JS(e)) = \frac{P_1 l_1}{P}$ and $L(JS(f)) = 0$, inequality (5.6) also holds in this case. Therefore, the theorem holds when $k = 1$.

(b) Induction Step:
*Induction Hypothesis*: For all $k$ (the number of jobs in *JS* whose length is not zero) such that $1 \le k < N$, $FT_{DEQ}(JS) \le \left(2 - \frac{2}{k+1}\right)\cdot A(JS(e)) + \left(2 - \frac{2}{k+1}\right)\cdot L(JS(f))$.
We now prove that it is also true when $k = N$. Since $A(JS(e))$ and $L(JS(f))$ are not affected by the jobs whose length is zero, we only need to prove the claim for the case where every job in *JS* is of non-zero length. Without loss of generality we assume that after executing for a period of time, $\tau$, $J_1$ finishes execution and all processors are then reallocated according to DEQ. Denote the remaining execution time of $J_i$ with $r_i$. For the remaining $(N-1)$ jobs we have the following relations:

$$\forall J_i \in JS_{para},\ \ r_i = l_i - \tau \quad \text{and}$$

$$\forall J_i \in JS_{equi},\ \ r_i = l_i - \frac{\tau \bar{p}}{P_i}.$$

Denote the jobs that originally belong to $JS_{equi}$ and are now allocated $p_i = P_i$ processors with $JS'_{para}$ ($JS'_{para} \subseteq JS_{equi}$). Let $JS'_{equi} = JS_{equi} - JS'_{para}$. By the time $J_1$ finishes execution, all jobs have been in the system for $\tau$ units of time. Therefore,

$$FT_{DEQ}(JS) \ \le \ N\tau +$$

$$FT_{DEQ}\left(\left\{(P_i, l_i - \frac{\tau\bar{p}}{P_i}): J_i \in JS_{equi}\right\} \cup \left\{(P_i, l_i - \tau): J_i \in JS_{para}\right\}\right) \tag{5.7}$$

According to the induction hypothesis, the second term of the right hand side of (5.7) satisfies the following relation:

$$\begin{aligned} &FT_{DEQ}\left(\left\{(P_i, l_i - \frac{\tau\bar{p}}{P_i}): J_i \in JS_{equi}\right\} \cup \left\{(P_i, l_i - \tau): J_i \in JS_{para}\right\}\right) \\ &\leq C \cdot A\left(\left\{(P_i, l_i(e) - \frac{\tau\bar{p}}{P_i}): J_i \in JS_{equi}\right\}\right) + C \cdot L\left(\left\{(P_i, l_i(f)): J_i \in JS_{equi}\right\}\right) \\ &\qquad + C \cdot \sum_{i \in JS_{para}} (l_i - \tau). \end{aligned} \tag{5.8}$$

Note that $JS_{equi} = JS'_{equi} \cup JS'_{para}$. For the job set $JS'_{equi}$ and $JS'_{para}$, the following relations hold:

$$\forall J_i \in JS'_{equi},\ l_i(e) > \frac{\tau\bar{p}}{P_i} \text{ and}$$

$$\forall J_i \in JS'_{para},\ l_i(e) = \frac{\tau\bar{p}}{P_i}.$$

Therefore,

$$A\left(\left\{\left(P_i, l_i(e) - \frac{\tau\bar{p}}{P_i}\right): J_i \in JS_{equi}\right\}\right) = A\left(\left\{\left(P_i, l_i(e) - \frac{\tau\bar{p}}{P_i}\right): J_i \in JS'_{equi}\right\}\right) \tag{5.9}$$

In order to make use of the squashed area bound, we arrange the jobs in $JS'_{equi}$ as $J_{j_1}, J_{j_2}, \ldots, J_{j_{k_e'}}$, $k_e' = |JS'_{equi}|$. in increasing order of the remaining work where $J_{j_i} \in JS'_{equi}$. Arrange the jobs in $JS'_{para}$ as $J_{m_1}, J_{m_2}, \ldots, J_{m_{k_p'}}$, $k_p' = |JS'_{para}|$, in increasing order of the amount of work. According to (5.1), the squashed area bound of $JS'_{equi}$ is :

$$A\left(\left\{\left(P_i, l_i(e) - \frac{\tau\bar{p}}{P_i}\right): J_i \in JS'_{equi}\right\}\right)$$

$$= \frac{1}{P}\sum_{i=1}^{k_e'}(k_e' - i + 1)P_{j_i}\left(l_{j_i}(e) - \frac{\tau\bar{p}}{P_{j_i}}\right)$$

$$= \frac{1}{P}\sum_{i=1}^{k_e'}(k_e' - i + 1)P_{j_i}l_{j_i}(e) \; - \; \frac{1}{P}\sum_{i=1}^{k_e'}(k_e' - i + 1)\tau\bar{p}$$

$$= \; A\left(\left\{(P_i, l_i(e)) \colon J_i \in JS'_{equi}\right\}\right) - \; \frac{k_e'(k_e' + 1)}{2P}\,\tau\bar{p}$$

$$= \; A(JS'_{equi}(e)) - \frac{k_e'(k_e' + 1)}{2P}\,\tau\bar{p}. \tag{5.10}$$

Since $L(JS_{para}(f)) = \sum_{i \, \in \, JS_{para}} l_i, \quad \sum_{i \, \in \, JS_{para}} (l_i - \tau) = L(JS_{para}(f)) - k_p\tau.$ Therefore, we have :

$$L(JS_{equi}(f)) + \sum_{i \, \in \, JS_{para}} (l_i - \tau) = L(JS(f)) - k_p\tau.$$

For simplicity let $C = 2 - \dfrac{2}{N+1}$. Combining this equation with (5.7), (5.8), (5.9) and (5.10), we obtain the upper bound on $FT_{DEQ}(JS)$ :

$$\begin{aligned} FT_{DEQ}(JS) \leq \; & N\tau + C \cdot A(JS'_{equi}(e)) + C \cdot L(JS(f)) \\ & - C \cdot k_p\tau - C \cdot \frac{k_e'(k_e' + 1)}{2P}\,\tau\bar{p} \end{aligned} \tag{5.11}$$

Note that $A(JS_{equi}(e)) = A(JS'_{equi}(e)) + \dfrac{1}{P}\sum_{i=1}^{k_p'}(k_e' + k_p' - i + 1)\tau\bar{p}$. Inequality (5.11) is equivalent to :

$$\begin{aligned} FT_{DEQ}(JS) \leq \; & N\tau + C \cdot A(JS_{equi}(e)) - \; \frac{C}{P}\sum_{i=1}^{k_p'}(k_e' + k_p' - i + 1)\tau\bar{p} \\ & + C \cdot L(JS(f)) - C \cdot k_p\tau - C \cdot \frac{k_e'(k_e' + 1)}{2P}\,\tau\bar{p} \end{aligned}$$

Since $C \cdot A(JS(e)) + C \cdot L(JS(f)) = C \cdot A(JS_{equi}(e)) + C \cdot L(JS(f))$, it will be equivalent to prove that the right hand side of the above inequality is no more than $C \cdot A(JS_{equi}(e)) + C \cdot L(JS(f))$ in order to prove the theorem. That is,

$$N\tau + C \cdot A(JS_{equi}(e)) - \frac{C}{P}\sum_{i=1}^{k_p{}'}(k_e{}' + k_p{}' - i + 1)\tau\bar{p} + C \cdot L(JS(f))$$

$$- C \cdot k_p\tau - C \cdot \frac{k_e{}'(k_e{}' + 1)}{2P}\tau\bar{p} \ \le \ C \cdot A(JS_{equi}(e)) + C \cdot L(JS(f)) \qquad (5.12)$$

Note that (5.12) is equivalent to

$$N\tau - C \cdot k_p\tau - C \cdot \frac{\tau}{P}\sum_{i=1}^{k_p{}'}(k_e{}' + k_p{}' - i + 1)\bar{p} - C \cdot \frac{k_e{}'(k_e{}' + 1)}{2P}\tau\bar{p} \le 0$$

In order to prove the above relation, it is both sufficient and necessary to prove that:

$$N \le C \cdot k_p + \frac{C}{P}\sum_{i=1}^{k_p{}'}(k_e{}' + k_p{}' - i + 1)\bar{p} + C\,\frac{k_e{}'(k_e{}' + 1)}{2P}\,\bar{p},$$

which is equivalent to

$$NP \le Ck_pP + C\sum_{i=1}^{k_p{}'}(k_e{}' + k_p{}' - i + 1)\bar{p} + \frac{C}{2}\,k_e{}'(k_e{}' + 1)\bar{p} \qquad (5.13)$$

Note that $k_e + k_p = N$. (5.13) is equivalent to :

$$k_eP \le (C - 1)k_pP + \frac{C}{2}\,k_e(k_e + 1)\bar{p}$$

Since $C = 2 - \dfrac{2}{N + 1}$, the above inequality is equivalent to

$$k_eP\ (N + 1) \le (N - 1)k_pP + Nk_e(k_e + 1)\bar{p}. \qquad (5.14)$$

Again, because $k_e + k_p = N$, the right hand side of the inequality is equivalent to:

$$Pk_p(N - 1) + N\bar{p}k_e(k_e + 1) \ = \ Pk_p(k_e + k_p - 1) + N\bar{p}k_e(k_e + 1)$$

According to (5.4) in observation (b), we have the following relation:

$$Pk_p(k_e + k_p - 1) + N\bar{p}k_e(k_e + 1) \ge Pk_pk_e + Pk_p(k_p - 1) + Pk_e(k_e + 1)$$

Note that $k_p(k_p - 1) \ge 0$. Therefore,

$$Pk_p(k_e + k_p - 1) + N\bar{p}k_e(k_e + 1) \ge Pk_e(k_p + k_e + 1) = Pk_e(N + 1).$$

Therefore, the theorem holds when $k = N$.

That is,

$$FT_{DEQ}(JS) \le \left(2 - \frac{2}{k+1}\right) A(JS(e)) + \left(2 - \frac{2}{k+1}\right) L(JS(f)) \;\text{ for }\; 1 \le k \le N. \qquad \square$$

Note that a lower bound on the optimal flow time, $Opt(JS)$, is $A(JS(e)) + L(JS(f))$ according to Lemma 5.1. That is,

$$A(JS(e)) + L(JS(f)) \;\le\; Opt(JS).$$

Since $FT_{DEQ}(JS) \le \left(2 - \frac{2}{N+1}\right) \cdot (A(JS(e)) + L(JS(f)))$ according to Theorem 5.1, the following relation holds:

$$FT_{DEQ}(JS) \le \left(2 - \frac{2}{N+1}\right) \cdot Opt(JS).$$

That is, an upper bound on the competitive ratio for DEQ is $2 - \frac{2}{N+1}$. This result implies that in the worst case the mean response time of DEQ is within a factor of $\left(2 - \frac{2}{N+1}\right)$ of the optimal.

## 5.3. Summary

In this chapter, we have shown that DEQ is $\left(2 - \frac{2}{N+1}\right)$ – *competitive* in terms of minimizing the mean response time. That is, in the worst case, the mean response time of DEQ over all possible job sets is no more than $\left(2 - \frac{2}{N+1}\right)$ times the optimal.

Note that in the same situation a lower bound on the competitive ratio for scheduling parallel jobs is $\left(2 - \frac{2}{N+1}\right)$ [21]. Our result has shown that an upper bound on the competitive ratio of DEQ is equal to the lower bound on the competitive ratio of any policy that does not utilize job execution time to make scheduling decisions. This implies that DEQ yields the optimal competitive ratio when scheduling without new arrivals and without knowledge of job execution times.

# Chapter 6

# Conclusions and Future Research

## 6.1. Conclusions

The goals of this thesis are to use competitive analysis to study dynamic processor allocation policies under realistic job and workload models and provide insight into the utility of these policies. In pursuit of these goals we use competitive analysis to devise and evaluate scheduling techniques for scheduling parallel jobs in a multiprocessor environment.

Our study is conducted in a dynamic environment in which processors may be preempted and reallocated during job execution. We assume that the scheduler has no information about the job arrival and execution times. We address the problems of minimizing the makespan and the mean response time respectively.

**• Minimizing the makespan:**

We use competitive analysis to devise an optimal competitive policy, *OptComp*, for scheduling two parallel jobs on multiprocessors. This suggests a new method to devise optimal competitive scheduling policies (i.e., policies which yields optimal competitive ratio). Compared with *OptComp*, the Dynamic Equipartition policy (DEQ) does not yield the optimal competitive ratio for scheduling two parallel jobs. However, the maximum relative ratio of DEQ to *OptComp* is only $1.175729$. Note that DEQ is much simpler than *OptCompt*. Therefore, we conclude that DEQ is a good policy for scheduling two parallel jobs.

We also extend the previous results for scheduling sequential jobs to parallel job scheduling. We have shown that both upper and lower bounds on the competitive ratio for any work-conserving policy are $(2 - \frac{1}{P})$ in the cases where parallelism profiles have multiple phases and there are new arrivals. This implies that any work-conserving policy yields the optimal competitive ratio if the objective function is minimizing the makespan. Note that these results are identical to their sequential counterparts [8][13][25].

In order to distinguish different work-conserving policies we introduce the notion of robustness by considering the case where there are jobs that may execute infinitely due to programming errors. Our result shows that DEQ yields the optimal competitive ratio, while the dynamic proportional partition (DPP) policy (which is also a work-conserving policy) does not produce a fixed competitive ratio. Our results show that DEQ is robust in the presence of infinite jobs.

- **Minimizing the mean response time:**

  Using competitive analysis we study the problem of scheduling $N$ parallel jobs in order to minimize the mean response time. Our study considers scheduling parallel jobs with single-phased parallelism profiles. We improve the previous results of scheduling parallel jobs in a static environment [31][30][29][17] and prove that in a dynamic environment an upper bound on the competitive ratio of DEQ is $\left(2 - \frac{1}{N+1}\right)$. Since a lower bound on the competitive ratio for any policy which does not know the job execution time is also $\left(2 - \frac{1}{N+1}\right)$, our result implies that DEQ yields the optimal competitive ratio.

Recent studies have shown that using the information of the work to execute will improve the mean response time [19][24][2][11]. However, it may not be possible to obtain the precise information about the job execution time. Instead, an estimate of execution time could be used when making scheduling decisions [2]. Our results have shown that DEQ leads to the best possible competitive ratio when the job execution time is not available to the scheduler at the time of scheduling. Note that DEQ does not require any information about job execution time, and it is easy to implement in real systems. Therefore, we conclude that DEQ is a competitive scheduling policy.

Our results shows that when complete information about job characteristics is not available to the scheduler, it is advisable to space-share processors equally among all applications provided that they have enough parallelism.

### 6.2. Direction for Future Research

Our analysis in scheduling $N$ jobs with single-phased parallelism profiles has provided insight into the analysis of this problem for parallel jobs with multiple-phased parallelism profiles. It is interesting to carry our future research in this direction.

# Appendix

The following are the equations for $\alpha$ and $R_1$ for the optimal competitive policy *Policy*($\alpha$) discussed in Section 4.2. Note that $\alpha$ is obtained by setting $R_1 = R_2$. They are presented in terms of $x$ and $y$ to simplify the formulae. Since $P_1$ and $P_2$ can be expressed as a fraction of $P$, we let $P_1 = m\ P$ and $P_2 = n\ P$.

$$
\begin{aligned}
x \ = \ & 4\,n\,m - 8\,n^2\,m - 8\,n\,m^2 + 5\,n^3\,m + 10\,n^2\,m^2 - 3\,n^3\,m^2 - 3\,n^2\,m^3 + 5\,n\,m^3 - n^4\,m \\
& - m^4\,n + (4\,n^3\,m + 8\,n^2\,m^2 + 4\,n\,m^3 - 48\,n^3\,m^2 - 48\,n^2\,m^3 - 16\,n^4\,m - 16\,m^4\,n \\
& + 104\,n^4\,m^2 + 160\,n^3\,m^3 - 232\,n^4\,m^3 - 232\,n^3\,m^4 + 104\,n^2\,m^4 - 104\,n^5\,m^2 - 104\,n^2\,m^5 \\
& + 160\,n^5\,m^3 + 230\,n^4\,m^4 + 49\,n^6\,m^2 + 160\,n^3\,m^5 + 49\,n^2\,m^6 - 50\,n^6\,m^3 - 100\,n^5\,m^4 \\
& - 10\,n^7\,m^2 - 100\,n^4\,m^5 - 50\,n^3\,m^6 + 15\,n^6\,m^4 + 20\,n^5\,m^5 + 6\,n^7\,m^3 + 15\,n^4\,m^6 \\
& + 6\,n^3\,m^7 - 10\,n^2\,m^7 + n^8\,m^2 + m^8\,n^2 + 24\,n\,m^5 - 16\,n\,m^6 + 4\,n\,m^7 \\
& + 24\,n^5\,m - 16\,n^6\,m + 4\,n^7\,m\,)^{1/2}
\end{aligned}
$$

$$
y \ = \ n\,m + n^2\,m^2 - n^3\,m + n\,m^3 - 2\,n\,m^2 - n^2 + 2\,n^3 - n^4
$$

$$
\alpha \ = \ \frac{x}{2\,y}
$$

Replacing $\alpha$ in $R_1$ (or equivalently $R_2$ ), we have :

$$
R_1 \ = \ \frac{\dfrac{x\,n}{2\,y} - 2\,m + \dfrac{x\,m}{2\,y} + 1 - 2\,n - \dfrac{x}{2\,y} + n^2 + n\,m}{-\dfrac{x\,n}{2\,y} + \dfrac{x\,n^2}{2\,y} + \dfrac{x\,n\,m}{2\,y} - m}
$$

# Bibliography


[1] K. P. Belkhale and P. Banerjee, ‘‘Approximate Algorithms for the Partitionable Independent Task Scheduling Problem’’, *1990 International Conference on Parallel Processing*, pp. 72-75, 1990.

[2] T. B. Brecht, ‘‘Multiprogrammed Parallel Application Scheduling in NUMA Multiprocessors’’, CSRI-303, Computer Systems Research Institute, University of Toronto, Toronto, April, 1994.

[3] S. Chiang, R. K. Mansharamani, and M. K. Vernon, ‘‘Use of Application Characteristics and Limited Preemption for Run-To-Completion Parallel Processor Scheduling Policies’’, *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 33-44, Nashville, TN, May, 1994.

[4] E. G. Coffman, **Computer and Job-Shop Scheduling Theory**, John Wiley and Sons, New York, 1976.

[5] X. Deng and C. H. Papadimitriou, ‘‘Competitive Distributed Decision-Making’’, *Information Processing 92*, *Vol.* I, pp. 350-356, Elsevier Science Publishers B. V., 1992.

[6] 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.

[7] A. Feldmann, J. Sgall, and S. H. Teng, ‘‘Dynamic Scheduling on Pararrel Machines’’, *Foundations of Computing Science*, pp. 111-120, 1991.

[8] M. R. Garey and R. L. Graham, ‘‘Bounds for Multiprocessor Scheduling with Resource Constraints’’, *SIAM Journal of Computing*, *Vol.* 4, *No.* 2, pp. 187-200, June, 1975.

[9] T. Gonzalez and S. Sahni, ‘‘Preemptive Scheduling of Uniform Processor Systems’’, *Journal of the ACM*, *Vol.* 25, *No.* 1, pp. 92-101, January, 1978.

[10] R. L. Graham, ‘‘Bounds for Certain Multiprocessor Anomalies’’, *Bell System Technical Journal*, *Vol.* 45, pp. 1563-1581, 1966.

[11] K. Guha, **Using Parallel Program Characteristics in Dynamic Multiprocessor Allocation Policies**, M.Sc. Thesis, Department of Computer Science, York University, Toronto, Ontario M3J 1P3, May, 1995.

[12] 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.

[13] L. Hall and D. B. Shmoys, ‘‘Approximation Schemes for Constrained Scheduling Problems’’, *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pp. 134-141, October, 1989.

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

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

[16] 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.

[17] W. Ludwig and P. Tiwari, ‘‘The Power of Choice in Scheduling Parallel Tasks’’, TR1190, Computer Science Department, University of Wisconsin-Madison, Madison, November, 1993.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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, May, 1989.

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

[25] D. B. Shmoys, J. Wein, and D. P. Williamson, ‘‘Scheduling Parallel Machines On-line’’, *Foundations of Computer Science*, pp. 131-140, 1991.

[26] D. D. Sleator, ‘‘A 2.5 Times Optimal Algorithm for Packing in Two Dimensions’’, *Information Processing Letters*, *Vol*. 10, *No*. 1, pp. 37-40, February, 1980.

[27] D. D. Sleator and R. E. Tarjan, ‘‘Amortized Efficiency of List Update and Paging Rules’’, *Communications of the ACM*, *Vol*. 28, *No*. 2, pp. 202-208, 1985.

[28] 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.

[29] J. Turek, W. Ludwig, J. L. Wolf, L. Fleischer, P. Tiwari, J. Glasgow, U. Schwiegelshohn, and P. S. Yu, ‘‘Scheduling Parallelizable Tasks to Minimize Average Response Time’’, *Proceedings of the 6th Annual Symposium on Parallel Algorithms and Architectures*, June, 1994.

[30] J. Turek, U. Schwiegelshohn, J. L. Wolf, and P. S. Yu, ‘‘A Significantly Smarter Bound for a Slightly Smarter SMART Algorithm’’, RC 19422 (84462), IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, February, 1994.

[31] J. Turek, U. Schwiegelshohn, J. L. Wolf, and P. S. Yu, ‘‘Scheduling Parallel Tasks to Minimize Average Response Time’’, *Proceedings of the 5th SIAM Symposium on Discrete Algorithms*, pp. 112-121, May, 1994.

[32] C. S. Wu, Processor Scheduling in Multiprogrammed Shared-Memory NUMA Multiprocessors, M.Sc. Thesis, University of Toronto, Toronto, Ontario, October, 1993.

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