

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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

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

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

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

U·M·I

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

Order Number 9111827

Job scheduling on a hypercube

Zhu, Yahui, Ph.D.

The Ohio State University, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

JOB SCHEDULING ON A HYPERCUBE

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By

Yahui Zhu, B.S., M.S.

* * * * *

The Ohio State University

1990

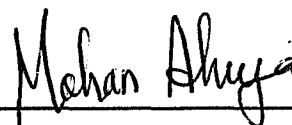
Dissertation Committee:

Prof. Mohan Ahuja

Prof. Ponnuswamy Sadayappan

Prof. Judith D. Gardiner

Approved by



Adviser

Department of Computer
and Information Science

To my family

ACKNOWLEDGMENTS

I would like to express my greatest thanks to my advisor, Professor Mohan Ahuja, who guided my research with deep insights and great encouragement. He was always available whenever I needed his help. To Professor P. Sadayappan and Professor J. Gardiner, members of my advisory committee, I express my sincere appreciation for their helpful comments.

I also like to express my sincere thanks to Professor T. Long, who guided one of my minor studies in Computer Theory and whose graceful teaching style I will never forget. My sincere thanks also go to Professor T. Carlson of Mathematics Department, with whom I did the other minor study in Applied Logics. I also wish to thank the following professors: M. Liu, G. Collins, T. H. Lai, D. Jayasimha, C. K. Chang, and C. H. Huang, and Dr. Y. N. Lien and Dr. G. I. Chen, for their help and influence in one way or another.

To many of my fellow students and friends, I express my thanks as well. They made my study here very enjoyable.

Finally, I would like to thank my wife, Jian Li. Without her support in numerous aspects, it would have been impossible for me to finish my study here.

VITA

November 1964 Born, Xian, China

1981–1985 B.S., Computer Science, Northwest
Telecommunication Engineering Institute,
Xian, China

1986–1987 M.S., Computer and Information Science,
The Ohio State University, Columbus,
Ohio

PUBLICATIONS

1. "An $O(n \log n)$ Feasibility Algorithm for Preemptive Scheduling of n Independent Jobs on a Hypercube," (with M. Ahuja) *Information Processing Letters*, 35 (1), 1990, pp. 7-11.
2. "Preemptive Job Scheduling on a Hypercube," (with M. Ahuja) in *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, August 13-17, 1990, pp. (I) 301-304.
3. "Job Scheduling on a Hypercube," (with M. Ahuja) in *Proceedings of the 10th International Conference on Distributed Computing Systems*, Paris, France, May 28-June 1, 1990, pp. 510-517.
4. "An Efficient Distributed Algorithm for Finding Articulation Points, Bridges, and Biconnected Components in Asynchronous Networks," (with M. Ahuja) in *Proceedings of the 9th Conference on Foundations of Software Technology and Theoretical Computer Science*, Bangalore, India, December 19-21, 1989, pp. 99-108.
5. "A New Distributed Algorithm for Minimum Weight Spanning Trees Based on Echo Algorithms," (with M. Ahuja) in *Proceedings of the 9th International Conference on Distributed Computing Systems*, Newport Beach, CA, June 5-9, 1989, pp. 3-10.

6. "A New Distributed Algorithm for Biconnected Component Detection in Asynchronous Networks," (with M. Ahuja) in *Proceedings of the First Annual IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, May 22-23, 1989, pp. 65-72.

Fields of Study

Major Field: Computer and Information Science

TABLE OF CONTENTS

ACKNOWLEDGMENTS	iii
VITA	iv
LIST OF FIGURES	viii
CHAPTER	PAGE
I INTRODUCTION	1
1.1 Parallel Processing	1
1.2 Hypercube Systems	5
1.3 Job Scheduling	13
1.4 Outline and Significance of Research	16
II NONPREEMPTIVE SCHEDULING ON A HYPERCUBE	21
2.1 Introduction	21
2.2 Previous Research	23
2.3 Preliminaries	28
2.4 The LDF Algorithm and Its Performance	31
2.5 An Absolute Lower Bound	37
2.6 Conclusion	45
III PREEMPTIVE SCHEDULING ON A HYPERCUBE	46
3.1 Introduction	46
3.2 Previous Research	46
3.3 Definitions	48
3.4 Feasibility Algorithm	49
3.5 Correctness and Analysis	56
3.6 On Searching for the Minimum Finish Time	63

3.7	Conclusion	63
IV	A NEW ALGORITHM FOR PREEMPTIVE SCHEDULING	65
4.1	Introduction	65
4.2	Megiddo's Search Method	66
4.3	Initial Algorithm Derivation	69
4.4	Detailed Algorithm	71
4.5	An Example	78
4.6	Correctness and Analysis	81
4.7	Conclusion	83
V	SUMMARY AND FUTURE RESEARCH	85
	BIBLIOGRAPHY	89

LIST OF FIGURES

FIGURE		PAGE
1	A shared-memory machine and a message-passing one.	4
2	Examples of hypercubes.	6
3	Embedding a 3×4 mesh into a 4-dimensional hypercube.	8
4	A 4-ary 2-cube.	12
5	Multiprogramming on a hypercube.	17
6	An example of two dimensional bin packing.	25
7	An example of FFDH packing.	27
8	An example of a stair-like profile.	30
9	The LDF algorithm.	31
10	An example LDF schedule.	32
11	The worst case LDF schedule and the corresponding optimal schedule.	35
12	LDF versus FFDH schedules for squares.	38
13	The optimal schedules.	41
14	The forced schedule.	42
15	The feasibility algorithm.	52
16	The schedule from our feasibility algorithm.	53

17	The schedule from Chen and Lai's feasibility algorithm.	55
18	The profile from our feasibility algorithm after grouping.	57
19	A balanced search tree.	62
20	The modified feasibility algorithm.	72
21	The feasible schedule when "found" is returned.	76
22	The search algorithm for minimum finish time schedule.	77
23	Finding the minimum finish time schedule.	79
24	Binary search over an ordered array.	83

CHAPTER I

INTRODUCTION

1.1 Parallel Processing

High-performance computers are increasingly in demand in the areas of scientific and engineering applications such as: structure analysis, weather forecasting, fusion energy research, medical diagnosis, aerodynamics simulations, artificial intelligence, and industrial automation. Many of these challenges depend on using superpower computers to solve them within reasonable time periods. The rapid advance of technology has made it possible to build faster computers. Over the past four decades, computers have been evolved over four generations: from the first generation (1940-50s) based on vacuum tubes, to second generation (1950-60s) based on transistors, to third generation (1960-70s) based on small- and medium-scale integrated (SSI/MSI) circuits, and to fourth generation (1970s and beyond) based on large- and very-large-scale integrated (LSI/VLSI) devices. But only limited performance increasing can be obtained by simply increasing the speed of electronic components and logic circuits. At most, the electronic signals can be transmitted by the speed of light.

To increase the computing speeds further, *parallel processing* is needed. It means the exploitation of concurrent events in the computing process. Concurrency implies

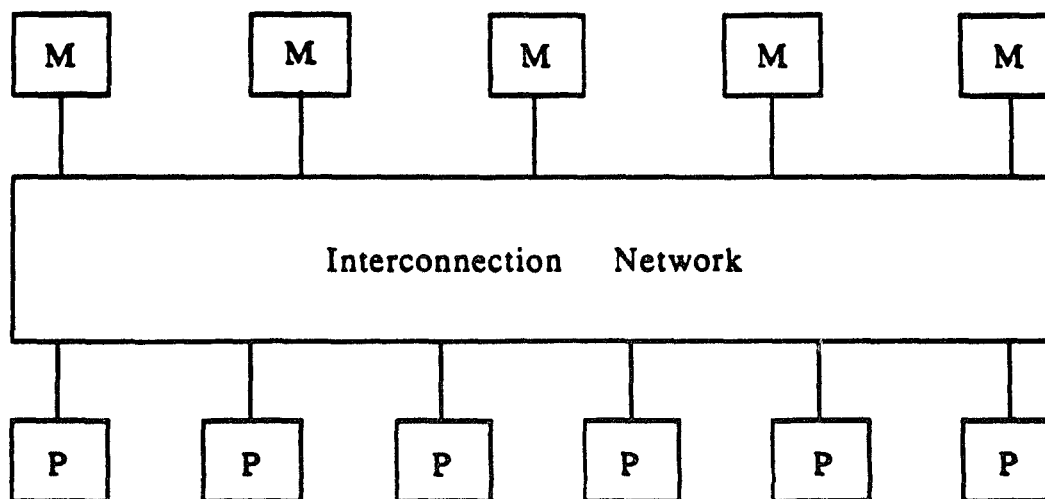
parallelism, simultaneity, and pipelining. Parallel events may occur in multiple resources during the same time interval; simultaneous events may occur at the same time instant; and pipelined events may occur in overlapped time spans. These concurrent events are attainable in a computer system at various processing levels.

Over the years, many parallel computers have been designed for parallel processing. They can be classified into three architectural configurations ([36]): pipeline computers, array processors, and multiprocessor systems. The three parallel approaches to computer system design are not mutually exclusive. A pipeline computer overlaps computations to exploit *temporal parallelism*. Such computers include earlier vector processors, such as Control Data's Star-100 and Texas Instruments' ASC, and more recent vector processors, such as Cray-1, Cyber-205, and Fujitsu VP-200. An array processor uses multiple synchronized arithmetic logic units to achieve *spatial parallelism*. Such computers include Illiac-IV and Burroughs Scientific Processor (BSP). A multiprocessor system achieves *asynchronous parallelism* through a set of interactive processors. Such computers include C.mmp and Cm* systems developed by Carnegie-Mellon University, IBM 3081, Cray 2, and many newer systems. Pipeline computers and array processors rely more on very fast components and pipelined operations. Such machines are quite expensive, and performance improvement is increasingly difficult to achieve. In contrast, the performance of multiprocessor systems can be significantly improved simply by adding more processors, faster processors, and better interconnections among them. Most of general purpose parallel systems built today belong to the multiprocessor configuration.

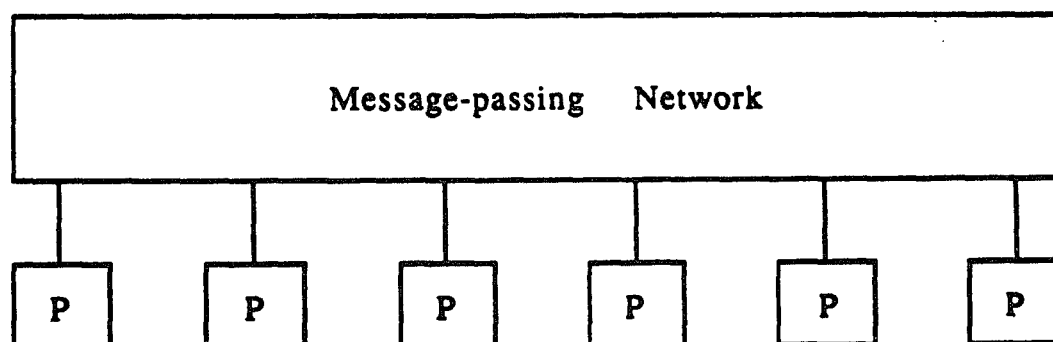
Depending on how the memory modules and processors are interconnected, multiprocessors can be further classified into two categories: *shared-memory* machines and *message-passing* machines. Figure 1 shows the difference between a shared-memory multiprocessor and a message-passing one. A brief discussion of these two types of systems is given below.

In a shared-memory system, a global memory is shared by all the processors through an interconnection network. The advantage such a system is that users can view the system as an extension of the traditional uniprocessor system. But it is very difficult to build high performance large system using the global memory approach. One problem arises when two or more processors access the same memory location at the same time, expensive hardware or software protocols are required to resolve the conflict among the processors. Another problem is that the memory access time must be kept small because memory references are very frequent operations of a program. When the number of processors becomes larger, the two problems become more severe and they make the design of the interconnection network harder.

In a message-passing system, each processor has its own memory, and processors exchange information through message passing. In a system, each processor is directly connected to a subset of the other processors (its "neighbors"). Messages between non-neighbors must be passed through intermediate processors connected by a network. Because a processor can pass messages more quickly to its neighbors than to processors not directly connected to it, tasks that need extensive interprocessor communication should be placed on neighboring processors. If each processor has



Shared-memory Machine



Message-passing Machine

Figure 1: A shared-memory machine and a message-passing one.

most of the data it will need, then the number of messages between processors can be kept relatively small, and the number of processors in the system can be made large. This approach overcomes the two problems in a shared-memory system.

It is more difficult to program on a message-passing system than on a shared-memory system. The placement of data and programs in a message-passing system plays an important role in its system performance. An interconnection scheme that makes it easier to achieve such placement is hypercubes. In this dissertation we study the job scheduling, which is one aspect of the placement, on such systems.

1.2 Hypercube Systems

A binary m -dimensional hypercube (or m -cube) is an undirected graph $G = (V, E)$ with 2^m number of nodes. Each node $p \in V$ is represent by a binary number $p_m p_{m-1} \cdots p_1$. Two nodes are connected iff their binary representations differ in exactly one bit. In a hypercube multiprocessor, each node is a processor that has its own CPU and local main memory. Two neighboring processors can communicate directly with each other over the edge connecting them. Figure 2 shows the hypercube topologies for $n \leq 4$.

An m -cube can be partitioned into subcubes. A d -dimensional subcube (or d -subcube), where $0 \leq d \leq m$, can be described by $m - d$ fixed bits and d "don't-care" bits. For example, a 2-subcube in a 5-cube can be described as $01*1*$, where "*" stands for a don't-care bit. The four processors in the 2-subcube are 01010, 01011, 01110, and 01111.

Hypercube structure has been studied by many research and many properties of

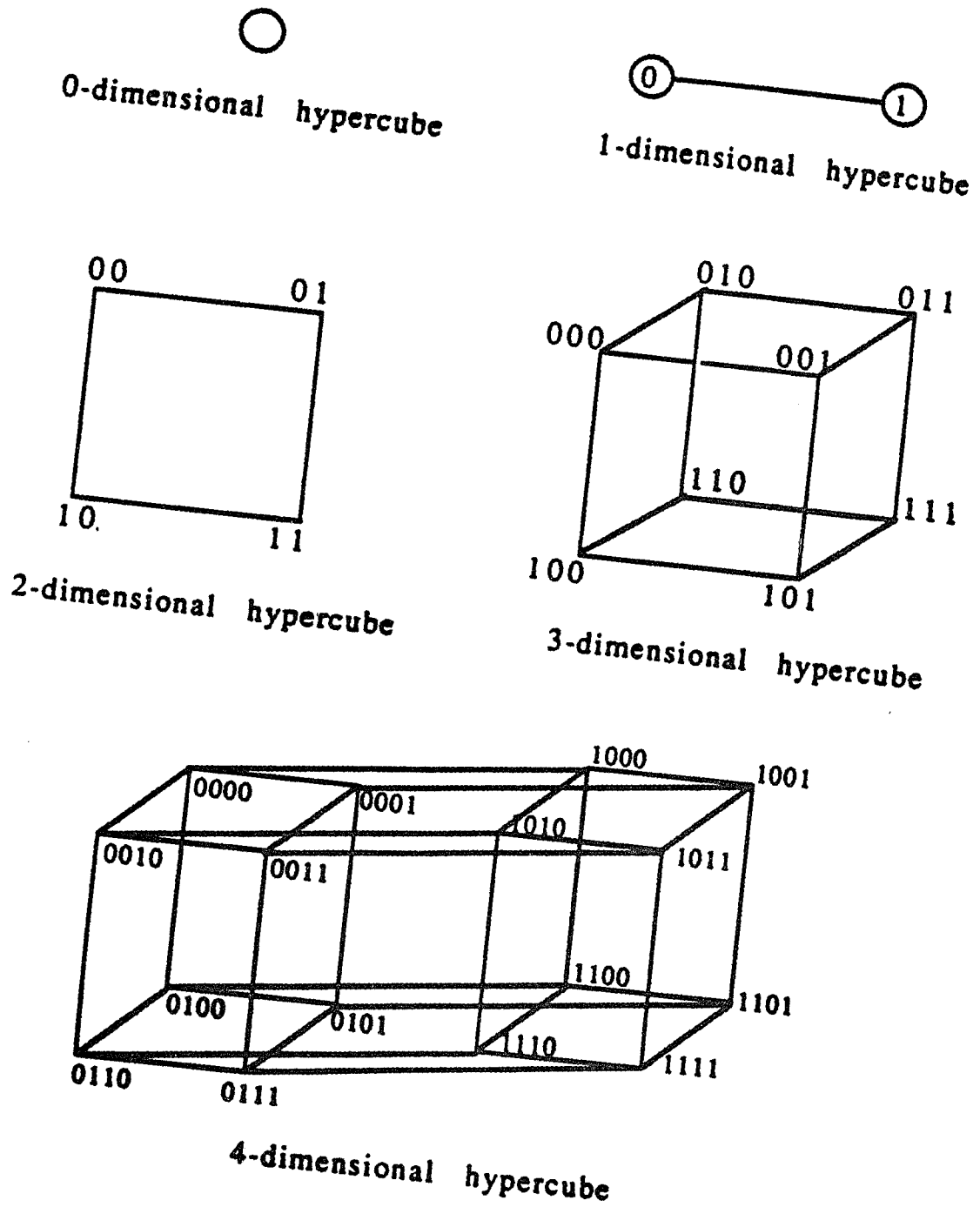


Figure 2: Examples of hypercubes.

hypercube have been discovered. One important property is the embeddibility. Many topologies, such as rings, trees [4, 5, 72], meshes [9, 34, 48, 61], and pyramids [43], can be embedded in hypercubes so that neighboring nodes are mapped to closed connected nodes (sometimes neighbors) in hypercubes. Figure 3 illustrates the embedding of 3×4 mesh into a 4-dimensional hypercube.

Programming on hypercubes is made easy by the good embeddibility property. The communication structures used in the fast Fourier transformation, bitonic sorting, partial differential equation, and convolution can be embedded similarly into hypercubes. Since a great many scientific applications use mesh, tree, FFT, or sorting interconnection structures, the hypercube is a good candidate for a general-purpose parallel architecture.

Hypercubes have small diameters and rich interconnections. In an m -cube, the number of nodes is $N = 2^m$ and the maximum internode distance (or diameter) is m , and each node is connected with m neighbors. Thus, many problems with less regular communication patterns can still be efficiently mapped into hypercubes. Compared with a fully connected network K_N , the hypercube diameter is larger, but the node degree (or fanout) is reduced from $N - 1$ to $\log_2 N$. Other standard architectures with small degree, such as meshes, trees, or bus systems, have either a large diameter (e.g., \sqrt{N} for a mesh) or a resource that becomes a bottleneck in many applications because too much communication must pass through it (as the root of a tree, or a shared bus). Thus, hypercube architecture balances node connectivity, communication diameter, algorithm embeddibility, and programming ease. This balance makes them suitable

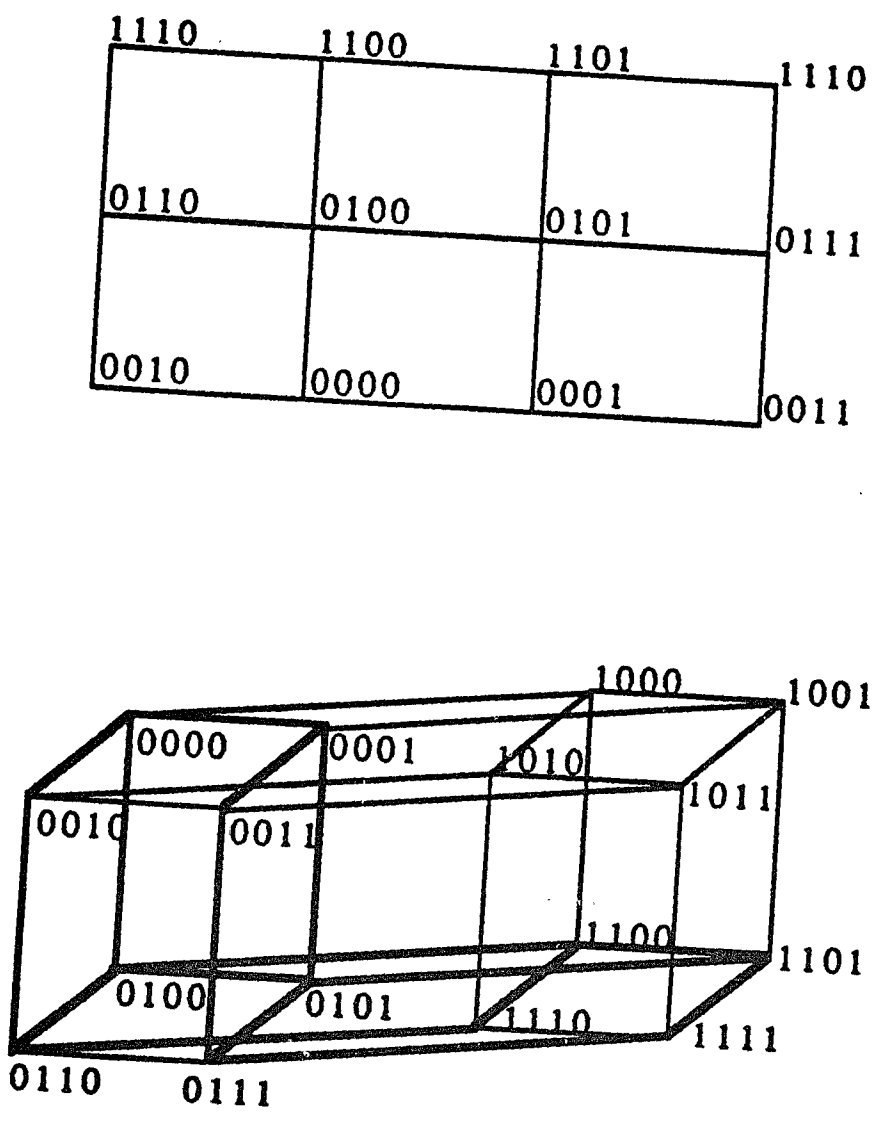


Figure 3: Embedding a 3 x 4 mesh into a 4-dimensional hypercube.

for an unusually broad class of computational problems.

A good survey of the history of the real implementation of hypercube systems can be found in [32]. Here we give a summary. In 1962, Squire and Palais at the University of Michigan carried out a detailed paper design of a hypercube computer [59]. Around 1975, IMS Associates, an early manufacturer of personal computers, announced a 256-node commercial hypercube based on the Intel 8080 microprocessor, but the machine was not produced. In 1977, Sullivan and his colleagues at Columbia University presented a proposal for a large hypercube called the Columbia Homogeneous Parallel Processor, which would have contained up to a million processors [67, 68]. In the same year, Pease published a study of the “indirect” binary n -cube architecture using a multistage interconnection network of the omega type for implementing the hypercube topology [54]. These early hypercube designs were impractical because the circuit technologies at that time could not provide the large number of logic and memory elements they required.

The situation began to change rapidly in the early 1980's as powerful 16/32-bit microprocessors could be implemented on one IC chip, and RAM densities moved into 100,000 to 1,000,000-bit-per-chip range. The first working hypercube computer — a 64-node Cosmic Cube at Caltech — was developed in 1983. Since then, Caltech researchers have built several similar hypercubes and successfully applied them to numerous scientific applications, demonstrating impressive performance improvements over conventional machines of comparable cost [28].

Influenced primarily by the Caltech work, several companies developed the first

generation commercial hypercubes around 1985. These are Intel Personal Supercomputer (or iPSC/1), Ametek's System/14, and NCUBE Corporation's NCUBE/ten. The iPSC/1 has 128 nodes, and each node has a 16-bit 80286/127 CPU as its node processor. The System/14 hypercube can have up to 256 nodes, each employing an 80286/287-based CPU and an 80186 processor for communication management. The NCUBE/ten can accommodate up to 1024 nodes, each based on a VAS-like 32-bit custom processor with a peak performance of 0.5 MFLOPS; thus, a fully configured NCUBE system has a potential throughput of around 500 MFLOPS. Compared with a traditional vector supercomputer such as the Cray-1, which has a peak throughput of 160 MFLOPS, NCUBE/ten can have a much higher performance at a relatively lower cost. Some later introduced hypercube-style machines with supercomputing potential include the Caltech/JPL Mark III [55], the Connection Machine [33], and the Floating Point Systems (FPT) T-series.

The second generation hypercubes, such as iPSC/2 and Ametek 2010, were introduced around 1988. The major improvement is the communication speed between nodes. Since the first generation hypercubes use *store-and-forward* technique to transmit packets, each node along the communication path has to store the entire packet and retransmit again to the next node. This incurs a lot of overhead. The second generation hypercubes reduced the overhead by using some new techniques, such as *circuit switching* in iPSC/2 [53], and *wormhole routing* in Ametek 2010 [23]. In circuit switching, a physical circuit is first established from the source node to the destination node and remains intact till the message transmission is over. Once the circuit

is established, the message transmission time is almost independent of the distance between the source and destination. In Intel iPSC/2, for example, messages move between sender and receiver at a speed of 2.8 megabytes per second [53]. The future generations of hypercube systems are predicted to have even faster communication speeds [1].

Many current hypercube systems use binary n -cube interconnection networks. It is a special case of the family of k -ary n -cubes, i.e., cubes with n dimensions and k nodes in each dimension. In such a cube system, we use n for the *dimension* and k for the *radix*. Dimension, radix, and number of nodes are related by the equation

$$N = k^n, \quad (k = \sqrt[n]{N}, \quad n = \log_k N). \quad (1.1)$$

A node in the k -ary n -cube can be identified by n -digit radix k address, $a_n a_{n-1} \cdots a_1$. The i th digit of the address, a_i , represents the node's position in the i th dimension. Each node can forward messages to its upper and lower neighbor in each dimension, i , with addresses $a_n \cdots a_i \oplus 1 \cdots a_1$ and $a_n \cdots a_i \ominus 1 \cdots a_1$, respectively, (\oplus and \ominus represent *mod k* + and - operations). Figure 4 shows a 4-ary 2-cube, which is also known as a mesh (with wrap-around connections) or a torus. Two dimensional mesh connections have been employed by several recent commercial systems, such as iWarp and Ametek 2010. As pointed out by several studies [22, 39, 71], low-dimensional k -ary n -cube may be the architecture for future median sized (around 256 processors) message-passing multiprocessors.

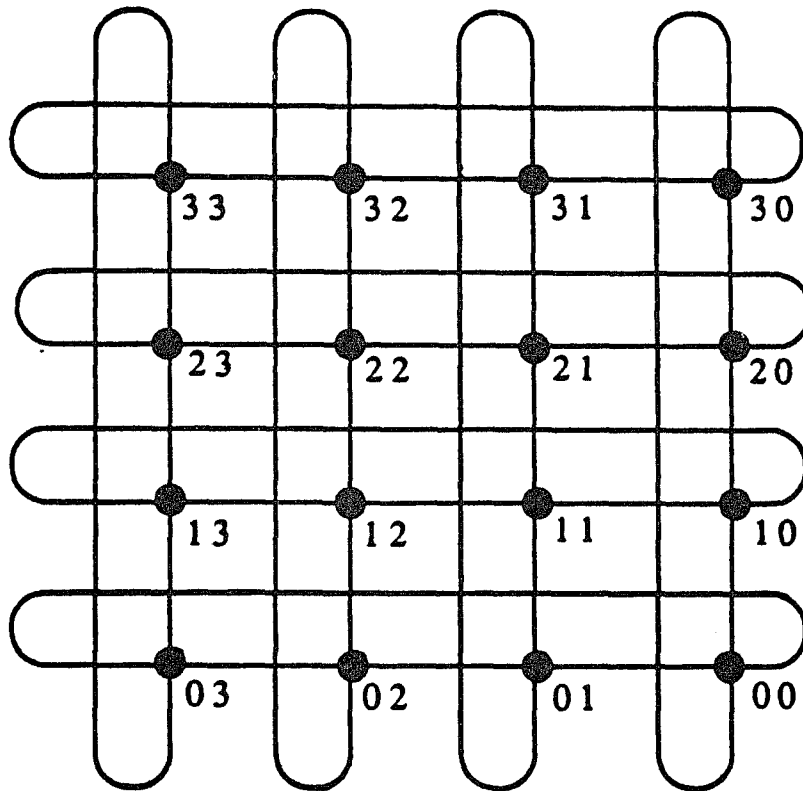


Figure 4: A 4-ary 2-cube.

1.3 Job Scheduling

To be able to use any computer systems efficiently, we need to consider many optimality issues from various aspects. From operating system point of view, we need to avoid any wasting of resources in a computer system. One of the most important resources is the CPU processing time. We would like to avoid the situation that some processors are idle while others are overloaded. This requires us to study ways to assign jobs to a computer system. The problem is known as the scheduling problem, which we will discuss next.

In classical job scheduling theory, a problem can be classified by a 3-field classification $\alpha|\beta|\gamma$ [31]. Suppose that a set of jobs need to be processed on a number of machines. The three fields are:

- **Machine environment:** It specifies what kind of machines the jobs are to be scheduled on. The most frequently studied machine models are *identical machines* where all the machines have the same execution speed, and *uniform machines* where the machines may have different speeds.
- **Job characteristics:** It specifies what kind of jobs needed to be scheduled. For example, a job can be *preemptive*, meaning that the execution of the job can be interrupted and resumed at a later time; otherwise, it is *nonpreemptive*; there may exist *precedence relation* between the jobs, meaning that some jobs have to be finished before some other jobs can be started; a job can also have a given *release time* and a given *deadline*, meaning the job cannot be executed until its

release time and must be finished by its deadline.

- **Optimality criteria:** It refers to what kind of optimality function is used. Usually, the optimality criterion is to minimize the finish time of a job schedule.

By considering the different combination of the above three fields, one can form various scheduling problems. Many researchers have studied scheduling problems over the past 30 years, and a large amount of results have been published. It turns out that many problems are NP-Complete, which means that it is unlikely to find optimal solutions for them. Some important results can be found in [18, 29, 31].

One important assumption used by the classical scheduling model is that one job requires one machine. But with the advance of computer network and parallel computers, this assumption may not be true. As a result, more and more researchers begin to consider scheduling problems in distributed and parallel systems, where jobs usually have more requirement than those in the classical scheduling.

A distributed computing system consist of a number of processors. Each processor is an individual computer, and all the processors are connected through a communication network and can send messages to each others. Here the communication cost is usually much higher than that in a parallel system. In a distributed system, a job is modeled as a set of communicating tasks. Each task is able to run on one processor. If two communicating tasks are assigned to two different processors, then there will be some time delay for each communication, otherwise, the communication cost is assumed to be zero. In order to finish the job efficiently, we need to distribute the tasks onto different computers. The objective is to distribute the tasks evenly among the

processors, and to allocate communicating tasks to the same processor or processors with close connections. A number of researchers have studied scheduling problems in distributed systems. The solutions can be classified into three categories: graph theoretical approaches [57, 65, 66], 0-1 integer programming approaches [16, 64], and heuristic approaches [26, 47].

In a parallel multiprocessor system, whether it is a shared-memory or a message-passing machine, one job may require many processors to run at the same time. This is because parallel programs need more than one processors to explore the parallelism existing in the applications. In fact, many parallel machines are built to satisfy this requirement. Many systems, such as the hypercubes, RP3 of IBM, Butterfly of BBN, Ultracomputer of New York University, and PASM of Purdue University, are partitionable or have the potential to be partitionable. For a job which requires certain number of processors, we can partition the system into subsystems and assign one subsystem with the required number of processors to the job. This creates a new research direction in scheduling theory.

Therefore, parallel job scheduling is actually done on two levels: process level and task level. Each level can be further refined into more levels. On process level, we need to: (i) determine the number of processors needed by the applications, and (ii) schedule the communicating macro-tasks of one job onto the required number of processors. Step (i) is called the *grain size determination problem*, which has been considered by [41, 73]. Step (ii) needs to determine how to balance the computation and communication to achieve good performance. Many techniques used in distributed

job scheduling may be applied here as well. Process level scheduling is usually done by a powerful parallel compiler. Once the process level schedule is done, we need to do the task level scheduling. On task level, we are given a set of jobs, with each job requiring certain number of processors for certain amount of time. The objective is to finish the set of jobs in minimum amount of time. Task level scheduling is usually done by the operating system.

On a hypercube system, the process level job scheduling as pointed out by Chen and Shin in [13], consists of three steps: (i) determination of the dimension d of a subcube required to process the incoming job, (ii) location of an idle subcube of dimension d , and (iii) assignment of the modules of the job to nodes of the d -subcube. The first step was formally treated in [14]. The third step is the well known embedding problem mentioned before. The second step is the subcube allocation problem and has recently attracted the attention of many researchers [13, 25, 42, 17].

In this study, we will study the task level job scheduling on a hypercube. Next we will introduce the problem and give an outline of our research presented in the remaining Chapters.

1.4 Outline and Significance of Research

We assume that the hypercube system supports *space-sharing multiprogramming*, as suggested by [1, 32, 55]. Here an entire hypercube is partitioned into subcubes, each job is assigned to a dedicated subcube and many jobs can be running simultaneously without interfering with each other. Figure 5 shows a snapshot of three jobs running on a 4-dimensional hypercube. Then, given a set of jobs with their subcube and their

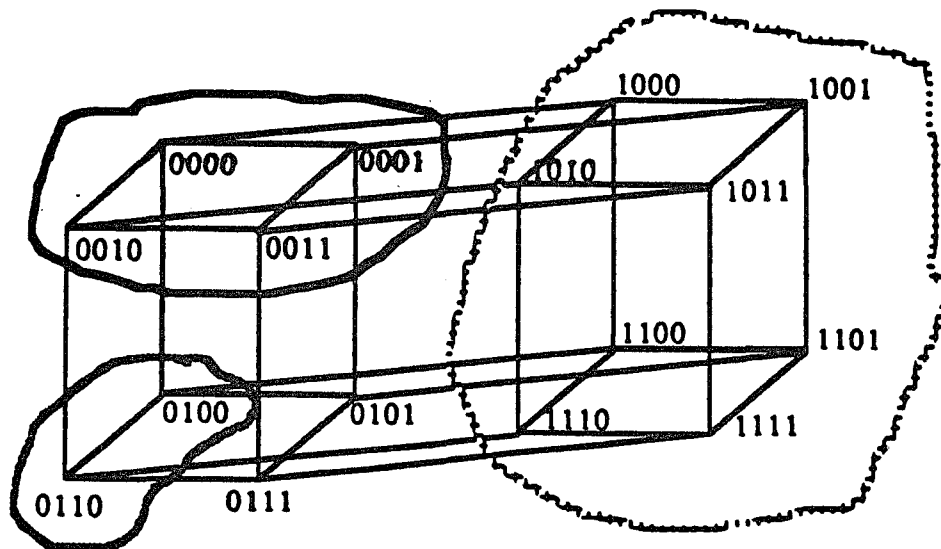


Figure 5: Multiprogramming on a hypercube.

running time requirements on such a hypercube system, it is important for the system to schedule them so that they can be finished as early as possible.

The formal definition of the problem, which has been introduced and studied by Chen and Lai [10, 11], is as follows. Consider an m -cube and a set J of n independent jobs. Each job J_i , where $1 \leq i \leq n$, is an ordered pair (d_i, t_i) , where d_i is an integer and t_i is a rational number satisfying $0 \leq d_i \leq m$ and $t_i > 0$. The ordered pair means that job J_i requires a d_i -subcube (2^{d_i} processors) for t_i units of time. We wish to assign d_i -subcube(s) to job J_i for all n jobs, so that their overall finish time (the time at which all jobs are finished) is minimized. In other words, we want to find the *minimum finish time schedule* for the given jobs. The *minimum finish time* is also called the *optimal time*, and the *minimum finish time schedule* is called the *optimal schedule*.

In Chapter II, we discuss the scheduling for *nonpreemptive* jobs, i.e., no job can be stopped during its execution. The problem is NP-Complete, which means it is unlikely that one can find an optimal schedule within polynomial time. In this case, we investigate approximation algorithms. An approximation algorithm is able to generate a schedule whose finish time is within a constant bound of the optimal time. We will present a simple approximation algorithm called LDF. We will show that the algorithm has a bound no worse than 2. The algorithm LDF also possesses some favorable properties compared with previously proposed algorithms. We will also show that there exists a $(1 + \sqrt{6})/2 \approx 1.7247$ lower bound for a class of approximation algorithms including LDF.

One important practical advantage of the LDF algorithm is that a system scheduler can perform very well without knowing the job execution times. This is especially true when the job execution times are hard to get. The lower bound result means that it is unlikely to find simple heuristic algorithms that can perform substantially better than LDF.

In Chapter III, we discuss the scheduling for *preemptive* jobs, i.e., each job may be preempted before its completion but will resume its execution at a later time. In this case, the optimal preemptive schedule can be found in polynomial time. We will first present an $O(n \log n)$ feasibility algorithm, which can determine if it is feasible to finish the job set by a given deadline T . If it is feasible, then the algorithm can generate a feasible schedule that has at most $n-2$ preemptions. The optimal schedule can be found through a binary search over a time interval by calling the feasibility algorithm repeatedly. We will show that there exists a time interval which is shorter than the one proposed by the earlier algorithm. the minimum finish time can be obtained through a binary search over the time interval.

Our feasibility algorithm not only runs faster than a previous algorithm, but also generates a schedule with fewer number of preemptions. These improvements are important since many scheduling algorithms require an efficient feasibility algorithm as a building block. Further, as research presented in the sequel shows, our feasibility algorithm leads well to the development of an even more efficient algorithm for finding minimum finish time schedule.

In Chapter IV, we propose a new algorithm with improved running time for finding

the optimal preemptive schedule. Compared with the algorithm presented in Chapter III, whose running time depends on the numerical values of the time requirements t_i of the jobs, the running time of our new algorithm depends only on the number of the jobs (i.e., n). Based on an advanced search technique, the new algorithm can be used to find the optimal schedule in $O(n^2 \log^2 n)$ time.

Finally, in Chapter V, we summarize our contribution and give several future research directions.

CHAPTER II

NONPREEMPTIVE SCHEDULING ON A HYPERCUBE

2.1 Introduction

In this chapter, we restrict our attention to nonpreemptive scheduling on a hypercube. Let us first recapitulate the problem below. Given an m -cube and a set J of n independent jobs J_1, \dots, J_n . Each job J_i is an ordered pair (d_i, t_i) , meaning that job J_i requires a d_i -subcube (2^{d_i} processors) for t_i units of time. Each job is *nonpreemptive*, i.e., it cannot be interrupted during its execution. We wish to assign d_i -subcube(s) to job J_i for all n jobs, so that their finish time is minimized.

The problem is NP-Complete because it embeds an identical machine nonpreemptive scheduling problem which has been proven as NP-Complete in classical scheduling [29, 70]. Thus, it is unlikely to find a polynomial time algorithm for the problem unless $P = NP$. Thus we need to look for polynomial approximation algorithms that can find a solution within a constant bound of the optimal one. Since the bound is one of the most important performance measures of an approximation algorithm, we would like to find an algorithm whose bound is as small as possible.

The commonly used approximation algorithms are called *list scheduling*. In list scheduling, all jobs are first put into a list in a certain order, and each job is scheduled

one after another in this order. The way a list is constructed is very important, even for the same algorithm, since different lists may result in very different performance bounds.

To measure the performance of approximation nonpreemptive scheduling algorithms, we use two bounds that have been introduced by Coffman *et. al.* in [19]. Let $L = (J_1, \dots, J_n)$ be a *particular* list containing jobs in J in certain order. Let $A(L)$ be the finish time of the schedule generated by *any* algorithm A using L . Let $OPT(L)$ be the optimal finish time, i.e., the finish time of the schedule for the jobs in L generated from an optimal algorithm.

Definition 1 Let t_{\max} be the longest time requirement of the n jobs, i.e., $t_{\max} = \max\{t_i : 1 \leq i \leq n\}$.

Definition 2 Let ρ be a constant. Then ρ is called the absolute bound of algorithm A , if for every list L of jobs in J we have

$$A(L) \leq \rho OPT(L). \quad (2.1)$$

Definition 3 Let ρ, ρ' be constants. Then ρ is called the asymptotic bound of algorithm A , if for every list L of jobs in J we have

$$A(L) \leq \rho OPT(L) + \rho' t_{\max}.$$

The absolute bound appears to be a better measure of performance when the number of jobs is small, while the asymptotic bound is a better measure when the number of jobs is large. This is because when the number of jobs is large, the term containing $OPT(L)$ will be the dominating factor compared with that containing t_{\max} .

In the following, we will first survey earlier related research. Then we will present and analyze the performance of an approximation algorithm called LDF. We will also prove a lower bound result for a class of nonpreemptive scheduling algorithms. Finally we conclude this chapter.

2.2 Previous Research

In [10], Chen and Lai proposed an algorithm, called LDLPT (*Largest Dimension Largest Processing Time*). In LDLPT, jobs are ordered by decreasing dimension, and jobs with same dimensions are ordered by decreasing time. Their algorithm can be stated as follows:

Let $L = (J_1, \dots, J_n)$ be a list of the n jobs in LDLPT order. Schedule each job from L , one after another, to the earliest available smallest indexed subcube.

Their algorithm is an extension of the famous LPT algorithm proposed by Graham [30]. In LPT, jobs do not have the dimensional requirement. Graham proved that LPT has an absolute bound $4/3 - 1/(3m')$, where m' is the number of identical machines. Extending Graham's proof, Chen and Lai showed that the absolute bound of LDLPT is $2 - 1/2^{m-1}$ for $m \geq 2$. For a large cube dimension m , such as a Connection Machine where m can be as high as 20, the second term will approach zero, and the absolute bound will be about 2. Although they did not discuss the asymptotic bound of LDLPT, it can be easily shown to be 1.

Since late 1970s, many researchers have studied the *two dimensional bin packing* problem, which is closely related to the hypercube scheduling defined above. Two dimensional bin packing was first introduced by Baker, Coffman and Rivest [3]. As illustrated in Figure 6, we are given an “open-ended” rectangle, or a two dimensional bin R , of width w and a set of n rectangles, or pieces, organized into a list $L = (p_1, p_2, \dots, p_n)$. Each rectangle is defined by an ordered pair $p_i = (w_i, h_i)$, $1 \leq i \leq n$, corresponding to the width (w_i) and height (h_i). The goal is to pack them into R , so as to minimize the total height of the packing. The rectangles must be packed orthogonally, i.e., no rotations are allowed: all rectangles must have their width parallel to the bottom of R .

This problem can be interpreted as a job scheduling problem as follows. The height of a piece is the amount of processing time a job requires, and its width is the amount of processors the job needs. If we restrict the width w of R and the width of each piece w_i to be a power of 2, then the problem is exactly the same as hypercube scheduling problem we introduced before. In fact, this restricted case has been considered before, under the name of *strongly divisible* by Coffman, Garey and Johnson in [21]. It is worth mentioning that, as far as we know, no one has related the hypercube scheduling problem with the two dimensional bin packing before.

In the following, we provide some important results discovered in two dimensional bin packing. A good survey of this area can be found in [20].

In [3], Baker *et. al.* considered a variety of packing algorithms based on the “bottom up left justified” (or *BL* for short). In a *BL* packing, rectangles are packed

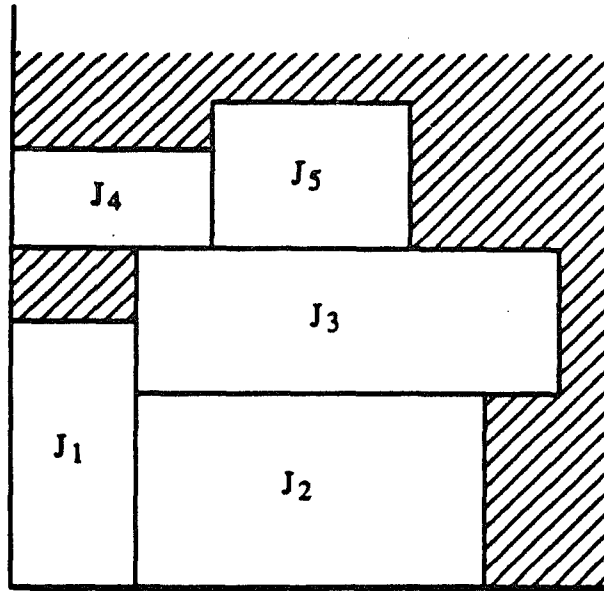


Figure 6: An example of two dimensional bin packing.

in turn, and each item being placed as near to the bottom of R as it will fit and then as far to the left as it can be placed at that bottom-most level. (See Figure 6, where the packing is actually done by the BL rule.) Depending on how (if at all) the set of rectangles is initially preordered, it turns out that only one ordering, *decreasing width* (or DW), can have a finite absolute bound. The algorithm, called BLDW, is proved to have an absolute bound 3. As another measurement of performance, they considered the special case of squares ($h_i = w_i$). For this case, BLDW has an absolute bound 2.

In [19], Coffman, Garey, Johnson, and Tarjan proposed an algorithm called FFDH (*First Fit Decreasing Height*) “level” algorithm. FFDH first order the rectangles by decreasing height. Then the packing is constructed as a sequence of *levels*, i.e., each rectangle is placed so that its bottom rests on one of these levels. The first level is simply the bottom of the bin. Each subsequent level is defined by a horizontal line drawn through the top of the tallest rectangle on the previous level. Each rectangle is placed left-justified on the first (i.e., lowest) level if in which it can fit. If none of the current levels has room, a new level is started. See Figure 7 for an example of an FFDH packing. The absolute bound of FFDH is proved to be 2.7.

In [58], Sleator developed a very smart algorithm which reduced the absolute bound to 2.5. His algorithm first packs the rectangles with width greater than half of the width of R on the bottom of R . Then R is divided into two halves to pack the remaining rectangles by using a modified FFDH algorithm.

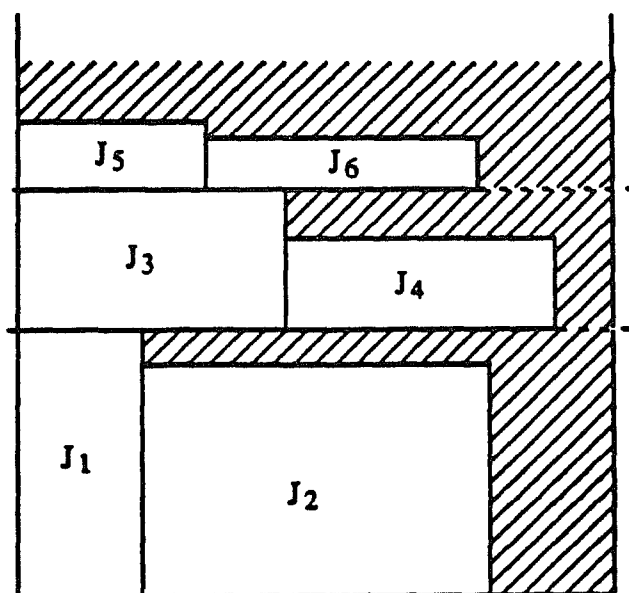


Figure 7: An example of FFDH packing.

It is understandable that the performance of the above bin packing algorithms are worse than similar algorithm for the hypercube job scheduling, because two dimensional bin packing is more general than hypercube job scheduling. Later, Coffman, *et. al.* considered the FFDH algorithm for strongly divisible case [21] (or for the hypercube job scheduling). They proved that the absolute bound in this case is less than 2, which is similar to that of LDLPT.

2.3 Preliminaries

Definition 4 A processor interval (or p-interval) $[a, b]$ denotes the set of processors $[a, b] = \{p \in V : a \leq p \leq b\}$ where $a, b \in V$ and $a \leq b$. Processor a and b are called the start point and end point of the p-interval, respectively. Also let $|[a, b]|$ be the number of processors in $[a, b]$.

Let $[u, v]$ be a p-interval, with $u = u_m \cdots u_1$ and $v = v_m \cdots v_1$ in binary. It is clear that $[u, v]$ forms a k -subcube iff

- $u_m \cdots u_{k+1} = v_m \cdots v_{k+1}$,
- $u_k \cdots u_1 = 00 \cdots 0$, and
- $v_k \cdots v_1 = 11 \cdots 1$.

In that case, $[a, b]$ is said to be a *basic* subcube. Note that every k -dimensional basic subcube $[a, b]$ can be divided into a number 2^{k-h} of h -dimensional basic subcubes ($h \leq k$): namely,

$$[a, a + 2^h - 1], [a + 2^h, a + 2(2^h) - 1], \dots, [b - 2^h + 1, b].$$

All the subcubes mentioned later in this paper are basic, so we will omit “basic” for simplicity.

An m -cube can be described as a p -interval of $[0, 2^m - 1]$. The m -cube can also be divided into a number of consecutive p -intervals. Such division will help us to describe job scheduling on the cube.

Definition 5 Let $l \geq 1$. A division of an m -cube is a list of l consecutive p -intervals $\langle [a_1, b_1], \dots, [a_l, b_l] \rangle$ (for some $l > 0$) satisfying $a_1 = 0$, $b_l = 2^m - 1$, and $(\forall i : 1 \leq i < l : a_{i+1} = b_i + 1)$.

Definition 6 A profile of a schedule is a function F that maps a processor $p \in V$ to a time $f = F(p)$, meaning that processor p has been busy until time f , and f is called the finish time of p .

A profile records the finish time of all the processors. Such information is needed by a scheduling algorithm to assign the next job. For hypercube scheduling, a profile function F maps a p -interval $[a, b]$ to a time $f = F([a, b])$, where $\forall p \in [a, b] : F(p) = f$. Let $\langle [a_1, b_1], \dots, [a_l, b_l] \rangle$ be a division of an m -cube. We define a profile of a cube schedule as follows.

Definition 7 A profile P of an m -cube schedule S can be described as a sequence of l ordered pairs $\langle ([a_1, b_1], f_1), \dots, ([a_l, b_l], f_l) \rangle$ satisfying $(\forall i : 1 \leq i \leq l : F([a_i, b_i]) = f_i)$. A stair-like profile P is a profile that satisfies $(\forall i : 1 \leq i < l : f_{i+1} < f_i)$.

Figure 8 shows an example of a stair-like profile. Stair-like profiles will be used later for preemptive scheduling.

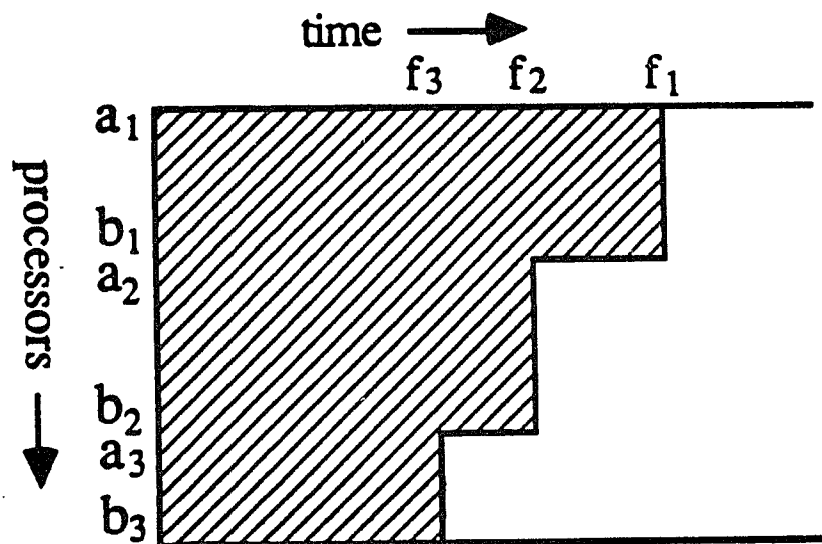


Figure 8: An example of a stair-like profile.

Algorithm LDF;

/* The algorithm LDF receives a job set of n jobs and an m for the m -cube; it returns the a schedule for the n jobs to be run on the m -cube. */

1. Let $L = (J_1, \dots, J_n)$ be in decreasing dimension.
 2. Schedule J_i to the earliest available, smallest indexed d_i -subcube.
-

Figure 9: The LDF algorithm.

2.4 The LDF Algorithm and Its Performance

We propose a new algorithm called LDF (*Largest Dimension First*) below. Here we construct $L = (J_1, \dots, J_n)$ of the n jobs by decreasing dimension, i.e., $d_1 \geq \dots \geq d_n$. The scheduling part of the algorithm is similar to that of the LDLPT and BLDW algorithms summarized before. More specifically, when scheduling job J_i , we assign J_i to the earliest available d_i -subcube; if several d_i -subcubes have the same earliest available time, J_i is assigned to the one with the smallest index. The formal presentation of the algorithm is listed in Figure 9.

An example schedule generated by LDF is shown in Figure 10. From Figure 10, we observe the following property for any LDF schedule. No processor may be idle before s_n – the starting time of J_n , the last job scheduled.

Let $LDF(L)$ and $OPT(L)$ be the finish time of the LDF schedule and the optimal schedule for jobs in L , respectively. We have the following theorem.

Theorem 2.4.1 *The absolute bound of LDF is $LDF(L)/OPT(L) \leq 2 - 1/2^m$, and*

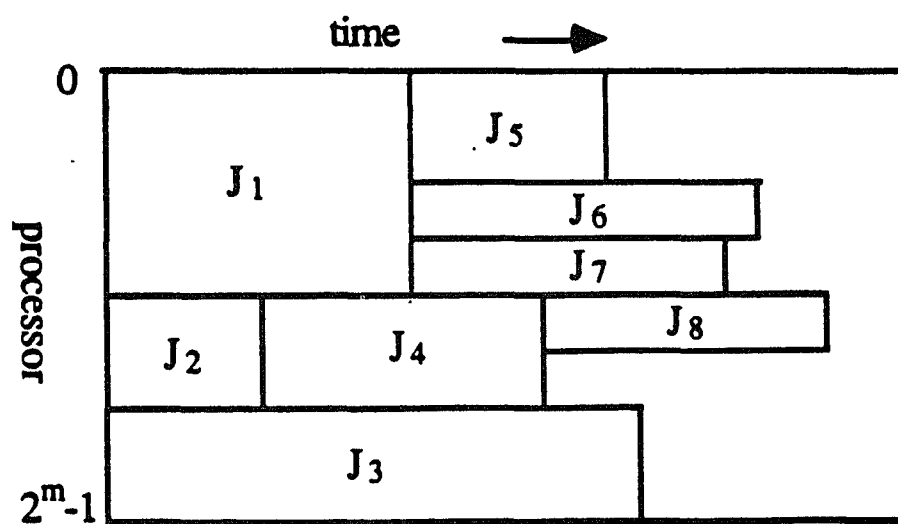


Figure 10: An example LDF schedule.

the bound is tight.

Proof: Let J_k be the first job in L that finishes at $\text{LDF}(L)$. Then we could just consider the prefix list $L' = (J_1, \dots, J_k)$. Since the remaining jobs in L will not increase $\text{LDF}(L)$ and will not decrease $\text{OPT}(L)$, therefore, considering L' will not decrease the final bound. Let s_i be the start time of J_i . First, we have

$$\text{LDF}(L) = s_k + t_k. \quad (2.2)$$

By the property that no processor is idle before s_k , we obtain

$$2^m s_k \leq \sum_{i=1}^{k-1} 2^{d_i} t_i. \quad (2.3)$$

Any schedule of L' has to take $\sum_{i=1}^k 2^{d_i} t_i$ area, so we have

$$\sum_{i=1}^k 2^{d_i} t_i \leq 2^m \text{OPT}(L). \quad (2.4)$$

Then

$$2^m \text{LDF}(L) = 2^m (s_k + t_k) \quad (\text{from (2.2)}) \quad (2.4)$$

$$= 2^m s_k + 2^m t_k \quad (2.5)$$

$$\leq \sum_{i=1}^{k-1} 2^{d_i} t_i + 2^m t_k \quad (\text{from (2.3)}) \quad (2.6)$$

$$= \sum_{i=1}^k 2^{d_i} t_i + 2^m t_k - 2^{d_k} t_k \quad (2.7)$$

$$\leq 2^m \text{OPT}(L) + (2^m - 2^{d_k}) t_k. \quad (\text{from (2.4)}) \quad (2.8)$$

Dividing both sides by 2^m , and since $t_{\max} \leq \text{OPT}(L)$, we get

$$\text{LDF}(L) \leq \text{OPT}(L) + ((2^m - 2^{d_k})/2^m) t_k \quad (2.9)$$

$$\leq \text{OPT}(L) + ((2^m - 1)/2^m)t_{\max} \quad (2.10)$$

$$\leq (2 - 1/2^m)\text{OPT}(L). \quad (2.11)$$

Thus we have proved the absolute bound.

To show the tightness of the bound, consider the following job set $J = \{J_i : 1 \leq i \leq 2^{2^m} - 2^m + 1\}$ where

$$J_i = \begin{cases} (0, 1) & \text{if } 1 \leq i \leq 2^{2^m} - 2^m; \\ (0, 2^m) & \text{if } i = 2^{2^m} - 2^m + 1. \end{cases} \quad (2.12)$$

The list is $L = (J_1, \dots, J_{2^{2^m} - 2^m + 1})$. The LDF and the optimal schedules are shown in Figure 11. In the LDF schedule, the first $2^{2^m} - 2^m$ jobs in L fill the m -cube up to time $2^m - 1$ completely; and the last job results the final time to be $2^{m+1} - 1$. In the optimal schedule, we can assign the last job first on the p -interval $[0, 1]$ up to time 2^m , and use the other jobs to fill the remaining area up to time 2^m . (Actually this is simply the worst case example for list scheduling on 2^m identical processors.) \square

Corollary 2.4.2 *The LDF algorithm has an optimal asymptotic bound.*

Proof: By inequation (2.10) above. \square

Compared with LDLPT and FFDH, LDF has comparable performance. Our proof for LDF is much simpler than that for LDLPT, which takes several pages.

Next we compare the performance of LDF with that of FFDH for a special case when each job is a *square*, i.e., when each $J_i = (d_i, t_i)$ satisfies $2^{d_i} = t_i$. Although packing squares makes more sense in the context of stock packing rather than job scheduling, this case is frequently used as another performance measure.

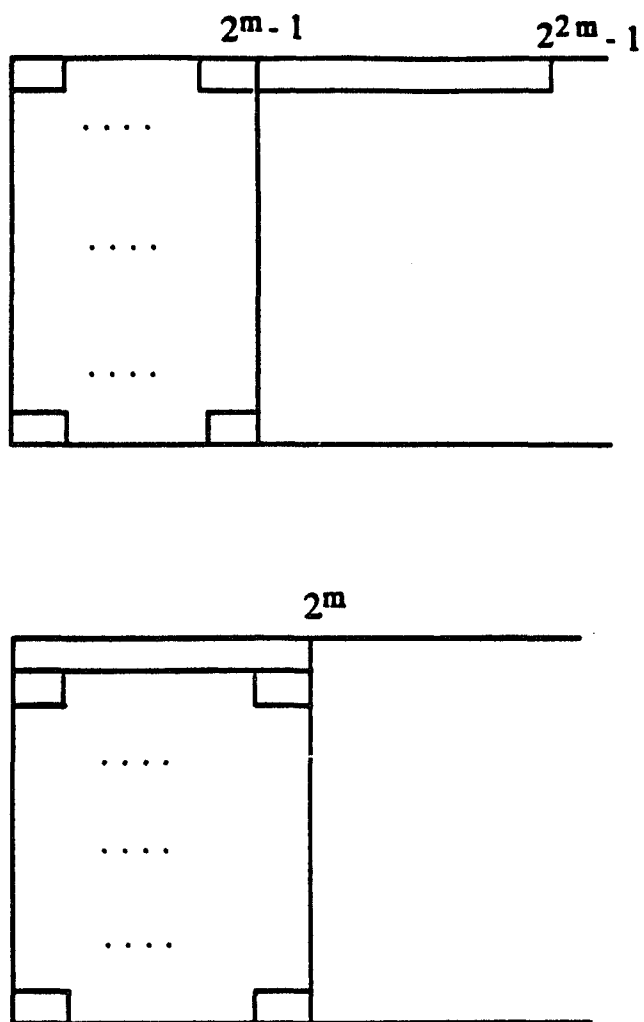


Figure 11: The worst case LDF schedule and the corresponding optimal schedule.

Theorem 2.4.3 *The LDF algorithm can generate an optimal schedule for squares, i.e., when each $J_i = (d_i, t_i)$ satisfies $2^{d_i} = t_i$.*

Proof: We prove the result by induction on the number of jobs in a job set J . When $|J| = 1$, the result is obviously true. Suppose that the LDF algorithm is optimal for any J such that $|J| < n$. Now consider a job set J with $|J| = n$, and suppose the jobs are ordered by decreasing dimension. After scheduling the first $n - 1$ jobs using LDF, we will get an optimal schedule by induction hypothesis. Let its profile $P(n - 1)$ be

$$\langle ([a_1, b_1], f_1), \dots, ([a_l, b_l], f_l) \rangle.$$

Since 2^{d_i} (or t_i) of each J_i , $1 \leq i \leq n - 1$, is a multiple of 2^{d_n} (or t_n), one can readily see the fact that $P(n - 1)$ is stair-like and all the $|[a_j, b_j]|$'s and all the $(f_j - f_{j-1})$'s, where $1 \leq j \leq l$ (define $f_0 = 0$), are multiples of 2^{d_n} (or t_n). Let $T_{n-1}^* = f_l$ represent the largest finish time in $P(n - 1)$. Now consider the scheduling of J_n . Let T_n^* represent the largest finish time in $P(n)$. Two cases arise:

- (i) $l > 1$ in $P(n - 1)$. Since $f_2 - f_1$ is a multiple of t_n and f_1 is the smallest finish time in $P(n - 1)$, J_n is scheduled by LDF onto $[a_1, a_1 + 2^{d_n} - 1]$ for t_n time without exceeding T_{n-1}^* . Thus, $T_n^* = T_{n-1}^*$. Therefore, T_n^* is the optimal finish time for J since T_{n-1}^* is already the optimal one for the first $n - 1$ jobs.
- (ii) $l = 1$ in $P(n - 1)$. Then T_n^* has to be greater than T_{n-1}^* in any optimal schedule, and must be a multiple of t_n . LDF will schedule J_n on $[0, 2^{d_n} - 1]$. Then $T_n^* = T_{n-1}^* + t_n$. But this T_n^* is the smallest multiple of t_n following T_{n-1}^* , so the schedule must be an optimal one.

By induction, the theorem holds. □

On the other hand, FFDH will not perform optimal for this case. Figure 12 illustrates this point. The advantage of LDF over FFDH is thus apparent.

2.5 An Absolute Lower Bound

For approximation solutions, one would also like to know whether the results can be improved further. In [8], various lower bounds were found for the so called *on-line* algorithms. An algorithm is on-line if, given a list of jobs $L = (J_1, \dots, J_n)$, it

1. schedules the jobs in the order given by L ,
2. schedules each job J_i without looking ahead at any $J_j, j > i$,
3. never moves a job already scheduled.

The above three algorithms LDF, LDLPT, and FFDH are on-line with different pre-ordered job lists.

First we observe that when job lists are not preordered, the absolute lower bound obtained in [8] (Theorem 1) still applies to hypercube scheduling because the construction there does not depend on job widths.

Theorem 2.5.1 ([8]) *Let A be an on-line algorithm. For any $\delta > 0$, there is a list L for which $A(L) > (2 - \delta)OPT(L)$.*

Thus, when jobs are not properly ordered, every on-line algorithm can perform so badly that it comes arbitrarily close to doubling the finish time of an optimal scheduling.

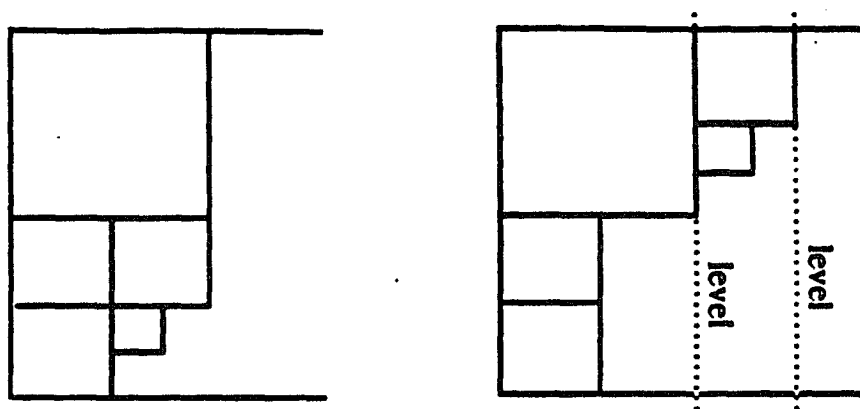


Figure 12: LDF versus FFDH schedules for squares.

So far, we have seen algorithms using decreasing dimension (e.g., LDLPT and LDF) and decreasing time (e.g., FFDH). For hypercube scheduling, two other job orderings are possible: *increasing dimension* and *increasing time*. But it appears that these two orderings would perform worse than decreasing dimension. Intuitively, these two ordering can create “hole” easily during the scheduling. Moreover, Baker *et al.* [3], have shown that job orderings by both increasing dimension and increasing time result in infinite bound for BL algorithm. We shall thus concentrate further on decreasing dimension. Next we prove an absolute lower bound for the schedules generated by any on-line algorithm using decreasing dimension lists.

Theorem 2.5.2 *Let A be any on-line algorithm. There is a list L ordered by decreasing dimension such that $A(L) \geq ((1 + \sqrt{6})/2)OPT(L) > 1.7247OPT(L)$.*

Proof: Consider a 5-cube. Let the list L be $L = L_1L_2L_3L_4$ where:

- L_1 consists of 4 jobs of size $(3, 1)$,
- L_2 consists of 4 jobs of size $(2, 2 - x)$,
- L_3 consists of 4 jobs of size $(1, 2 + x)$,
- L_4 consists of 9 jobs of size $(0, 4)$.

where $0 < x < 1$.

The strategy may be outlined as follows. Suppose we have an algorithm A which has an absolute bound $A(L)/OPT(A)$. As we allocate jobs one by one from L using algorithm A , we need to enforce that all the partial schedules are within the bound.

Doing this will force us to schedule the jobs such that the final schedule will have a finish time greater than a function of x . We then choose x in such a way that the function takes its largest value. This largest value will be the lower bound. The problem now is how to choose the value x .

First, we notice that the optimal schedules of lists L_1 , L_1L_2 , $L_1L_2L_3$, and $L_1L_2L_3L_4$ (or L) are shown in Figure 13. The optimal finish times of these schedules are:

- $\text{OPT}(L_1) = 1$,
- $\text{OPT}(L_1L_2) = 2$,
- $\text{OPT}(L_1L_2L_3) = 3 - x$,
- $\text{OPT}(L) = \text{OPT}(L_1L_2L_3L_4) = 4$.

Next, we will show that by enforcing the schedules of L_1 , L_1L_2 , $L_1L_2L_3$ to satisfy

$$A(L_1)/\text{OPT}(L_1) < A(L)/\text{OPT}(L), \quad (2.13)$$

$$A(L_1L_2)/\text{OPT}(L_1L_2) < A(L)/\text{OPT}(L), \quad (2.14)$$

and

$$A(L_1L_2L_3)/\text{OPT}(L_1L_2L_3) < A(L)/\text{OPT}(L), \quad (2.15)$$

we are inevitably led to a schedule of L as shown in Figure 14 such that

$$A(L)/\text{OPT}(L) = (7 - x)/4. \quad (2.16)$$

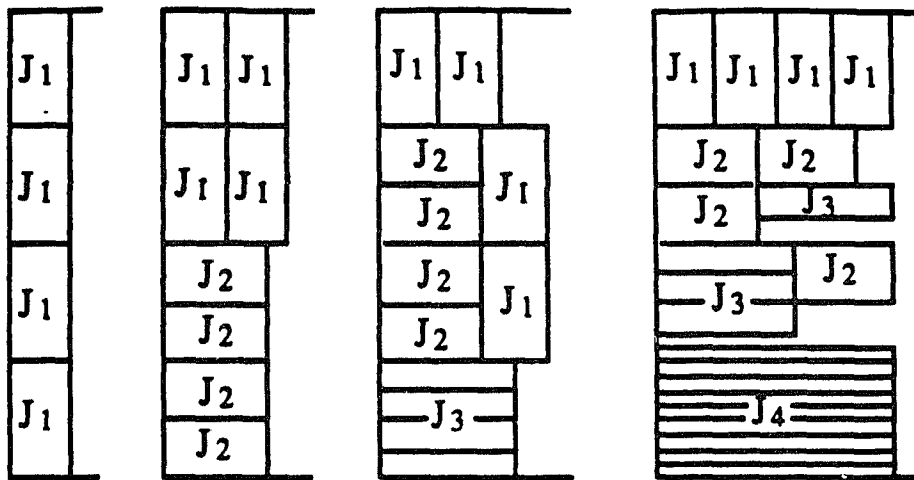


Figure 13: The optimal schedules.

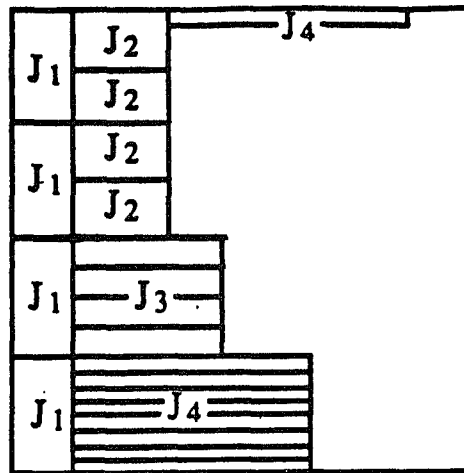


Figure 14: The forced schedule.

In the following, we shall look for the appropriate value of x so that the final schedule in Figure 14 is enforced. First, consider the partial scheduling of L_1 . To satisfy (2.13), we require

$$1/1 \leq A(L_1)/\text{OPT}(L_1) < (7 - x)/4 = A(L)/\text{OPT}(L). \quad (2.17)$$

Also, the algorithm must not schedule two jobs in L_1 above each other. Otherwise, (2.13) or

$$2/1 \geq A(L)/\text{OPT}(L) = (7 - x)/4 \quad (2.18)$$

will be violated. Thus the x that satisfies both of the requirements is

$$-1 \leq x < 3. \quad (2.19)$$

Next, consider the partial scheduling of L_1L_2 . Because of the schedule of L_1 explained above, we have to schedule jobs in L_2 over jobs in L_1 . To satisfy (2.14), we have

$$(1 + (2 - x))/2 \leq A(L_1L_2)/\text{OPT}(L_1L_2) < (7 - x)/4 = A(L)/\text{OPT}(L). \quad (2.20)$$

Also, the algorithm must not schedule two jobs in L_2 above each other. Otherwise, (2.14) or

$$(1 + 2(2 - x))/2 \geq A(L)/\text{OPT}(L) = (7 - x)/4 \quad (2.21)$$

will be violated. Combining the two requirements leads to

$$-1 < x \leq 1. \quad (2.22)$$

Now consider the partial scheduling of $L_1L_2L_3$. Since jobs in L_2 can only cover half of the cube, the jobs in L_3 can still be put on top of jobs in L_1 . For (2.15), we require

$$\begin{aligned} (1 + (2 + x))/(3 - x) &\leq A(L_1L_2L_3)/\text{OPT}(L_1L_2L_3) \\ &< (7 - x)/4 = A(L)/\text{OPT}(L). \end{aligned} \quad (2.23)$$

Again, the algorithm must not schedule two jobs in L_3 over a job in either L_2 or L_3 . Otherwise, (2.15) or

$$(1 + (2 - x) + (2 + x))/(3 - x) \geq A(L)/\text{OPT}(L) = (7 - x)/4 \quad (2.24)$$

will be violated. Combining the two inequations, we get

$$5 - 2\sqrt{6} \leq x < 7 - 2\sqrt{10}. \quad (2.25)$$

Combining (2.19), (2.22), (2.25) and $0 < x < 1$, we get

$$5 - 2\sqrt{6} \leq x < 7 - 2\sqrt{10}. \quad (2.26)$$

After the above scheduling of $L_1L_2L_3$, we can see that jobs in L_4 are forced to be scheduled the way we wanted, i.e., at least one job of L_4 has to be over a job from L_2 or L_3 . From (2.16), we see that $(7 - x)/4$ gets the largest value by letting x have its smallest value. From (2.26), we can choose $x = 5 - 2\sqrt{6}$. Then $A(L)/\text{OPT}(L) = (1 + \sqrt{6})/2$.

Thus, we have shown that it is impossible to have

$$\max \left\{ \frac{A(L_1)}{\text{OPT}(L_1)}, \frac{A(L_1L_2)}{\text{OPT}(L_1L_2)}, \frac{A(L_1L_2L_3)}{\text{OPT}(L_1L_2L_3)}, \frac{A(L)}{\text{OPT}(L)} \right\} < (1 + \sqrt{6})/2. \quad (2.27)$$

The theorem is, therefore, proven. \square

2.6 Conclusion

In this chapter, we have presented an algorithm called LDF. The LDF algorithm is very simple and has almost the same performance as the earlier algorithms (i.e., LDLPT and FFDH). It requires a decreasing dimension job list, and schedules each job to the earliest available subcube. It has an absolute bound less than 2 and an optimal asymptotic bound.

One important practical advantage of the LDF algorithm is that a system scheduler can perform very well without knowing the job execution times. This is especially true when the job execution times are hard to get.

The LDF algorithm tells us that as long as we can schedule jobs in decreasing dimension order, we can get an absolute bound less than 2. The asymptotic bound of LDF being optimal means that as the number of jobs becomes large, the system tends to be fully utilized by using LDF.

We have also proved a lower bound for the absolute bounds of a class of algorithms including LDF, LDLPT, and FFDH. This result tells us that it is unlikely to find simple heuristic algorithms that can perform much better than these algorithms.

CHAPTER III

PREEMPTIVE SCHEDULING ON A HYPERCUBE

3.1 Introduction

In this chapter, we shall restrict our attention to the preemptive hypercube scheduling problem. Consider an m -cube and a set J of n independent jobs. Each job $J_i = (d_i, t_i)$ is preemptive, i.e., it may be interrupted before its completion, but its execution will be resumed at a later time, possibly on a different subcube. All the preemptions are assumed to take no time. In this case, the minimum finish time schedule can be found in polynomial time. Our goal is to find an algorithm that has fast running time and can generate a schedule with a small number of preemptions.

This chapter is organized as follows. First, we summarize previous research results. We also provide related definitions. Next we present a feasibility algorithm. We also prove the correctness and analyze the time complexity of our feasibility algorithm. Finally, we give concluding remarks.

3.2 Previous Research

In [6], Blazewicz, Drabowski and Weglarz studied a multiprocessor scheduling model similar to two dimensional bin packing problem. The difference is that each job, or

rectangle in two dimensional bin packing, can now be preempted during execution.

More specifically, the problem is as follows

Suppose we have n' jobs and m' identical machines. Each job may require p , where $1 \leq p \leq m'$, machines for t units of time. All the jobs are independent. The goal is to find an optimal preemptive schedule.

First they studied a restricted case when all the jobs may require either of the two different numbers of machines, say either p_1 or p_2 , where $1 \leq p_1, p_2 \leq m'$. For this case, they gave an $O(n)$ algorithm. For the general case, when jobs can require any different number of machines, they formulated the solution as a linear programming problem. Although this proved that the optimal schedule can be found in polynomial time, the resulting algorithm is too costly to be used for large values of n and m .

In [11], Chen and Lai also studied the nonpreemptive hypercube scheduling problem. They developed a *feasibility algorithm*, which is used to decide if it is feasible to finish all the jobs by a deadline T and generate a schedule (called *feasible schedule*) if this can be done. To find the minimum finish time schedule, their algorithm uses a binary search over a time interval by calling their feasibility algorithm repeatedly. Their feasibility algorithm may be summarized as follows:

The jobs are ordered according to decreasing dimensions, i.e., $d_1 \geq d_2 \geq \dots \geq d_n$. Job J_1 is scheduled on the first d_1 -subcube. Each of the remaining jobs is scheduled in the smallest indexed subcube of its dimension as early as possible and as long as possible.

The feasibility algorithm runs in $O(n^2)$ time and it produces a feasible schedule which can have up to $n(n-1)/2$ number of preemption. Since the optimal time cannot be longer than $\sum_{i=1}^n t_i$, the optimal schedule can be found through a binary search over the time interval. Thus, the total running time for finding the optimal schedule is $O(n^2 \log(\sum_{i=1}^n t_i))$.

In the following, we will first develop a faster feasibility algorithm which can also generates a schedule with much fewer number of preemptions. Also, we will show that the time interval in the binary search can be slightly reduced.

3.3 Definitions

Since we consider the problem of scheduling a set of jobs $J = \{J_i : 1 \leq i \leq n\}$ on an m -cube to meet a given deadline T , we need a few definitions related to the deadline T .

Definition 8 For a deadline T , the remaining processing time (or RPT) of processor p is $T - f$, where f is the finish time of processor p .

Let P be a profile of an entire cube schedule defined by

$$\langle ([a_1, b_1], f_1), \dots, ([a_l, b_l], f_l) \rangle.$$

For a deadline T , the RPT of a p -interval is defined as:

Definition 9 Let f_i be the finish time of a p -interval $[a_i, b_i]$. The RPT of the p -interval, denoted by r_i , is defined as $r_i = T - f_i$.

Since the p-intervals with no RPTs left in S cannot be used to schedule the remaining jobs, we may simply keep those p-intervals with nonzero RPTs (i.e., $f_i < T$) in the profile P . From now on, we use l to stand for the number of such p-intervals (not necessarily consecutive) in P .

Definition 10 Let Υ be $\max\{(\sum_{i=1}^n 2^{d_i} t_i)/2^m, t_{\max}\}$, i.e., the lower bound for the finish time of any schedule.

Clearly, the deadline T given by our feasibility algorithm must satisfy $T \geq \Upsilon$.

3.4 Feasibility Algorithm

We derive below a feasibility algorithm for scheduling a job set $J = \{J_i : 1 \leq i \leq n\}$, where $J_i = (d_i, t_i)$, on an m -cube to meet a given deadline T . Assume that the jobs have been ordered so that $(\forall i : 1 \leq i < n : d_i \geq d_{i+1})$. We schedule the jobs one by one in this order. Let $S(i)$ and $P(i)$ be the schedule and the profile after J_i is scheduled, respectively. Let $S(0)$ be the initial schedule when no job has been scheduled yet, and let the (stair-like) profile $S(0)$ be $P(0) = \langle ([0, 2^m - 1], 0) \rangle$. For ease of presentation, we may view $S(0)$ as the schedule obtained after a dummy job $J_0 = (m, 0)$ is scheduled.

Let l be the number of p-intervals with nonzero RPTs in $P(i - 1)$. If $l = 0$, then J_i cannot be scheduled. Otherwise, let the profile $P(i - 1)$ be

$$\langle ([a_1, b_1], f_1), \dots, ([a_l, b_l], f_l) \rangle$$

and assume that it is stair-like. Let $r_j = T - f_j$ be the RPT of $[a_j, b_j]$. Then the p-intervals in $P(i - 1)$ are also ordered in increasing order of their RPTs.

Depending on which one is applicable, $J_i = (d_i, t_i)$ is scheduled by applying one of the following four rules:

R0. If $t_i > r_l$, then J_i cannot be scheduled, return “infeasible”.

R1. If $t_i < r_1$, then J_i is scheduled entirely on subcube $[a_1, a_1 + 2^{d_i} - 1]$ from f_1 to $f_1 + t_i$.

R2. If $(\exists j : 1 \leq j \leq l : t_i = r_j)$, then J_i is scheduled entirely on subcube $[a_j, a_j + 2^{d_i} - 1]$ to use up all its RPT r_j .

R3. If $(\exists j : 1 \leq j < l : t_i > r_j \wedge t_i < r_{j+1})$, then J_i is scheduled in subcube $[a_j, a_j + 2^{d_i} - 1]$ to use up its RPT r_j and the remaining time $t_i - r_j$ of J_i is scheduled on subcube $[a_{j+1}, a_{j+1} + 2^{d_i} - 1]$ from f_{j+1} to $f_{j+1} + (t_i - r_j)$.

After assigning J_i , the algorithm generates the new schedule $S(i)$. In Section 3.4, we shall show that the profile $P(i)$ is kept stair-like.

The above four rules are actually inspired by the work of Sahni [62]. In [62], Sahni gave an algorithm for scheduling n' jobs with different deadlines on m' identical machines. His algorithm first orders the jobs into sets, each set corresponds to a deadline. The algorithm schedules the set of jobs with the earliest deadline first, then the set of jobs with the next deadline, and so on. During the scheduling of jobs in each set, the algorithm uses a number of rules similar to the above ones. But here, the jobs in our algorithm may require a subcube, while the jobs there only require one machine.

The above ideas are formalized as an algorithm called *F-Schedule* in Figure 15. We assume that the n jobs have already been ordered by decreasing dimension. During the scheduling of job J_i , subroutine $Find(d_i, t_i, flag, a_j, a_{j+1})$ produces a *flag* which is a number between 0 and 3 corresponding to the four cases, and two start points a_j and a_{j+1} in $P(i-1)$. Note that a_{j+1} is used only for case 3. To illustrate how the algorithm works, we give an example below.

Example: Consider a 4-cube, a deadline $T = 4$, and a job set J with five jobs ordered by decreasing dimension as $J_1 = (3, 2)$, $J_2 = (2, 3)$, $J_3 = (1, 3.5)$, $J_4 = (1, 1.5)$, and $J_5 = (0, 3.5)$. Figure 16 shows the resulting schedule from our feasibility algorithm. The scheduling of each job is explained below:

1. Job $J_1 = (3, 2)$ is scheduled on the first 3-subcube by rule R1, and the profile

$P(1)$ is

$$\langle ([0, 7], 2), ([8, 15], 0) \rangle.$$

2. Job $J_2 = (2, 3)$ is scheduled by rule R3, and the profile $P(2)$ is

$$\langle ([4, 7], 2), ([8, 11], 1), ([12, 15], 0) \rangle.$$

3. Job $J_3 = (1, 3.5)$ is also scheduled by rule R3, and the profile $P(3)$ is

$$\langle ([4, 7], 2), ([10, 11], 1), ([12, 13], 0.5), ([14, 15], 0) \rangle.$$

4. Job $J_4 = (1, 1.5)$ is scheduled by rule R1, and the profile $P(4)$ is

$$\langle ([4, 5], 3.5), ([6, 7], 2), ([10, 11], 1), ([12, 13], 0.5), ([14, 15], 0) \rangle.$$

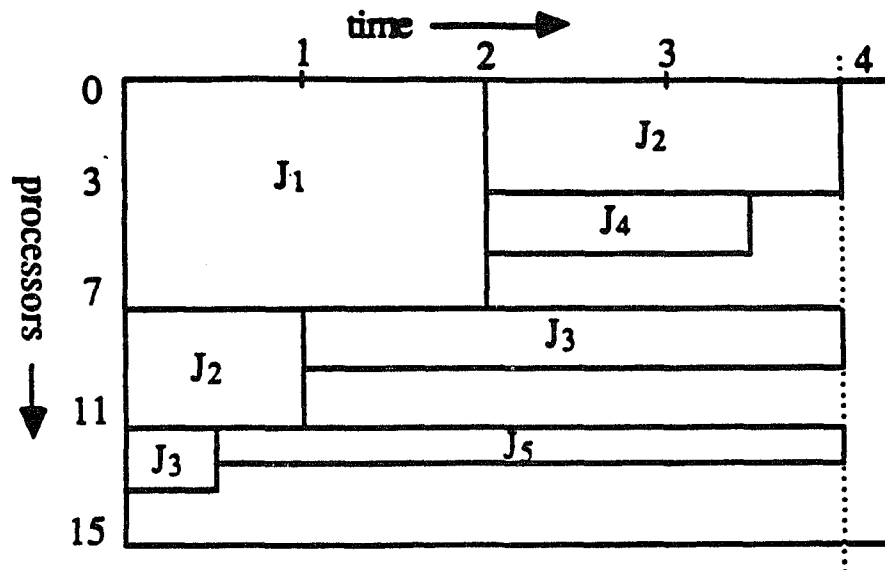


Figure 16: The schedule from our feasibility algorithm.

5. Job $J_5 = (0, 3.5)$ is scheduled by rule R2, and the profile $P(5)$ is

$$\langle ([4, 5], 3.5), ([6, 7], 2), ([10, 11], 1), ([13, 13], 0.5), ([14, 15], 0) \rangle.$$

It is instructive to make a comparison here between our feasibility algorithm and the algorithm by Chen and Lai [11]. Their algorithm maintains a stair-like profile with *consecutive* p-intervals throughout the scheduling. Consider the same example as one in Figure 16 and the use of Chen and Lai's feasibility algorithm. The resulting schedule is shown in Figure 17. We see clearly that the profile in Figure 17 is stair-like and has consecutive p-intervals.

Like Chen and Lai, we realize that it would be difficult to manage each profile $P(i - 1)$ and to schedule the next job J_i if $P(i - 1)$ were not stair-like. However, we also observe that a stair-like profile with *consecutive* p-intervals is unnecessary. Consider the profiles of the schedules in Figure 16 and Figure 17, each profile consists of those p-intervals with nonzero RPTs in each figure. They are actually *identical*, in the sense that both profiles contain the same number of pairs, and for the j th pair $([a_j, b_j], f_j)$ in one profile, we have the j th pair $([a'_j, b'_j], f_j)$ in the other profile such that $|[a_j, b_j]| = |[a'_j, b'_j]|$. This fact becomes more obvious if we compare the space not occupied by the jobs in Figure 17 and Figure 18, where Figure 18 is obtained by grouping together those p-intervals with zero RPTs in Figure 16. (Dotted lines are used to show how Figure 18 is constructed.) The identity of the two profiles is actually maintained during the entire scheduling session for the two algorithms. By choosing the appropriate subcube(s) for each job, our algorithm can reduce both the running time and the number of preemptions. This is what we are going to show

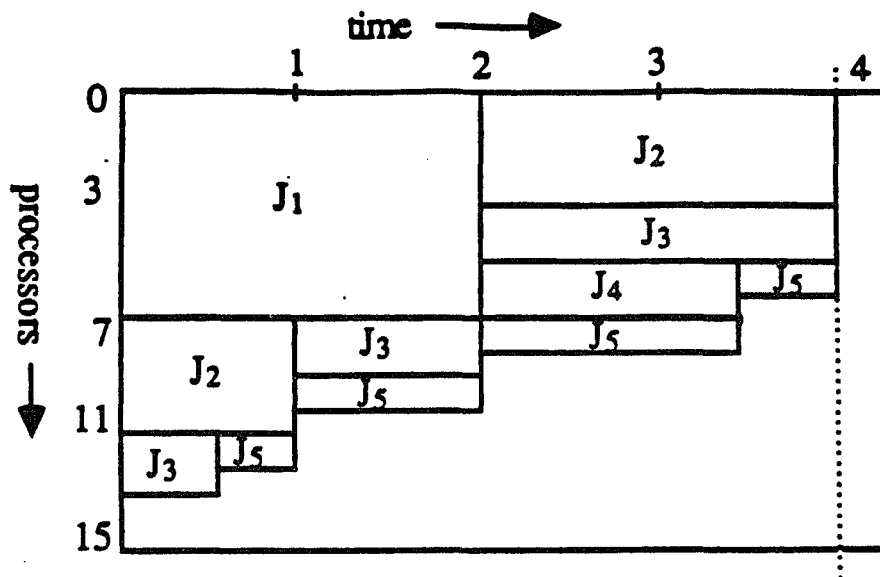


Figure 17: The schedule from Chen and Lai's feasibility algorithm.

next.

3.5 Correctness and Analysis

Suppose the next job to be scheduled is J_i , and let $S(i-1)$ and $P(i-1)$ be defined as before.

Lemma 3.5.1 *For any $1 \leq i \leq n$ and $J_i = (d_i, t_i)$, (1) for any p-interval $[a, b]$ in $P(i-1)$, $|[a, b]|$ is a multiple of 2^{d_i} , and (2) after J_i is scheduled, $P(i)$ is stair-like.*

Proof: The lemma is clearly true when $i = 1$. For $i \geq 2$, suppose that (1) and (2) are true for any number less than i , we prove that they hold for i . Then by induction, the lemma is true.

To prove (1), we observe that since $[a, b]$ comes from some p-interval $[a', b']$ in $P(i-2)$ after job J_{i-1} is assigned, $|[a, b]|$ must be equal to one of the following three: $2^{d_{i-1}}$, $|[a', b']| - 2^{d_{i-1}}$, or $|[a', b']|$. By induction hypothesis, $|[a', b']|$ is a multiple of $2^{d_{i-1}}$. Furthermore, $d_{i-1} \geq d_i$, so $2^{d_{i-1}}$ is a multiple of 2^{d_i} . Hence claim (1) holds for i .

To prove (2), we need to check only those p-intervals in $P(i-1)$ that are affected by the assignment of J_i . If J_i is scheduled by R1 or R2, then the proof is straightforward. Now suppose J_i is scheduled by R3 on the two d_i -subcubes starting at a_j and a_{j+1} in pairs in $P(i-1)$. Then $[a_j, a_j + 2^{d_i} - 1]$ has no RPT left in $S(i)$, so it is simply deleted from $P(i)$. We need to show that the new finish time $f_{j+1} + (t_i - r_j)$ of $[a_{j+1}, a_{j+1} + 2^{d_i} - 1]$ is less than f_j (the finish time of the remaining p-interval $[a_j + 2^{d_i}, b_j]$). But

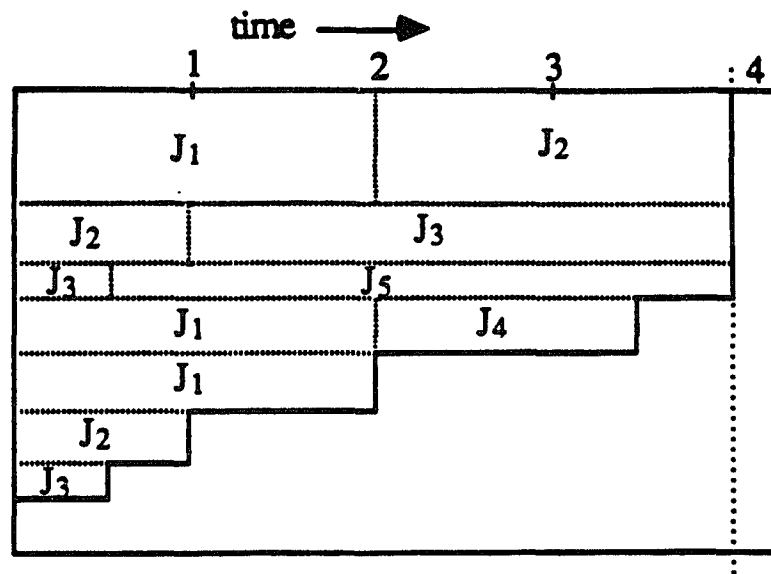


Figure 18: The profile from our feasibility algorithm after grouping.

this is true because of the way the two subcubes are chosen. Hence claim (2) holds for i . \square

This lemma shows the importance of scheduling jobs in decreasing dimension order. By doing so, J_i can be scheduled on the d_i -subcube starting at any a_j in $P(i-1)$. The stair-like property will be used in the proof of the next theorem.

Theorem 3.5.2 *The algorithm generates a feasible schedule if and only if the given jobs can be feasibly scheduled.*

Proof: We need only to prove the “if” part. Let S be any feasible schedule of the job set J with deadline T . Assume that in S jobs J_0, J_1, \dots, J_{i-1} are scheduled as in $S(i-1)$. Clearly this assumption holds for $i = 1$. We show that S can be modified such that J_i is also scheduled in S in $S(i)$. Then by induction, S can be transformed to $S(n)$, the schedule generated by the algorithm.

Let $P(i-1) = \langle ([a_1, b_1], f_1), \dots, ([a_i, b_i], f_i) \rangle$ be the profile of $S(i-1)$. It must be true that $f_i + t_i \leq T$, since in $S - S(i-1)$ job J_i is scheduled between times f_i and T . Thus, the algorithm is able to schedule J_i and generate schedule $S(i)$.

Suppose in $S(i)$ job $J_i = (d_i, t_i)$ is scheduled on subcube $A = [a_j, a_j + 2^{d_i} - 1]$ from f_j to $\Pi = f_j + t_i$ by R1 (in which case $j = 1$) or R2, or on subcube A from f_j to T and also on subcube $B = [a_{j+1}, a_{j+1} + 2^{d_i} - 1]$ from f_{j+1} to $\Pi' = f_{j+1} + (t_i - r_j)$ by R3, where a_j, a_{j+1}, f_j , and f_{j+1} are from $P(i-1)$. If in S , J_i is scheduled in the same manner as in $S(i)$, then we are done. Otherwise, we modify S by rearranging the jobs J_i, \dots, J_n in $S - S(i-1)$ so that J_i is scheduled as in S just as in $S(i)$:

(1) Divide the time interval $[0, T]$ into intervals of equal length δ , with each interval called a *t-interval*, so that each job in S is preempted or finished only at an endpoint of some t-interval. This can always be done by choosing δ sufficiently small. For a t-interval θ , let $J(\theta) = \{J_k : i \leq k \leq n, J_k \text{ is scheduled in } S \text{ over } \theta\}$.

(2) Divide the m -cube into 2^{m-d_i} d_i -subcubes across the entire interval $[0, T]$. Then line up the jobs in $J(\theta)$ over each t-interval θ in $[0, T]$, so that no job is scheduled on two d_i -subcubes. (Note that $(\forall J_k \in J(\theta) : d_k \leq d_i)$.)

(3) To achieve the final rearrangement, we first move J_i to time interval $[T - t_i, T]$. We require that $T' = T - t_i$, Π , and Π' are endpoints of the t-intervals as well. Let $\Delta' = \{\theta' : \theta' \text{ is a t-interval on the left of } T' \text{ and } J_i \in J(\theta')\}$, and let $\Delta = \{\theta : \theta \text{ is a t-interval on the right of } T' \text{ and } J_i \notin J(\theta)\}$. Then the sizes of Δ' and Δ must be equal. Let g be any 1-1 function between Δ' and Δ , i.e., we have $\Delta = \{\theta : \theta = g(\theta') \wedge \theta' \in \Delta'\}$. Now we move J_i over t-intervals in Δ' to d_i -subcubes over t-intervals in Δ . Let $\theta' \in \Delta'$, and let $\theta = g(\theta') \in \Delta$. By the stair-like property of $P(i-1)$ (from induction hypothesis and lemma 3.5.1), the number of d_i -subcubes in $S - S(i-1)$ over θ' is no more than the one over θ . Since J_i is over θ' but not over θ , there are (at least) 2^{d_i} processors over θ occupied either by jobs in $J(\theta) - J(\theta')$ or by empty p-intervals. So over θ , we can either find a d_i -subcube or arrange one (by moving around jobs in $J(\theta)$), so that it only contains jobs in $J(\theta) - J(\theta')$ or empty p-intervals. Thus, we can interchange job J_i over θ' with the one in the d_i -subcube over θ .

(4) Finally, we move J_i now in $[T', T]$ to the desired subcubes and time intervals, depending on the rule used for scheduling J_i in $S(i)$:

If R2 is used, then $(T' = f_j \wedge T = \Pi)$. For each θ in $[T', T]$, we interchange J_i in its d_i -subcube with the one in A and we are done.

If R1 is used, then $(T' > f_1 \wedge \Pi > f_1)$. For each θ in $[T', T]$, we do the same as when R2 is used. We swap J_i in subcube A over $[T', T]$ with the one in A over $[f_1, \Pi]$. Because A extends from f_1 to T in $S - S(i - 1)$, the swapping can be done.

If R3 is used, then $(f_j > T' > f_{j+1}) \wedge (f_j > \Pi' > f_{j+1})$. For each θ in $[f_j, T]$, we do the same as when R2 is used. But for each θ in $[T', f_j]$, we interchange J_i with that in B . We swap J_i in subcube B over $[T', f_j]$ with the one in B over $[f_{j+1}, \Pi']$. Since B extends from f_{j+1} to T in $S - S(i - 1)$, the swapping can be done. \square

Theorem 3.5.3 *The feasibility algorithm generates a feasible schedule with at most $\min\{n - 2, 2^m - 1\}$ number of preemptions.*

Proof: The first term in the above expression follows almost exactly from the arguments in [62]. J_1 is scheduled with no preemption, and each of the remaining jobs may have at most one preemption. Hence, the maximum number of preemptions is $n - 1$. This number can be reduced to $n - 2$ by scheduling the first $n - 1$ jobs using the algorithm and assigning job J_n entirely on the d_n -subcube $[a_l, a_l + 2^{d_n} - 1]$ starting at f_l in $P(n - 1)$. (Recall that f_l has the smallest finish time.)

Each preempted job J_i must occupy a d_i -subcube and finish on deadline T , with the exception of the d_n -subcube for J_n on which no job is preempted. Since there are at most 2^m processors in an m -cube, the theorem is then proven. The worst case comes when all the preempted jobs plus J_n need 0-subcubes. This can be seen as the

worst case for preemptive scheduling on 2^m identical processors (see Theorem 2.6 of [18]). \square

Now we examine the running time of the algorithm. The algorithm is implemented by using a balanced search tree, which can be either an AVL tree or a 2-3 tree [35]. An example balance search tree is shown in Figure 19.

Theorem 3.5.4 *The feasibility algorithm runs in $O(n \log n)$ time.*

Proof: Obviously the task of sorting jobs in decreasing dimension order, if needed, takes $O(n \log n)$ time. We shall show that each job assignment can be done in $O(\log n)$ time. Then the theorem follows.

Let each node in the search tree have two fields $[a, b]$ and f , meaning that the p-interval $[a, b]$ has the finish time f . Initially, there is one node with its two fields as $[0, 2^m - 1]$ and 0. After each job assignment, at most one new $f \neq T$ is resulted (from R1 or R3), and we insert one new node in the search tree with the assigned subcube and this f as the two fields. When a subcube is used up after a job assignment (from R2 or R3), we delete the assigned subcube from the p-interval in its original node. For the case when this deleting results in an empty p-interval, then the node is deleted from the search tree. Thus, the total number of nodes in the search tree will be at most $n + 1$. Each of these node insertions or deletions, together with any needed rebalancing, can be done in $O(\log n)$ time.

Determining which rule to apply for each job also takes $O(\log n)$ time. This includes no more than a constant number of searches on the tree. \square

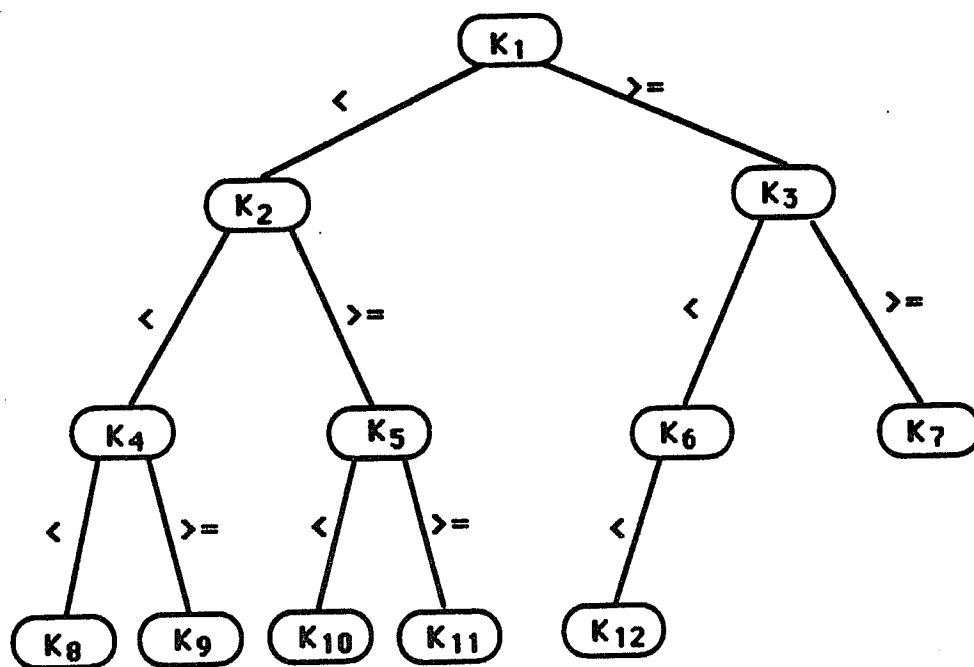


Figure 19: A balanced search tree.

3.6 On Searching for the Minimum Finish Time

Next we show that to find the minimum finish time, we can do a binary search over a time interval from Υ to 2Υ . Recall that $\Upsilon = \max\{\sum_{i=1}^n 2^{d_i} t_i / 2^m, t_{\max}\}$, i.e., the lower bound for the finish time of any schedule. This time interval is less than the time interval from Υ to $\sum_{i=1}^n t_i$, used by Chen and Lai algorithm.

Theorem 3.6.1 *The minimum finish time schedule can be found by calling the feasibility algorithm no more than $O(\log \Upsilon)$ times.*

Proof: First we prove that it is feasible to schedule J with deadline 2Υ . From inequation (2.4) in the proof of Theorem 2.4.1 in Chapter II, we have

$$T_{\text{LDF}} \leq \left(\sum_{i=1}^k 2^{d_i} t_i \right) / 2^m + ((2^m - 2^{d_1}) / 2^m) t_k \quad (3.1)$$

$$\leq \left(\sum_{i=1}^n 2^{d_i} t_i \right) / 2^m + ((2^m - 1) / 2^m) t_k \quad (3.2)$$

$$\leq \Upsilon + ((2^m - 1) / 2^m) \Upsilon \quad (3.3)$$

$$\leq 2\Upsilon. \quad (3.4)$$

Thus, the finish time of the LDF schedule for J is no greater than 2Υ . Then by Theorem 3.5.2, J can be scheduled with deadline 2Υ . Using a binary search, we see that the minimum finish time can be found by calling the feasibility algorithm no more than $O(\log \Upsilon)$ times. \square

3.7 Conclusion

In this chapter, we have studied the problem of preemptive scheduling of n independent jobs on an m -cube. A new feasibility algorithm is presented that runs

in $O(n \log n)$ time and generates a schedule with at most $\min\{n - 2, 2^m - 1\}$ preemptions, while a previous known feasibility algorithm runs in $O(n^2)$ time and produces a schedule with up to $n(n - 1)/2$ preemptions. The minimum finish time schedule is obtained through a binary search over a time interval of length $\Upsilon = \max\{\sum_{i=1}^n 2^{d_i} t_i / 2^m, \max\{t_i : 1 \leq i \leq n\}\}$. This length is less than $O(\sum_{i=1}^n t_i)$, which is used by the previous algorithm. Thus, the total running time for finding the minimum finish time schedule is $O(n \log n \log \Upsilon)$.

Since feasibility testing is one of the basic operation in job scheduling and many scheduling algorithms require an efficient feasibility algorithm as a building block, improvements resulting from our feasibility algorithm are important. Further, as research presented in the sequel shows, our algorithm leads well to the development of a polynomial algorithm for finding minimum finish time schedule.

It appears unlikely to reduce the running time of our feasibility algorithm even further. Since scheduling job by decreasing dimension appears to play a fundamental role in both nonpreemptive and preemptive scheduling, any algorithm would need a sorting procedure which takes $O(n \log n)$ time.

In our studies, we have assumed that preemption takes no time. Although the assumption is very common in scheduling theory, it would be interesting to consider the case where job preemption cost is not negligible.

CHAPTER IV

A NEW ALGORITHM FOR PREEMPTIVE SCHEDULING

4.1 Introduction

From the studies in the previous chapter, we have learned that we can find the minimum finish time through a binary search over a time interval. The problem with such a binary search is that the number of calls to the feasibility algorithm depends on the numerical values of the job times t_i 's, and finding the exact value of the minimum finish time may take a long time. So, the complexity of the resulting algorithm for finding the minimum finish time schedule is not a polynomial in n , the number of jobs to be scheduled.

We show below that through an advanced search technique due to Megiddo [50], the exact value of the minimum finish time can be found by calling our feasibility algorithm no more than $O(n \log n)$ times. The total running time for finding the minimum finish time schedule will be shown to be $n^2 \log^2 n$. The time complexity now depends only on n . Therefore, such an algorithm has a truly polynomial complexity in n .

In the following, we will first introduce Megiddo's search technique. We will apply the technique to find the minimum finish time schedule for the problem at hand.

4.2 Megiddo's Search Method

Many combinatorial optimization problems can be formulated as linear minimization problems subjected to certain constraints. In [50], Megiddo considered a linear minimization problem (called problem A) and a ratio minimization problem (called Problem B).

Problem A:

$$\begin{aligned} & \text{Minimize } c_1x_1 + \dots + c_nx_n \\ & \text{such that } \mathbf{x} = (x_1, \dots, x_n) \in D, \end{aligned}$$

Problem B:

$$\begin{aligned} & \text{Minimize } (a_0 + a_1x_1 + \dots + a_nx_n)/(b_0 + b_1x_1 + \dots + b_nx_n) \\ & \text{such that } \mathbf{x} \in D. \end{aligned}$$

where D is assumed to be the solution space of vectors of length n for the problems.

An example of a practical ratio minimization problem is that of minimizing cost-to-time ratio. Megiddo proved the following relationship between the time complexity of problem B and that of problem A.

Theorem 4.2.1 *If problem A is solvable within $O(p(n))$ comparisons and $O(q(n))$ additions then problem B is solvable in time $O(p(n)(q(n) + p(n)))$.*

The proof given in [50] is constructive and leads to an algorithm for solving problem B. This result is not directly related to our hypercube scheduling problem. But

technique he developed is very general and applicable to solving hypercube scheduling problem. In his case, he solved problem B by calling the algorithm for problem A. In our case, we will try to find the minimum finish schedule by calling our feasibility algorithm. Therefore, we will explain the technique in his proof. This technique will be used later for developing an algorithm for our problem.

Meggido explored the following rather standard trick for solving ratio minimization problems. Given problem B, pick a real number t and get the following equation:

$$(a_0 + a_1x_1 + \dots + a_nx_n)/(b_0 + b_1x_1 + \dots + b_nx_n) = t, \quad (4.1)$$

or

$$(a_1 - b_1t)x_1 + \dots + (a_n - b_nt)x_n = tb_0 - a_0. \quad (4.2)$$

Next solve problem A with parameters $c_i = a_i - tb_i$, where $1 \leq i \leq n$.

Suppose that v is the optimal value of problem A. Then depending on the value of t , we can have the following three cases:

- If $v = tb_0 - a_0$, then t is the optimal value of problem B; and the optimal solution vector x for problem A (for achieving the optimal value v) is also an optimal solution vector for problem B.
- If $v < tb_0 - a_0$, then we know that a smaller t should be picked.
- If $v > tb_0 - a_0$, then we know that a greater t should be picked.

This procedure continues until the "correct" value t^* is found. The question is how to limit the number of tests of t before the "correct" t^* is found. Assume we already

have an algorithm for problem A, called A-algorithm. Then the question becomes how to use A-algorithm to search for this t^* .

Depending on the value of t , the A-algorithm may follow different paths when solving A with parameters $c_i = a_i - tb_i$, $1 \leq i \leq n$. These paths form a directed search tree where branching points correspond to comparisons made by the A-algorithm.

At the beginning the parameters are linear functions (possibly constants) of t . Consider the first comparison made by the A-algorithm. Since the algorithm compares two linear functions of t , the outcome of the comparison may depend on t . However, in any case there will be at most one critical value, say t_1 , such that for all $t \leq t_1$, one of the functions compared is greater than or equal to the other, and for $t \geq t_1$ the other function is greater than or equal to the first one. Thus, the comparison may partition the real line into two halves: $[-\infty, t_1]$ and $[t_1, \infty]$. The comparison corresponds to a branching point with an outgoing degree of 2, and the two subtrees correspond to the two halves: $[-\infty, t_1]$ and $[t_1, \infty]$.

As A-algorithm continues its execution, more comparisons are made. Each comparison in the algorithm introduces a branching point in the search tree. In general, a comparison occurs when the search is in a subtree over an interval $[e, f]$, where e and f are some real numbers satisfying $-\infty \leq e < f \leq \infty$. Since the comparison involves two linear functions of t , there will be a critical value t' in $[e, f]$ and the corresponding branching point will have two subtrees corresponding to the two subintervals $[e, t']$ and $[t', f]$.

The above observation gives the following algorithm for solving problem B: solve problem A parametrically over an interval which reduces throughout the computation, and search for the “correct” value t^* . At each branching point reached in the tree of the A-algorithm, the corresponding critical value of t is tested by running A-algorithm with t fixed at the critical value. Then the appropriate branch is selected and the next branching point is considered. At the end, the optimal value of problem A will be given in the form of a linear function $v(t)$ defined over an interval $[e, f]$ which contains t^* . The value t^* is then calculated by solving $v(t) = tb_0 - a_0$.

4.3 Initial Algorithm Derivation

We have introduced Megiddo’s search technique above. The method is very general and very powerful. In fact, it has been used before by Martel for a classical scheduling problem [49]. We will use the search technique for our preemptive hypercube job scheduling problem, and the new algorithm to be developed will be called the *search algorithm*.

Let T^* denote the minimum finish time needed to schedule the n jobs J_i ’s in J on the m -cube. To find the minimum finish time schedule, we only need to find T^* . Since once we know T^* , we can obtain the minimum finish time schedule by running the feasibility algorithm with T^* as the deadline.

We will search for this T^* as an unknown deadline by calling the feasibility algorithm as a guide. Whenever we need to do a comparison that depends on the unknown deadline, we generate a testing value, T , such that if $T^* \leq T$ then the comparison will be true and if $T^* > T$ then the comparison will be false. To determine the result

of the comparison, we call the feasibility algorithm with a deadline T . This is similar to determining which branch to take at a branching point in Megiddo's algorithm for problem B.

But how can we determine the optimal value T^* for our problem? In Megiddo's algorithm, the final value of t^* is found by solving an equation. Here we make the following important observation for our case. The actual value of T^* is found *iff* the feasibility algorithm running with the testing deadline T can successfully schedule all the jobs and it completes some job, say J_i , using the entire longest RPT of the profile $P(i - 1)$, i.e., $t_i = r_i$. (Similar observation is made in [49] as well.) This fact is apparent. Suppose we are given a smaller deadline T , then we will not be able to schedule J_i since the longest RPT will be too short. On the other hand, if we are given a larger T , then the longest RPT will be too long for any job to use all of it.

Therefore, in the search algorithm to be developed, we will try to detect this condition. We need to make some modifications to our feasibility algorithm for this. When the feasibility algorithm is called with a deadline T , the modifications corresponds to the three cases that may arise:

1. $T = T^*$, the above fact will be detected. We make the feasibility algorithm return "found";
2. $T < T^*$, then one of the jobs in J will not be able to finish within T . We make the feasibility algorithm return "infeasible";
3. $T > T^*$, then all the jobs in J can be scheduled before T . We make the feasibility algorithm return "feasible".

The modified feasibility algorithm is listed in Figure 20.

4.4 Detailed Algorithm

The search algorithm basically works in a similar manner as the feasibility algorithm does. Before, the feasibility algorithm can determine where to schedule each job by comparing t_i of a job J_i with the RPTs of the current profile. Since we do not know the T^* , the search algorithm will make calls to the modified feasibility algorithm to determine where and how to schedule each job. Each call will generate a testing deadline T , and such a call is referred as a *test*. Eventually, one of these testing deadlines will cause the feasibility algorithm to return “found”. Once “found” is returned, we know the feasible schedule generated by the feasibility algorithm is the minimum finish time schedule.

During the computation, the nonzero RPT values r_i of the p-intervals in a profile will be recorded as linear functions of T^* . Initially, $r_1 = T^*$. (Recall that $P(0) = \langle ([0, 2^m - 1], 0) \rangle$.) After J_1 is scheduled, we will have two p-intervals with their RPTs being $r_1 = T^* - t_1$ and $r_2 = T^*$, respectively. So after each job J_i is scheduled, the r_j values in $P(i - 1)$ are updated (as we will describe), and a new set of linear functions of T^* will be formed in $P(i)$.

Now assume that after the jobs J_1, \dots, J_{i-1} have been scheduled, we have a stair-like profile $P(i - 1)$ (for $l \geq 1$)

$$\langle ([a_1, b_1], f_1), \dots, ([a_l, b_l], f_l) \rangle.$$

Each f_i , where $1 \leq i \leq l$, is a linear function of T^* . Since $r_i = T^* - f_i$, we shall use

Algorithm F' -Schedule;

/ The algorithm generates a feasible preemptive schedule on an m -cube for a job set J containing n jobs with deadline T . It also returns a message which can be: infeasible, feasible, found */*

```

F([0,  $2^m - 1$ ]) := 0;    /* all processors are initially free */
message := 'feasible';
for  $i := 1$  to  $n$  do
    Find( $d_i, t_i, flag, a_j, a_{j+1}$ );
    case flag of
        0: return('infeasible');    /* can return before the loop is finished */
        1: print('schedule',  $J_i$ , 'on', [ $a_j, a_j + 2^{d_i} - 1$ ], 'from',  $f_j$ , 'to',  $f_j + t_i$ );
           F([ $a_j, a_j + 2^{d_i} - 1$ ]) :=  $f_j + t_i$ ;
        2: print('schedule',  $J_i$ , 'on', [ $a_j, a_j + 2^{d_i} - 1$ ], 'from',  $f_j$ , 'to',  $T$ );
           F([ $a_j, a_j + 2^{d_i} - 1$ ]) :=  $T$ ;
           if  $j = l$  then message := 'found';    /* the minimum finish time is detected */
        3: print('schedule',  $J_i$ , 'on', [ $a_j, a_j + 2^{d_i} - 1$ ], 'from',  $f_j$ , 'to',  $T$ );
           print('schedule',  $J_i$ , 'on', [ $a_{j+1}, a_{j+1} + 2^{d_i} - 1$ ], 'from',  $f_{j+1}$ , 'to',
                $f_{j+1} + (t_i - r_j)$ );
           F([ $a_j, a_j + 2^{d_i} - 1$ ]) :=  $T$ ;
           F([ $a_{j+1}, a_{j+1} + 2^{d_i} - 1$ ]) :=  $f_{j+1} + (t_i - r_j)$ 
    endcase
endfor;
return(message);    /* reach here only if not returned in case 0 */

```

Figure 20: The modified feasibility algorithm.

both the r_i and the f_i of $[a_i, b_i]$ in expressing our results. Let $r_i = x_i T^* + y_i$ (and $f_i = T^* - r_i = (1 - x_i)T^* + y_i$). Hence, both r_i and f_i are linear functions of T^* and can be represented by an ordered pair (x_i, y_i) . So when we refer to the updates of r_i and f_i , we mean the updates of the values of x_i and y_i .

Now we describe the search algorithm to schedule J_i .

First we do a test on the first p-interval to determine whether $t_i < r_1 = x_1 T^* + y_1$. We call the feasibility algorithm with a testing deadline $T = (t_i - y_1)/x_1$. Depending upon the message returned, we have three different cases:

Case (1). The test returns “infeasible”, we know that $t_i < r_1$. Job J_i should be scheduled on the first p-interval, we schedule J_i on $[a_1, a_1 + 2^{d_i} - 1]$ by using rule R1 in the feasibility algorithm. We obtain $P(i)$ by updating the first element $([a_1, b_1], f_1)$ in $P(i - 1)$ as follows:

- if $a_1 + 2^{d_i} - 1 = b_1$, then the element is replaced by one new element $([a_1, b_1], f_1 + t_i)$;
- else the element is replaced by two new elements $([a_1, a_1 + 2^{d_i} - 1], f_1 + t_i)$ and $([a_1 + 2^{d_i}, b_1], f_1)$.

Case (2). The test returns “feasible”, then $t_i > r_1$. We need to do more tests on other p-intervals in $P(i - 1)$ to determine which p-interval(s) J_i should be scheduled on. We claim that if j , where $1 \leq j < l$, is the largest indexed p-interval which cause the feasibility algorithm to return “feasible”, then the job should be scheduled on the j th and $(j + 1)$ th p-intervals. Two questions need to be clarified in this claim:

- The first one is why we need to select this largest j th p-interval. It will be easy to understand this claim if we connect a test call on an p-interval in the search algorithm with a comparison operation in the feasibility algorithm. Actually, a “feasible” test of J_i on a p-interval with RPT r_j (a linear function of T^*) in the search algorithm is exactly like a $t_i < r_j$ comparison in the feasibility algorithm. Hence, the two p-intervals suggested for the search algorithm in the above claim are exactly the same two p-intervals the feasibility algorithm would choose.
- The second one is how we can make sure that there exists a $(j + 1)$ th p-interval. The answer is that the execution of the search algorithm is actually a run of feasibility algorithm with a deadline T^* . All the jobs must be able to be scheduled by this deadline, so there must be a $(j + 1)$ th p-interval whose r_{j+1} is greater than t_i .

Now we need to find the j th p-interval. To determine whether $t_i < r_j = x_j T^* + y_j$, we call the feasibility algorithm with a testing deadline $T = (t_i - y_j)/x_j$. If the feasibility algorithm returns “infeasible”, then $t_i < r_j$ and the desired index is greater than j . If the feasibility algorithm returns “feasible”, we know that $t_i > r_j$, and the desired index is at most j . Continue to do such tests until the desired j is found.

Once the index j is found, we know J_i should be scheduled on subcube $[a_j, a_j + 2^{d_i} - 1]$ and subcube $[a_{j+1}, a_{j+1} + 2^{d_i} - 1]$ by using rule R3 in the feasibility algorithm. We obtain $P(i)$ by updating both the j th and the $(j + 1)$ th elements of $P(i - 1)$ as follows:

- for the j th element $([a_j, b_j], f_j)$,

- if $a_j + 2^{d_i} - 1 = b_j$, then the element is simply deleted,
 - else the element is replaced by one new element $([a_j + 2^{d_i}, b_j], f_j)$;
- for the $(j + 1)$ th element $([a_{j+1}, b_{j+1}], f_{j+1})$,
 - if $a_{j+1} + 2^{d_i} - 1 = b_{j+1}$, then the element is replaced by one new element $([a_{j+1}, b_{j+1}], f_{j+1} + (t_i - r_j))$,
 - else the element is replaced by two new elements $([a_{j+1}, a_{j+1} + 2^{d_i} - 1], f_{j+1} + (t_i - r_j))$ and $([a_{j+1} + 2^{d_i}, b_{j+1}], f_{j+1})$.

Case (3). The test returns “found”. This means we have found the optimal T^* on the first test, i.e., $t_i = r_1$ and $T^* = T = (t_i - y_1)/x_i$.

Notice that a “found” can also be returned by a test during the search in case (2). Suppose a “found” returned in a test of J_i on an RPT, say r_k ($1 \leq k \leq l$). Then we have $t_i = r_k$ and $T^* = T = (t_i - y_k)/x_k$. We should be careful about what causes the return of a “found”. If $k = l$, then J_i used the entire longest RPT, and the “found” should be returned by the feasibility algorithm. If $k < l$, then we must have a later job, say J_j , that has its t_j equal to the longest RPT of the profile $P(j - 1)$. This is because the feasibility algorithm returns a “found” iff there is a job that uses entire longest RPT. This situation is described in Figure 21.

A complete high level description of the search algorithm, called Optimal-Schedule, is given in Figure 22.

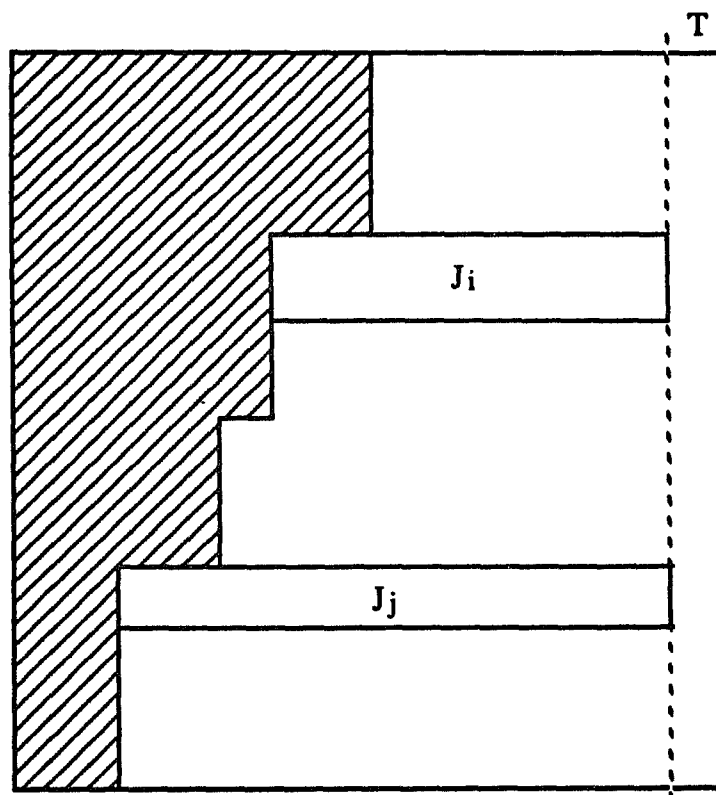


Figure 21: The feasible schedule when “found” is returned.

ALGORITHM Optimal-Schedule;

/ The algorithm can generate an minimum finish time preemptive schedule on an m -cube for a job set J containing n jobs. */*

1. Sort the n jobs in J by decreasing dimension.
 2. $i := 1$; $P(0) := \langle ([0, 2^m - 1], 0) \rangle$.
 3. if "found" is returned during the search,
 then goto step 5 immediately;
 else schedule J_i on the p-interval(s) and update the corresponding element(s)
 in $P(i - 1)$ to get $P(i)$.
 4. If $i < n$, then $i := i + 1$, and goto step 3.
 5. Return the feasible schedule from the feasibility algorithm as the minimum finish time schedule.
-

Figure 22: The search algorithm for minimum finish time schedule.

4.5 An Example

We will trace through a detailed example in this section to show how the algorithm works.

Example. Consider a 4-cube, and five jobs ordered by decreasing dimension as $J_1 = (3, 4)$, $J_2 = (2, 4)$, $J_3 = (2, 3.5)$, $J_4 = (1, 3.5)$, and $J_5 = (1, 3.5)$. The scheduling of each job, together with the updating of RPTs, is illustrated in Figure 23. The steps are explained below:

- a) Initially, we have a profile $P(0)$ which is

$$\langle ([0, 15], 0) \rangle$$

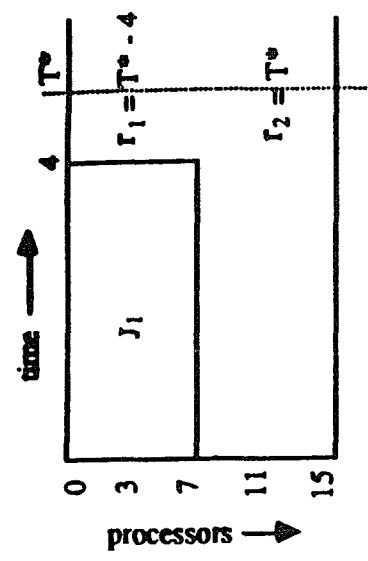
with $r_1 = T^*$.

- b) To schedule job $J_1 = (3, 4)$, the search algorithm uses $T = (4 - 0)/1 = 4$ which causes the feasibility algorithm to return “infeasible”. So the algorithm schedules J_1 on the first p-interval (actually the only one). The new profile $P(1)$ is

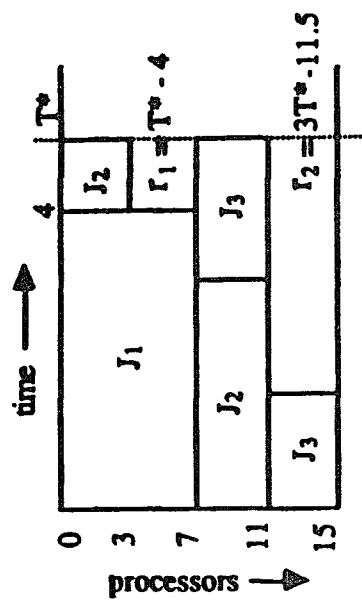
$$\langle ([0, 7], 4), ([8, 15], 0) \rangle$$

with $r_1 = T^* - 4$, and $r_2 = T^*$.

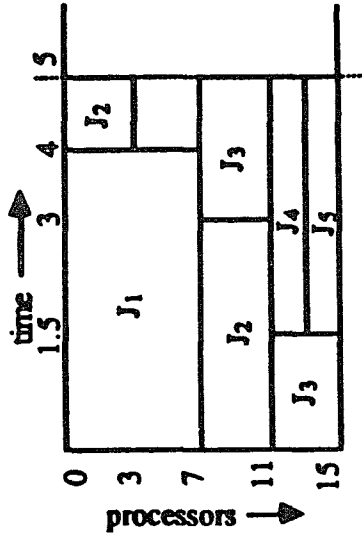
- c) To schedule job $J_2 = (2, 4)$, the search algorithm tries on the first p-interval by setting $T = (4 - (-4))/1 = 8$. This test causes the feasibility algorithm to return “feasible”. But a test on the second p-interval returns “infeasible”. Hence, the first p-interval is the largest p-interval tested “feasible”. So the



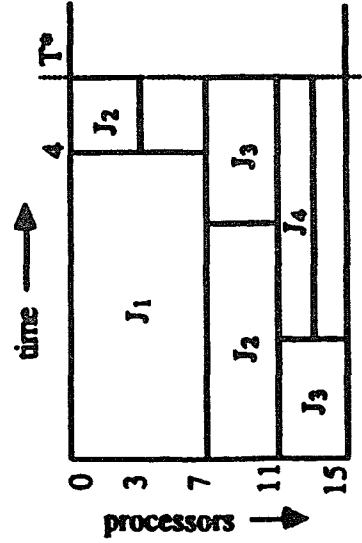
a) The initial profile.



b) After Job 2 scheduled.



c) After Job 3 scheduled.



d) After Job 4 scheduled, $T^* = 5$.

e) The minimum finish time schedule.

Figure 23: Finding the minimum finish time schedule.

search algorithm schedules J_2 to use up the RPT of the first p-interval and schedules the remaining $4 - (T^* - 4) = 8 - T^*$ time of J_2 on the second p-interval. The new profile $P(2)$ is

$$\langle ([4, 7], 4), ([8, 11], 8 - T^*), ([12, 15], 0) \rangle$$

with $r_1 = T^* - 4$, $r_2 = 2T^* - 8$, and $r_3 = T^*$.

- d) To schedule job $J_3 = (2, 3.5)$, the search algorithm tests the three p-intervals by calling the feasibility algorithm three times, and finds out that the second one is the largest p-interval that causes the feasibility algorithm to return “feasible”. So the search algorithm schedules J_3 on the second and third p-intervals. The update profile $P(3)$ is

$$\langle ([4, 7], 4), ([8, 11], 11.5 - 2T^*) \rangle$$

with $r_1 = T^* - 4$, and $r_2 = 3T^* - 11.5$.

- e) To schedule $J_4 = (1, 3.5)$, the search algorithm tests on the two p-intervals. The first one causes the feasibility algorithm to return “feasible” and the second one “found”. So the search algorithm finds out $T^* = T = (3.5 - (-11.5))/3 = 5$ after the second test call. Thus the search algorithm finds the minimum finish time even before the last job J_5 has been tested.
- f) The minimum finish time schedule is the schedule generated by the feasibility algorithm in step e).

4.6 Correctness and Analysis

From the above example, it is easy to observe that the search algorithm schedules each job in exactly the same way as the feasibility algorithm does. The difference between the two algorithms is that the feasibility algorithm uses a known deadline T and can determine where to schedule a job, while the search algorithm use an unknown deadline T^* and has to rely on the feasibility algorithm to determine where to schedule a job and to decide if the schedule is an optimal schedule. This relationship between the two algorithms is evident in the derivation of the search algorithm. Therefore, the search algorithm can always find the minimum finish schedule for a job set. We summarize the result in the following theorem.

Theorem 4.6.1 *For a given job set, the search algorithm always generates the minimum finish time schedule.*

We now consider the time complexity of the search algorithm.

Theorem 4.6.2 *The search algorithm runs in $O(n^2 \log^2 n)$ time.*

Proof: Consider the search algorithm listed in Figure 22. Step 1 takes $O(n \log n)$ time. But the time complexity of the algorithm will be determined by step 3, which we will analyze in detail below.

Step 3 needs to efficiently do search, insertion, and deletion of tuples in a profile . Although the balanced search tree used in the feasibility algorithm appears to be a good choice, it cannot not be used here. This is because the finish time f_i 's in the

profiles of the search algorithm are linear functions of T^* . They cannot serve as the keys in the nodes of the search tree.

The data structure we use here is an ordered linear array. The size of the array could be at most $n + 1$, since the profiles used by the search algorithm could have at most $n + 1$ p-intervals. Each array element is a record with 4 fields, which are used to store a 4-tuple, say (a_j, b_j, x_j, y_j) . The first two numbers in the tuple represent the j th p-interval, while the remaining two represent the parameters of the linear function for r_j . To schedule job J_i , we need to search for the appropriate p-interval stored in the array. Since the array is ordered and it has at most i elements corresponding to the i p-intervals in the current profile $P(i - 1)$, we can use a binary search over the array elements to locate the desired p-interval within $O(\log i)$ steps. Each binary search step makes one test call to the feasibility algorithm. So the cost of finding the p-interval is $O(\log i)$ calls to the feasibility algorithm. (See Figure 24.)

Once we have located the p-interval(s), J_i can be scheduled and we can update the array through changing, deleting, or inserting element(s) as described in the search algorithm while maintaining the ordering. Such updating takes $O(i)$ time. To schedule all of the n jobs, we need

$$O(1) + O(2) + \cdots + O(n) = O(n^2) \quad (4.3)$$

time for array updatings, and at most a total number of

$$O(\log 1) + O(\log 2) + \cdots + O(\log n) = O(n \log n) \quad (4.4)$$

calls to the feasibility algorithm. These calls dominate the time complexity of the algorithm. Hence, the minimum finish time schedule can be found in $O(n^2 \log^2 n)$

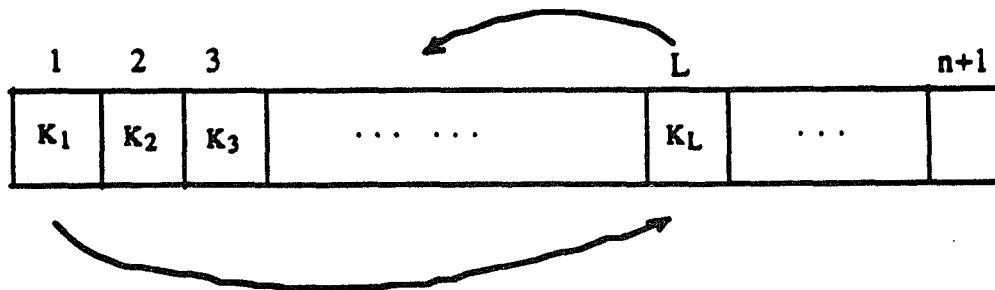


Figure 24: Binary search over an ordered array.

time. □

4.7 Conclusion

In this chapter, we have further studied the preemptive hypercube scheduling. We have developed a new algorithm that can generate the minimum finish time schedule in $O(n^2 \log^2 n)$ time. The search algorithm follows the idea of our feasibility algorithm to schedule each job in an decreasing dimensional order. During the scheduling of each job, the algorithm tries to search the minimum finish time. The searching of the minimum finish time needs to make calls to our feasibility algorithm. The advantage of the new search algorithm is that the time complexity is independent of the times required by the job set.

One possible further research may be to find faster algorithms for the minimum

finish time schedule. One possible approach is to find ways to compute the minimum finish time directly. For a special case where all jobs have only two different dimensional requirements, there exists a formula to compute T^* [6, 7].

CHAPTER V

SUMMARY AND FUTURE RESEARCH

We have studied scheduling of n independent jobs on an m -dimensional hypercube computer system. Each job J_i is allowed to require a d_i -subcube for t_i units of time. We have considered finding the minimum finish time schedule for a given set of n independent jobs. Jobs can be preemptive or nonpreemptive. We have developed new algorithms for both scheduling problems. The contributions of this study are as follows.

- For nonpreemptive scheduling:

- We have proposed an approximation algorithm called LDF. It has an absolute bound less than 2 and an optimal asymptotic bound. An important practical advantage of the LDF algorithm is that as long as we can schedule jobs by decreasing dimension, we can have more than 50 percent system efficiency.
- We have proved a lower bound for the absolute bound of a class of algorithms including LDF, LDLPT, and FFDH. This result tells us that it is unlikely to find simple heuristic algorithms in this class that can perform much better than these algorithms.

- For preemptive scheduling:

- We have developed a feasibility algorithm that runs in $O(n \log n)$ time and generates a schedule with at most $\min\{n - 2, 2^m - 1\}$ preemptions.
- We have presented an algorithm that can generate the minimum finish time schedule in $O(n^2 \log^2 n)$ time. The algorithm is based on our feasibility algorithm and Megiddo's search technique which can avoid the drawbacks of binary search method.

Compared with the classical scheduling theory, the area of scheduling jobs on hypercubes, or on other message-passing multiprocessors, is rather new and very few problems have been studied. As hypercubes and message-passing systems become more and more popular, we believe that scheduling problems on such systems will attract more and more attention. The following directions appear to be interesting areas for future research.

The first direction is to consider extending hypercube scheduling problems in "classical" manners. As we stated in Chapter I, jobs may have many characteristics in the classical scheduling theory. This is certainly true for hypercube scheduling. For nonpreemptive hypercube scheduling, we may consider cases where jobs have precedence relation. For preemptive hypercube scheduling, we may consider cases where jobs have different deadlines, release times, or resource constraints. For example, Plehn have recently proposed a linear programming solution for cases where jobs have different release times and deadlines [56]. But his algorithm is of very high order of polynomial complexity. Reducing the complexity of his algorithm could be

an interesting research topic.

The second direction is to solve *fragmentation problem*. When a cube becomes fragmented, even if there is a sufficient number of processors available in the cube, they may not form a subcube large enough to accommodate an incoming job. The fragmentation problem does not occur in our scheduling algorithms because we use decreasing dimension. This situation may cause serious problems in dynamic hypercube scheduling, where jobs come and go randomly. Our work may be extended to dynamic scheduling. Thus efficient methods for reducing fragmentation must be developed. Some methods, such as job migration [15], compaction [37], and virtual cube [12], have been suggested. But solutions proposed so far are only for simple cases. Many open problems in this area still need to be solved.

The third direction is to consider studying the scheduling on hypercube systems with *folding* capability. In some systems, such as the Connection Machine, jobs can be executed under *virtual-machine* model [69]. A user program may require a 20-cube with $V = 2^{20}$ processors. Depending on the available hardware, the system may allocate $P = 2^{16}$ physical processors. The *virtual-processor ratio* is $V/P = 16$. In other words, we *fold* the required subcube into a smaller one, and the user program will run at about of 1/16 the speed of the physical processor. Doing this increases the user program execution time by about 16 times. But from the system point of view, doing this may reduce the idle smaller subcubes in the system so that the the system utilization is increased. For such systems, one needs to consider ways of incorporating this folding capability of jobs into our scheduling algorithms.

Finally, the fourth direction is to consider job scheduling on k -ary n -cubes. Low-dimensional k -ary n -cubes have several advantages such as message-passing efficiency over the binary cubes. They have been suggested for future median sized cube systems, and several commercial systems based on meshes (or k -ary 2-cubes) have been built. So far, only special cases of job scheduling on k -ary n -cubes have been studied, for example, job scheduling on binary cubes and job scheduling for mesh systems [46]. Compared with the special cases, job scheduling problem on general k -ary n -cubes appears to be much harder.

BIBLIOGRAPHY

- [1] W. C. Athas and C. L. Seitz, "Multicomputers: message-passing concurrent computers," *IEEE Computer*, Vol. 21, No. 8, pp. 9-24, Aug. 1988.
- [2] M. Ahuja and Y. Zhu, "An $O(n \log n)$ feasibility algorithm for preemptive scheduling of n independent jobs on a hypercube", *Inform. Process. Letters*, 35 (1), pp. 7-11, 1990.
- [3] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest, "Orthogonal packings in two dimensions," *SIAM J. Comput.*, vol.9, pp. 846-855, 1980.
- [4] S. N. Bhatt and I. C. F. Ipsen, "How to embed trees in hypercubes," TR-443, Yale University, Dec. 1985.
- [5] S. N. Bhatt, F. Chung, T. Leighton, and A. Rosenberg, "Optimal simulations of tree machines," *Proc. 27th Annu. Symp. Foundations Comp. Sci.*, pp. 274-282, Oct. 1986.
- [6] J. Blazewicz, M. Drabowski and J. Weglarz, "Scheduling independent 2-processor tasks to minimize schedule length," *Inform. Process. Letters*, 18 (5), pp. 267-263, June 1984.
- [7] J. Blazewicz, M. Drabowski and J. Weglarz, "Scheduling multiprocessor tasks to minimize schedule length," *IEEE Trans. on Comp.*, 35 (5), pp. 389-393, May 1986.
- [8] D. J. Brown, B. S. Baker, and H. P. Katseff, "Lower bounds for on-line two-dimensional packing algorithms," *Acta Informat.*, vol.18, pp. 207-225, 1982.
- [9] T. F. Chan and Y. Saad, "Multigrid algorithms on the hypercube multiprocessor," *IEEE Trans. on Comp.*, C-35, No. 11, pp. 969-977, Nov. 1986.
- [10] G. I. Chen, and T. H. Lai, "Scheduling independent jobs on hypercubes," in *Proc. 5th Symp. on Theo. Aspects of Comp. Sci.*, LNCS 294, pp. 273-280, 1988.
- [11] G. I. Chen, and T. H. Lai, "Preemptive scheduling of independent jobs on a hypercube," *Inform. Process. Letters*, vol. 28, pp. 201-206, 1988.

- [12] G. I. Chen, and T. H. Lai, "of independent jobs on a hypercube," *Proc. Internat. Conf. on Parallel Processing*, pp. II 73-76, 1989.
- [13] M. S. Chen, and K. G. Shin, "Processor allocation in an N-cube multiprocessor using Gray codes," *IEEE Trans. Comput.*, C-36, pp. 1396-1407, 1987.
- [14] M. S. Chen, and K. G. Shin, "On relaxed squashed embedding of graphs into a hypercube," *SIAM J. Comput.*, Vol. 18, No. 6, pp. 1226-1244, Dec. 1989.
- [15] M. S. Chen, and K. G. Shin, "Subcube allocation and task migration in hypercube multiprocessor," *IEEE Trans. Comput.*, Vol. 39, No. 9, pp. 1146-1155, Sept. 1990.
- [16] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efa, "Task allocation in distributed data processing," *IEEE Computer*, 13 (11), pp. 57-69, Nov. 1980.
- [17] P.J. Chuang and N.F. Tzeng, "Dynamic processor allocation in hypercube computers," *Proc of Internat. Symp. on Computer Architecture*, pp. 40-49, 1990.
- [18] E. G. Coffman, Jr. (ed.), *Computer and Job Shop Scheduling Theory*, New York: John Wiley & Sons, 1979.
- [19] E. G. Coffman, Jr., M. R., Garey, D. S. Johnson, and R. E. Tarjan, "Performance bounds for level-oriented two-dimensional packing algorithms," *SIAM J. Comput.*, vol. 9, pp. 808-826, 1980.
- [20] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "Approximation algorithms for bin-packing - an updated survey," in *Algorithm Design for Computer System Design*, (G. Ausiello, M. Lucertini, P. Serafini, ed.), pp. 49-106. New York: Springer-Verlag, 1984.
- [21] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson, "Bin packing with divisible item sizes," *Journal of Complexity*, no. 3, pp. 406-428, 1987.
- [22] W. J. Dally, "Performance analysis of k -ary n -cube interconnection networks," *IEEE Trans. on Comp.*, Vol. 39, No. 6, pp. 775-785, June 1990.
- [23] W. J. Dally and C. L. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," *IEEE Trans. on Comp.*, pp. 547-553, May 1987.
- [24] D. Z. Djokovic, "Distance-preserving subgraphs of hypercubes," *J. Combinatorial Theory Theory Ser. B*, 14 (3), pp. 143-146, 1977.
- [25] S. Dutt and J. P. Hayes, "On allocating subcubes in a hypercube multiprocessor," *Proc. of 3rd Conf. on Hypercube Computers and Applications*, pp. 801-810, Jan. 1988.

- [26] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer*, Vol. 15, pp. 50-56, 1982.
- [27] S. Foldes, "A characterization of hypercubes," *Discrete Math.*, 17 (2), pp. 155-159, 1977.
- [28] G. Fox, "The performance of the Caltech hypercube in scientific calculations," Report CALT-68-1298, California Institute of Technology, Apr. 1985.
- [29] M. R. Garey, and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-completeness*, San Francisco: W. H. Freeman, 1979.
- [30] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. Appl. Math.*, vol. 17, pp. 416-429, 1969.
- [31] R. L. Graham, E. L. Lawler, J. K. Lenstra and A. H. G. Rinnooy Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Proc. of Discrete Optimization*, 1977.
- [32] J. P. Hayes, T. N. Mudge, Q. F. Stout, S. Colley, and J. Palmer, "A microprocessor based hypercube supercomputer," *IEEE Micro*, vol. 6, no. 5, pp. 6-17, Oct., 1986.
- [33] W. D. Hills, *The connection machine*, Cambridge, Mass.: MIT Press, 1985.
- [34] C. T. Ho and S. L. Johnson, "On the embedding of arbitrary meshes in boolean cubes with expansion two dilation two," *Proc. of the 1987 Internat. Conf. on Parallel Processing*, pp. 188-191, 1987.
- [35] E. Horowitz, and S. Sahni, *Fundamentals of Data Structures*, Potomac, Md.: Computer Science Press, 1976.
- [36] K. Hwang, and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984.
- [37] C. H. Huang and J. Y. Juang, "A partial compaction scheme for processor allocation in hypercube multiprocessors," *Proc. Internat. Conf. on Parallel Processing*, pp. (I) 211-217, 1990.
- [38] D. S. Johnson, "Fast algorithms for bin packing," *J. Comp. Sys. Sci.*, vol. 8, pp. 272-314, 1974.
- [39] S. L. Johnson, "Communication efficient basic linear algebra computations on hypercube architectures," *J. of Parallel and Distributed Processing*, 4 (2), pp. 133-172, Apr. 1987.

- [40] D. G. Kafura, and V. Y. Shen, "Task scheduling on a multiprocessor system with independent memories," *SIAM J. Comput.*, vol. 6, pp. 167-187, 1977.
- [41] B. Kruatrachue and T. Lewis, "Grain Size Determination for parallel processing," *IEEE Software*, 5 (1), pp. 23-33, Jan. 1988.
- [42] J. Kim, C.R. Das, and W. Lin, "A processor allocation scheme for hypercube computers," *Proc. Internat. Conf. on Parallel Processing*, pp. (II) 231-238, 1989.
- [43] T. H. Lai and W. White, "Mapping pyramid algorithms into hypercubes," OSU-CISRC-1/89-TR 4, Ohio State University, 1989.
- [44] K. Li and K. H. Cheng, "Complexity of resource allocation and job scheduling problems in partitionable mesh connected systems," *Proc. First IEEE Symp. on Parallel and Distributed Processing*, pp. 358-365, 1989.
- [45] K. Li and K. H. Cheng, "Job scheduling in partitionable mesh connected systems," *Proc. Internat. Conf. on Parallel Processing*, pp. (II) 65-72, 1989.
- [46] K. Li and K. H. Cheng, "Job scheduling in PMCS using a 2DBS as the system partitioning scheme," *Proc. Internat. Conf. on Parallel Processing*, pp. (i) 119-122, 1990.
- [47] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Trans. on Comp.*, C-37 (11), pp. 1384-1397, 1988.
- [48] Y. E. Ma and L. Tao, "Embedding among toruses and meshes," *Proc. of the 1987 Internat. Conf. on Parallel Processing*, pp. 178-187, 1987.
- [49] C. Martel, "Preemptive scheduling to minimize maximum completion time on uniform processors with memory constraints," *Oper. Res.*, vol. 33, no. 6, 1360-1380, 1985.
- [50] N. Megiddo, "Combinatorial optimization with rational objective functions," *Math. Oper. Res.*, vol. 4, 414-424, 1979.
- [51] M. Mulder, " $(0,\lambda)$ -graphs and n-cubes," *Discrete Math.*, 28 (2), pp. 179-188, 1979.
- [52] M. Mulder, "n-cubes and median graphs," *J. Graph Theory*, 4 (1), pp. 107-110, 1980
- [53] S. F. Nugent, "The iPSC/2 direct-connect communications technology," *Proc. of 3rd Conf. on Hypercube Computers and Applications*, pp. 56-60, Jan. 1988.

- [54] M. C. Pease, "The indirect binary n -cube microprocessor array," *IEEE Trans. Comp.*, C-26, No. 5, pp. 458-473, May 1977.
- [55] J. C. Peterson, J. O. Tuazon, D. Lieberman, and M. Pneil, "The Mark III hypercube-ensemble concurrent computer," in *Proc. Internat. Conf. on Parallel Processing*, pp. 71-75, 1985.
- [56] J. Plehn, "Preemptive scheduling of independent jobs with release times and deadlines on a hypercube," *Inform. Process. Letters*, 34, pp. 161-166, 1990.
- [57] G. S. Rao and H. S. Stone, "Assignments of tasks in a distributed processor system with limited memory," *IEEE Trans. on Comp.*, C-28 (4), pp. 291-299, 1979.
- [58] D. K. D. B. Sleater, "A 2.5 times optimal algorithm for bin packing in two dimensions," *Inform. Process. Letters*, 10, pp. 37-40, 1980.
- [59] J. S. Squire and S. M. Palais, "Programming and design considerations for a highly parallel computer," *AFIPS Conf. Proc.*, Vol. 23, pp. 395-400, 1963.
- [60] Y. Saad and M. H. Schultz, "Data communication in hypercubes," TR 428, Yale University, Oct. 1985.
- [61] Y. Saad and M. H. Schultz, "Topological properties of hypercubes," *IEEE Trans. on Comp.*, 37 (7), pp. 867-872, 1988.
- [62] S. Sahni, "Preemptive scheduling with due dates," *Oper. Res.*, vol. 27, no. 5, pp. 925-934, 1979.
- [63] C. L. Seitz, "The cosmic cube," *Comm. ACM*, vol. 20, pp. 22-33, 1985.
- [64] J. B. Sinclair, "Efficient computation of optimal assignments for distributed tasks," *Journal of Parallel and Distributed Computing*, Vol. 4, pp. 342-361, 1987.
- [65] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Computer*, 10 (1), pp. 85-93, June 1977.
- [66] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *IEEE Computer*, 11 (7), pp. 97-106, July 1978.
- [67] H. Sullivan and T. R. Bashkow, "A large scale homogeneous, fully distributed parallel machine, I," *Proc. 4th Ann. Symp. on Comp. Architecture*, pp. 105-117, 1977.

- [68] H. Sullivan, T. R. Bashkow, and D. Klappholz, "A large scale homogeneous, fully distributed parallel machine, II," *Proc. 4th Ann. Symp. on Comp. Architecture*, pp. 118-124, 1977.
- [69] T. W. Tucker and G. G. Robertson, "Architecture and applications of the Connection Machine," *IEEE Computer*, Vol. 21, No. 8, pp. 26-38, Aug. 1988.
- [70] J. D. Ullman, "NP-Complete scheduling problems," *J. Comp. Sys. Sci.*, vol. 10, pp. 384-393, 1975.
- [71] P. M. B. Vitanyi, "Locality, communication, and interconnect length in multi-computers," *SIAM J. Comp.*, 17 (4), pp. 659-672, Aug. 1988.
- [72] A. Y. Wu, "Embedding of tree networks into hypercubes," *Journal of Parallel and Distributed Computing*, 2 (3), pp. 238-249, August 1985.
- [73] M. Y. Wu and D. D. Gajski, "Hypertool: a programming aid for message-passing systems," *IEEE Trans. on Parallel and Distributed Systems*, Vol.1, No.3, pp. 330-343, July 1990.
- [74] Y. Zhu and M. Ahuja, "Job scheduling on a hypercube," *Proc. 10th Internal. Conf. on Distributed Computing Systems*, pp. 510-517, 1990.
- [75] Y. Zhu and M. Ahuja, "Preemptive Job scheduling on a hypercube," *Proc. Internal. Conf. on Parallel Processing*, pp. (I) 301-304, 1990.