Order Number 9405605

Continuous task scheduling on unrelated multiprocessing systems

Yeh, Henry Tang-Yu, Ph.D.

City University of New York, 1993

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

CONTINUOUS TASK SCHEDULING ON UNRELATED MULTIPROCESSING SYSTEMS

By

Henry Tang-Yu Yeh

A Dissertation submitted to the Graduate Faculty in Business in partial fulfillment of the requirements for the degree of Doctor of philosophy, The City University of New York

1993

This manuscript has been read and accepted for the Graduate Faculty in Business in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

6/21/93
Date

Chair of Examining Committee

6-24-93
Date

Donald Kiesenbergh
Executive Officer

Dr. Georghios P. Sphicas

Dr. William Chien

Dr. Nicholas Georgantzas, Outside reader

Supervisory Committee

The City University Of New York

ABSTRACT

CONTINUOUS TASK SCHEDULING ON

UNRELATED MULTIPROCESSING SYSTEMS

by

HENRY TANG-YU YEH

Advisors: Professors David Dannenbring, Georghios Sphicas, and William Chien

ABSTRACT

A class of problems which concerns the nonpreemptive scheduling of independent tasks on a set of processors with different capabilities are investigated. Each processor can process one task at a time, and a task can only be processed on one processor at a time. Basically, we are given a set of tasks $J=\{J_1, J_2, \ldots, J_n\}$ to be assigned to a set of processors $P=\{P_1, P_2, \ldots, P_m\}$, where each processor P_i can only execute a subset of tasks S_i, $1 \le i \le m$, and $\bigcup_{i=1}^{m} S_i = J$. The objective is to minimize the makespan, which is the total elapsed time from the start of execution until all tasks are finished.

We assume that no processors can work indefinitely without any rest. That is, each processor, after working contiunously for a certain period of time, must take a rest for a specified interval of time. A maximum working time, w_i, and a minimum resting time, r_i, are specified for each processor P_i, where $w_i \ge 1$

and $r_i \geq 1$ for $1 \leq i \leq m$. P_i can work continuously for at most w_i time units. After that, it must rest for at least r_i consecutive time units. Our problem is: Can we construct a feasible schedule which assigns the tasks to the processors in such a way that at any time unit there exist at least (or at most) p ($1 \leq p \leq m$) processors working. That is, all processors do not rest (or do not work) simultaneously. Furthermore, if such a schedule exists, we want all the tasks to be finished in the shortest time. Processors are unrelated in the sense that the time required for the execution of a task on a processor is a function of both the task and the processor. This models the situation in which general purpose processors have specialized capabilities that permit them to execute certain tasks more efficiently than others. This problem is referred to as continuous task scheduling on unrelated multiprocessing systems. The most significant assumption in this research relates to the working and resting conditions. There is no previous research about adding the w_i and r_i constraints. The continuous task scheduling problem may be applied to banks, toll booths, fast food stores, manufacturers, hospitals, schools, computer local area networks, etc.

We formulate this problem as an exact integer programming model, which can then be solved using standard integer programming codes such as GAMS (General Algebraic Modeling Systems). Then, we find an alternative—a sequential solution method to relax and decompose the exact model. Finally, due to the limitation of the

integer programming package, we develop an heuristic algorithm, with two versions of a reassignment subroutine, to solve the problem. There are two phases in the algorithm: the first phase is to find an optimal feasible schedule for the case of at least one processor working in every time period problem by using A* algorithm, and the second phase is to adjust the solution pattern to meet the at least or at most p processors working requirement for each time period. We then compare the solution times and the limits on problem size for both the exact integer programming model and the heuristic algorithm, and study the behavior of the algorithms.

The effectiveness of the heuristic approach or algorithm has been demonstrated: Its accuracy is 99% of the exact models and about 10^4 faster than that of GAMS. The maximum problem size that can be run by GAMS is approximately 5 processors and 50 tasks, while problems with at least 10 processors and 100 tasks can be easily run by the heuristic approach. For the heuristic, most of the time is used in the reassignment subroutine. For those problem instances that needed task reassignments, the more reassignments required, the more CPU time was needed. The CPU time for the heuristic did not increase when the problem size increased. The efficiency of the heuristic is due to the estimation function h, which makes the heuristic insensitive to the problem size with respect to the computation time.

In summary, the contributions of this thesis are: We defined the continuous task scheduling problem, proved it to be NP-complete, formulated it as an integer programming model and provided a constraint-relaxed and decomposed method, and developed a heuristic algorithm to solve the problem. The primary limitation of this research is that the accuracy of our heuristic algorithm is 99.5% of the exact model.

Acknowledgements

First of all, I express my sincere gratitude and respect to Professors David Dannenbring, Georghios Sphicas, and William Chien for their guiding my growth in every sense. They not only taught me how to do research, but also gave me exquisite and novel ideas when we discussed problems occurring in our society. I also thank Professor Nicholas Georgantzas for his constructive comments on the final draft of this dissertation.

I also extend my special thanks to Professors David Dannenbring and William Chien. Their invaluable guidance and suggestions have benefitted me throughout my entire learning process in becoming a competent researcher.

I would like to express my appreciation to Professors Donald Vredenburgh and Georghios Sphicas for their financial support, constant encourgement and remarkable patience, which have helped me finish the dissertation stage of my Ph.D. study.

I should thank my wife, Rachel Yeh, for her loving care during all these enduring years. Finally, I would like to express my highest gratitude to my parents, Mr. Wei-Chien Yeh and Mrs. Cheng Li-Chen Yeh, for their generous support, constant love, friendship and encourgement throughout my entire doctoral study here at Baruch.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE

INTRODUCTION

1.1 Introduction

Scheduling Problems arise in many practical situations under a wide variety of guises. Many such problems are basically optimization problems having the following form: Given a collection of tasks to be scheduled on a particular processing system, we seek a feasible schedule that minimizes (or in some cases, maximizes) the value of a given objective function, subject to various constraints. Thus, it is not surprising to find that much of the work on scheduling theory has been devoted to the design and analysis of optimization algorithms—algorithms that construct an optimal feasible schedule for a given instance of a scheduling problem.

Throughout human history, mankind has been troubled with the tough problem of optimally utilizing limited resources, to accomplish various tasks or objectives. The problem is not new although contemporary manifestations of this problem in forms such as the energy crisis or inflation may have been unseen before. An attempt to cope with such problems demands that one develop some plans or schedules. The analysis and study of such plans, particularly with respect to their optimality and other properties under various scenarious of objectives and constraints, constitute the field known as scheduling theory. This theory thus deals with

a concretization of continuing human experience.

In general, the theory of scheduling is concerned with the optimal allocation of scarce resources to activities over time. Of obvious practical importance, it has been the subject of extensive research in the past several decades. In view of the fact that the above description allows for a wide variety of problem types, it should come as no surprise that the development of the theory has gone hand in hand with the refinement of a detailed problem classification, for which the ultimate foundation was laid in the classic book *Theory of Scheduling* by Conway, Maxwell, and Miller in 1967 (Conway, Maxwell, and Miller 1967). Since then, considerable progress has been made in solving scheduling problems.

One of the major research areas in scheduling concerns multiprocessing systems. With the rapid advances in technology, it is now possible to build systems consisting of many processors. These processors may be computers, other machines, or even people; the tasks here may be computer programs, items to be assembled in an assembly-line manufacturing system, or even human affairs. These systems may be classfied based on how they are used. For example, in computer operating systems, when the processors are used to execute and process at the program level, where each processor executes its own independent task, one speaks about multiple-task systems which are designed to improve the overall system throughput, rather than the overall execution time of a

single program. When the overall execution time of a single program has to be improved, that program can be divided into cooperating tasks. Several such tasks may be executed in parallel due to the availability of a number of processors. Those systems are known as multiprocessing systems.

Basically, in this research field, the problem generally consists of a set $J=\{J_1,J_2,J_3,\ldots,J_n\}$ of tasks and another set $P=\{P_1,P_2,P_3,\ldots,P_m\}$ of procesors. The objective is to assign all of the tasks to processors, subject to various constraints, such that some criterion is optimized. For books concerning the field of scheduling theory and results, see (Baker 1974), (Coffman 1976), (Rinnooy Kan 1976), (Lenstra 1977), (French 1982), (Dempster, Lenstra, and Rinnooy Kan 1982), (Bellman, Esogbue, and Nabeshima 1982), (Hax and Candea 1984). For surveys, see (Gonzales 1977), (Graham, Lawler, Lenstra, and Rinnooy Kan 1979), (Graves 1981), (Lawler, Lenstra, and Rinnooy Kan 1982), (Johnson 1983), (Kindervater, Lenstra, and Rinnooy Kan 1989). For an annotated bibliography, see (Lenstra and Rinnooy Kan 1983).

1.2 Motivation and Research Questions

In the multiprocessing system studied here we assume that no processor can perform all kinds of tasks, and that the time required for execution is a function of both the task and the processor. Such a system has specialized capabilities that allow execution of certain tasks more efficiently than others. In our

research, if a processor can execute a task, it always takes an integer-number of time units to complete the task. If the processing time is a general integer time, then our problem will become the knapsack problem with side constraints. This variation will not be dealt with here and will be left for future research. We further assume that no processor can work indefinitely without any rest or maintenance. That is, after processing jobs for a certain period of time, each processor must rest for a minimum amount of time. These restrictions are expressed as a maximum working time w_i and a minimum resting time r_i for each processor P_i, where $w_i \geq 1$ and $r_i \geq 1$ for $1 \leq i \leq m$. P_i can work continuously for at most w_i time units. After that, it must rest for at least r_i consecutive time units.

Our research questions are: Can we construct a feasible schedule which assigns tasks to processors in such a way that during any time unit there exist at least (or at most) p processors working, where $1 \leq p \leq m$? That is, we require that not all processors rest (or work) simultaneously. Furthermore, if such a schedule exists, we wish to complete all tasks in the shortest amount time. We call this problem the continuous task scheduling problem. Can we prove that even to determine whether a feasible continuous working schedule exists is NP-complete in the strong sense? Can we find polynomially solvable cases for some w_i and r_i constraints? Can we formulate this problem as an exact integer programming model? Can we develop an algorithm which can solve

this problem more efficiently? All the questions we raise here will be discussed and answered in this study.

We will formulate and solve this problem as a generic integer programming model and develop a specialized branch-and-bound-based heuristic algorithm for the problem. We will also discuss their performance.

In what follows we will discuss some potential applications for the continuous task scheduling problem.

A. Computer Local Area Networks

Local area network (LAN) systems consist of several (m) autonomous processors, which are geographically dispersed to execute a group of (n) programs. Each processor has its own capability, a maximum continuous working time w_i and a minimum continuous resting time r_i, and takes integer units of processing time to execute the programs. Our goal is to find a continuous working schedule, which means that at every time unit there are at least (or at most) p computers processing, or that whenever we inspect the operations of the systems, we can always find at least (or at most) p processors processing, where $1 \leq p \leq m$. That is, we do not want all the processors to rest simultaneously, because if that happens, the utilization and efficiency of the whole local area network systems will deteriorate accordingly. Furthermore, if we can find such schedules, we desire a schedule that will complete all the programs in the least amount of time (i.e., the minimum makespan).

B. Hospital Management

A public hospital had its budget cut by the government this year. Therefore, the administration of the hospital tries to reduce personnel cost, since that cost is the largest portion of the new budget. There are a group of m nurses and a set of n tasks. Each nurse has his or her own capability and talent, a maximum continuous working time w_i and a minimum continuous resting r_i, $1 \le i \le m$. Each nurse gets regular pay for the regular working hours R (usually 8 hours per day). He or she will get overtime pay (one and a half of the regular pay) for the extra overtime hours worked. The goals of the administration are the following:

1. The hospital administration desires a continuous working schedule which ensures that at every time unit (minute or hour) there are at least p ($1 \le p \le m$) nurses on duty, where p is the minimum number of nurses needed to staff the unit.
2. Under the continuous working schedules, since the makespan in the regular working time horizon is fixed, the administration wants to minimize the number of nurses on duty.
3. In the overtime working horizon, because the number of nurses, y ($y \le m$), working is fixed, for each y, the administration seeks to minimize the makespan in the overtime working horizon.

1.3 Summary of Research Results

The research decribed here encompasses the following:

(1) The problem is formulated as several exact integer programming models.

(2) We show how the exact model can be constraint-relaxed and decomposed into two submodels, thereby increasing the efficiency in solving this problem.

(3) The exact integer program is solved using GAMS (General Algebraic Modeling Systems)-ZOOM as a solver installed on an IBM 386 PC with an 80387 math coprocessor.

(4) The continuous task scheduling problem is shown to be NP-complete.

(5) A two phase heuristic algorithm is presented and implemented, featuring two reassignment subroutines.
Phase one of the algorithm solves the at least one processor working case. Based on the derived makespan, phase two of the algorithm solves the at least or at most p processors working in every time period,

(6) Solution times and the limits of the problem size are compared for the exact model and the two-phase algorithm we developed. The performances of the two reassignment subroutines are also compared, The relationship between average makespan and p, for the at least or at most p cases, are compared. Finally, we also study the average behavior and perform a sensitivity analysis on the heuristic algorithm.

1.4 Overview of the Dissertation

This thesis is organized as follows. In Chapter 1 we have discussed the motivation for this research, defined the problem, raised the research questions, illustrated some potential applications, and outlined the main results. Chapter 2 reviews previous research on the unrelated multiprocessing task scheduling problem. In Chapter 3, we prove the continuous task scheduling problem to be NP-complete, formulate the problem as several exact integer programming models, develop the constraint-relaxation and decomposition approach to solve the problem optimally, and provide a survey of integer programming software packages. Chapter 4 develops a two-phase algorithm with alternate reassignment subroutines. In Chapter 5, the experimental design is discussed, and we present results that demonstrate the performance of the algorithm. In the last chapter, conclusions and suggestions for future research are given.

CHAPTER TWO

SCHEDULING PROBLEMS ON UNRELATED MULTIPROCESSING SYSTEMS

2.1 Scheduling in Multiprocessing Systems

Scheduling a number of tasks on more than one processor working in parallel is a frequently encountered problem. The processors may be computers, other machines, or even people. The tasks may be computer programs, items to be assembled, or other human activities. By varying the above processor and/or task characteristics, a huge number of problems can be derived and studied (Gonzales 1977), (Lagewag, Lawler, Lenstra, and Rinnooy Kan 1981), (Lageweg, Lawler, Lenstra, and Rinnooy Kan 1982), (Dror, Stern, and Lenstra 1987), (Chen and Lai 1988), (Janiak 1989), (Simons and Warmuth 1989), (Carreno 1990), (So 1990). The classification of scheduling in multiprocessing systems is shown in Table 2-2 (at the end of the chapter).

For example, when considering the tasks, the execution time may be different for each task, there may be partial orders among the tasks, tasks may be preempted and restarted later without any extra cost, some tasks may appear periodically, there may be release times and deadlines for tasks, or other resources may be required during the execution of some tasks. When considering the processors, the processors may be equal or unequal in their task processing capabilities. The processors are equal (identical) with respect to tasks when the processing time of each task does not depend on the assigned processor. That is, if t_{ij} denotes the

processing time of task j on processor i, then for the identical processors case $t_{ij}=t_j$, which also means that each task has a unique execution time unaffected by the processor to which the task is assigned.

Finally, processors can be unrelated in the sense that there is no notion of a fast processor always requiring less time than a slow processor, irrespective of the task being executed. Rather, the time required for the execution of a task on a processor is a function of both the task and the processor. This models the situation in which general purpose processors have specialized capabilities that permit them to execute certain jobs more efficiently than others. For example, some processors may have high-speed arithmetic capabilities, access to needed data bases, a large high-speed memory, access to required peripheral devices, or other facilities associated with them which would make them especially well-suited for executing certain jobs. Due to the advance of industrial technology and the maturity of local area networks, the unrelated multiprocessor systems tend to be more and more common. It is thus worthwhile to investigate the various aspects of task scheduling in unrelated multiprocessing systems (Gonzalez, Lawler, Sahni 1990).

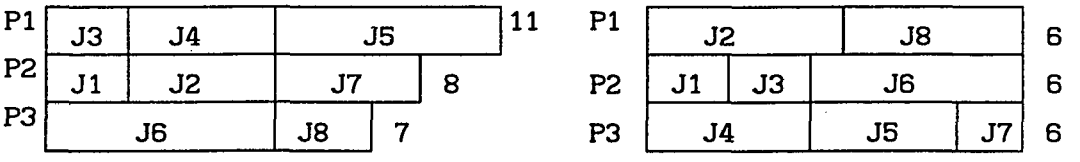
2.2 Unrelated Multiprocessing Task Scheduling Problem

Basically, the unrelated multiprocessing task scheduling

problem has a set of n independent tasks, $J=\{J_1,J_2,\ldots,J_n\}$, to be scheduled on a set of m processors, $P=\{P_1,P_2,\ldots,P_m\}$. The execution time of task J_j executed by processor P_i will be denoted as t_{ij} where $t_{ij} > 0$. This model of a multiprocessor system was introduced in (Bruno, Coffman, Sethi 1974a, 1974b).

Let $[T_{ij}]$ denote the m*n matrix of processing time. We use the Gantt Chart, illustrated in Figure 2-1, to represent schedules. If S is a schedule, $s_j(S)$ and $f_j(S)$ denote the starting time and finishing time, respectively, of task J_j in S. The finishing time (i.e., length or makespan) of a schedule S is denoted by $F(S)=\max_j\{f_j(S)\}$. A schedule which yields the minimum makespan $F^*(S)$ will be called optimal. An example for m=3 and n=8 is shown in Figure 2-1.

$$[tij] = \begin{bmatrix} 2 & 3 & 1 & 4 & 6 & 5 & 2 & 3 \\ 1 & 4 & 1 & 5 & 5 & 4 & 3 & 4 \\ 3 & 2 & 3 & 2 & 3 & 5 & 1 & 2 \end{bmatrix}$$



Schedule S, F(S)=11 An Optimal Schedule, $F^*(S)=6$

Figure 2–1 An example for m=3, n=8

It is well known that the problem of finding an optimal schedule is NP-hard even for the case of two identical processors

(Bruno, Coffman, and Sethi 1974b), (Karp 1972), (Sahni 1974), (Horowitz and Sahni 1976), (Garey and Johnson 1979). Informally, this means that obtaining an algorithm of polynomial time complexity that can generate a schedule with minimum finishing time is as hard as obtaining an algorithm of polynomial time complexity for such problems as the travelling saleperson problem, the integer knapsack problem, and the maximum clique problem (Karp 1972), (Sahni 1974), (Garey and Johnson 1979). The theory of NP-completeness plus the diversity of problems (more than 300 in Garey and Johnson 1979) indicates that in all likelihood any problem which is NP-complete does not admit to an algorithm of polynomial time complexity. This puts added importance on the development of approximation algorithms. Surveys and discussion of the worst-case analysis of approximation algorithms can be found in (Garey, Graham, and Johnson 1978), (Fisher 1980).

Bruno, Coffman, and Sethi (Bruno, Coffman, and Sethi 1974b) first addressed the problem of minimizing the finishing time in an unrelated multiprocessor system, but only gave approximation algorithms for the case of identical processors. Horowitz and Sahni (Horowitz and Sahni 1976) were the first to give exact and approximate algorithms for the unrelated case.

They used the dynamic programming approach and the overall computing time is $O(\min\{nF, m^n\})$ where F is the finishing time of the schedule obtained by executing each job on the processor for which its time is minimal.

Horowitz and Sahni (Horowitz and Sahni 1976) also proposed an approximation algorithm for the unrelated case. The idea is similar to that of the exact algorithm. The fractional error is not greater than ϵ for an algorithm of time $O(n(n/\epsilon)^{m-1})$.

Polynomial time approximation algorithms for the general unrelated case were first studied by Ibarra and Kim (Ibarra and Kim 1977). Five algorithms were presented, each of which was guaranteed to be at most m times worse than optimal in the worst case. In addition, four of the five were proved to be exactly m times worse than optimal in the worst case. The fifth algorithm was left as an open problem---its effectiveness was shown to be between 2 and m times worse than optimal. Table 2-1 summarizes the behavior of their algorithms.

Table 2-1 Behavior of Heuristic Algorithms in Ibarra and Kim

Algorithm	Run Time	Worst-case bound for F/F^*	Is the bound the best possible
IK-1	O(mn)	m	yes
IK-2	O(mnlogn)	m	yes
IK-3	O(mnlogn)	m	yes
IK-4	$O(mn^2)$	m	yes
IK-5	$O(mn^2)$	m	?

Ibarra and Kim also designed an approximation algorithm for the case of two unrelated processors which runs in time O(nlogn) and the performance guarantee is $F/F^* \leq (\sqrt{5}+1)/2$, where F is the

finish time of the schedule obtained by the algorithm and F^* is the optimal finish time. Moreover, the bounds they provide for each algorithm are the best possible under their algorithms (see Table 2-1 for details).

Investigating the "average" behavior of heuristics for the unrelated multiprocessor task scheduling problem via large-scale computational testing was first done by De and Morton (De and Morton 1980). They analyzed the shortcomings of the heuristics of Ibarra and Kim and came out with a new heuristic. Forty four hundred test cases were run. The results from this new heuristic were compared with the best solutions of Ibarra and Kim's heuristics and with the solutions obtained by a branch-and-bound procedure. It is observed that, although the heuristic takes negligible time compared to the branch-and-bound procedure, its performance is within 1.3 percent of the latter, on average. The heuristic also performs better than a composite heuristic whose output, for any given problem, is the best among all previously suggested heuristics. But no theoretical analysis of the worst case bound is given.

The worst case performance bounds of Ibarra and Kim's algorithm (Ibarra and Kim 1977) are all O(m). Seeing this, Davis and Jaffe (Davis and Jaffe 1981) proposed several other heuristics, each of which was shown to have a worst-case performance ratio of $O(\sqrt{m})$. The principle is to set up an efficiency threshold which avoids assigning a task to a processor

that is inefficient in executing this task.

Davis and Jaffe also show that Ibarra and Kim's fifth algorithm (Algorithm IK-5) can create a schedule which is (1+logm) worse than optimal and concludes that it is a promising candidate for an order of magnitude better than their $O(\sqrt{m})$ algorithms. Finally, Spinrad (Spinrad 1985) shows that Ibarra and Kim's fifth algorithm can be arbitrarily close to O(m) worse than optimal for any value of m, making it significantly worse than Davis and Jaffe's algorithms with respect to worst-case behavior.

The worst-case ratio of all the above heuristics is a function of m, the number of processors. Potts (Potts 1985) proposed a linear programming heuristic and showed that the heuristic has a best possible worst-case performance ratio of 2. But the computational requirement is polynomial in n and exponential in m.

All of the above surveyed algorithms either have a worst-case performance ratio that is a function of m or require exponential running time in the worst case.

2.3 Scheduling Tasks to Processors with Different Capabilities and Other cases

Another direction in dealing with an NP-complete problem is tosimplify the structure of the problem in the hope that some meaningful polynomially solvable subcases may appear. In the context of the unrelated multiprocessor task scheduling problem, the

parameters are n (the number of tasks), m (the number of processors), and t_{ij}'s (the task processing times). We have already seen that the case of m=2 is still NP-complete; the case of constant n is trivially polynomially solvable ($O(m^n)$) (Coffman and Sethi 1974).

There has been other work on the scheduling of tasks on unrelated multiprocessors system. Preemptive scheduling of tasks on unrelated multiprocessors was studied by several researchers (Lawler and Labetoulle 1978; Bertossi and Bonuccelli 1985; Gonzales, Lawler, and Sahni 1990). Goyal (Goyal 1976) considered the case that each unit-execution-time task specifies the processor on which it is to be executed and there is a partial order among the tasks. It is shown that the problem of determining optimal schedules is NP-complete. Jaffe (Jaffe 1980) studied a typed task system which consists of different types of tasks to be scheduled on different types of processors. There are partial orders among the tasks. Buten and Shen (Buten and Shen 1973) investigated the case of a system consisting of two classes of processors with each task requiring execution on both kinds of processors. Liu and Liu (Liu and Liu 1978) considered the system where a task can be assigned only to processors of certain types. In their model, there are r different types of functionally dedicated processors and processors of the same type are identical. Each task specifies an execution time required and the type of processor on which it can be executed. There is also a

partial order among the jobs. Kubale (Kubale 1987) studies a system which contains tasks that simultaneously require two dedicated processors for their processing. Finally, communication costs among the processors are also considered by others (Price and Pooch 1982; Bannister and Trivedi 1983; Shen and Tsai 1985, Hansen and Giauque 1986, and Sinclair 1987).

2.4 Chapter Summary

In Section 2.1, we examined a classification of scheduling problems for multiprocessing systems based on the processors or the tasks or both. Since the unrelated multiprocessing systems tend to be more common, we concluded that it is worthwhile to investigate the various aspects of task scheduling in the unrelated multiprocessing systems.

In Section 2.2, we introduced the unrelated multiprocessing task scheduling problem. We first showed a Gantt chart to illustrate the concept of an optimal schedule, then compared and summarized previous researchers' contributions in this area. In Section 2.3, we showed that there is another direction to simplify the structure of the problem to get some polynomially sovable cases.

At the end of this chapter, Table 2-2 provides a classification scheme for scheduling in multiprocessing systems. The basic ideas of our research are derived from Goyal and others (Goyal 1976; Lawler and Labetoulle 1978; Bertossi and Bonuccelli

1985; and Gonzales, Lawler, and Sahni 1990). They use a linear programming approach to formulate their models and develop their own algorithms to compare with the models. Our model differs from theirs in that we add the working and resting time constraints on each processor and seek to find a feasible continuous schedule. We will discuss in greater detail our assumptions and models in the following chapters.

Table 2-2

Classification of the Scheduling in Multiprocessing Systems

	IT		DT	
	NP	PP	NP	PP
IP	Coffman & Sethi(LP,PNP) Garey & Johnson(H,PNP)	Martel(H,PA)	Jaffe(H,WCS)	Bruno, Coffman and Sethi (H,PA)
UP	Horowitz(DP,WCS) Ibarra & Kim(H,PA) De & Morton(LP,ABS) Davis & Jaffe(H,WCS) Potts(LP,WCS) Sinclair(H,PNP) Gonzalez,Lawler & Sahni (H,PA)	Bertossi& Bonucelli (LP, PA)	Kubale (H,PNP)	Horowitz and Sahni (H,PA)

Remark:

IPITNP: Identical Processors, Independent Tasks, Nonpreemptive Processing
IPDTNP: Identical Processors, Dependent Tasks, Nonpreemptive Processing
IPITPP: Identical Processors, Independent Tasks, Preemptive Processing
IPDTPP: Identical Processors, Dependent Tasks, Preemptive Processing

UPITNP: Unrelated Processors, Independent Tasks, Nonpreemptive Processing
UPDTNP: Unrelated Processors, Dependent Tasks, Nonpreemptive Processing
UPITPP: Unrelated Processors, Independent Tasks, Preemptive Processing
UPDTPP: Unrelated Processors, Dependent Tasks, Preemptive Processing

---

(Methodology, Main Contributions)

(LP,PNP)——————→(Linear Programming, Prove NP)
(LP,ABS)——————→(Linear Programming, Average Behavior Study)
(LP,WCS)——————→(Linear Programming, Worst Case Study)
(LP,PA)——————→(Linear Programming, Polynomial Approximation)
(H,PNP)——————→(Heuristics, Prove NP)
(H,WCS)——————→(Heuristics, Worst Case Study)
(H,PA)——————→(Heuristics, Polynomial Approximation)
(DP,WCS)——————→(Dynamic Programming, Worst Case Study)

CHAPETR THREE

MODELING THE CONTINUOUS TASK SCHEDULING PROBLEM

3.1 Introduction to the Continuous Task Scheduling Problem

The continuous task scheduling problem begins with a set $J=\{J_1, J_2, \ldots, J_n\}$ of tasks and a set $P=\{P_1, P_2, \ldots, P_m\}$ of processors. Each processor P_i can only execute a nonempty subset S_i of tasks and $\bigcup_{i=1}^{m} S_i=J$. We assume that no processor can work indefinitely without any rest. That is, each processor, after processing for a certain period of time, must rest for at least a predetermined interval of time. Thus, there is a maximum continuous processing time, w_i, and a minimum continuous resting time, r_i, for each processor P_i, $1 \le i \le m$. P_i can process continuously for at most w_i time units. After that, it must rest for at least r_i time units. The problem is: Can we assign tasks to processors in such a way that in any time unit there exists at least (or at most) p processors working, $1 \le p \le m$. If such a schedule exists, all the tasks are to be finished in the shortest amount of time.

A schedule is called feasible if and only if every task is assigned to a processor that can execute it, and for every time unit between the first time unit in which a task is executed and the time unit when all tasks have been finished, at least (or at most) p tasks are being executed (or, equivalently at least, or at most, p processors are executing tasks). An optimal schedule has the shortest makespan among all possible feasible schedules. Our

problem is to find an optimal feasible schedule if one exists.

Unfortunately, as we will show below, this problem, like many other scheduling problems (French 1982; Garey and Johnson 1979; Lenstra and Rinnooy Kan 1983; and Rinnooy Kan 1976), is NP-complete. Hence it does not seem likely that a polynomial time algorithm for solving this problem exists.

The class of NP-complete problems (Garey and Johnson 1979) has been the focus of considerable theoretical and practical interests for many years. The class contains many well-known and much-studied problems, such as the traveling salesperson problem, the integer knapsack problem, and the graph coloring problem, and is characterized by two important properties:

(1) No NP-complete problem is known to be solvable by a polynomial time algorithm; and

(2) If we had a polynomial time algorithm for one of the NP-complete problems, we could obtain polynomial time algorithms for all the NP-complete problems.

In light of these two properties it is widely conjectured that no NP-complete problem can be solved by a polynomial time algorithm. For this reason the NP-complete problems are frequently considered to be computationally "intractable". (Karp 1972).

To prove a problem NP-complete, we must show that this problem is an NP problem and there exists an already proven NP-complete problem which can be polynomially reduced to this problem. To prove a problem NP-complete in the strong sense, we

further require that the problem to be transformed is itself NP-complete in the strong sense and the largest number in the constructed instance does not grow exponentially as a function of the largest number of the problem instance to be transformed. Refer to (Garey and Johnson 1979) for further details.

To prove that the continuous task scheduling problem is NP-complete, we know that the problem of finding an optimal schedule is NP-complete even for the case of two identical processors (Bruno, Coffman, and Sethi 1974b; Karp 1972; Sahni 1974; Horowitz and Sahni 1976; and Garey and Johnson 1979). This can be regarded as a special case of the continuous task scheduling problem, for $w_i=n$, $r_i=n$, $1\leq i\leq 2$, where n is the number of given tasks for the case of two identical processors. If finding an optimal schedule is NP-complete for the two identical processor case, then the more general continuous task scheduling problem, which can be polynomially reducible to the case of two identical processors, is also NP-complete.

3.2 To Determine Whether the Continuous Task Scheduling Problem is Strong NP-Complete

Although we are interested in finding optimal solutions for the continuous task scheduling problem, we will show that it is equally hard to find a feasible solution to the continuous task scheduling problem. That is, the problem of finding a feasible schedule for the continuous task scheduling problem is NP-complete in the strong sense.

Definition 3-1 Feasible Continuous Task Scheduling Problem

Instance: Unit-execution-time task set $J=\{J_1, J_2, \ldots, J_n\}$, processor set $P=\{P_1, P_2, \ldots, P_m\}$, and each processor P_i can execute a nonempty set $S_i \subset J$ of tasks, where $\bigcup_{i=1}^{m} S_i = J$. Associated with each processor P_i, $1 \le i \le m$, there is a pair of integers (w_i, r_i) where $w_i \ge 1$ and $r_i \ge 1$.

Question: Can each task J_j, $1 \le j \le n$, be assigned to be executed by a processor P_i, where $J_j \in S_i$, and such that the following conditions hold:

(1) Each processor P_i, $1 \le i \le m$, continuously executes at most w_i tasks;

(2) When a processor P_i, $1 \le i \le m$, stops executing tasks, it cannot resume executing tasks until r_i time units have passed;

(3) Every processor can begin to execute its first task at any time unit; and

(4) Between the first time unit that has any task executed and the last time unit when all processors complete their execution, there must not exist a time unit where all processors are resting.

We will show that the 3-PARTITION problem can be reduced to the continuous task scheduling problem. The 3-PARTITION problem was proved to be NP-complete in the strong sense (Garey and Johnson 1975).

Definition 3-2 The 3-Partition Problem

Instance: Set A of 3m elements, a bound $B \in Z^+$, and a size $s(a) \in Z^+$. For each $a \in A$, $B/4 < s(a) < B/2$ and $\sum_{a \in A} s(a) = mB$.

Question: Can A be partitioned into m disjoint sets $A_1, A_2, \ldots, A_m$ such that, for $1 \le i \le m$, $\sum_{a \in A} s(a) = B$ (Note that each A_i must therefore contain exactly three elements from A)?

Lemma 3-1 The continuous task scheduling problem is NP-complete in the strong sense.

Proof: We will transform the **3-Partition** problem, which is NP-complete in the strong sense, to the continuous task scheduling problem. Let $A=\{a_1, a_2, \ldots\ldots, a_{3m}\}$, $B \in Z^+$ be an arbitary instance of 3-partition. Denote the size of a_i by $s(a_i)$ for $1 \le i \le 3m$. The corresponding continuous task scheduling problem instance is given by $J=\{J_1, J_2, \ldots, J_{mB+m+1}\}$, $P=\{P_0\} \cup \{P_1, P_2, \ldots, P_{3m}\}$, and $S_i=J$ for

$0 \le i \le 3m$. That is, we have $mB+m+1$ tasks to be accomplished by $3m+1$ processors, and every processor can execute every task. Let $w_i = s(a_i)$ for $1 \le i \le 3m$, $w_0 = 1$ and $r_i = mB+m$ for $1 \le i \le 3m$, $r_0 = B$. This transformation clearly can be performed in polynomial time. We next show that set A has a 3-Partition if and only if the constructed continuous task scheduling problem instance has a feasible schedule.

Suppose that A has a 3-partition. Without loss of generality, let $A_1, A_2, \ldots, A_m$ denote this partition and $A_i = \{a_{i_1}, a_{i_2}, a_{i_3}\}$ for $1 \le i \le m$, where $1 \le i_1, i_2, i_3 \le 3m$. A feasible schedule of the corresponding continuous task scheduling problem can be obtained as follows: assign $m+1$ tasks to be executed by P_0 at time unit 1, B+2, 2B+3,.....,mB+m+1. This leaves m separate blocks of unfilled time units, each of whose length is exactly B. (That is, no processor is scheduled to execute tasks at those time units as yet). But from the definition of 3-Partition, $\sum_{j=1}^{3} s(a_i) = \sum_{j=1}^{3} w_{i_j} = B$ for $1 \le i \le 3m$. We can partition the remaining processor set $\{P_1, P_2, \ldots, P_{3m}\}$ into m disjoint subsets according to A_1, A_2, A_3,....., A_m. each subset of processors will execute B tasks at a separate time block unfilled by P_0 (which set of processors execute in which time block is immaterial). Since $\sum_{i=1}^{3m} s(a_i) = \sum_{i=1}^{3m} w_i = mB$ and P_0 executes m+1 tasks, all mB+m+1 task will be accomplished and in each time unit at least one processor is executing tasks.

On the other hand, suppose that the continuous task scheduling problem has a feasible schedule. Since we have only

$mB+m+1$ jobs, any feasible schedule will be completed within time $mB+m+1$. Furthermore, because $r_i=mB+m$, processor P_i can execute at most w_i tasks within time $mB+m+1$ for $1 \le i \le 3m$. Altogether, P_1 through P_{3m} can execute at most $\sum_{i=1}^{3m} w_i = mB$ tasks in any feasible schedule. That leaves $m+1$ tasks to be executed by processor P_0. But since $w_0=1$ and $r_0=B$, the only possible schedule for processor P_0 to execute $m+1$ jobs within time $mB+m+1$ is to assign those $m+1$ tasks to be executed by processor P_0 at time unit 1, $B+2$, $2B+3, \ldots\ldots, mB+m+1$. This leaves m separate blocks of time units, each of whose length is exactly B to be filled by processor P_1 through processor P_{3m}. Therefore, processor P_i must exactly execute w_i tasks for $1 \le i \le 3m$. Moreover, the execution of these w_i tasks must be continuous in time due to the constraint that $r_i=mB+m$. Since $B/4 < s(a_i)=w_i < B/2$ for $1 \le i \le 3m$ and there are m time blocks of length B unfilled, each block must be exactly filled by three processors for a feasible schedule to exist. From the way processor P_1 through P_{3m} is partitioned in the m blocks, we can partition $A=\{a_1, a_2, \ldots, a_{3m}\}$ accordingly into m disjoint sets, the size of each set being exactly B. Thus, if we have a feasible continuous task schedule, we have a 3-partition for set A.

Q.E.D.

Example 3-1 Let $A=\{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9\}$, $s(a_1)=26$, $s(a_2)=27$, $s(a_3)=28$, $s(a_4)=29$, $s(a_5)=30$, $s(a_6)=31$, $s(a_7)=41$, $s(a_8)=43$, $s(a_9)=45$. Then $m=3$ and $B=100$. We have $mB+m+1=304$ tasks to be executed by processors $P=\{P_0, P_1, P_2, \ldots\ldots, P_9\}$ where $w_i=s(a_i)$, $r_i=mB+m=303$ for $1\le i\le 9$, and $w_0=1$, $r_0=B=100$. A feasible continuous task schedule is for processor P_0 to execute $m+1=4$ jobs at time unit 1, 102, 203, and 304, respectively, for processors P_1, P_4, P_9 to execute another 100 jobs at time unit 2 to 101, for processors P_2, P_5, P_8 to execute 100 jobs from time unit 103 to 202, and for processors P_3, P_6, P_7 to execute the remaining 100 tasks from time unit 204 to 303. Consequently, $\{a_1, a_4, a_9\}$, $\{a_2, a_5, a_8\}$, and $\{a_3, a_6, a_7\}$ is a 3-partition of A.

3.3 Previous Problem Formulations by Integer Programming

Certain preemptive scheduling problems of unrelated processors can be formulated and solved as linear programming problems. For example, the general problem (Lawler and Labetoulle 1978) that Lawler and Labetoule deal with is that of finding optimal preemptive schedules for independent tasks on unrelated processors. They show that certain specific scheduling problems of this type, e.g., minimization of makespan, can be formulated and solved as linear programming problems. They also show that the linear programming formulations provide a means for establishing upper bounds on the number of preemptions required for an optimal schedule.

As part of the general problem formulation, they assume that there are m processors, indexed by $i=1,2,\ldots,m$ and n tasks, indexed by $j=1,2,\ldots,n$. A processor can work on only one task at a time, and a task can be worked on by only one processor at a time. The processing of a task may be interrupted at any time and resumed at a later time, by the same processor or a different processor. There is no cost and no time loss associated with such an interruption or preemption. The input data for a problem instance include m*n positive numbers p_{ij}, where p_{ij} represents the total processing time required to complete task j on processor i. If task j is processed by processor i for a total time t_{ij}, then in order for the task to be completed, it is necessary that

$$\sum_{i=1}^{m} \frac{t_{ij}}{p_{ij}} = 1$$

For a given feasible schedule, the last point in time at which task j is processed is its completion time C_j. The first and most important problem that Lawler and Labetoulle consider is that of finding a feasible schedule for which "makespan" or maximum completion time, $C_{max}=\max_j\{C_j\}$, is minimized. They assume that all tasks are available for processing at time t=0. For any feasible schedule of n tasks on m unrelated processors, it is evident that the values of C_{max} and t_{ij} for the schedule constitute a feasible solution to the following linear programming problem:

Minimize C_{max}

Subject to

$$\sum_{i=1}^{m} \frac{t_{ij}}{p_{ij}} = 1, \qquad j=1,2,\ldots,n \qquad (3\text{-}1)$$

$$\sum_{i=1}^{m} t_{ij} \leq C_{max}, \qquad j=1,2,\ldots,n \qquad (3\text{-}2)$$

$$\sum_{j=1}^{n} t_{ij} \leq C_{max}, \qquad i=1,2,\ldots,m \qquad (3\text{-}3)$$

$$t_{ij} \geq 0 \qquad (3\text{-}4)$$

De and Morton (De and Morton 1980) consider the nonpreemptive case of the problem. As shown below, it can be formally expressed as an integer program. Let

$$X_{ij}=\begin{cases}1, & \text{if task } j \text{ is processed on processor } i,\ i=1,2..,m,\ j=1,2,..,n.\\ 0, & \text{otherwise.}\end{cases}$$

and t_{ij}=Processing time of task j on processor i. Then the integer program is as follows:

Minimize M

$$\text{Subject to} \quad \sum_{j=1}^{n} t_{ij} X_{ij} \le M \qquad i=1,.....,m \qquad (3\text{-}5)$$

$$\sum_{i=1}^{m} X_{ij}=1 \qquad j=1,.....,n \qquad (3\text{-}6)$$

$$X_{ij}=0 \text{ or } 1 \qquad (3\text{-}7)$$

Where M is the makespan.

Potts (Potts 1985) presents a formulation of the problem which uses zero-one variables X_{ij} ($i=1,2,...,m; j=1,2,...,n$), where

$$X_{ij}=\begin{cases}1, & \text{if task } j \text{ is processed on processor } i,\ i=1,..,m,\ j=1,..,n.\\ 0, & \text{otherwise.}\end{cases}$$

P_{ij} is the integer processing time of task j on processor i, C_{max} represents the maximum completion time which is actually the same as the makespan in the previous model [De and Morton, 1980]. They refer to the X_{ij} as assignment variables. The problem can be written as:

Minimize C_{max}

$$\text{subject to} \quad \sum_{j=1}^{n} P_{ij}X_{ij} \le C_{max} \qquad i=1,\ldots,m \qquad (3\text{-}8)$$

$$\sum_{i=1}^{m} X_{ij}= 1 \qquad j=1,\ldots,n \qquad (3\text{-}9)$$

$$X_{ij} \in \{0,1\} \qquad i=1,\ldots,m; j=1,\ldots,n \qquad (3\text{-}10)$$

Constraints (3-8) ensure that C_{max} is at least as large as the total processing time on any processor, while constraints (3-9) and (3-10) ensure that each task is processed by exactly one processor.

Basically, De & Morton and Potts' formulations are the same, while their contributions are different, as shown in Chapter 2. De & Morton investigate the average behavior of the heuristics, while Potts shows a best possible worst-case performance ratio of 2 and the computation requirement which is polynomial in n and exponential in m.

In the following sections, the assumptions that we adopt will be listed, then the continuous task scheduling problem can be formulated as an integer linear program. The at-least-one-processor-working case is formulated first. After that, two formulations for the at-least-p-processors-working case are developed. These two integer programs are different in their objective functions: one is to minimize the makespan; the other is to maximize the total sum of the cumulative number of tasks processed.

Comparing the at-least-one- and at-least-p-processors working cases, we show that we can obtain an optimal solution for the at-least-p-processors-working case, from the at-least-one-processor-working case, if the latter represents a

feasible solution to the former case. We also develop an integer program for the at-most-p-processors-working case. Finally, we develop a constraint-relaxation and decomposition technique and an iterative sequential procedure for the at-least-one-processor-working case.

3.4 Formulating the Continuous Task Scheduling Problem by Integer Programming

The assumptions that we make for the continuous task scheduling problem are:

1. A set of m unrelated processors, $P=\{P_1, P_2, \ldots, P_m\}$.
2. A set of n independent tasks, $J=\{J_1, J_2, \ldots, J_n\}$.
3. The processing of the tasks is nonpreemptive.
4. Each processor can only execute a nonempty subset of tasks S_i, and $\bigcup_{i=1}^{m} S_i = J$
5. Each processor can execute one task at a time. A task can be executed by at most one processor at a time.
6. Associated with each processor P_i, $1 \le i \le m$, there is a pair of integers w_i and r_i, ($w_i \ge 1$, $r_i \ge 1$), where w_i is the maximum continuous processing time and r_i is the minimum continuous resting time. P_i can continuously process at most w_i time units, after that, it must rest continuously for at least r_i time units.
7. Task processing time is in unit-execution-time.
8. $n \ge m$

9. There must be at least (or at most) p ($1 \leq p \leq m$) processors working during each time unit between the start and finish of all tasks.
10. At the beginning of the schedule, each processor may choose to rest for any number of time units or not to rest at all.

The significant differences between this research and the previous research in the literature are: (1) continuous scheduling (i.e., the at-least-p-processors working and the at-most-p-processors working constraints), (2) the w_i and r_i constraints.

The assumptions listed above permit the following three cases (a, b, and c):

Case (a):

We consider the case where $t_{ij}=1$ for all i,j and formulate the problem as an integer program and develop a specialized branch-and-bound algorithm to solve the problem.

Case (b):

For t_{ij}=general integer processing time, the formulations and the branch-and-bound algorithm will still be valid if we change the assumption (3) from nonpreemptive to preemptive and replace each task with a set of unit-execution-time tasks (the number of these new tasks is equal to the original processing time of the replaced task).

Case (c):

For the nonpreemptive general integer processing time case, the

continuous task scheduling problem will become an integer knapsack problem with side constraints. This problem will be left for future research.

Let us now define our notation for the formulations. Additional definitions of the decision variables will be given with the corresponding formulations.

Recall that we provided a potential application to hospital management in Chapter 1; we will use this example to formulate our model. We first formulate this model in the regular working time horizon, based on the fixed makespan M_r, to minimize the number of processors y, $1 \le y \le m$. After we get this minimum number of processors, we will formulate this model in the overtime working horizon to determine the minimum makespan for a given y.

Exact Formulation in the Regular Working Time Horizon

MODEL [RW-AL1]

(henceforth will be called MODEL [AL1])

$$X_{ijk} = \begin{cases} 1, & \text{if task } j \text{ is processed on processor } i \text{ at time unit } k. \\ 0, & \text{otherwise.} \end{cases}$$

$$Z_{ik} = \begin{cases} 1, & \text{if processor } i \text{ executes a task at time unit } k. \\ 0, & \text{otherwise.} \end{cases}$$

$$Y_i = \begin{cases} 1, & \text{if processor } i \text{ is used.} \\ 0, & \text{otherwise.} \end{cases}$$

$$\text{Min} \sum_{i=1}^{m} Y_i \qquad (3\text{-}11)$$

Subject to:

$$k^{*}Z_{ik} \leq M_r \qquad i=1,\ldots,m;\ k=1,\ldots,n \qquad (3\text{-}12)$$

$$\sum_{i \in I_j} \sum_{k=1}^{n} X_{ijk} = 1 \qquad j=1,\ldots,n \qquad (3\text{-}13)$$

$$\sum_{j=1}^{n} X_{ijk} = Z_{ik} \qquad i=1,\ldots,m;\ k=1,\ldots,n \qquad (3\text{-}14)$$

$$\sum_{i=1}^{m} Z_{ik} + \left(\frac{1}{n}\right) \sum_{i=1}^{m} \sum_{q=1}^{k-1} Z_{iq} \geq 1 \qquad k=1,\ldots,n \qquad (3\text{-}15)$$

$$\sum_{q=k}^{k+r_i-1} Z_{iq} \leq (1 - Z_{i,k-1} + Z_{ik}) * r_i \qquad i=1,\ldots,m;\ k=2,\ldots,n-r_i+1 \qquad (3\text{-}16)$$

$$\sum_{q=k}^{k+W_i} Z_{iq} \leq W_i \qquad i=1,\ldots,m;\ k=1,\ldots,n-W_i \qquad (3\text{-}17)$$

$$Z_{ik} \leq Y_i \qquad i=1,\ldots,m;\ k=1,\ldots,n \qquad (3\text{-}18)$$

$$X_{ijk},\ Z_{ik},\ Y_i \in \{0,1\} \qquad i=1,\ldots,m;\ j=1,\ldots,n;\ k=1,\ldots,n \qquad (3\text{-}19)$$

The objective of this formulation is to minimize the number of processors to be used in the regular working time horizon. The

interpretation of this formulation will be discussed together with the next model. Constraint (3-18) shows the relationship and connection between Y_i and Z_{ik}. The explanations for the rest of the constraints are reserved until we present the next model.

Exact Formulations for the Overtime Working Horizon

MODEL [OW-AL1]

(henceforth will be called MODEL [AL1])

$$X_{ijk}=\begin{cases} 1, & \text{if task } j \text{ is processed on processor } i \text{ at time unit } k. \\ 0, & \text{otherwise.} \end{cases}$$

$$Z_{ik}=\begin{cases} 1, & \text{if processor } i \text{ executes a task at time unit } k. \\ 0, & \text{otherwise.} \end{cases}$$

Minimize M_o (3-20)

Subject to:

$$k^*Z_{ik} \le M_o \qquad i=1,\ldots,m;\ k=1,\ldots,n \qquad (3\text{-}21)$$

$$\sum_{i\in I_j} \sum_{k=1}^{n} X_{ijk} = 1 \qquad j=1,\ldots,n \qquad (3\text{-}22)$$

$$\sum_{j=1}^{n} X_{ijk} = Z_{ik} \qquad i=1,\ldots,m;\ k=1,\ldots,n \qquad (3\text{-}23)$$

$$\sum_{i=1}^{m} Z_{ik} + \left(\frac{1}{n}\right) \sum_{i=1}^{m} \sum_{q=1}^{k-1} Z_{iq} \ge 1 \qquad k=1,\ldots,n \qquad (3\text{-}24)$$

$$\sum_{q=k}^{k+r_i-1} Z_{iq} \le (1-Z_{i,k-1}+Z_{ik})*r_i \qquad \begin{matrix} i=1,\ldots\ldots,m \\ k=2,\ldots\ldots,n-r_i+1 \end{matrix} \qquad (3\text{-}25)$$

$$\sum_{q=k}^{k+W_i} Z_{iq} \le W_i \qquad \begin{matrix} i=1,\ldots\ldots,m \\ k=1,\ldots\ldots,n-W_i \end{matrix} \qquad (3\text{-}26)$$

$$X_{ijk},\ Z_{ik} \in \{0,1\} \qquad \begin{matrix} i=1,\ldots\ldots,m \\ j=1,\ldots\ldots,n \\ k=1,\ldots\ldots,n \end{matrix} \qquad (3\text{-}27)$$

The objective (3-20) is to minimize the makespan M_0. Constraint (3-21) defines the makespan, which indicates that the makespan M_0 should be at least as large as the index of the time unit in which the last task is executed. Constraint (3-22) ensures that all tasks must be completed and every task is executed exactly once. Constraint (3-23) defines the Z_{ik} variables, which show the relationship and connection between X_{ijk} and Z_{ik} and limits each processor to at most one task during any given time period. Constraint (3-24) ensures that there is at least one processor working during every single time unit until all the tasks are executed. The second term $(\frac{1}{n}) \sum_{i=1}^{m} \sum_{q=1}^{k-1} Z_{iq}$ becomes 1, and forces the first term $\sum_{i=1}^{m} Z_{ik} \ge 0$ (i.e., a redundant constraint) when all tasks have been finished. Constraint (3-25) ensures that each processor meets the minimum resting time requirement once the rest begins. There are four possible sets of values {(0,0), (0,1), (1,0), (1,1)} for $(Z_{i,k-1}, Z_{i,k})$. Only in the case of (1,0) is the constraint in effect, corresponding to the situation in which

processor i switches from working to resting and the summation of Z_{iq} from $q=k$ to $k+r_i-1$ is forced to be 0. The other three combinations will result in redundant constraints. Constraint (3-26) ensures that each processor does not exceed the limit on the maximum contiunous working requirement.

MODEL [AL_1] is only valid for the case in which at least one processor must be working in each time unit before all tasks are completed. It can be extended to the case which requires at least p processors be working, as will be shown in **MODEL** [AL_p–Min] and **MODEL** [AL_p–Max]. These two models both ensure that at least p processors are working in each time unit until all tasks are completed, but have different objective functions. **MODEL** [AM_p] extends the basic model to restrict the continuous schedule to require at most p processors working in each time unit until all tasks are completed. These models are our major formulations. We will use these formulations to experiment and compare with our heuristic algorithms later. Next, let us show the at-least-p-processors-working case:

MODEL [AL_p–Min]

$$X_{ijk}=\begin{cases} 1, & \text{if task } j \text{ is processed on processor } i \text{ at time unit } k. \\ 0, & \text{otherwise.} \end{cases}$$

$$Z_{ik}=\begin{cases} 1, & \text{if processor } i \text{ executes a task at time unit } k. \\ 0, & \text{otherwise.} \end{cases}$$

$$C_k=\begin{cases} 1, \text{ if } (n-\sum_{i=1}^{m}\sum_{q=1}^{k-1} Z_{iq}) \geq p, \text{ where p is an integer number and } k=2,\ldots,n \\ 0, \text{ otherwise.} \end{cases}$$

Minimize M_o (3-20)

Subject to:

$C_1=1$ (3-28)

$$n-\sum_{i=1}^{m}\sum_{q=1}^{k-1} Z_{iq} \geq p*C_k \qquad k=2,\ldots,n \qquad (3\text{-}29)$$

$$\sum_{i=1}^{m} Z_{ik} \geq p*C_k \qquad k=1,\ldots,n \qquad (3\text{-}30)$$

$$\sum_{i=1}^{m} Z_{ik} \leq m*C_k \qquad k=1,\ldots,n \qquad (3\text{-}31)$$

$$k*Z_{ik} \leq M_o \qquad i=1,\ldots,m;\ k=1,\ldots,n \qquad (3\text{-}21)$$

$$\sum_{i\in I_j}\sum_{k=1}^{n} X_{ijk}= 1 \qquad j=1,\ldots,n \qquad (3\text{-}22)$$

$$\sum_{j=1}^{n} X_{ijk}=Z_{ik} \qquad i=1,\ldots,m;\ k=1,\ldots,n \qquad (3\text{-}23)$$

$$\sum_{i=1}^{m} Z_{ik} + \left(\frac{1}{n}\right)\sum_{i=1}^{m}\sum_{q=1}^{k-1} Z_{iq} \geq 1 \qquad k=1,\ldots,n \qquad (3\text{-}24)$$

$$\sum_{q=k}^{k+r_i-1} Z_{iq} \leq (1-Z_{i,k-1}+Z_{ik})*r_i \qquad i=1,\ldots,m;\ k=2,\ldots,n-r_i+1 \qquad (3\text{-}25)$$

$$\sum_{q=k}^{k+W_i} Z_{iq} \leq W_i \qquad i=1,\ldots,m;\ k=1,\ldots,n-W_i \qquad (3\text{-}26)$$

$$X_{ijk},\ Z_{ik},\ C_k \in \{0,1\} \qquad \begin{matrix} i=1,\ldots\ldots,m \\ j=1,\ldots\ldots,n \\ k=1,\ldots\ldots,n \end{matrix} \qquad (3\text{-}27)$$

Variable C_k is 1 as long as at least p tasks remain to be scheduled; Constraint (3-28) initializes C_k for the first time unit. Constraint (3-29) ensures that at least p processors are working in every time unit until all n tasks have been completed, which also means that there are always at least p remaining tasks to be scheduled at time period k until all tasks are completed. It forces C_k to 0 after all tasks have been scheduled. This is critical for the next two constraints to work. Constraint (3-30) ensures that at least p processors are working in every time unit when there are at least p remaining tasks to be scheduled (i.e., when C_k=1). After all tasks are completed, Constraint (3-30) becomes redundant (i.e., when C_k=0). Constraint (3-31), together with Constraint (3-24) will force C_k to be 1 if there are tasks remaining unscheduled. The interpretation of the remaining constraints are the same as in model [AL_1].

If we remove constraint (3-21), $k*Z_{ik} \le M_o$, and change the objective function from Min M_o to Max $\sum_{k=1}^{n} \left(\sum_{i=1}^{m} \sum_{q=1}^{k} Z_{iq} \right)$, we can maximize the sum of the cumulative number of tasks scheduled. As the objective here is maximizing, we refer to this formulation as **MODEL [AL_p–Max]**. The motivation of this modification is to eliminate the m*n constraints represented by (3-21). Such modification may save some memory space and computation time.

MODEL [ALp-Max]

$$X_{ijk}=\begin{cases}1, & \text{if task } j \text{ is processed on processor } i \text{ at time unit } k.\\ 0, & \text{otherwise.}\end{cases}$$

$$Z_{ik}=\begin{cases}1, & \text{if processor } i \text{ executes a task at time unit } k.\\ 0, & \text{otherwise.}\end{cases}$$

$$C_k=\begin{cases}1, & \text{if } (n-\sum_{i=1}^{m}\sum_{q=1}^{k-1} Z_{iq}) \geq p, \text{ where p is an integer number and } k=2,\ldots,n\\ 0, & \text{otherwise.}\end{cases}$$

$$\text{Max} \sum_{k=1}^{n} \left(\sum_{i=1}^{m} \sum_{q=1}^{k} Z_{iq} \right) \qquad (3\text{-}32)$$

Subject to:

$$C_1=1 \qquad (3\text{-}28)$$

$$n-\sum_{i=1}^{m}\sum_{q=1}^{k-1} Z_{iq} \geq p^*C_k \qquad k=2,\ldots,n \qquad (3\text{-}29)$$

$$\sum_{i=1}^{m} Z_{ik} \geq p^*C_k \qquad k=1,\ldots,n \qquad (3\text{-}30)$$

$$\sum_{i=1}^{m} Z_{ik} \leq m^*C_k \qquad k=1,\ldots,n \qquad (3\text{-}31)$$

$$\sum_{i\in I_j} \sum_{k=1}^{n} X_{ijk}= 1 \qquad j=1,\ldots,n \qquad (3\text{-}22)$$

$$\sum_{j=1}^{n} X_{ijk}=Z_{ik} \qquad i=1,\ldots,m \quad k=1,\ldots,n \qquad (3\text{-}23)$$

$$\sum_{i=1}^{m} Z_{ik} + \left(\frac{1}{n}\right) \sum_{i=1}^{m}\sum_{q=1}^{k-1} Z_{iq} \geq 1 \qquad k=1,\ldots,n \qquad (3\text{-}24)$$

$$\sum_{q=k}^{k+r_i-1} Z_{iq} \le (1-Z_{i,k-1}+Z_{ik})*r_i \qquad i=1,\ldots,m;\ k=2,\ldots,n-r_i+1 \qquad (3\text{-}25)$$

$$\sum_{q=k}^{k+W_i} Z_{iq} \le W_i \qquad i=1,\ldots,m;\ k=1,\ldots,n-W_i \qquad (3\text{-}26)$$

$$X_{ijk},\ Z_{ik},\ C_k \in \{0,1\} \qquad i=1,\ldots,m;\ j=1,\ldots,n;\ k=1,\ldots,n \qquad (3\text{-}27)$$

We can get the makespan from $\sum_{k=1}^{n} C_k$ if the solution is feasible. The minimum makespans obtained by solving models [ALp–Min] and [ALp–Max] should both be the same.

Models [AL1] and [ALp] have some interesting points which can be shown below:

Lemma 3-1. The solution set of the at-least-p-processors-working case is a subset of the solution set of the at-least-one-processor-working case.

Proof: Let the optimal solution set for the at-least-p-processors-working case be $S^*(P \ge p)$. The solution pattern is the following:

	1	2	.....	M-1	M
P1	J1	Jm+1	.....		J(M-1)m+1
P2	J2	Jm+2	.....		⋮
⋮	⋮	⋮	⋮	⋮	⋮
⋮	Jp	J2p	⋮	⋮	JMp
⋮	⋮	⋮	⋮	⋮	⋮
Pm	Jm	J2m	.....		Jn

m: # of processors

n: # of tasks

M: the min makespan

It is obvious that for a feasible solution:

$$(J_1)+(J_2)+\ldots\ldots+(J_p)+\ldots\ldots+(J_m)\geq p,$$

$$(J_{m+1})+(J_{m+2})+\ldots+(J_{2p})+\ldots+(J_{2m})\geq p,$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$(J_{(M-1)m+1})+(J_{(M-1)m+2})+\ldots+(J_{Mp})+\ldots\ldots+(J_n)\geq p. \qquad (3\text{-}34)$$

where J_1, J_2, . , J_p. . , J_m, ... , J_n are equal to 0 or 1 based on the task-to-processor assignments.

Similarly, let the optimal solution set for the at-least-one-processor-working case be $S^*(P\geq 1)$. Then, the solution pattern is:

	1	2	.....	M-1	M
P_1	J_1	J_{m+1}	.....		$J_{(M-1)m+1}$
P_2	J_2	J_{m+2}	.....		:
:	:	:		:	:
:	:	:		:	:
:	:	:		:	:
P_m	J_m	J_{2m}	.....		J_n

m: # of processors

n: # of tasks

M: the min makespan

where J_1, J_2, . . . , J_m, ... , J_n are also equal to 0 or 1 based on the optimal task-to-processor assignments. Furthermore, we must have

$$(J_1)+(J_2)+\ldots+(J_m)\geq 1$$

$$(J_{m+1})+(J_{m+2})+\ldots+(J_{2m})\geq 1$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$(J_{(M-1)m+1})+(J_{(M-1)(m+2)})+\ldots\ldots+(J_n)\geq 1. \qquad (3\text{-}35)$$

Let $S(P\geq p) \in S^*(P\geq p)$ be an optimal solution to the

at-least-p-processors-working case, with the minimum makespan equal to M_p. By (3-34) and (3-35), $S(P \geq p)$ is also a feasible solution to the at-least-one-processor working case since $p \geq 1$. Suppose $S(P \geq p) \notin S^*(P \geq 1)$; then there exists an optimal solution $S(P \geq 1) \in S^*(P \geq 1)$ with the minimum makespan less than M_p. Without loss of generality, assume that the makespan is $M_p - 1$. Since $S(P \geq p)$ is a feasible solution with makesapn M_p to the at-least-one-processor-working case, to construct a feasible schedule with makespan equal to $M_p - 1$, we must reassign the tasks previously scheduled for the M_{pth} time period to the first $M_p - 1$ time periods. By (3-34), the reconstructed schedule is also a feasible schedule for the at-least-p-processors-working case with makespan equal to $M_p - 1$. This contradicts the fact that $S(P \geq p) \notin S^*(P \geq p)$. Therefore, if $S(P \geq p)$ is one of the optimal solutions for the at-least-p-processor-working case, it is also the optimal solution for at-least-one-processor-working case. Here, $S^*(P \geq p) \subset S^*(P \geq 1)$, but not vice versa.

Q.E.D.

Exact Formulation for at-most-p-processor-working at every time unit

For the case in which at most p processors are allowed to be working in each time period, we do not need the condition that at least p tasks remain for processing in every time unit k, so we can drop the index variable C_k. But we need to have constraint (3-24) to ensure that at least one processor is working and all

tasks are finished before time period n. MODEL [AM_p] is shown below for the at-most-p-processors-working case.

MODEL [AM_p]

$$X_{ijk}=\begin{cases} 1, & \text{if task } j \text{ is processed on processor } i \text{ at time unit } k. \\ 0, & \text{otherwise.} \end{cases}$$

$$Z_{ik}=\begin{cases} 1, & \text{if processor } i \text{ executes a task at time unit } k. \\ 0, & \text{otherwise.} \end{cases}$$

P: an integer number, representing the maximum number of processors that can be working during any time period.

Minimize M_o (3-20)

Subject to:

$$\sum_{i=1}^{m} Z_{ik} \le P \qquad k=1,\ldots,n \qquad (3\text{-}33)$$

$$k*Z_{ik} \le M_o \qquad i=1,\ldots,m;\ k=1,\ldots,n \qquad (3\text{-}21)$$

$$\sum_{i\in I_j} \sum_{k=1}^{n} X_{ijk} = 1 \qquad j=1,\ldots,n \qquad (3\text{-}22)$$

$$\sum_{j=1}^{n} X_{ijk} = Z_{ik} \qquad i=1,\ldots,m;\ k=1,\ldots,n \qquad (3\text{-}23)$$

$$\sum_{i=1}^{m} Z_{ik} + \left(\frac{1}{n}\right) \sum_{i=1}^{m} \sum_{q=1}^{k-1} Z_{iq} \geq 1 \qquad k=1,\ldots\ldots,n \qquad (3\text{-}24)$$

$$\sum_{q=k}^{k+r_i-1} Z_{iq} \leq (1 - Z_{i,k-1} + Z_{ik}) * r_i \qquad \begin{matrix} i=1,\ldots\ldots,m \\ k=2,\ldots\ldots,n-r_i+1 \end{matrix} \qquad (3\text{-}25)$$

$$\sum_{q=k}^{k+W_i} Z_{iq} \leq W_i \qquad \begin{matrix} i=1,\ldots\ldots,m \\ k=1,\ldots\ldots,n-W_i \end{matrix} \qquad (3\text{-}26)$$

$$X_{ijk},\ Z_{ik} \in \{0,1\} \qquad \begin{matrix} i=1,\ldots\ldots,m \\ j=1,\ldots\ldots,n \\ k=1,\ldots\ldots,n \end{matrix} \qquad (3\text{-}27)$$

Constraint (3-33) ensures that there are at most p processors working at every time unit. The interpretation of the remaining constraints are the same as for the previous models.

An interesting potential application for the "at-most-p-processors-working" requirement can be provided as follows. Consider, for example, a situation with 5 machines available to process tasks. By solving the [AMp] model several times with different p values, we can examine the effect of having at most p machines working in any given time period on the minimum makespan. That is, the trade-off between the utilization and the reliability of the system can be captured in the above sensitivity analysis.

3.5 Constraint Relaxation and Decomposition of the Exact Model—— An Alternative to Solve the Continuous Task Scheduling Problem

In **MODEL [AL_1]**, constraints (3-22) and (3-23) show the connection and relationship between variables X_{ijk} and Z_{ik}. However, since there are six sets of constraints and i*j*k + i*k binary variables included, it takes a lot of memory for this formulation to be solved on computer. For example, if we have 5 processors and 50 tasks, there will be 5*50*50=12,500 X_{ijk} variables. Realistic problems can therefore be too large even for a mainframe computer. Thus, we need to find an alternative approach to solving the formulation.

If we remove constraints (3-22) and (3-23) from **MODEL [AL_1]**, the model will contain only four sets of constraints and i*k binary variables included in this formulation; we name this formulation **MODEL [Z_{ik}]** now, which can be listed below:

MODEL [Z_{ik}]

$$Z_{ik}=\begin{cases} 1, & \text{if processor } i \text{ executes a task at time unit } k. \\ 0, & \text{otherwise.} \end{cases}$$

Minimize M_0 (3-20)

Subject to:

$$k*Z_{ik} \le M_0 \qquad \begin{matrix} i=1,\ldots,m \\ k=1,\ldots,n \end{matrix} \qquad (3\text{-}21)$$

$$\sum_{i=1}^{m} Z_{ik} + \left(\frac{1}{n}\right) \sum_{i=1}^{m} \sum_{q=1}^{k-1} Z_{iq} \geq 1 \qquad k=1,\ldots\ldots,n \qquad (3\text{-}24)$$

$$\sum_{q=k}^{k+r_i-1} Z_{iq} \leq (1-Z_{i,k-1}+Z_{ik})*r_i \qquad \begin{matrix} i=1,\ldots\ldots,m \\ k=2,\ldots\ldots,n-r_i+1 \end{matrix} \qquad (3\text{-}25)$$

$$\sum_{q=k}^{k+W_i} Z_{iq} \leq W_i \qquad \begin{matrix} i=1,\ldots\ldots,m \\ k=1,\ldots\ldots,n-W_i \end{matrix} \qquad (3\text{-}26)$$

$$Z_{ik} \in \{0,1\} \qquad \begin{matrix} i=1,\ldots\ldots,m \\ k=1,\ldots\ldots,n \end{matrix} \qquad (3\text{-}27)$$

Solving this formulation gives Z_{ik}, which indicates whether processor i is working at time unit k. After that, we can solve Model $[X_{ij}]$ to obtain the task-to-processor assignments. Model $[X_{ij}]$ is listed below:

MODEL $[X_{ij}]$

$$X_{ij}=\begin{cases} 1, & \text{if task } j \text{ is processed on processor } i. \\ 0, & \text{otherwise.} \end{cases}$$

Let Zsum(i) be the sum of tasks that are assigned to processor i, where $Zsum(i)=\sum_{k=1}^{n} Z_{ik}$, and are obtained by solving Model $[Z_{ik}]$.

$$\text{Max} \sum_{J_j \in J} \sum_{i=1}^{m} X_{ij} \qquad (3\text{-}28)$$

$$\text{Subject to: } \sum_{P_i \in P} X_{ij}=1 \qquad j=1,\ldots\ldots,n \qquad (3\text{-}29)$$

$$\sum_{Jj \in J} X_{ij} \leq Zsum(i) \qquad i=1,\ldots\ldots,m \qquad (3\text{-}30)$$

$$X_{ij} \in \{1,0\} \qquad i=1,\ldots\ldots,m \quad j=1,\ldots\ldots,n \qquad (3\text{-}31)$$

The [AL1] model has been decomposed into two sub-models [Zik] and [Xij]. In the [Zik] model, we remove the constraints (3-22) and (3-23) of [AL1], and this model is equivalent to an assignment problem, which will provide the processor with a time period assignment matrix. Note here that the solution may or may not be feasible to the original problem. The [Xij] model is a transportation model, where we seek to obtain a feasible schedule using an input the solution to the [Zik] model. The idea is to solve the [Zik] and [Xij] models sequentially and iteratively until an optimal solution to the original problem is found. We begin by solving the [Zik] model. A minimum makespan M and the corresponding matrix Z are obtained. This assignment matrix indicates which processors will execute tasks in a given time period. Again, the Z matrix may or may not be feasible. We then feed the Z matrix into the [Xij] model and solve it. If the Z matrix is feasible, then the solution to the [Xij] model will give the optimal schedule. If the Z matrix is not feasible, we then return to the [Zik] model, add one additional constraint $M \geq M^{*}+1$, to enlarge the feasible region of the current makespan M^{*} (because relaxing the constraints may get an underestimate of makespan), and resolve the [Zik] model.

We can use the feasible/infeasible signal from the [X_{ij}] model to determine whether we have reached an optimal schedule or not. In modifying the [Z_{ik}] model, we can also add constraints $\sum_{i=1}^{m} Z_{ik} \leq Pmax(i)$, where Pmax(i) is the maximum number of tasks that processor i can execute, and $\sum_{i=1}^{m} Z_{ik} \geq Pmin(i)$, where Pmin(i) is the minimum number of tasks that processor i must execute. Both Pmax(i) and Pmin(i) are derived from the given original task-to-processor set. Such modification to **MODEL** [Zik] will result in an improved Z matrix, which can then be fed into the **MODEL** [X_{ij}].

The above procedure has been automated using the IP solver in GAMS. The number of iterations is arbitarily set equal to 25, but usually the procedure converges with only a fraction of the above limit on the number of iterations.

3.6 Some Discussions on Integer Programming Packages

There is a survey in the October 1990 issue of MS/OR Today for linear programming software packages, some of which have the capabilities to solve the integer programming formulations.

This survey reports that problems with a few hundred binary variables are solvable on IBM 386 type machines. If there are more variables, we have to use more powerful machines and high performance commercial solvers like MPSX, SCICONIC or LAMPS. Models with more than 500 binary variables require careful modeling to be solved. Some solvers offer advanced features like

OSL (optimization subroutine library) to facilitate the solution process.

Based on the results of the above surveys, we chose GAMS as the solver to obtain an optimal schedule in our experiment, by using GAMS-ZOOM on an IBM 386 PC with a math coprocessor 387 and Cornell's supercomputer to solve large size problem instances. Three tactics can be used to handle large problem sizes:

1. Reduce the number of periods, i.e., the range of the k index in the [AL_1], [AL_p-Min], [AL_p-Max] and [AM_p] formulations. This will reduce the number of binary variables. For instance, for the 5 processor and 50 task problem instance, we need at least 5x50x50=12,500 binary variables before. If we reduce the range of k index from 50 to say, 15, then the formulation requires only a little more than 5x50x15=3,750 binary variables. Of course, this approach is limited to those problem instances which have a solution with makespan less than or equal to 15. That is, we have to be very careful in choosing the problem instances. A single heuristic could be used to obtain an upper bound on k.

2. The Constraint-Relaxation approach can be used to solve larger problem instances. For a 5 processor and 50 task instance, the approach requires the solution of two double-index models, which contain a total of 500 (2x5x50) binary variables.

3. Once the GAMS-OSL version 2 solver is installed in Cornell Supercomputer, its limit can then be tested.

3.7 Chapter Summary

In Sections 3.1 and 3.2, we showed some important properties about NP-completeness and proved that the continuous task scheduling problem is NP-complete in the strong sense.

In Section 3.3, we reviewed previous mathematical programming formulations for scheduling on unrelated multiprocessing systems.

In Section 3.4, we have completed the following: (1) The formulation for the "at-least-one-processor-working" case denoted as Model [AL_1]. (2) Three mathematical programs are provided for the "at-least-p-processors-working" case (denoted as Model [AL_p-Min] and Model [AL_p-Max]), and for the "at-most-p-processors-working" case denoted as Model [AM_p], respectively. (3) We proved that the solution set of the "at-least-p-processors-working" case is a subset of the solution set of the at-least-one-processor-working case. An interesting potential application for the "at-most-p-processor-working" case was also provided.

In Section 3.5, we developed a constraint-relaxation solution approach. The procedure has been implemented using GAMS on an IBM 386 PC.

In Section 3.6, GAMS has been chosen to solve the integer programs, by using GAMS-ZOOM on an IBM 386 PC with a math coprocessor 387 and Cornell's supercomputer to solve large size problem instances.

CHAPTER FOUR

AN ALGORITHM FOR THE CONTINUOUS TASK SCHEDULING PROBLEM

4.1 Basic Ideas of the Algorithm for the Continuous task Scheduling Problem

Although the continuous task scheduling problem is NP-complete, an algorithm can be designed to solve this problem as will be described in this section. It should be noted that NP-completeness is with respect to the worst case. Thus, it is still possible that there exists an algorithm which can solve the continuous task scheduling problem efficiently for average cases.

Definition 4-1: Let [AL_p] denote the problem with the constraint that at least p processors must be working in every time period, and [AM_p] be the problem with the constraint that at most p processors can be working in every time period.

Our basic idea to solve the [AL_p] or [AM_p] problem is to solve the [AL_1] problem first, then, based on the [AL_1] solution, we can adjust and fulfill the p requirement for the [AL_p] or [AM_p] problem from the solution pattern for the [AL_1] problem. There are two phases in the algorithm, the first phase is to find an optimal schedule for the [AL_1] problem, and the second phase is to adjust the [AL_1] solution pattern according to either the [AL_p] or the [AM_p] requirement. In the following, we shall let $Z_{ik}=1$ denote that processor i is executing some task at time k and let $Z_{ik}=0$ denote that processor i is not executing any task at time k. Given

an instance of a continuous task scheduling problem with m processors, we can then represent a solution to the problem by the sequence $Z_{11}Z_{21}\ldots Z_{m1}Z_{12}Z_{22}\ldots Z_{m2}\ldots Z_{ik}$, $1\le k\le n$ and $1\le i\le m$, Z_{ik} is the last "1" in the sequence, but not necessarily Z_{mn}.

Let J be the task set, P be the processor set, S_i be the subset of J which can be executed by processor i. We have the following Example 4-1.

Example 4-1. Let $J=\{J_1, J_2, J_3, J_4, J_5, J_6, J_7, J_8, J_9, J_{10}\}$, $P=\{P_1, P_2, P_3, P_4, P_5\}$, $S_1=\{J_1, J_2, J_4\}$, $S_2=\{J_2, J_3, J_4\}$, $S_3=\{J_3, J_4, J_5, J_6\}$. $S_4=\{J_6, J_7, J_8\}$, $S_5=\{J_8, J_9, J_{10}\}$. $(w_1, r_1)=(w_3, r_3)=(2,1)$, $(w_2, r_2)=(1,2)$, and $(w_4, r_4)=(w_5, r_5)=(1,1)$. Figure 4-1 shows two feasible solutions for this [AL1] problem. The two sequences corresponding to these feasible solutions are 1111110100000111 and 11110101010001010001 respectively.

P1	J1	J2	0	J4		P1	J1	J2	0	J4
P2	J3	0	0			P2	J3	0	0	0
P3	J5	J6	0			P3	J5	J6	0	0
P4	J7	0	J8			P4	J7	0	J8	0
P5	J9	0	J10			P5	0	J9	0	J10
TIME	1	2	3	4			1	2	3	4

Figure 4-1 Two examples of feasible solutions

Definition 4-2: A sequence $Z_{11}Z_{21}\ldots Z_{m1}Z_{12}Z_{22}\ldots Z_{m2}\ldots$ is a legal sequence if this assignment satisfies the working and resting conditions and the continuous task scheduling requirement,

but not necessarily the processor-task relationship.

For instance, consider Example 4-2.

Example 4-2 $J=\{J_1, J_2, J_3, J_4, J_5, J_6\}$, $P=\{P_1, P_2, P_3\}$, $S_1=\{J_1\}$, $S_2=\{J_2, J_3, J_6\}$, $S_3=\{J_2, J_4, J_5\}$, $(w_1, r_1)=(2,1)$, $(w_2, w_2)=(1,1)$, and $(w_3, r_3)=(1,1)$. In this case, the schedule shown in Fig 4-2 is a legal sequence in that it satisfies the working-resting conditions and the continuous task scheduling requirement. Yet it is not a feasible solution, because P_1 can only execute one task and, in the solution, P_1 will have to execute two tasks. In other words, $Z_{12}=1$ is impossible as P_1 actually can not execute any task after time $t>1$ if it has already executed a task at time $t=1$.

P_1	1	1	0
P_2	1	0	1
P_3	1	0	1
time	1	2	3

Figure 4-2 A legal sequence satisfying the working/resting conditions

Obviously, a feasible sequence is also a legal sequence. But, not every legal sequence corresponds to a feasible solution. One solution approach is to first find a legal sequence with the minimum length and then determine whether there exists a feasible schedule corresponding to this legal sequence. If such a schedule exists, we have found an optimal solution; otherwise, we try to find the next shortest legal sequence and repeat the above process

until a feasible schedule is found to match the legal sequence. The algorithm considered here is simple, but not efficient, because we may need to search all legal sequences to find the optimal solution.

The efficiency of the above approach can be improved if the checking for feasibility is not delayed until a complete sequence is found. As we are constructing a legal sequence, we try to find a partial schedule which corresponds to the partial legal sequence. We shall now show that with this method, we can sometimes terminate the construction of this legal sequence prematurely because there can be no feasible schedule corresponding to this legal sequence. For instance, let us consider Example 4-2. Suppose that we have constructed the partial legal sequence shown in Figure 4-3.

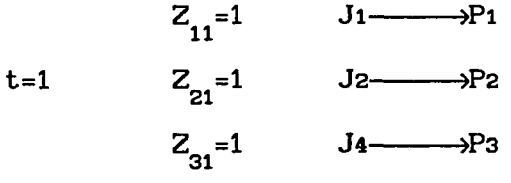


Figure 4-3 A partial legal sequence

Since $(w_1, r_1)=(2,1)$, P1 does not have to rest at time t=2. Therefore, a legal sequence may be $Z_{12}=1$ (meaning that a task will be assigned to P1). However, this is impossible because P1 has already executed all of the tasks that it can execute. Thus the construction of the legal sequence 1111... can now be terminated. Or, equivalently, if $Z_{11}=Z_{21}=Z_{31}=1$, then Z_{12} must be 0.

The above example is a special case in which we can quickly decide that Z_{12} must be 0. To illustrate the more general case, suppose we have already constructed a partial legal sequence which has a feasible partial schedule corresponding to it and our next step is to determine whether Z_{1k} should be 1 or 0. Let us further assume that as far as P_1 is concerned, it need not rest. Therefore, Z_{1k} may be assigned 1. Furthermore, in the partial feasible schedule, assume that all of the tasks that processor P_1 can execute have already been assigned, some of them to P_1 and the rest to other processors. Even though processor P_1 has no remaining tasks to be assigned, we can not necessarily conclude that Z_{1k} should be set to zero. This is because it may be possible to reassign tasks to the other processors freeing up one task that could then be assigned to P_1. To check this, we may call a reassignment subroutine. This subroutine will attempt to find a reassignment of tasks to processors in such a way that the partial legal sequence already constructed remains unchanged, although the specific assignment of tasks will change, and P_1 may be assigned a task to execute. If this subroutine does find such a reassignment, then Z_{1k} is assigned 1; otherwise Z_{1k} is assigned 0.

Let us illustrate our idea again by considering an example.

Example 4-3 In Example 4-2, suppose that the task set that P_1 can execute is changed from $S_1=\{J_1\}$ to $S_1=\{J_1, J_2\}$. Then after the partial sequence of Figure 4-3, we consider Z_{12}. Since $w_1=2$, Z_{12}

can be assigned 1. But all of the tasks that P_1 can execute have already been assigned. Therefore, a task reassignment, if possible, is required. The reassignment subroutine will find that by reassigning J_3 to P_2, J_2 can be assigned to P_1 and 1111 will be a legal partial sequence with a feasible partial schedule. Consequently, Z_{12} can be set to 1.

In summary, our algorithm is based upon the following basic ideas:

(1) The algorithm seeks the shortest legal sequence for [AL_1], which has a feasible schedule corresponding to it. Since this legal sequence is the shortest one, its corresponding feasible schedule must be optimal for [AL_1].

(2) Our algorithm constructs such a shortest legal sequence step by step by determining whether Z_{ik} can be 1 or 0. if $Z_{ik}=1$, we also record the task assigned to processor P_i at time $t=k$. Thus, each partially constructed legal sequence L has a corresponding partial task assignment A for [AL_1].

(3) After the basic optimal solution pattern for [AL_1] is derived, it may be necessary to move some tasks from time periods with excess tasks to those time periods with a shortage in tasks to fulfill the "at-least-p" or "at-most-p" requirement for every time period.

To determine whether $Z_{ik}=1$ or 0, three possible situations are considered:

Case (a): Processor P_i has to rest (i.e., it has worked w_i time

units or is already resting and has not completed r_i consecutive periods of rest). Assign $Z_{ik}=0$.

Case (b): Processor P_i need not rest and there is a task (possibly through a task reassignment) for P_i to execute. Assign $Z_{ik}=1$.

Case (c): Processor P_i need not rest but there are no more tasks for P_i to execute and, the reassignment subroutine fails to reassign a task for P_i to execute. Assign $Z_{ik}=0$.

(4) If setting $Z_{mk}=0$ violates the continuous scheduling requirement (i.e., $Z_{ik}=0$ for all i), and the at-least-one-processor-working requirement (i.e., $\sum_{i=1}^{m} Z_{ik}<1$), construction of this partial legal sequence can be terminated.

We have described the basic ideas of our algorithm to solve the [AL_p] or [AM_p] problem in this section. In so doing, we have not answered a very important question: How can we know that our algorithm will produce a shortest legal sequence for [AL_1] problem? This will be explained in the next section.

4.2 [PHASE ONE]: The Specialized Branch-and-Bound Algorithm for the [AL1] Problem

Given an instance, our task is to find a legal sequence L of the shortest length that has a corresponding feasible schedule. The makespan of this feasible schedule is $\lfloor len(L)/m \rfloor$, where len(L) is the length of the feasible schedule and m is the number of processors. The concept of our algorithm is similar to that of the A^* algorithm (Slagle and Bursky, 1968; Hart, 1971; Nilson, and Raphael, 1980; Nilson, 1980; Pearl, 1984).

Figure 4-4 shows a typical tree constructed during the execution of our algorithm which follows the concept of the A^* algorithm. "B" means bounded nodes. A bounded node is a node for which the tree can not go further and gets an infeasible solution. "G" means goal nodes. A goal node is a node for which the legal sequence reaches an optimal solution. The unbold-line rectangular boxes enclose nodes which have been generated but unexpanded. All other nodes are expanded. Note that the tree is constructed by determining whether $Z_{ik}=1$ or 0. As can be seen, one of our problems in designing such an algorithm is to select an appropriate node to expand. An A^* algorithm always selects a least cost node to expand; we follow what principle here as well. We first explain how to evaluate the node-selection function f. The node-selection function f used in our algorithm consists of two parts, g(n) and h(n), where g(n)=the cost of the current path from start node s to n, with g(s)=0

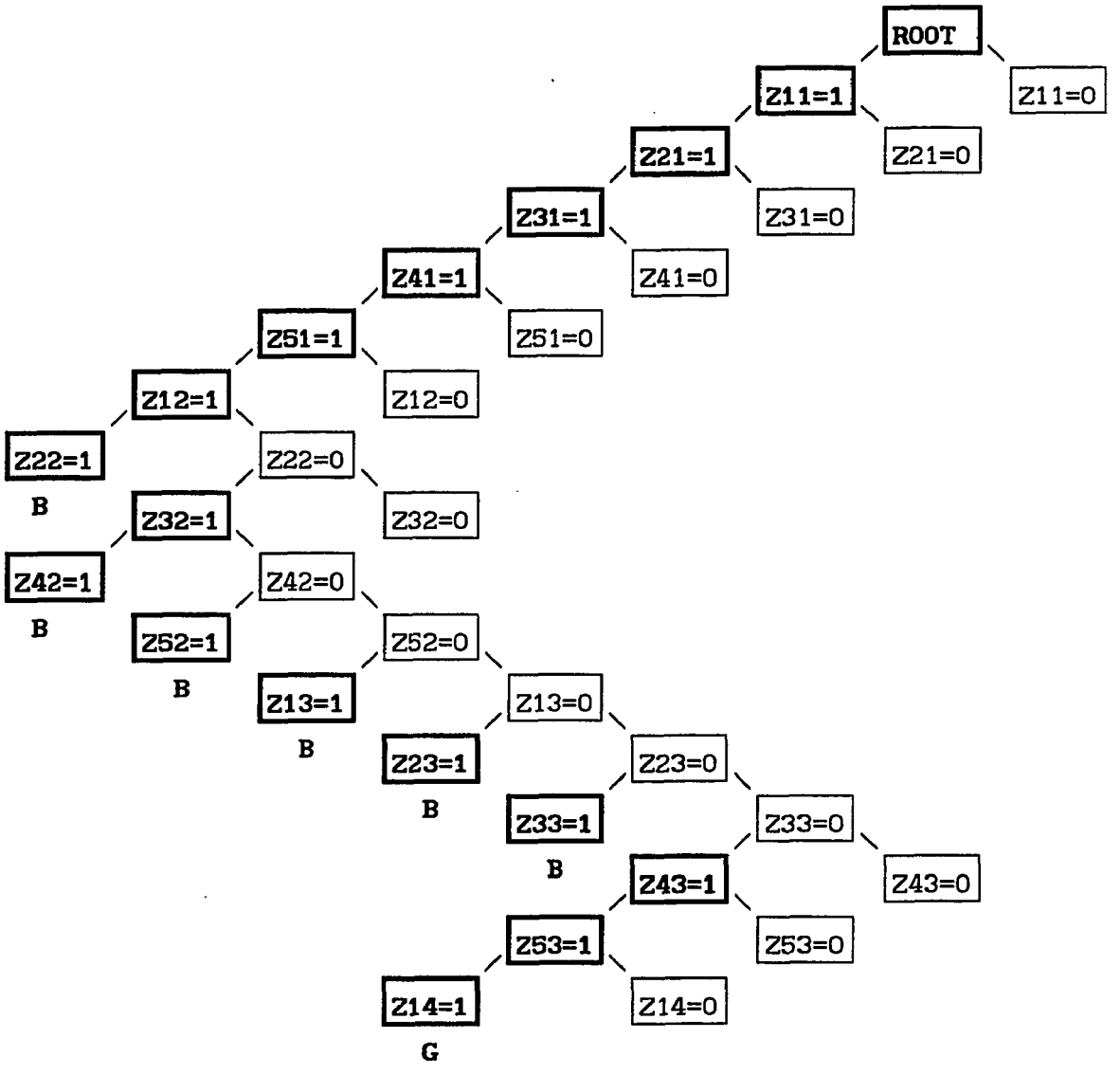


Figure 4-4 A typical tree constructed during the execution of the algorithm

g(n)=the cost of the current path from start node s to n, with g(s)=0

h(n)=an estimate of the cost of the best path from n to a goal node, where h(goal node)=0.

Since our goal is to find the shortest legal sequence that has a corresponding feasible schedule, the cost will be the length of the legal sequence. g(n) is just the length of the partial legal sequence from the root of the search tree to the current node, and h(n) is an estimate of the remaining length required to form a complete schedule without considering the processor-task relationship and the continuous task scheduling requirement. The idea is illustrated in Figures 4-5 and 4-6 using Example 4-1.

The corresponding partial legal sequence is $Z_{11}Z_{21}Z_{31}Z_{41}$=1111. Suppose now that we expand node Z_{41}=1. Its two children are Z_{51}=1 and Z_{51}=0. We have $g(Z_{51}=1)=g(Z_{51}=0)=5$ because their partial legal sequences are 11111 and 11110 respectively. To calculate h, we must form a complete schedule from the partial schedule. This is done without regard to processor-task assignments or to the continuity constraint. The complete schedule will obey the working and resting conditions of processors. Figure 4-5 shows the schedule extended from partial legal sequence 11111 and Figure 4-6 shows the schedule extended from 11110. From Figure 4-5, $h(Z_{51}=1)=11$. From Figure 4-6, $h(Z_{51}=0)=15$.

P_1	J_1	1	0	1
P_2	J_3	0	0	
P_3	J_5	1	0	
P_4	J_7	0	1	
P_5	1	0	1	
TIME	1	2	3	4

Figure 4–5 A legal sequence without continuity condition (f=16)

P_1	J_1	1	0	1
P_2	J_3	0	0	0
P_3	J_5	1	0	0
P_4	J_7	0	1	0
P_5	0	1	0	1
TIME	1	2	3	4

Figure 4–6 A legal sequence without continuity condition (f=20)

From the definition of h, the following lemma is obvious.

Lemma 4–1. h is an underestimate of the actual cost and can be calculated in time proportional to the number of remaining unscheduled tasks.

Since the cost of Figure 4-5 (cost=16) is less than that for Figure 4-6 (cost=20), we next choose the node Z_{51}=1 to expand the tree. This is repeated until a goal node is reached. From this goal node, we then go back to the original root and expand all of the possible [011...1] sequences, compare the cost of the goal node (the sequence [111...1]) with the costs of the [011...1] sequences, and pick up the least cost of the sequence to ensure that an optimal solution is derived.

4.3 The Reassignment Subroutine

4.3.1 Version A

In this section, we show how the REASSIGNMENT subroutine works. There are two versions. Let's first introduce version A with an example, then version B in section 4.3.2.

Example 4-4 Let $J=\{J_1, J_2, J_3, J_4\}$, $P=\{P_1, P_2, P_3\}$, and $S_1=\{J_1, J_2\}$, $S_2=\{J_1, J_2, J_3\}$, $S_3=\{J_3, J_4\}$. $(w_1,r_1)=(2,1)$, and $(w_2, r_2)=(w_3, r_3)=(1,2)$. Suppose that the assignment of the first time unit is as shown in Figure 4-7.

P_1	J_1
P_2	J_2
P_3	J_3
TIME	1

Figure 4-7 Task Assignment at the first time unit

When we consider time unit 2, P_1 can still execute a task ($w_1=2$), but both of the tasks that P_1 can execute have already been assigned. Therefore, if P_1 is to work in period 2, some task previously assigned to another processor must be reassigned to P_1.

Definition 4-3 An assignment is called **partial** if there exists at least one task which has not been assigned yet. Those unassigned tasks are called **free** tasks.

Figure 4-7 is a partial assignment. The only free task is J_4.

Definition 4-4 Given a partial assignment X, the **partial assignment graph** corresponding to X is a directed bipartite graph, defined as follows. The vertex set $V=J \cup P$ and the edge set $E=\{(P_i, J_j) \mid J_j$ is assigned to P_i in X$\} \cup \{(J_j, P_i) \mid J_j \in S_i$ and J_j is not assigned to $P_i\}$.

The partial assignment graph corresponding to Figure 4-7 of Example 4-4 is shown in Figure 4-8.

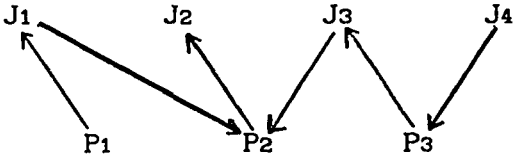


Figure 4-8 A partial assignment graph

Definition 4-5 If P_i can execute a task but all tasks that P_i can execute have been assigned, P_i is called a **critical** processor, P_c, and the situation is called critical.

Definition 4-6 A critical situation is said to be **remediable** if and only if there is a way to reassign tasks such that the critical processor will become noncritical and the current partial legal sequence (the current execution patterns of all processors) remains unchanged. That is, the working and resting time units remain unchanged. Only the task-to-processor assignments are changed.

Definition 4-7 Let A_k be the set of tasks already assigned to P_k in a partial assignment, for $1 \le k \le m$. Let P_c be a critical processor. Then P_j, is called a **donating** processor P_d with respect to P_c if and only if $A_j \cap S_c \neq \phi$. That is, processor P_j is a donating processor if one or more of the tasks assigned to that processor can also be performed by the critical processor.

In Example 4-4, P_1 is the critical processor at time unit 2. We have $A_1=\{J_1\}$, $A_2=\{J_2\}$, and $A_3=\{J_3\}$, $S_1 \cap A_2=\{J_2\}$, $S_1 \cap A_3=\emptyset$. Therefore P_2 is the only donating processor with respect to P_1. This critical situation is remediable since a schedule as shown in Figure 4-9 exists.

P_1	J_1	J_2
P_2	J_3	
P_3	J_4	
TIME	1	2

Figure 4-9 A schedule which is remediable

Lemma 4-2 A critical situation is remediable if and only if there is a path from a free task to a critical processor in the partial assignment graph.

Proof: Suppose that there is a path $J_f \longrightarrow P_{i_1} \longrightarrow J_{j_1} \longrightarrow \ldots \longrightarrow P_{i_k} \longrightarrow J_{j_k} \longrightarrow P_d$ in the partial assignment graph where J_f is a free task and P_d is a donating processor. We can remedy the critical situation by the following procedures. For each J_{j_x},

$1 \le x \le k-1$, reassign Jj_x to Pi_{x+1}. Assign Jf to Pi_1 and Jj_k to Pd. The number of tasks assigned to each processor remains unchanged except that of Pd, which has increased by one. But since Pd is a donating processor, one of the tasks which are assigned to Pd can be reassigned to the critical processor. Thus we have remedied the critical situation.

On the other hand, suppose that the critical situation is not remediable. Then one of the tasks which is assigned to one of the donating processors must be reassigned to the critical processor. If there is no path from any free task to any donating processor, the number of tasks assigned to at least one processor will be decreased by one since the number of tasks assigned to the critical processor will be increased by one. The current partial legal sequence has been changed. This is contradictory to the assumption that the critical situation is remediable.

Q.E.D.

In Figure 4-8 there is a path from $J4$ (a free task) to $P2$ (the donating processor). Therefore the critical situation can be remedied by using this path. The result has been shown in Figure 4-9.

We next present a detailed procedure for reassignment subroutine Version A.

Algorithm: The Reassignment Subroutine Version A

Input: A partial assignment together with a critical processor P_i

Output: A new partial assignment, if possible, to remedy the critical situation.

Step 1: Construct the partial assignment graph.

Let X be a directed bipartite graph defined as follows. The vertex set $V=J \cup P$ and the edge set $E=\{(P_i, J_j) \mid J_j$ is assigned to P_i in $X\} \cup \{(J_j, P_i) \mid J_j \in S_i$ and J_j is not assigned to $P_i\}$.

Step 2: Determine the free tasks and the donating processors.

Let A_k be the set of tasks already assigned to P_k in a partial assignment for $1 \leq k \leq m$. Set of free tasks is defined as $J-(\bigcup_{k=1}^{m} A_k)$.

Let P_i be a critical processor

Then P_j, $j \neq i$, is called a donating processor with respect to P_i, if $A_j \cap S_i \neq \emptyset$.

Step 3: **For** each free task **do**

For each donating processor **do**

if there exists a directed path that begins with the free task vertex and ends at this donating processor vertex, **then go to** Step 5.

Step 4: **Return** with failure.

Step 5: Reassign the tasks according to the path found from the free task to the donating processor, assign the task

originally assigned to the donating processor to the critical processor and **return** successfully.

Lemma 4-3 The **REASSIGNMENT** subroutine is correct and of time complexity $O(m^2n^2)$, where m is the total number of processors and n is the total number of tasks.

Proof: The correctness is due to Lemma 4-2. Since there are at most O(n) free tasks and O(m) donating processors, the algorithm will terminate after at most O(mn) iterations. Each iteration can be implemented by either depth first search or breadth first search in $O(|E|)=O(mn)$ time. Therefore the total time complexity is $O(m^2n^2)$.

Q.E.D.

The flowchart of the reassignment subroutine version A is shown in Figure 4-10.

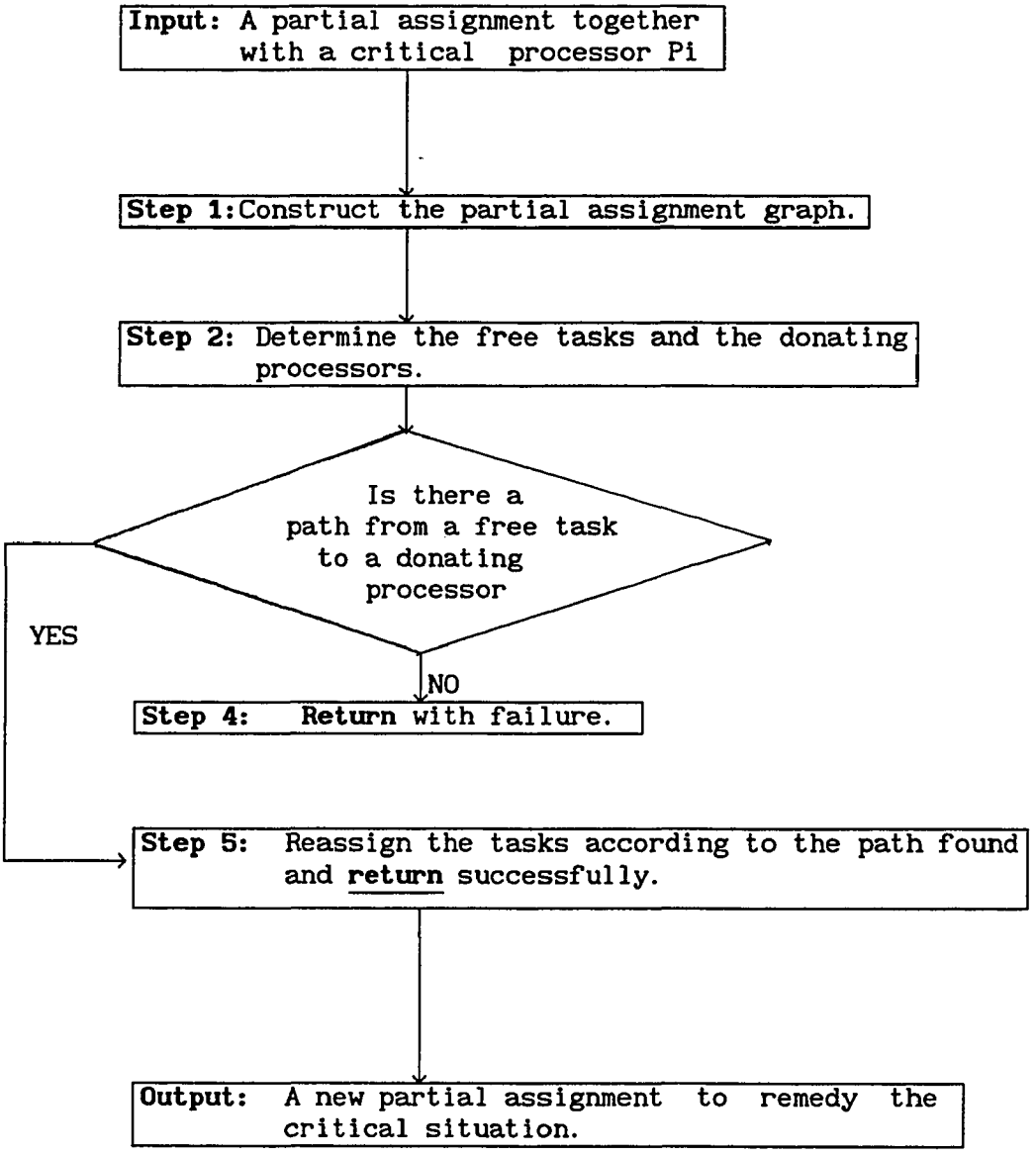


Figure 4-10 THE FLOWCHART OF REASSIGNMENT SUBROUTINE VERSION A

4.3.2 Version B

In this section, we show an alternative REASSIGNMENT subroutine. We repeat Example 4-4 here as Example 4-5.

Example 4-5 Let $J=\{J_1, J_2, J_3, J_4\}$, $P=\{P_1, P_2, P_3\}$, and $S_1=\{J_1, J_2\}$, $S_2=\{J_1, J_2, J_3\}$, $S_3=\{J_3, J_4\}$. $(w_1,r_1)=(2,1)$, and $(w_2,r_2)=(w_3,r_3)=(1,2)$. Suppose that the assignment of the first time unit is as shown in Figure 4-11.

P_1	J_1
P_2	J_2
P_3	J_3
TIME	1

Figure 4-11 Task assignment at the first time unit

Definitions 4-3, 4-5, and 4-6 remain unchanged here. In addition, we have:

Definition 4-8 Let A_k be the set of tasks already assigned to P_k in a partial assignment for $1 \le k \le m$. Then the set of free tasks F, can be defined as $F=J-(\bigcup_{k=1}^{m} A_k)$.

Definition 4-9 Let P_i be a critical processor. Then P_j, $j \neq i$, is called a **donating** processor P_d with respect to P_i if $A_j \cap S_i \neq \phi$ and P_q, $q \neq i \neq j$, is called a **linking** processor with respect to P_j if $F \cap S_q \neq \emptyset$ and $A_q \cap S_j \neq \emptyset$.

REASSIGNMENT RULE: Given a partial assignment with a critical processor P_c, first we determine the free tasks F, linking processors P_q, and donating processors P_d. If there is at least one free task, linking processor, and donating processor:

Reassign (1) a free task to the linking processor.

(2) a task previously executed by the linking processor to the donating processor.

(3) a task previously executed by the donating processor to the critical processor.

and **return** to main algorithm successively. Otherwise, return with failure.

In Example 4-5, P_1 is the critical processor at time unit 2. We have $A_1=\{J_1\}$, $A_2=\{J_2\}$, $A_3=\{J_3\}$, $F=\{J_4\}$. Since $A_2 \cap S_1=\{J_2\}$, P_2 is a donating processor with respect to P_1. Furthermore, since $F \cap S_3=\{J_4\} \neq \emptyset$ and $A_3 \cap S_2=\{J_3\} \neq \emptyset$, P_3 is a linking processor with respect to P_2. This critical situation is remediable since a schedule as shown in Figure 4-12 exists. i.e. Assign J_4 to P_3, J_3 to P_2 and J_2 to P_1.

P_1	J_1	J_2
P_2	J_3	
P_3	J_4	
TIME	1	2

Figure 4-12 A schedule which is remediable

Algorithm: The Reassignment Subroutine Version B

Input: A partial assignment together with a critical processor P_i.

Output: A new partial assignment, if possible, to remedy the critical situation.

Step 1: Determine the **free tasks**, **donating** and **linking processors**.

Let A_k be the set of tasks already assigned to P_k in a partial assignment for $1 \le k \le m$, then the **free tasks** set are defined as $F=J-(\bigcup_{k=1}^{m} A_k)$.

Let P_i be a critical processor. Then P_j, $j \neq i$, is called a **donating** processor with respect to P_i if $A_j \cap S_i \neq \emptyset$, and P_q, $q \neq i \neq j$, is called a **linking** processor with respect to P_j if $A_q \cap S_j \neq \emptyset$ and $F \cap S_q \neq \emptyset$.

if P_j and P_q exist, **go to Step 3**.

Step 2: **Return** with failure.

Step 3: **Reassign** (1) a free task to the linking processor.

(2) a task previously executed by the linking processor to the donating processor.

(3) a task previously executed by the donating processor to the critical processor.

return successfully.

Lemma 4-4 The **REASSIGNMENT** subroutine is of time complexity $O(m^2n)$, where m is the total number of processors and n is the total number of tasks.

Proof: There are at most O(n) free tasks, O(m) donating processors, and O(m) linking processors. The algorithm will terminate after at most $O(m^2n)$ iterations. So this algorithm can be implemented by search in $O(|E|)=O(m^2n)$ iterations. Therefore, the total time complexity is $O(m^2n)$.

Q.E.D.

Lemma 4-5 Not all remediable critical conditions can be remedied using REASSIGNMENT SUBROUTINE Version B.

proof: In reassignment subroutine Version A, we examine every possible intersection among the processors, while in Version B, we only search the intersection among any three processors. Version B should take less processing time, but Version A may be more successful in finding legal reassignments. Version B would not be successful if there are more than three processors along the path from a free task to the critical processor. The following example demonstrates one case in which Version A works, but Version B fails.

Example 4-6 There are five processors and ten tasks, $S_1=\{J_1, J_2, J_3\}$, $S_2=\{J_3, J_4, J_5\}$, $S_3=\{J_5, J_6, J_7\}$, $S_4=\{J_7, J_8, J_9\}$, $S_5=\{J_9, J_{10}\}$, and $(w_1, r_1)=(3, 1)$, $(w_2, r_2)=(2, 1)$, $(w_3, r_3)=(2, 1)$, $(w_4, r_4)=(2, 1)$, $(w_5, r_5)=(1, 2)$. After implementing the algorithm presented in section 4.2, we have the following solution for [AL_1] problem in Figure 4-13.

P_1	J_1	J_2	1^*
P_2	J_3	J_4	
P_3	J_5	J_6	
P_4	J_7	J_8	
P_5	J_9	0	
Time	T_1	T_2	T_3

Figure 4-13 An example of a partial feasible schedule

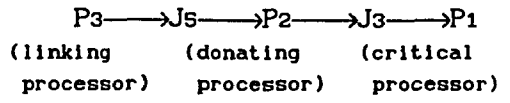
P_1 can still work, but no tasks are available for assignment, and there is a free task J_{10}. If we apply reassignment subroutine Version A, we can find a path, and the reassignment subroutine Version A can be implemented successfully here:

$J_{10} \longrightarrow P_5 \longrightarrow J_9 \longrightarrow P_4 \longrightarrow J_7 \longrightarrow P_3 \longrightarrow J_5 \longrightarrow P_2 \longrightarrow J_3 \longrightarrow P_1$

(free task) (donating processor) (critical processor)

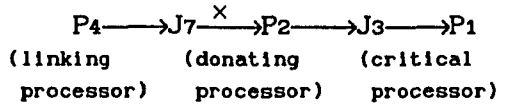
If we apply reassignment subroutine Version B, There are six combinations we have to take into account: (P_1, P_2, P_3), (P_1, P_2, P_4), (P_1, P_2, P_5), (P_1, P_3, P_4), (P_1, P_3, P_5), (P_1, P_4, P_5).

(1) For P_1, P_2, and P_3, there is a path:



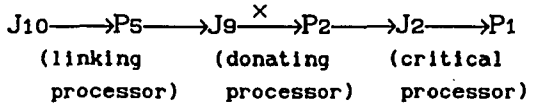
but no path connects J_{10} (free task) to P_3

(2) For P_1, P_2, and P_4, there is no such path:



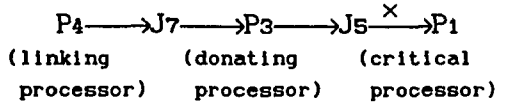
but no path connects J_{10} (free task) to P_4 or J_7 to P_2.

(3) For P_1, P_2, and P_5, there is no such path:



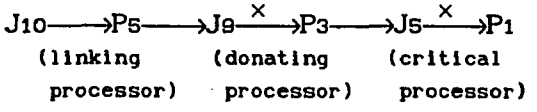
while no path connects J_9 to P_2.

(4) For P_1, P_3, and P_4, there is no such path:



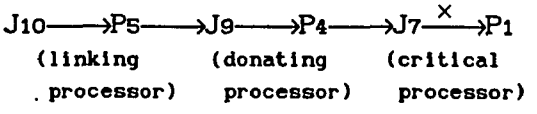
while no path connects the free task J_{10} to P_4 or J_5 to P_1.

(5) For P_1, P_3, and P_5, there is no such path:



while no path connects J_9 to P_3 and J_5 to P_1.

(6) For P_1, P_4, and P_5, there is no such path:



while no path connects J_7 to P_1.

All six combinations fail when using the reassignment subroutine Version B.

Q. E. D.

Although reassignment subroutine Version B can not guarantee to remedy all remediable conditions, the subroutine saves O(n) iterations, which also means that Version B can save some computation time. In pratice, the quality of solutions are comparable to that for Version A according to the experimental results (which are reported in Chapter 5).

The flowchart of reassignment subroutine version b is shown in Figure 4-14.

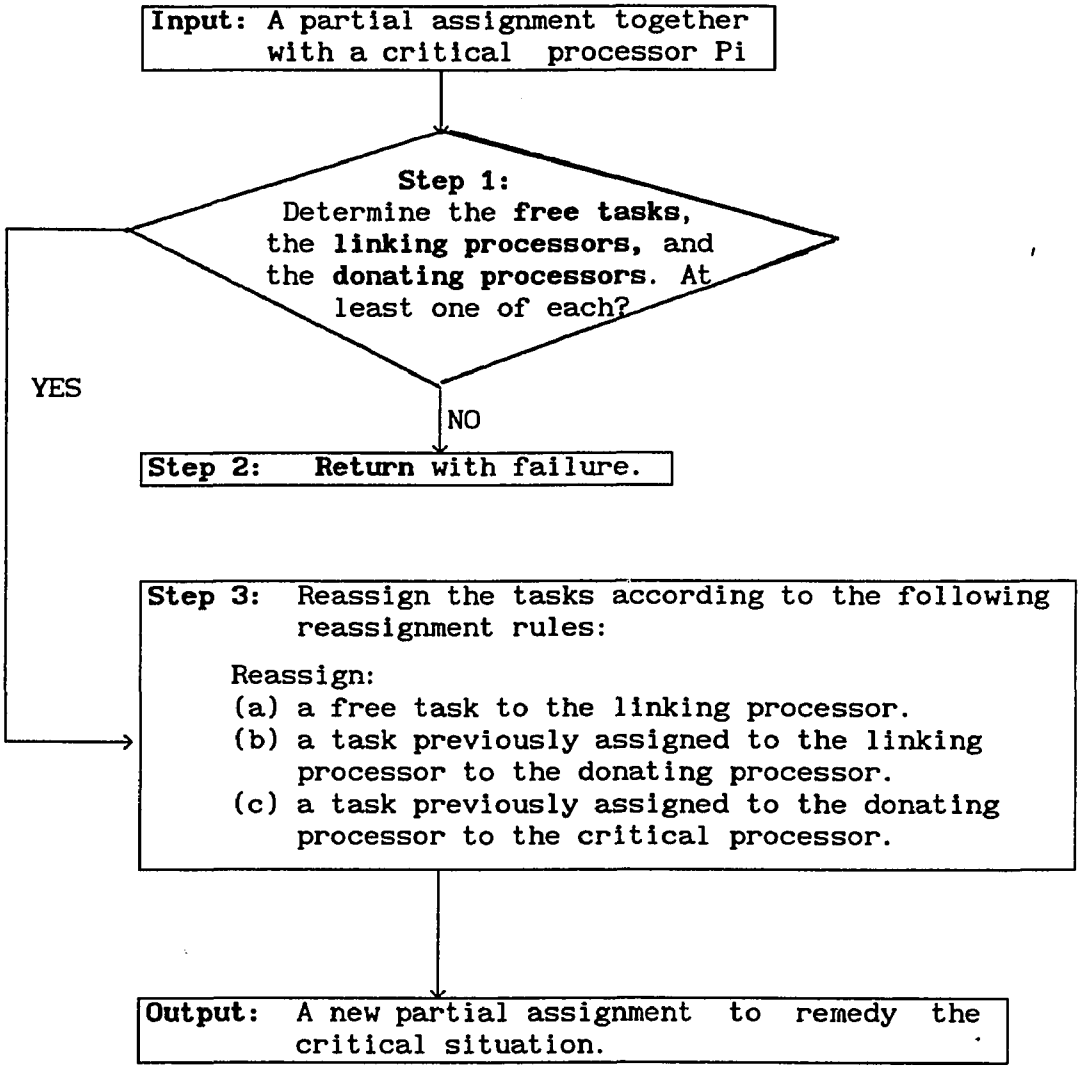


Figure 4-14 The Flowchart of Reassignment Subroutine Version B

4.4 [PHASE TWO]: Algorithm for the [AL_p] or [AM_p] Continuous Task Scheduling Problem

After we derive an optimal feasible schedule for the [AL_1] problem, the solution pattern can be used to solve the [AL_p] or [AM_p] problem. In this section we describe how this can be done.

We first discuss the underlying approach used to solve the [AL_p] problem. There are two ways to check for the at least p or at most p constraints. One way is to check the constraints while developing a legal solution. The second way is to check the constraints after obtaining a legal solution. The first approach is more complicated and time-consuming. Thus, we chose the second approach. The number of tasks to be processed during each time unit is counted to determine whether the constraints on the number of processors working are violated or not. If one or more constraints are violated, two adjustment procedures are available for modifying the [AL_1] solution so that it becomes feasible. These two adjustment procedures, Right-to-Left Adjustment Method or the Column-Checking Method, are described below. We note that the makespan obtained by relaxing the "at least p " or "at most p" constraints is a lower bound on the makespan possible when these constraints are considered.

4.4.1 The Right-to-Left Adjustment Method for the [ALp] problem [RTLAMALP]:

We denote the Right-to-Left adjustment method as **[RTLAMALP]**. This procedure considers the "at least p" constraint for each time period, beginning with the last time period and proceeding from right to left until the first time period is checked. Whenever the number of tasks assigned to a time period is less than p, tasks are shifted from earlier time periods to this time period to satisfy the constraint. This is done by first choosing the processor with the largest difference between work and rest time (W-R) and then moving the tasks assigned to that processor to later time periods, so as to increase by one the number of processors working in the period for which the ALp constraint is violated. If there are still fewer than p tasks assigned, we choose the processor with the next largest (W-R), and move that processor's task back to further increase the processors working in that period. This process is continued until all time periods have been checked and a feasible solution is found or until no further adjustments can be made, in which case the solution is infeasible.

4.4.2 The Column Checking Method for [ALp] problem [CCMALP]:

We denote the Column Checking method as **[CCMALP]**. This procedure provides an alternative method of checking for constraint satisfaction and making adjustments where needed. This method begins with the solution pattern for the [AL1] problem. Let

the total number of tasks assigned to each time period be C_k, $k=1,\ldots,M$. If $C_k-p>0$, then C_k-p tasks are in excess of the minimum needed, and can be reassigned to some other periods; if $C_k-p=0$, then this time period exactly meets the AL_p requirement. If $C_k-p < 0$, then $p-C_k$ tasks must be reassigned to this time period from one or more of the time periods that have excess tasks assigned. Feasibility, therefore, requires that the summation of C_k-p from 1 to makespan be greater than or equal to 0. This procedure is similar to a reassignment of tasks among the processors, only here it is a reassignment of tasks among time periods.

Two similar adjustment methods can be used to solve the [AM_p] problem as we will next show.

4.4.3 The Left-to-Right Adjustment Method for [AM_p] problem [LTRAMAMP]:

We denote this method as **[LTRAMAMP]**. This procedure is similar to **[RTLAMALP]**. Here, however we begin with time period one's constraint. If it is not violated, we check the next time period. If it is violated, we must move (C_k-p) tasks to the next time period. The criteria used to pick which tasks should be moved is to select the C_k-p processors with tasks assigned in that period that have the least total working and resting time. Moving tasks for these processors to the next time period will have the smallest impact on the solution pattern. Once the first period constraint is satisfied, we continue by checking and adjusting, when needed, for each subsequent period.

4.4.4 The Column Checking Method for [AM_p] problem [CCMAMP]:

We denote this method as **[CCMAMP]**. As with the [AL_p] problem we begin with the optimal schedule for the [AL_1] problem, and let the total number of tasks assigned to each time period be defined as C_k, $k=1,\ldots,M$. If $C_k>p$, then C_k-p tasks must be reassigned to other periods to meet the AM_p requirement; if $C_k=p$, this time period exactly satisfies the [AM_p] requirement; if $C_k<p$, although this time period also meets the [AM_p] requirement, C_k-p tasks can be reassigned to this time period from those periods that have too many tasks assigned.

Feasibility requires that the summation of C_k-p from 1 to makespan be less than or equal to 0. For purpose of this procedure, if a processor is available to work in some time period (based on working and resting conditions) but no task has been assigned, a "1" is placed in the task assignment matrix for that processor in that time period. We define $C_k-p=0$ if $C_k\leq p$ and there are no "1s" in this time period. If $C_k<p$, then $C_k-p \triangleq -\min$ {# of "1s" in this time period, $|C_k-p|$}. We can shift tasks from the "excess" time periods ($C_k-p>0$) to the "shortage" time periods ($C_k-p<0$). This procedure is similar to a reassignment of tasks among the processors, only here it is a reassignment of task among time periods. If the sum of the C_k-p values over the time periods ≤ 0, then a feasible solution must exist. Otherwise, no feasible solution exist for this makespan.

We now present the formal algorithm.

The Complete Algorithm for [AL_P] or [AM_P] Problem:

[PHASE ONE]: The Specialized Branch-and-Bound Algorithm for the [AL_1] Problem

Input: Unit-execution-time task set $J=\{J_1, J_2, \ldots\ldots, J_n\}$, processor set $P=\{P_1, P_2, \ldots\ldots, P_m\}$, and each processor P_i can execute a nonempty set $S_i \subset J$ of tasks, where $\bigcup_{i=1}^{m} S_i = J$. Associated with each processor P_i, $1 \le i \le m$, there is a pair of integers (w_i, r_i) corresponding to maximum continuous working time and minimum continuous resting time, where $w_i \ge 1$ and $r_i \ge 1$.

Output: A continuous task schedule with the minimum makespan for the [AL_1] problem, if possible.

Step 1: Put the start node s in OPEN.
(OPEN is a storage place for all generated and yet unexpanded nodes. The two children of the start node are $Z_{11}=1$ and $Z_{11}=0$).

Step 2: If OPEN is empty, exit with failure. No feasible continuous schedule can be found.

Step 3: Remove from OPEN and place in CLOSED a node n whose cost function value f(n) is the minimum among all the nodes in OPEN.

(If there is more than one node with the same minimum f value, the node corresponding to $Z_{ik}=1$ will be selected. If all open nodes with the same minimum f value are of $Z_{ik}=0$, then the latest generated node will be selected. CLOSED is a storage place for the expanded nodes).

Step 4: If n is a goal node, identify the solution obtained by tracing back the pointers from n to s, and exit successfully. Otherwise,

if n is $Z_{mk}=0$ and $\sum_{i=1}^{m} Z_{ik}=0$,

(the continuity constraint fails in the current time period) terminate the expansion of this node. Go to step 2.

Otherwise, go to step 5.

(For node $n \neq s$, n is either $Z_{ik}=1$ or $Z_{ik}=0$. If n is $Z_{ik}=1$ and there is only one task j left to be assigned and task j can be executed by processor i, then n is a goal node).

Step 5: Expand n. Let $n=Z_{j1}$, $1 \leq j \leq m$, then its two children are $Z_{ik}=1$ and $Z_{ik}=0$, where $i=j+1$, $k=1$ or $i=1$, $k=1+1$. The value of Z_{ik} can be determined as follows.

(1) If processor P_i must rest in time period k, Z_{ik} can only be assigned 0.

(2) If $Z_{ik}=1$ will lead to a simultaneous rest of all processors in the next time period, Z_{ik} can only be assigned 0.

(3) If Z_{ik} can be assigned 1, check if there is any task in S_i that has not been assigned yet. if the answer is affirmative, pick the lowest numbered task and assign it to P_i. If there is no such task, call the **REASSIGNMENT** subroutine. If the reassignment attempt fails, Z_{ik} can only be assigned 0.

(The purpose of the **REASSIGNMENT** subroutine is to find a feasible task assignment without changing the current partial legal sequence.

Step 6: For every successor n' of n:

(1) If n' is not already in OPEN or CLOSED, estimate h(n'), (an estimate of the cost of the best path from n' to some goal node), and calculate f(n')=g(n')+h(n') where g(n')=g(n)+1. Put n' into OPEN.

(2) If n' is already in OPEN or CLOSED, direct its pointers along the path yielding the lowest g(n').

(3) If n' required pointer adjustment and was found in CLOSED, put it back into OPEN.

Step 7: Go to Step 2

[PHASE TWO]: The Algorithm for [AL_p] or [AM_p] Problem

Input: A continuous task schedule with the minimum makespan for the [AL_1] problem.

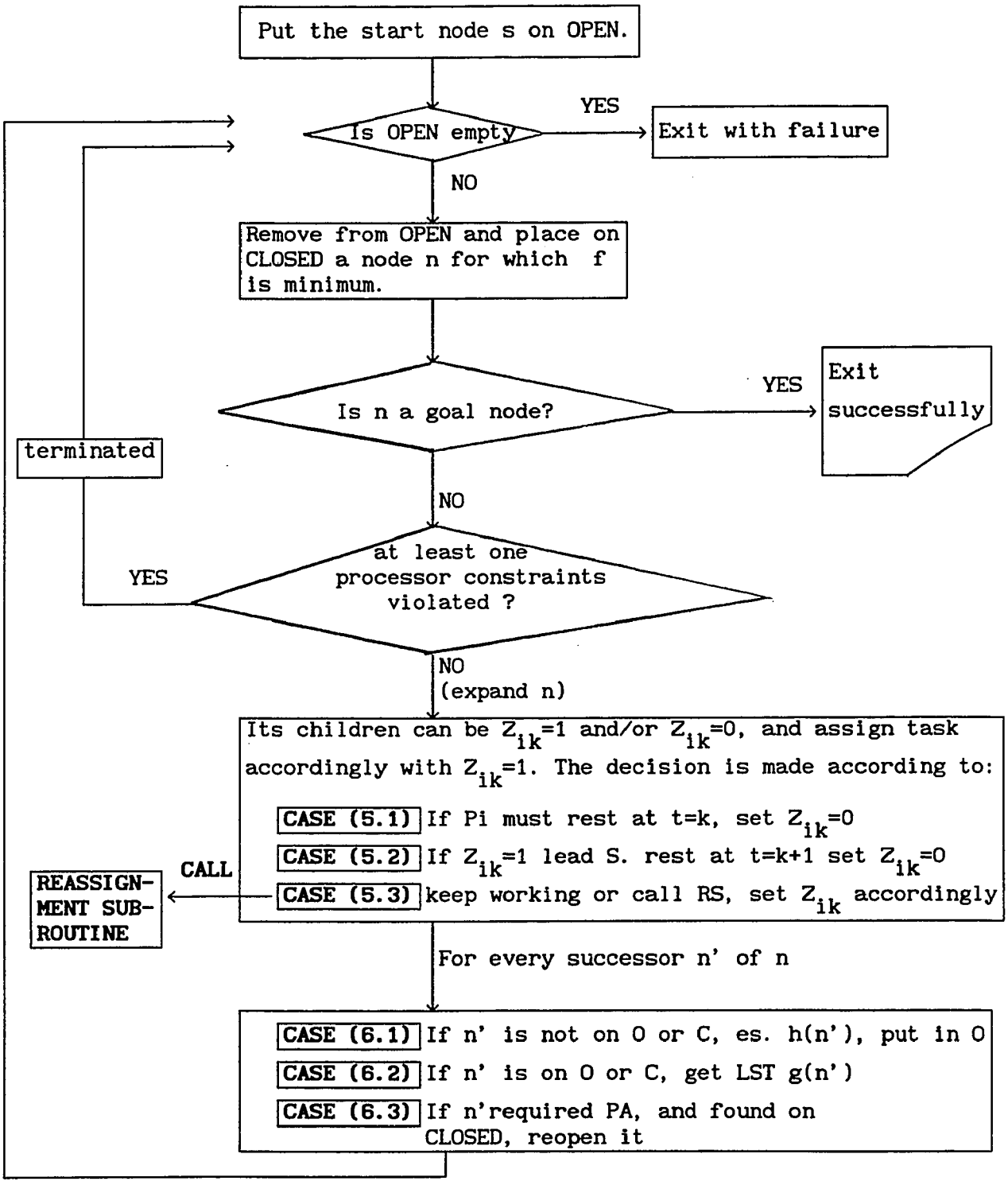
Output: A feasible continuous task schedule for the [AL_p] or [AM_p] problem, if possible.

Step 8: If the at least p processors working requirement is met for all time periods, stop, a feasible solution for the at least p processors working requirement is reached; Otherwise, use the Right-to-Left Adjustment Method or the Column-Checking Method to adjust the legal solution pattern. If there is still no feasible solution for the at least p processors working requirement, report an optimal schedule for the [AL_1] problem.

Step 9: If the at most p processors working requirement is met for all time periods, stop, a feasible solution for AM_p is reached; Otherwise, use the Left-to-Right Adjustment Method or the Column-Checking Method to adjust the legal solution pattern. If there is still no feasible solution for [AM_p] problem, report an optimal feasible schedule for for [AL_1] problem.

The flowchart of the complete algorithm is shown in Figure 4-15.

[PHASE ONE]


Put the start node s on OPEN.
Is OPEN empty
YES
Exit with failure
NO
Remove from OPEN and place on CLOSED a node n for which f is minimum.
Is n a goal node?
YES
Exit successfully
terminated
NO
at least one processor constraints violated ?
YES
NO
(expand n)
Its children can be Z_{ik}=1 and/or Z_{ik}=0, and assign task accordingly with Z_{ik}=1. The decision is made according to:
CASE (5.1) If Pi must rest at t=k, set Z_{ik}=0
CASE (5.2) If Z_{ik}=1 lead S. rest at t=k+1 set Z_{ik}=0
CASE (5.3) keep working or call RS, set Z_{ik} accordingly
CALL
REASSIGN-MENT SUB-ROUTINE
For every successor n' of n
CASE (6.1) If n' is not on O or C, es. h(n'), put in O
CASE (6.2) If n' is on O or C, get LST g(n')
CASE (6.3) If n'required PA, and found on CLOSED, reopen it

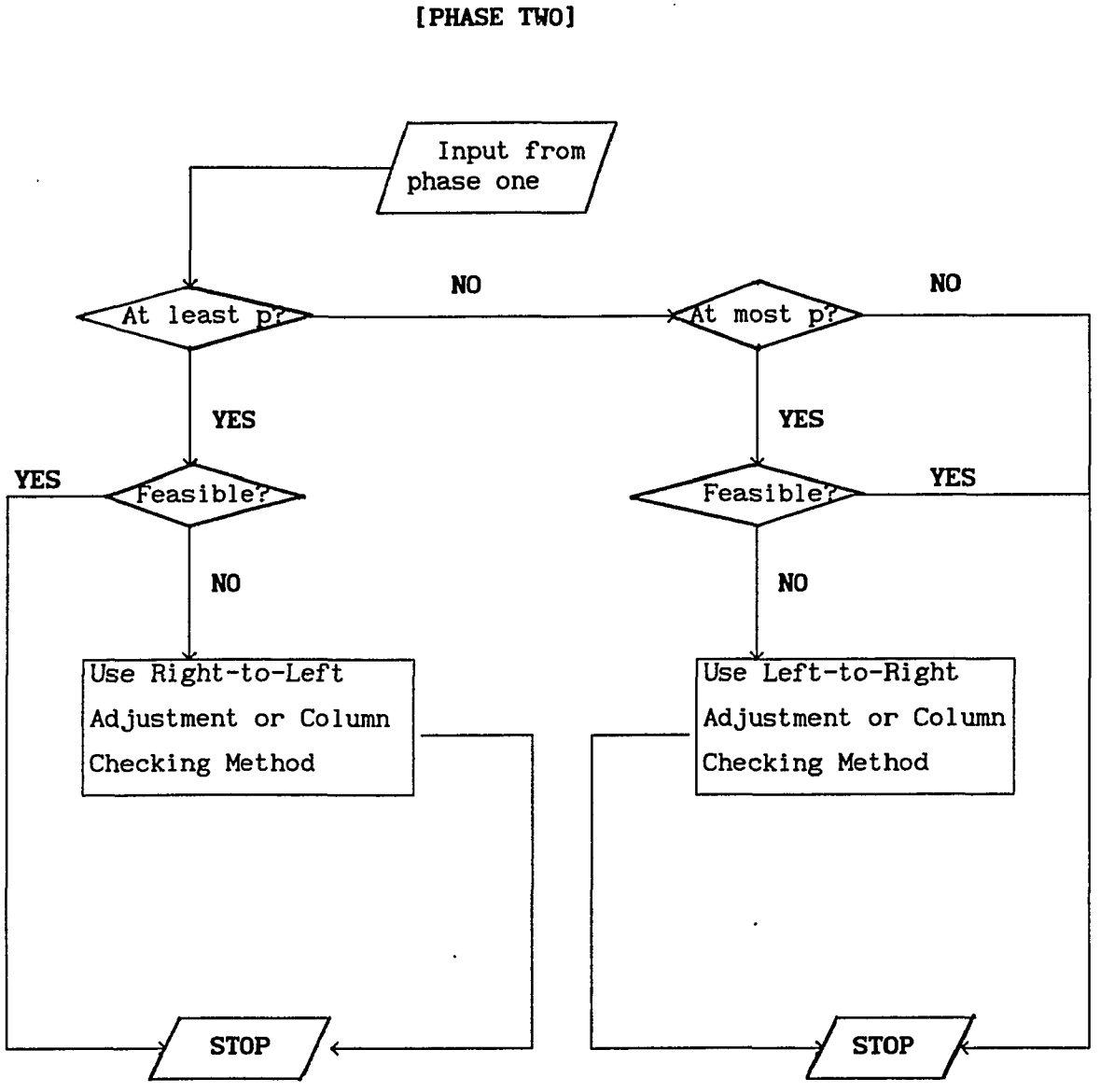


Figure 4-15 Flowchart of the Problem [AL_p] or [AM_p] Algorithm

There are two phases in the algorithm, the first phase is to find an optimal [AL_1] solution pattern and the second phase is to adjust the pattern according to the AL_p or AM_p requirement for each unit time period. In the second phase, the right-to-left adjustment method and the column checking method are used to adjust for the AL_p requirement, and the left-to-right adjustment method and the column-checking method to adjust for the AM_p requirement.

The algorithm can be demonstrated using Example 4-1. Let us show the algorithm step by step.

STEP 1: Initialize Problem Data:

GETTASK( ):
Total tasks: 10

GETPROCESSOR( ):
Total processors: 5

GETTASKSET( ):
S_1={J_1, J_2, J_4}
S_2={J_2, J_3, J_4}
S_3={J_3, J_4, J_5, J_6}
S_4={J_6, J_7, J_8}
S_5={J_8, J_9, J_{10}}

TIMETABLE( ):

	P_1	P_2	P_3	P_4	P_5
W	2	1	2	1	1
R	1	2	1	1	1

STEP 2: Create and Select Start Node.

STEP 3 and 4:

Step 3.1:

(1)

P_1	0	1	1	0
P_2	1	0	0	1
P_3	1	1	0	
P_4	1	0	1	
P_5	1	0	1	
	T_1	T_2	T_3	T_4

(1) P_1 starts from 0, cost=f=1+16=17

(2)

P_1	1	1	0	1
P_2	1	0	0	
P_3	1	1	0	
P_4	1	0	1	
P_5	1	0	1	
	T_1	T_2	T_3	T_4

(2) P_1 starts from 1, cost=f=1+15=16

(3)

P_1	J_1	1	0	1
P_2	1	0	0	
P_3	1	1	0	
P_4	1	0	1	
P_5	1	0	1	
	T_1	T_2	T_3	T_4

Choose (2), since (2) has the minimum cost, and assign J_1 to P_1 in (3).

Step 3.2:

(1)

P_1	J_1	1	0	1
P_2	0	1	0	
P_3	1	1	0	
P_4	1	0	1	
P_5	1	0	1	
	T_1	T_2	T_3	T_4

(1) P_2 starts from 0, cost=f=2+14=16

(2)

P_1	J_1	1	0	1
P_2	1	0	0	
P_3	1	1	0	
P_4	1	0	1	
P_5	1	0	1	
	T_1	T_2	T_3	T_4

(2) P_2 starts from 1, cost=f=2+14=16

(3)

P_1	J_1	1	0	1	
P_2	J_4	0	0		
P_3	1	1	0		
P_4	1	0	1		
P_5	1	0	1		
	T_1	T_2	T_3	T_4	

Choose (2). If f values are the same, the node corresponding to $Z_{ik}=1$ will be selected. Assign J_4 to P_2 in (3).

Step 3.3:

(1)

P_1	J_1	1	0	1	
P_2	J_4	0	0		
P_3	0	1	1		
P_4	1	0	1		
P_5	1	0	1		
	T_1	T_2	T_3	T_4	

(1) P_3 starts from 0, cost=f=3+13=16

(2)

P_1	J_1	1	0	1	
P_2	J_4	0	0		
P_3	1	1	1		
P_4	1	0	1		
P_5	1	0	1		
	T_1	T_2	T_3	T_4	

(2) P_3 starts from 1, cost=f=3+13=16

(3)

P_1	J_1	1	0	1	
P_2	J_4	0	0		
P_3	J_5	1	0		
P_4	1	0	1		
P_5	1	0	1		
	T_1	T_2	T_3	T_4	

Choose (2).
Assign J_5 to P_3 in (3).

Step 3.4:

(1)

P_1	J_1	1	0	1	
P_2	J_4	0	0	1	
P_3	J_5	1	0		
P_4	0	1	0		
P_5	1	0	1		
	T_1	T_2	T_3	T_4	

(1) P_4 starts from 0, cost=f=4+13=17

(2)

P_1	J_1	1	0	1	
P_2	J_4	0	0		
P_3	J_5	1	0		
P_4	1	1	1		
P_5	1	0	1		
	T_1	T_2	T_3	T_4	

(2) P_4 starts from 1, cost=f=4+12=16

(3)

P_1	J_1	1	0	1	
P_2	J_4	0	0		
P_3	J_5	1	0		
P_4	J_7	0	1		
P_5	1	0	1		
	T_1	T_2	T_3	T_4	

Choose (2), since (2) has the minimum cost, and assign J_7 to P_4 in (3).

Step 3.5:

(1)

P_1	J_1	1	0	1	
P_2	J_4	0	0	1	
P_3	J_5	1	0		
P_4	J_7	0	1		
P_5	0	1	0		
	T_1	T_2	T_3	T_4	

(1) P_5 starts from 0, cost=f=5+12=17

(2)

P_1	J_1	1	0	1	
P_2	J_4	0	0		
P_3	J_5	1	0		
P_4	J_7	0	1		
P_5	1	0	1		
	T_1	T_2	T_3	T_4	

(2) P_5 starts from 1, cost=f=5+11=16

(3)

P_1	J_1	1	0	1	
P_2	J_4	0	0		
P_3	J_5	1	0		
P_4	J_7	0	1		
P_5	J_9	0	1		
	T_1	T_2	T_3	T_4	

Choose (2), since (2) has the minimum cost, and assign J_9 to P_5 in (3).

Step 3.6:

(1)

P_1	J_1	0	1	1	
P_2	J_4	0	0		
P_3	J_5	1	0		
P_4	J_7	0	1		
P_5	J_9	0	1		
	T_1	T_2	T_3	T_4	

(1) P_1 starts from 0, cost=f=6+10=16

(2)

P_1	J_1	1	0	1
P_2	J_4	0	0	
P_3	J_5	1	0	
P_4	J_7	0	1	
P_5	J_9	0	1	
	T_1	T_2	T_3	T_4

(2) P_1 starts from 1, cost=f=6+10=16

(3)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	
P_3	J_5	1	0	
P_4	J_7	0	1	
P_5	J_9	0	1	
	T_1	T_2	T_3	T_4

Choose (2).
Assign J_2 to P_1 in (3).

Step 3.7: P_2 must rest, so must assign 0.

Step 3.8:

(1)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	
P_3	J_5	0	1	
P_4	J_7	0	1	
P_5	J_9	0	1	
	T_1	T_2	T_3	T_4

(1) P_3 starts from 0, cost=f=8+8=16

(2)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	
P_3	J_5	1	0	
P_4	J_7	0	1	
P_5	J_9	0	1	
	T_1	T_2	T_3	T_4

(2) P_3 starts from 1, cost=f=8+8=16

(3)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	
P_3	J_5	J_3	0	
P_4	J_7	0	1	
P_5	J_9	0	1	
	T_1	T_2	T_3	T_4

Choose (2).
Assign J_3 to P_3 in (3).

Step 3.9: P_4 must rest, so must assign 0.

Step 3.10: P_5 must rest, so must assign 0.

Step 3.11: P_1 must rest, so must assign 0.

Step 3.12: P_2 must rest, so must assign 0.

Step 3.13: P_3 must rest, so must assign 0.

Step 3.14:

(1)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	1
P_3	J_5	J_3	0	
P_4	J_7	0	0	
P_5	J_9	0	1	
	T_1	T_2	T_3	T_4

(1) P_4 starts from 0, cost=f=14+3=17

(2)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	
P_3	J_5	J_3	0	
P_4	J_7	0	1	
P_5	J_9	0	1	
	T_1	T_2	T_3	T_4

(2) P_4 starts from 1, cost=f=14+2=16

(3)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	
P_3	J_5	J_3	0	
P_4	J_7	0	J_8	
P_5	J_9	0	1	
	T_1	T_2	T_3	T_4

Choose (2), since (2) has the minimum cost, and assign J_8 to P_4 in (3).

Step 3.15:

(1)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	1
P_3	J_5	J_3	0	
P_4	J_7	0	J_8	
P_5	J_9	0	0	
	T_1	T_2	T_3	T_4

(1) P_5 starts from 0, cost=f=15+2=17

(2)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	
P_3	J_5	J_3	0	
P_4	J_7	0	J_8	
P_5	J_9	0	1	
	T_1	T_2	T_3	T_4

(2) P_5 starts from 1, cost=f=15+1=16

(3)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	
P_3	J_5	J_3	0	
P_4	J_7	0	J_8	
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

Choose (2), since (2) has the minimum cost, and assign J_{10} to P_5 in (3).

STEP 4, 5 and 6

(1)

P_1	J_1	J_2	0	0
P_2	J_4	0	0	1
P_3	J_5	J_3	0	
P_4	J_7	0	J_8	
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

(1) P_1 starts from 0, cost=f=16+1=17

(2)

P_1	J_1	J_2	0	1
P_2	J_4	0	0	
P_3	J_5	J_3	0	
P_4	J_7	0	J_8	
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

(2) P_1 starts from 1, cost=f=16

(3)

P_1	J_1*	J_2	0	J_4*
P_2	J_3	0*	0	
P_3	J_5	J_6	0	
P_4	J_7	0	J_8	
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

Choose (2), and call the reassignment subroutine. There is a free task J_6 for P_3. Reassign J_3 to P_2 and then J_4 to P_1. All tasks are finished and the goal node is reached.

[PHASE TWO]:

STEP 8: Assume we need at least 2 processors working in every time period.

1. Right-to-Left Adjustment Method for [ALp] problem [RTLAMLP].

The right-to-Left adjustment method checks for the [AL2] requirement for every time period beginning with the last time period. T_4 needs one task to fulfill the [AL2] requirement. Both P_1 and P_3 have the same largest (W-R), but P_1 is already executing a task in period T_4. Therefore, we choose P_3 and move its tasks back two time periods.

Step 8.1.1

(1)

					W-R	
P_1	J_1	J_2	0	J_4	1	
P_2	J_3	0	0		-1	
P_3	J_5*	J_6*	0		1	Select P_3 to shift
P_4	J_7	0	J_8		0	
P_5	J_9	0	J_{10}		0	
	T_1	T_2	T_3	T_4		

(2)

P_1	J_1	J_2	0	J_4
P_2	J_3	0	0	
P_3	0	0	J_5*	J_6*
P_4	J_7	0	J_8	
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

This satisfies the [ALp] requirement for T_4.

Step 8.1.2

					W-R	
P_1	J_1	J_2	0	J_4	1	
P_2	J_3	0	0		-1	
P_3	0	0	J_5*	J_6*	1	
P_4	J_7	0	J_8		0	Select P_4 to shift
P_5	J_9	0	J_{10}		0	or P_5 to shift
	T_1	T_2	T_3	T_4		

T_3 meets the [AL_P] requirement. We continue checking T_2 and find T_2 needs one task. Either processor P_4 or P_5 can be selected on the basis of the W-R value and J_7 or J_9 can be shifted from T_1 to T_2. This satisfies the requirement for T_2 without violating the [AL_P] requirement in T_3. We continue by checking T_1 where the [AL_P] requirement is already met and a feasible solution has been found.

P_1	J_1	J_2	0	J_4
P_2	J_3	0	0	
P_3	0	0	J_5	J_6
P_4	0	J_7	0	J_8
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

2. Column-Checking Method for [AL_P] problem [CCMALP].

Step 8.2.1

(1)

P_1	J_1	J_2	0	J_4
P_2	J_3	0	0	
P_3	J_5	J_6	0	
P_4	J_7	0	J_8	
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

(2)

P1	$J1_*$	J2	0	J4
P2	$J3_*$	0	0	1
P3	J5	J6	0	1
P4	J7	0	J8	
P5	J9	0	J10	
	T1	T2	T3	T4
Ck	5	2	2	1
Ck-2	+3	0	0	-1

The "1s" in the tasks-to-processors table indicate those processors which can work in additional time periods based on working/resting constraints but do not have tasks assigned to them. We can shift tasks from the "excess" time periods (periods with $C_k-p>0$) to the "shortage" time periods (periods with $C_k-p<0$). This procedure is similar to a reassignment of tasks among the processors, only here it is a reassignment of tasks among time periods. In this example, the summation of C_k-p values over the four time periods is >0, indicating a feasible solution may exist.

Note that period T_4 needs one additional task, Period T_1 has 3 excess tasks, and that P_2 and P_3 have tasks assigned to T_1 and are available for work in T_4. Therefore we could reassign either task J_3 or task J_5 from period T_1 to period T_4. Ths solution below shows the result of reassigning task J_3.

(3)

P_1	J_1	J_2	0	J_4 *
P_2	0*	0	0	J_3
P_3	J_5	J_6	0	
P_4	J_7	0	J_8	
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

The solution is now feasible.

STEP 9: Assume that we desire at-most-3 processors working during any given time period.

1. **Left-to-Right Adjustment Method for [AM_p] problem [LTRAMAMP].**

Step 9.1.1

(1)

					W+R	
P_1	J_1*	J_2	0	J_4	4	
P_2	J_3*	0	0		1	Select P_2 to shift
P_3	J_5	J_6	0		2	Select P_3 to shift
P_4	J_7	0	J_8		3	
P_5	J_9	0	J_{10}		3	
	T_1	T_2	T_3	T_4		

We begin with time period T_1, which has two more tasks assigned than are allowed by the at-most-p constraint. We select the processors with the least total (W+R), since they will have the least impact on the solution pattern when shifting the tasks. T_1 needs 2 processors to rest. We select the two processors (P_2 and P_3) with the least total (W+R) and shift their tasks to the right by one time period. The at-most-3 constraint is now satisfied for T_1.

(2)

P_1	J_1	J_2*	0	J_4*	
P_2	0	J_3*	0*		
P_3	0	J_5	J_6		
P_4	J_7	0	J_8		
P_5	J_9	0	J_{10}		
	T_1	T_2	T_3	T_4	

Step 8.1.2: T_2 does not violate the at-most-3 requirement.

Step 8.1.3: T_3 does not violate the at-most-3 requirement.

Step 8.1.4: T_4 does not violate the at-most-3 requirement.

The solution is now feasible.

2. Column-Checking Method for [AM_p] problem [CCMAMP].

Step 9.2.1

(1) The [AL_1] solution pattern was:

P_1	J_{1*}	J_2	0	J_4
P_2	J_{3*}	0	0	
P_3	J_5	J_6	0	
P_4	J_7	0	J_8	
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

(2)

For column-checking purposes, both P_2 and P_3 are available for work in T_4, as indicated by the "1s" in the table below. Processors P_4 and P_5 must rest in period T_4, as indicated by the "0s".

P1	$J1_*$	J2	0	J4
P2	$J3_*$	0	0	1
P3	J5	J6	0	1
P4	J7	0	J8	0
P5	J9	0	J10	0
	T1	T2	T3	T4
Ck	5	2	2	1
Ck-3	+2	0	0	-2

We define $C_k-p=0$ if $C_k \leq p$ and there are no "1s" in this time period. If $C_k<p$, then $C_k-p \hat{=}$ -min {# of "1s" in this time period, $|C_k-p|$ }. We can shift tasks from the "excess" time periods ($C_k-p>0$) to the "shortage" time periods ($C_k-p<0$). This procedure is similar to a reassignment of tasks among the processors, only here it is a reassignment of tasks among time periods. If the sum of the C_k-p values over all the time periods is ≤ 0, then a

feasible solution may exist. Otherwise, no feasible solution exists for this makespan. Time period T_1 has two excess tasks and period T_4 can be assigned up to 2 additional tasks. Processors P_2 and P_3 can work in period T_4 and both have tasks assigned in T_1. Therefore, tasks J_3 and J_5 are reassigned from T_1 to T_4. This results in a feasible solution.

(3)

P_1	J_1	J_2	0	J_4 *
P_2	0	0	0	J_3 *
P_3	0	J_6	0	J_5
P_4	J_7	0	J_8	
P_5	J_9	0	J_{10}	
	T_1	T_2	T_3	T_4

In the following, we shall prove that the algorithm is correct.

Lemma 4-6 Algorithm will terminate in a finite number of steps.

Proof: We have n tasks and m processors. The continuous task scheduling requirement implemented in Step 4 guarantees that every path in the search tree (every legal sequence) is of maximum length mn. Since we expand the nodes from $Z_{11}=1$ first, until we reach the goal node, such expansions will need at most 2^{mn} number of nodes put into OPEN. After that we point back to the root reopen the $Z_{11}=0$ node, then expand it. This also requires at most 2^{mn} nodes put into OPEN. Therefore the total number of nodes put into OPEN (including those which are closed and later reopened) is at most $2^{m(n+1)}$. Since every iteration of the algorithm will remove one node from OPEN, the algorithm will terminate in a finite number of steps.

Q.E.D.

Let s denote the start node and r denote one of the goal nodes. An optimal path from s to r is represented by P^*_{s-r} and its cost is denoted by C^*. Also let $g^*(n)$ denote the cost of the best path from s to n and $h^*(n)$ denote the cost of the best path from n to a goal node.

Lemma 4-7 At any time before the Algorithm terminates, there exists an OPEN node n' on P^*_{s-r} with $f(n')\leq C^*$.

Proof: Assume that $P^*_{s-r}=s,n_1,n_2,.....,n',...,r$ and let n' be the node on P^*_{s-r} that is put into OPEN the most recently (there is at least one OPEN node on P^*_{s-r} because r is not closed until

termination). Since the path $s,n_1,n_2,\ldots\ldots,n'\ldots r$ is optimal, we have $f(n')=g^*(n')+h(n')\leq g^*(n')+h^*(n')=C^*$. The inequality is due to Lemma 4-1.

Q.E.D.

Lemma 4-8. Algorithm phase one will return an optimal solution if one exists.

Proof: Suppose Algorithm phase one terminates with a goal node t for which $f(t)=g(t)>C^*$. By Step 3 and 4, Algorithm phase one inspects nodes for compliance with the termination criterion only after it selects them for expansion. Hence, when t was chosen for expansion, it satisfies $f(t)\leq f(n)$, $\forall n \in$ OPEN, This means that, immediately prior to termination, all nodes on OPEN satisfy $f(n)>C^*$. This, however, contradicts Lemma 4-3 which guarantees the existence of at least one OPEN node n' with $f(n')\leq C^*$. Therefore the terminating t must have $g(t)=C^*$, which means that Algorithm phase one returns an optimal solution.

Q.E.D.

There is one important way where the efficiency of Algorithm phase one can be improved. Note that in Step 6, every generated node will be put into OPEN and in Step 3, a minimum cost generated node is then selected for expansion. Therefore, if a node is generated, it will first be put into OPEN and selected for expansion later even though this node is actually a goal node. In the following, we shall show that in some cases one of the

generated nodes can be expanded immediately without first putting it into OPEN.

Lemma 4-9 Assume that $Z_{ik}=1$ and $Z_{ik}=0$ are the two children of the current expanded node, n. if $Z_{ik}=1$ is possible (i.e., processor i can still work and there is an unassigned task which can be executed by P_i), then $Z_{ik}=1$ can be expanded in the next step; only $Z_{ik}=0$ need be put into OPEN.

Proof: When n is expanded, we have $f(n) \leq f(n')$, $\forall n' \in$ OPEN. From the definition of f and h, we have $f(Z_{ik}=1)= g(Z_{ik}=1)+ h(Z_{ik}=1)= (g(n)+1)+ (h(n)-1)= f(n)$. Also $g(Z_{ik}=1)=g(Z_{ik}=0)=g(n)+1$ and $h(Z_{ik}=1) \leq h(Z_{ik}=0)$. Therefore we can next select $Z_{ik}=1$ for expansion.

Q.E.D.

With the same proof as that of Lemma 4-5, we have:

Lemma 4-10 If Z_{ik} must be assigned 0 because processor P_i must rest at time k, then $Z_{ik}=0$ can be selected for expansion in the next step.

Finally, we note that Algorithm phase one is applied to the processor ordering of P_1, P_2, ..., P_m. That is, in each time unit, we first consider P_1, then P_2, then P_3, and so on. It is obvious that if the processor ordering is different, the application of the algorithm may produce different optimal legal sequences. The search efficiency of different legal sequences will be different, but the makespan will remain the same.

Definition 4-10 For the task set, let n be the total number of tasks, m be the total number of processors, and $\alpha=\frac{n}{m}$. We also define a parameter Q, which is the executable rate that tasks can be executed by processors. If Q=1, then all tasks can be executed by any processors; if Q_i=0.3, then 30% of the tasks can be worked by processor P_i. Q_i=0.3, for i=1,2,4,5 can be shown in Example 4-1, There are 3 tasks in S_1, S_2, S_4, and S_5 task sets, and the total number of tasks is 10, so Q_i=3/10=0.3 for S_1, S_2, S_4, and S_5, and Q_j=4/10=0.4, when j=3 for S_3. Note that different processor P_i may have different Q_i, for example, Q_1=0.2 for P_1, Q_2=0.3 for P_2, Q_3=0.4 for P_3, Q_4=0.5 for P_4, Q_5=0.6 for P_5.

Definition 4-11 A parameter OL is defined as the task set overlapping level among the processors. OL is dependent on Q. If (Q=0.3, OL=0.2), 30% of the tasks could be executed by P_1, P_2, P_4, and P_5 processors and among them approximately 20% of the task sets are overlapped. In Example 4-1, P_1 has Q_1=0.3, P_2 has Q_2=0.3, between S_1 and S_2 task sets, J_2 and J_4 are two overlapping tasks, so OL=2/10=0.2 for P_1 and P_2. The same as Definition 4-10, different processors P_i may have different OL_i.

For phase two of the algorithm, it is obvious that it will not affect the result of NP-completeness if every processor is capable of executing every task. From now on, unless stated otherwise, we assume that $S_i=J$ for $1\le i\le m$. If we can not find any feasible schedule under the above condition, it will not be possible for us to find a feasible schedule under the constraint that each processor P_i can only execute a nonempty subset S_i of tasks.

Lemma 4-11: If p>n, then the original [ALp] problem itself is infeasible.

Proof: If p>n, then there will be a shortage of p-n tasks at k=1, so that is impossible to construct a feasible schedule to fulfill the at least p requirement for the first time period. It is necessary that n>p to initiate and construct a feasible schedule.

Lemma 4-12: For the original [ALp] problem, a necessary condition to construct a feasible schedule is $n\ge M*p$; otherwise, it is infeasible.

Proof: Since it is necessary that at least p tasks are scheduled in each time period k=1,...,M, there must be at least M*p tasks to be scheduled to construct a feasible schedule, so $n\ge M*p$.

We next discuss the feasibility of $[AL_1]$ first, since the solution sets of the $[AL_2]$, $[AL_3]$,...,$[AL_m]$ are all subsets of that for $[AL_1]$. If we can not find any feasible soultion in $[AL_1]$,

then it is impossible to find any feasible schedule for [AL2], [AL3],...,[ALm].

Lemma 4-13. For the original [AL1] problem, if $w_k + r_k \leq \sum_{i=1}^{m} w_i$ for all k, then there always exists a feasible schedule.

Proof: For the original [AL1] problem, a feasible schedule can be constructed as follows. First, we assign w_1 tasks to be executed by processor P_1 in time unit 1 to w_1. Next we assign w_2 tasks to P_2 in time unit w_1+1 to w_1+w_2, w_3 tasks to P_3 in time unit w_1+w_2+1 to $w_1+w_2+w_3$, and so on until w_m tasks are assigned to P_m in time unit $\sum_{i=k}^{m-1} w_i+1$ to $\sum_{i=k}^{m} w_m$. Since $r_1 \leq \sum_{i=1}^{m} w_i - w_1$, P_1 is ready to execute w_1 tasks again. The whole assignment process can thus be repeated until all tasks are assigned.

Q.E.D.

Lemma 4-14 For the original [AMp] problem, if $w_k + r_k \leq \sum_{i=1}^{m} w_i$ for all k, then there always exists a feasible schedule for $p \geq 1$.

Proof: From Lemma 4-12 we know that there always exists a feasible schedule for [AL1] problem if $w_k + r_k \leq \sum_{i=1}^{m} w_i$, for all k. The solution for [AL1] is actually the same as for [AMp], when p=m (the total number of processors). Since the solution set of [AM1]$\subset$ [AM2]$\subset$...$\subset$[AMm], if we can find the solution for [AMp], then we can always find a feasible schedule for [AMm-1], [AMm-2],...,[AM1]. So for the [AMp] problem, if $w_k + r_k \leq \sum_{i=1}^{m} w_i$ for all k, then there always exists a feasible schedule for $p \geq 1$.

Q.E.D.

4.5 Chapter Summary

In Section 4.1, the basic ideas of the algorithm were presented. The algorithm seeks the shortest legal sequence that has a feasible schedule corresponding to it for the [AL_1] problem. The algorithm constructs such a shortest legal sequence step by step by determining whether Z_{ik} can be 0 or 1, where Z_{ik} is the binary variable indicating whether or not processor i executes some task at time unit k. We also discussed the idea for the problems [AL_p] and [AM_p].

In Sections 4.2, 4.3, and 4.4, a formal and complete algorithm was presented. Phase one for the [AL_1] problem and two versions of reassignment subroutines were given. We proved some results of the subroutines and showed that reassignment Version B is faster than Version A, but will not always find a feasible reassignment even if one exists. Then we described Phase two which finds solutions for the [AL_p] and [AM_p] problems. Detailed examples were provided and we proved that this algorithm would terminate in a finite number of steps and return an optimal solution if one exists for [AL_1] problem. In Section 4.4, based on the [AL_1] solution, we used **[RTLAMALP]** and **[CCMALP]** methods for the [AL_p] problem, and **[LTRAMAMP]** and **[CCMAMP]** methods for the [AM_p] problem and discussed the relationships between the [AL_1] solution and the [AL_p] and [AM_p] solutions.

CHAPTER FIVE

COMPUTATIONAL EXPERIMENTS AND RESULTS

5.1 Experimental Design

For the continuous task scheduling problem, our research question was: Can we construct a feasible schedule which assigns tasks to processors in such a way that at any time unit there exist at least or at most p processors working, where $1 \leq p \leq m$? Furthermore, if such a schedule does exist, we wish to complete all tasks in the shortest amonut of time. In the prior chapter we described an algorithm for solving this problem. In this chapter we discuss the computational tests performed and their results.

We tested the integer programming formulation and the heuristic algorithm for Model $[AL_p]$, where p=1,2,3,4,5 and for $[AM_p]$, where p=2,3,4,5 (since the makespan of p=1 for the $[AM_p]$ problem is always equal to the given number of tasks if there is a feasible schedule for the $[AL_1]$ problem), using a commercially available linear programming code, GAMS (General Algebraic Modeling Systems) on an IBM PC 386 (with a math coprocessor 387). Computation is truncated for excessive time or storage requirement using truncation criteria of maximum nodes=100,000, maximum iterations=1,000,000. These results are compared to those obtained from the heuristic using the same problem sets and the same computer equipment. The range of the index for k time periods in GAMS is derived from the heuristic solution (makespan solution plus three time units). We compared solution times and limits on

problem size for the IP approach with the heuristic. We also compared the efficiency of the heuristic with two reassignment subroutines (versions A and B) and studied the general behavior of the algorithm. We coded the heuristic in the TURBO C programming language, and the programs are available upon request. Table 5-1 summarizes the problem characteristics in the test sets.

Table 5-1 Problem characteristics in the experiment

Problem	p	Solver	Computation
[ALp]	1,2,3,4,5	GAMS	truncation criteria: maximum nodes=100,000 maximum iteration=1,000,000
[AMp]	2,3,4,5		
[ALp]	1,2,3,4,5	Heuristic	Coded in Turbo C programming
[AMp]	2,3,4,5		

To generate problems of differing difficulty and requirements, we used four parameters with varying levels. These parameters, to be defined in this section, represented the relative load, availability rate, executable rate, and the task set overlapping level.

Let n be the total number of tasks, m be the total number of processors, and $\alpha=\frac{n}{m}$, where α is the relative load. Let W be the maximum continuous working time, R be the minimum continuous resting time, and $\beta=\frac{W}{R}$, where β is the availability rate. For testing purposes, we used specific levels for $\alpha=\frac{n}{m}$=(low, medium, relatively high, high)=$(\frac{10}{5}, \frac{25}{5}, \frac{40}{5}, \frac{50}{5})$=(2, 5, 8, 10), and

$\beta=\frac{W}{R}=\{low,\ medium,\ high\}=(\frac{1}{1},\frac{3}{1},\frac{5}{1})=(1,\ 3,\ 5)$. We also defined a parameter Q, which is the executable rate measuring the proportion of tasks that can be processed by each processor. If Q=1, then all tasks can be executed by each of the processors; if Q=0.8, then the probability that a specific task can be worked on by a specific processor is 0.8. A parameter OL is defined as the task set overlapping level among the processors, OL is dependent on Q, if (Q=0.8, OL=0.75), 80% of the tasks could be executed by any given processor, and among the processors about 75% of the task sets were overlapped.

Five pairs of the levels of Q and OL were use for the computational experiments: (Q, OL)=(high, relative high, medium, relative low, low)=(1, 1), (.6, .5), (.6, .3), (.3, .4), (.3, .2).

Thus, these parameters were used to determine n, m, w_i, r_i, the size of processor task set, and the degree of overlapping.

5.2 Experimental Results

We will use VA to refer to the the algorithm with reassignment subroutine version A and VB to refer to the algorithm with reassignment subroutine version B. CPU times for VA and VB were quite small and are reported in 10^{-4} seconds while CPU time for GAMS is in seconds. For Model [AL_p], we ran 5 (levels of p)*3 (levels of α)*3 (level of β)*5 (levels of Q and OL)=300 test cases; for Model [AL_p], we ran 4 (levels of p)*4 (levels of α)*3 (levels of β)*5 (levels of Q and OL)=240 test cases; so the total

cases we examined was 540. Each case was solved once by heuristic VA, once by heuristic VB, and finally by GAMS.

The levels of the controlled parameters are α={low **(L)**, medium **(M)**, relatively high **(RH)**, high **(H)**}, β={low **(L)**, medium **(M)**, high **(H)**}, (Q, OL)={low **(L)**, relative low **(RL)**, medium **(M)**,relative high **(RH)**, high **(H)**}. There are 4*3*5=60 {α, β, (Q, OL)} combinations in the experiment. "I" stands for infeasible solution makespan in the tables. Table 5-2a is the summary of 5 processors and 10 tasks for the MODEL [AL_p] tests; the accuracy of the heuristic is 100% of the exact model. Table 5-2b is the summary of 5 processors and 25 tasks for the MODEL [AL_p] tests; the accuracy of the heuristic is about 99% of the exact model. Table 5-2c is the summary of 5 processors and 40 tasks for the MODEL [AL_p] tests; the accuracy of the heuristic is about 97% of the exact model. Table 5-2d is the summary of 5 processors and 50 tasks for the MODEL [AL_p] tests; the accuracy of the heuristic is about 100% of the exact model. Detail tables are shown in appendix A.

Table 5-4a is the summary of 5 processors and 10 tasks for the MODEL [AM_p] tests; the accuracy of the heuristic is about 98% of the exact model. Table 5-4b is the summary of 5 processors and 25 tasks for the MODEL [AM_p] tests; the accuracy of the heuristic is 100% of the exact model. Table 5-4c is the summary of 5 processors and 40 tasks for the MODEL [AM_p] tests; the accuracy of the heuristic is 100% of the exact model. Table 5-4d is the summary of 5 processors and 50 tasks for the MODEL [AM_p] tests;

the accuracy of the heuristic is 100% of the exact model. Detail tables are shown in appendix B.

Table 5-3 shows the summary for MODELs [AL_p]; the accuracy of the heuristic is about 99% of the exact model. Table 5-5 shows the summary for MODELs [AM_p]; the accuracy of the heuristic is about 99.5% of the exact model. Figures 5-1, 5-2, 5-3, 5-4 show the relationships between the average makespan and p for MODEL [AL_p]. Figures 5-5, 5-6, 5-7, 5-8 show the relationships between the average makespan and p for MODEL [AM_p].

5.3 Discussion of Experimental Results

As the tables show, we have demonstrated the effectiveness of the heuristic. For Model [AL_p], the accuracy is 99% of the exact model; for Model [AM_p], the accuracy is 99.5% of the exact model.

The CPU time required by the heuristics is about 10^{-4} that of GAMS. The maximum problem size that can be run by GAMS is approximately 5 processors and 50 tasks, while problems of a size of 10 processors and 100 tasks can easily be handled by the heuristic.

For the heuristic, most of the CPU time is used in the reassignment subroutine; The reassignment time for version A is about 90% of the total execution time and version B is about 80%. This can be easily seen from appendix A and B. The CPU time of version B is about 95% of that of version A. For the problems examined, the results obtained with version B were never worse than those for version A. Since little time is saved with version

B, Version A is suggested to use for the entire heuristic algorithm. Even for the test cases in which no reassignment was carried out, computation time was still needed to examine possible reassignments.

As would be expected, problems based on the medium level of Q and OL required more task reassignments than the other levels, and the more reassignments required, the more CPU time was needed.

From the summary tables (Tables 5-3 and 5-5), we see that CPU time for the heuristic did not increase when the problem size increased. The efficiency of the heuristic is due to the estimation function h, which makes the heuristic insensitive to the problem size with respect to the computation time. If h' never overestimates h, then our algorithm will find the shortest legal sequence. The amount of time it takes depends on the accuracy of h'. If h' is a good approximation to h, then our heuristic algorithm will execute fairly efficiently most of the time, even though there are individual situations in which h' is inaccurate and more search is required. We might want to ask how well our algorithm will perform on the average, possibly as a function of the particular h' we use. If we relax the constraint on h' so that it may overestimate h, then we may miss the shortest legal sequence. But notice that the difference that can exist between the legal sequence we find and the best legal sequence that we missed is h'-h. So if we can bound the error in h', we can bound the error of our solution. (See Harris (Harris 1974) for a more

complete discussion of this issue and the presentation of an algorithm for bandwidth search). For GAMS, when problem size increased from 5 processors and 25 tasks to 5 processors and 50 tasks, the CPU time was more than doubled.

For larger α (higher relative load), the CPU time for the heuristic did not increase much, but the solution makespan increased proportionally. This is because the number of tasks increases with α, and so the solution makespan increases to accomodate these additional tasks in the schedule. For larger β (higher processor availability), the total CPU time and solution makespan decreased. This is expected as the higher processor availability (more working time relative to resting time) increases the chance of finding a shorter feasible schedule. This, in turn, reduces computational effort.

There are six test problems whose results from VA and VB are different from GAMS. The reason is that we only guarantee the optimal solution for [AL_1] in the first phase, while the second phase in our algorithm is heuristic. Details are shown in Tables A-b.3, A-c.1, A-c.3, B-a.1.

As Tables A-a.3, A-a.4, A-a.5, A-b.2, A-b.3, A-b.4, A-b.5, A-c.2, A-c.3, A-c.4, A-c.5, A-d.3, A-d.4, A-d.5 show, there were 128 infeasible problems. The reasons that those problem instances resulted in infeasibility were: 1. There were not sufficient tasks (n) to fit the [AL_p] requirement, $n<p$ for the first time period or $n<M*p$ (the makespan times p), 2. The specific values for working

and resting time made it impossible to meet the continuity requirement; if $w_k + r_k > \sum_{i=1}^{m} w_i$ for all k, then there is no feasible solutions for the [AL1] problem, but we still need to find the relationship between w_i and r_i for the general [ALp] problem. For example, when p=2, can we find any rule for w_i and r_i which makes the problem infeasible or feasible? It is expected that these relationships will differ for every p, since a feasible solution for [AL1] may not be feasible for [AL2],...[ALm], etc.. In brief, we found the rule $w_k + r_k > \sum_{i=1}^{m} w_i$ for problem [AL1], but not for [AL2], [AL3],...,[ALm]. For the problem [AMp], if we can not find a feasible schedule for problem [AL1], then there is no feasible schedule for [AMp]; this has been shown before.

For both [ALp] and [AMp] problems, as Figures 5-1 through 5-8 show, when the number of processors and tasks is fixed, as p increases, the solution makespan will decrease. This is reasonable because the set relationship of the solution makespan of [ALp] and [AMp] problems are [ALp] $\subset$ [ALp-1] $\subset$...$\subset$ [AL1] and [AM1] $\subset$ [AM2] $\subset$...$\subset$ [AMp]. For [ALp] problems, the higher utilization rate of the processors in each time period will finish the tasks sooner if the number of the tasks are fixed. For [AMp] problems, the more processors standby (as p increased for fixed m), the more reliable the system, but the makespan will increase. Alternatively, if less processors standby, the system will be less reliable, but the makespan will decrease. Thus there is a trade-off between reliability and the makespan. As the number of

tasks increases, the solution makespan increases proportionally for fixed p for both [ALp] and [AMp] problems.

5.4 Chapter Summary

In Section 5.1, we outlined the experimental design for the test problems for [ALp] and for [AMp]. We generated 540 test cases, each case is solved three times using GAMS, VA and VB. The control parameters are: $\alpha=\frac{n}{m}$, where α is the relative load and $\beta=\frac{W}{R}$, where β is the availability rate, Q (executable rate) and OL (overlapping level).

In Section 5.2 and 5.3, the results show that for larger α, the CPU time for the heuristic did not increase much, but the solution makespan increased proportionally; For larger β (processor availability), the total CPU time and solution makespan decreased. The medium level of Q and OL resulted in more task reassignments than the other levels, and consequently more CPU time was needed. For Model [ALp], the accuracy of the heuristic is 99% of the exact model; for Model [AMp], the accuracy is 99.5% of the exact model. The CPU time required by the heuristics is about 10^{-4} that of GAMS. The maximum problem size that can be run by GAMS is approximately 5 processors and 50 tasks, while problems of a size of 10 processors and 100 tasks can easily be handled by the heuristic.

For the heuristic, most of the CPU time was used in the reassignment subroutine.

CPU time for the heuristic did not increase when the problem size increased. For GAMS, when problem size increased from 5 processors and 25 tasks to 5 processors and 50 tasks, the CPU time was more than doubled.

We only guarantee the optimal solution for [AL_1] in the first phase, while the second phase in our algorithm is heuristic.

Some problems were infeasible, due generally to the following: 1. There are not sufficient tasks to fit the [AL_p] requirement; 2. The arrangement of working and resting time restriction cannot meet the continuity requirement; 3. If we cannot find a feasible schedule for problem [AL_1], then we cannot find a feasible schedule for problem [AM_p].

When the number of processors and tasks is fixed, the solutiom makespan will decrease as p increases. The higher utilization rate of the processors in each time period will finish the tasks sooner if the number of the tasks are fixed. For [AM_p] problems, there is a trade-off between reliability and the makespan; increasing reliability results in a larger makespan. As the number of tasks increases the solution makespan increases proportionally for fixed p for both [AL_p] and [AM_p] problems.

Table 5-2a Summary of results for Model [ALp] with

5 processors and 10 tasks

PROCESSORS	TASKS	ALp	Average execution T *10^{-4}(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	10	1	13	12.8	8.6	0.86	0.86	100%	100%	0	2.8
5	10	2	13	12.8	8.8	1.06	1.2	100%	100%	0	3
5	10	3	13	13	10.2	0.86	1	100%	100%	0	2.2
5	10	4	13	13	10.1	1.06	0.9	100%	100%	0	2
5	10	5	13	13	10.1	1	0.9	100%	100%	0	2
Average			**13**	**12.9**	**9.56**	**0.97**	**0.97**	**100%**	**100%**	**0%**	**2.4**

Table 5-2b Summary of results for Model [ALp] with

5 processors and 25 tasks

PROCESSORS	TASKS	ALp	Average execution T *10^{-4}(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	25	1	13	12.6	9.5	1	0.8	100%	100%	0	7.3
5	25	2	12	12	10.2	0.8	0.8	100%	100%	0	7
5	25	3	14	13.8	10.2	0.8	0.8	93%	93%	7%	5.6
5	25	4	13	11.8	11.3	0.8	0.8	100%	100%	0	5.2
5	25	5	13	12.8	10.9	0.8	1.7	100%	100%	0	5
Average			**13**	**12.8**	**10.4**	**0.84**	**0.98**	**99%**	**99%**	**1.4%**	**6**

Table 5-2c Summary of results for Model [ALp] with

5 processors and 40 tasks

PROCESSORS	TASKS	ALp	Average execution T *10^{-4}(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	40	1	14	13.6	12.9	1.7	2.2	87%	87%	13%	12.2
5	40	2	13	12.8	12.6	2.2	2.3	100%	100%	0	10.5
5	40	3	13	12.8	13.3	2.3	2.2	100%	100%	0	9.5
5	40	4	12	11.8	14.6	2.2	2.2	100%	100%	0	I
5	40	5	13	12.7	13.3	2.2	0.3	100%	100&	0	I
Average			**13**	**12.7**	**13.3**	**2.12**	**1.84**	**97%**	**97%**	**2.6%**	**10.7**

Table 5-2d Summary of results for Model [ALp] with

5 processors and 50 tasks

PROCESSORS	TASKS	ALp	Average execution T *10^{-4}(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	50	1	14	13.7	12.6	0.3	0.3	100%	100%	0	15
5	50	2	13	12.8	13.6	0.3	0.3	100%	100%	0	15
5	50	3	14	13.8	19.7	0.3	0.3	100%	100%	0	12.5
5	50	4	12	11.8	22.2	0.3	0.3	100%	100%	0	I
5	50	5	13	12.8	22.8	0.3	0.3	100%	100%	0	I
Average			**13**	**13**	**18.2**	**0.3**	**0.3**	**100%**	**100%**	**0%**	**14**

Table 5-3 Summary for Model [ALp]

PROCESSORS	TASKS	ALp	Average execution T *10^{-4}(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	10	1	13	12.8	8.6	0.86	0.86	100%	100%	0	2.8
5	10	2	13	12.8	8.8	1.06	1.2	100%	100%	0	3
5	10	3	13	13	10.2	0.86	1	100%	100%	0	2.2
5	10	4	13	13	10.1	1.06	0.9	100%	100%	0	2
5	10	5	13	13	10.1	1	0.9	100%	100%	0	2
5	25	1	13	12.6	9.5	1	0.8	100%	100%	0	7.3
5	25	2	12	12	10.2	0.8	0.8	100%	100%	0	7
5	25	3	14	13.8	10.2	0.8	0.8	93%	93%	7%	5.6
5	25	4	13	11.8	11.3	0.8	0.8	100%	100%	0	5.2
5	25	5	13	12.8	10.9	0.8	1.7	100%	100%	0	5
5	40	1	14	13.6	12.9	1.7	2.2	87%	87%	13%	12.2
5	40	2	13	12.8	12.6	2.2	2.3	100%	100%	0	10.5
5	40	3	13	12.8	13.3	2.3	2.2	100%	100%	0	9.5
5	40	4	12	11.8	14.6	2.2	2.2	100%	100%	0	I
5	40	5	13	12.7	13.3	2.2	0.3	100%	100&	0	I
5	50	1	14	13.7	12.6	0.3	0.3	100%	100%	0	15
5	50	2	13	12.8	13.6	0.3	0.3	100%	100%	0	15
5	50	3	14	13.8	19.7	0.3	0.3	100%	100%	0	12.5
5	50	4	12	11.8	22.2	0.3	0.3	100%	100%	0	I
5	50	5	13	12.8	22.8	0.3	0.3	100%	100%	0	I
Grand Average			**13**	**12.8**	**13.1**	**1.1**	**1.1**	**99%**	**99%**	**1%**	**7.3**

Table 5-4a Summary of results for Model [AMp] with 5 processors and 10 tasks

PROCESSORS	TASKS	ALp	Average execution T *10^{-4}(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	10	2	13	12.6	16.3	0.8	0.8	93%	93%	7%	5
5	10	3	13	13	14.6	0.8	0.8	100%	100%	0	4
5	10	4	13	13	13.4	0.8	0.8	100%	100%	0	3.3
5	10	5	13	13	8.6	0.8	0.8	100%	100%	0	2.8
Average			**13**	**12.9**	**13.2**	**0.8**	**0.8**	**98%**	**98%**	**1.75%**	**15.1**

Table 5-4b Summary of results for Model [AMp] with 5 processors and 25 tasks

PROCESSORS	TASKS	ALp	Average execution T *10^{-4}(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	25	2	12	12	16.3	0.8	0.8	100%	100%	0	13
5	25	3	14	13.6	14.6	0.8	0.8	100%	100%	0	9.7
5	25	4	12	11.6	13.6	0.8	0.8	100%	100%	0	8.1
5	25	5	13	12.6	9.5	0.8	0.8	100%	100%	0	7.3
Average			**13**	**12.5**	**13.5**	**0.8**	**0.8**	**100%**	**100%**	**0%**	**9.5**

Table 5-4c Summary of results for Model [AMp] with

5 processors and 40 tasks

PROCESSORS	TASKS	ALp	Average execution T $*10^{-4}$(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	40	2	13	12.6	21.6	0.8	0.8	100%	100%	0	20
5	40	3	13	12.6	17.5	0.8	0.8	100%	100%	0	15
5	40	4	12	11.6	15.6	0.8	0.8	100%	100%	0	13.2
5	40	5	13	12.6	10.9	0.8	0.8	100%	100&	0	11.6
Average			**13**	**12.4**	**16.4**	**0.8**	**0.8**	**100%**	**100%**	**0%**	**14.9**

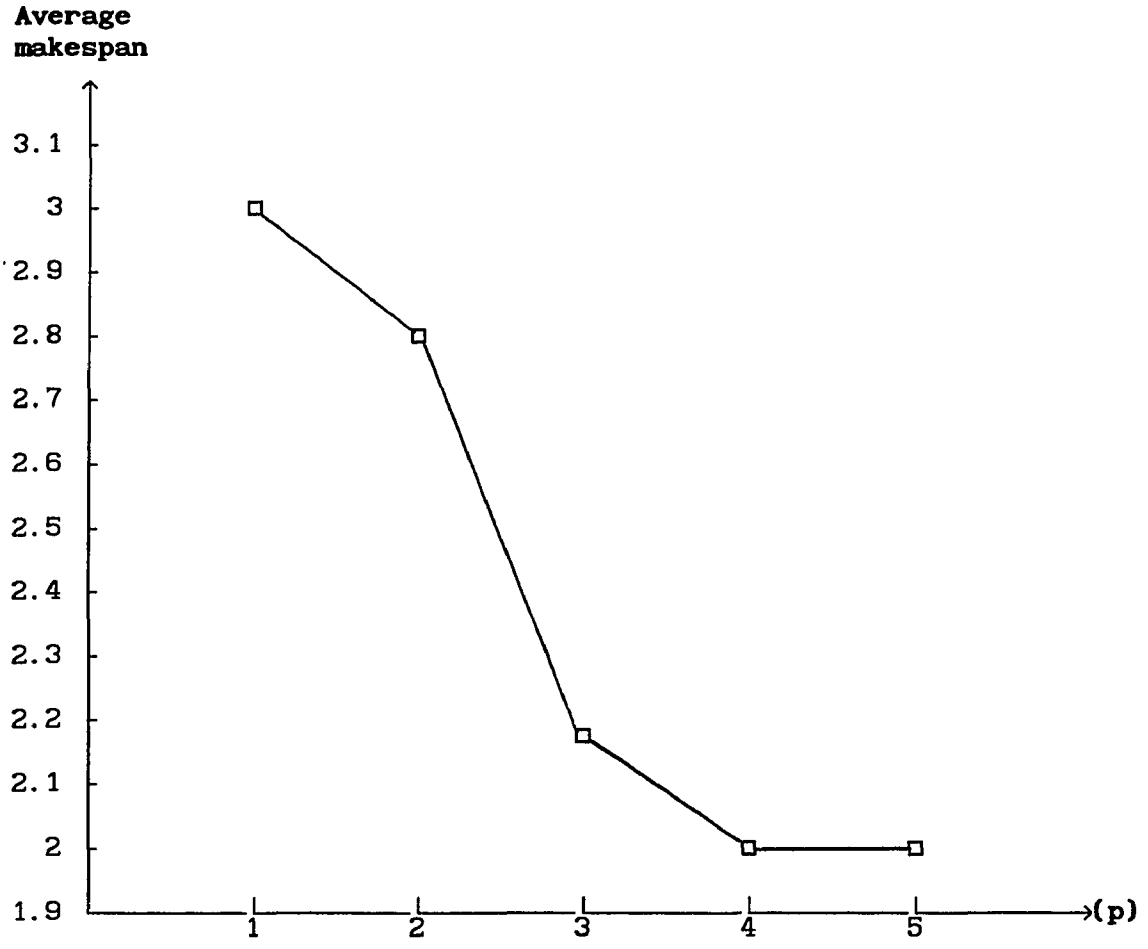
Table 5-4d Summary of results for Model [AMp] with

5 processors and 50 tasks

PROCESSORS	TASKS	ALp	Average execution T $*10^{-4}$(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	50	2	13	12.6	31.5	0.8	0.8	100%	100%	0	26
5	50	3	14	13.6	30.7	0.8	0.8	100%	100%	0	18.6
5	50	4	12	11.6	23.7	0.8	0.8	100%	100%	0	16.3
5	50	5	13	12.6	14	0.8	0.8	100%	100%	0	15
Average			**13**	**12.6**	**25**	**0.8**	**0.8**	**100%**	**100%**	**0%**	**19**

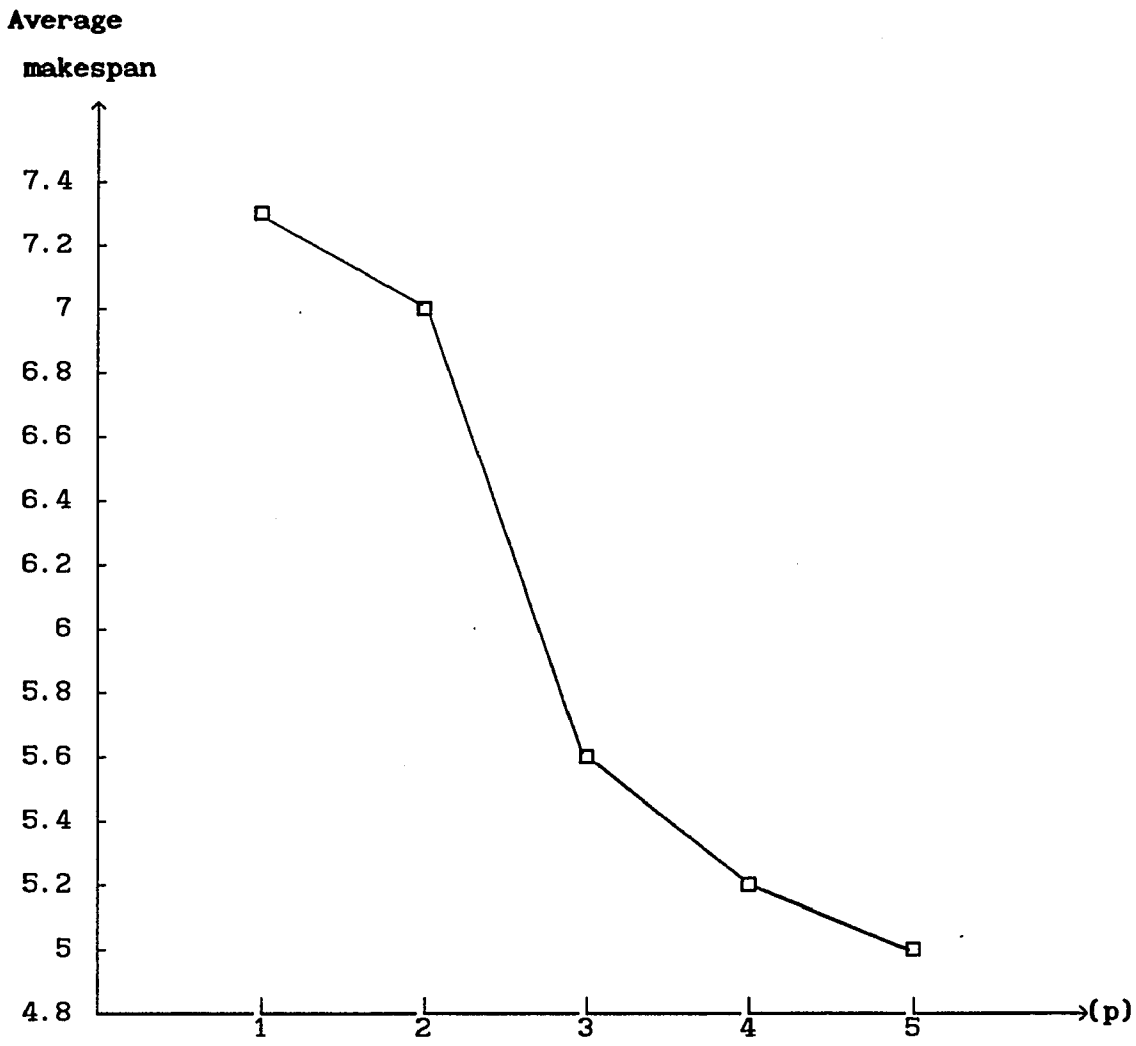
Table 5-5 Summary for Model [AMp]

PROCESSORS	TASKS	ALp	Average execution T *10^{-4}(s)(sec)			Number of successful reassignment		accuracy proportion		Average	
		p	VA	VB	GAMS	VA	VB	VA / GAMS	VB / GAMS	error	make-span
5	10	2	13	12.6	16.3	0.8	0.8	93%	93%	7%	5
5	10	3	13	13	14.6	0.8	0.8	100%	100%	0	4
5	10	4	13	13	13.4	0.8	0.8	100%	100%	0	3.3
5	10	5	13	13	8.6	0.8	0.8	100%	100%	0	2.8
5	25	2	12	12	16.3	0.8	0.8	100%	100%	0	13
5	25	3	14	13.6	14.6	0.8	0.8	100%	100%	0	9.7
5	25	4	12	11.6	13.6	0.8	0.8	100%	100%	0	8.1
5	25	5	13	12.6	9.5	0.8	0.8	100%	100%	0	7.3
5	40	2	13	12.6	21.6	0.8	0.8	100%	100%	0	20
5	40	3	13	12.6	17.5	0.8	0.8	100%	100%	0	15
5	40	4	12	11.6	15.6	0.8	0.8	100%	100%	0	13.2
5	40	5	13	12.6	10.9	0.8	0.8	100%	100&	0	11.6
5	50	2	13	12.6	31.5	0.8	0.8	100%	100%	0	26
5	50	3	14	13.6	30.7	0.8	0.8	100%	100%	0	18.6
5	50	4	12	11.6	23.7	0.8	0.8	100%	100%	0	16.3
5	50	5	13	12.6	14	0.8	0.8	100%	100%	0	15
Grand Average			**13**	**12.6**	**17**	**0.8**	**0.8**	**99.5%**	**99.5%**	**0.5%**	**11.8**



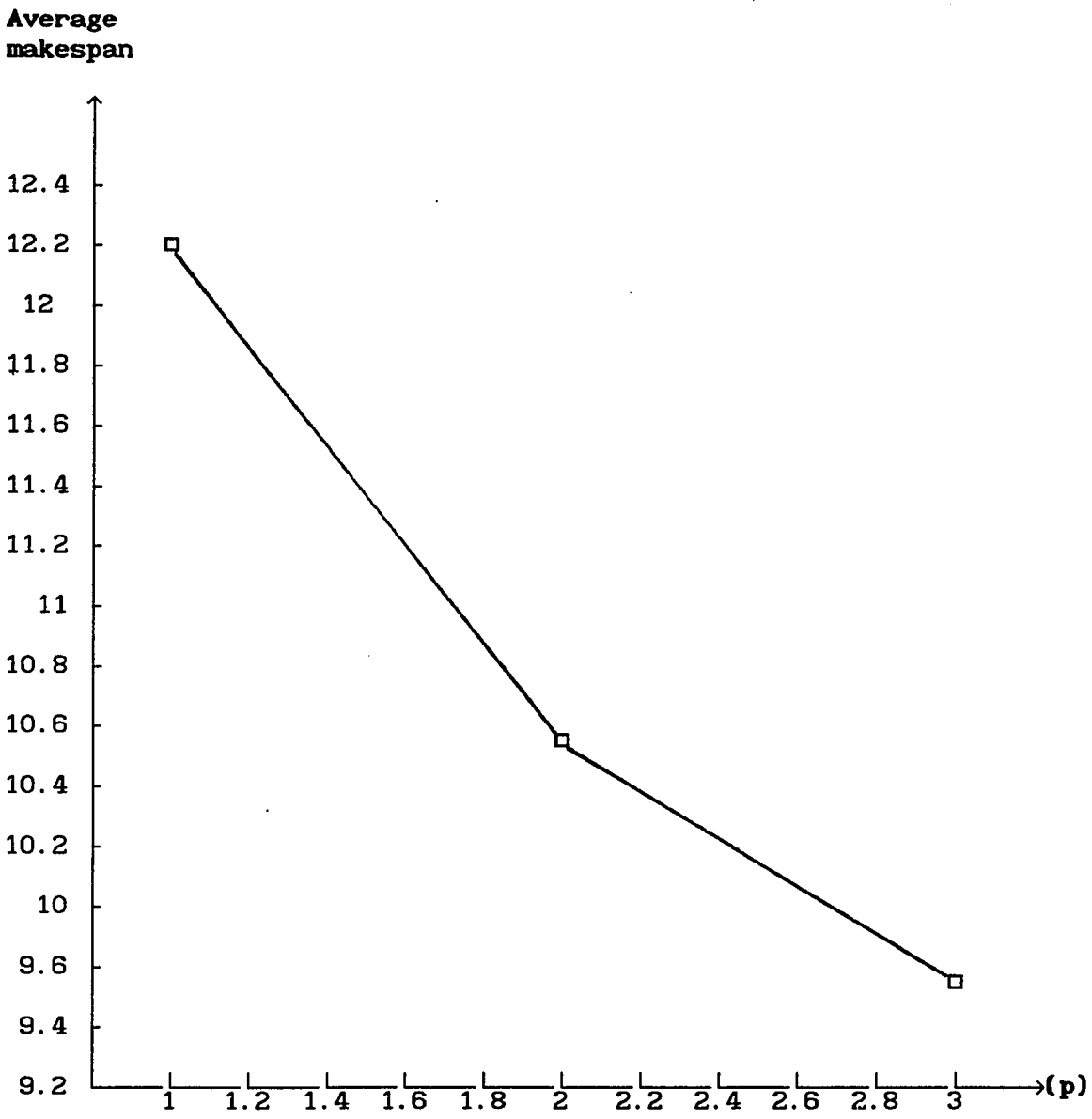
p=1,2,3,4,5 of 5 processors and 10 tasks for Models [ALp]

Figure 5-1 The relationship between average makespan and p for Model [ALp] with 5 processors and 10 tasks



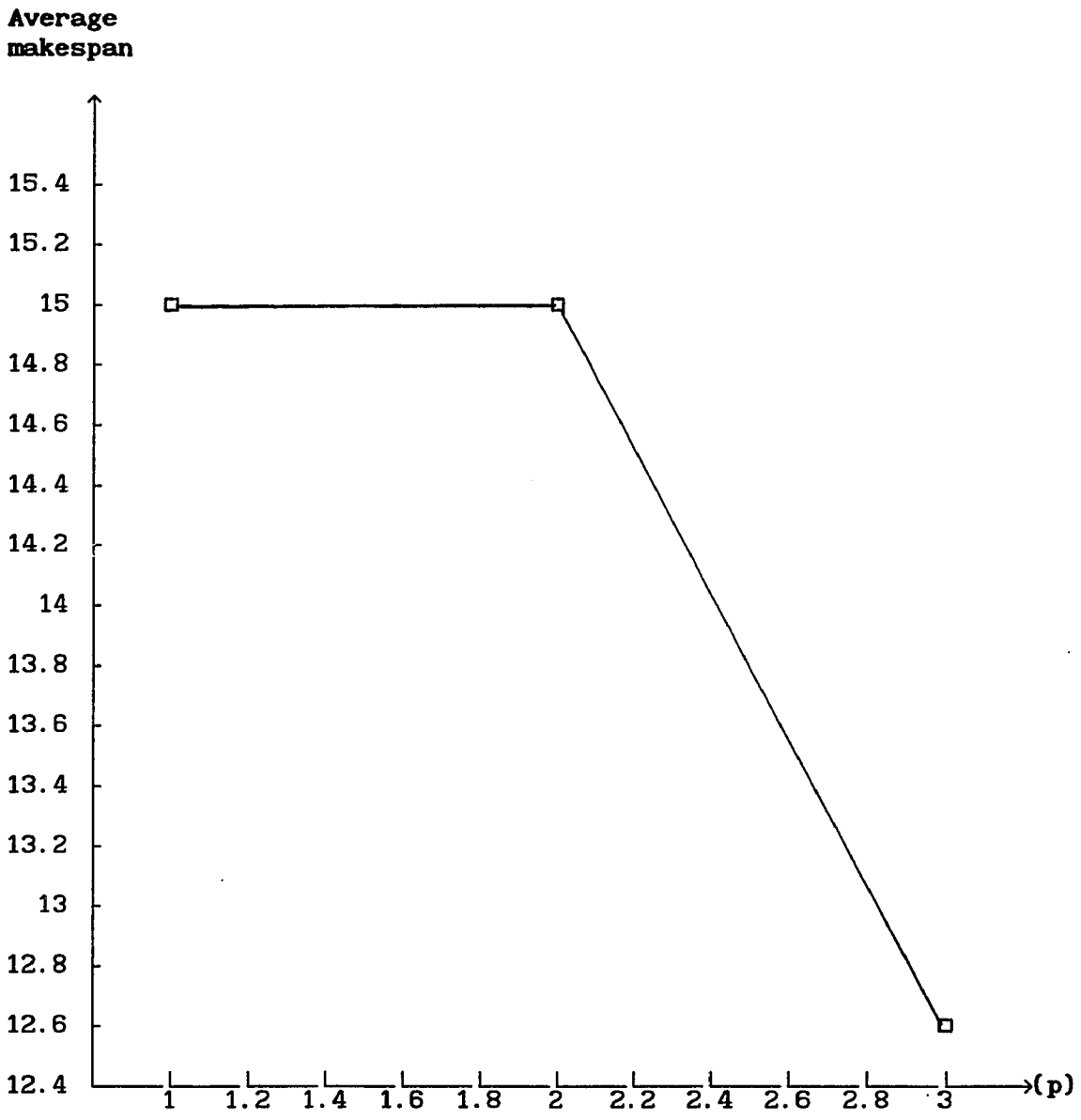
p=1,2,3,4,5 of 5 processors and 25 tasks for Models [ALp]

Figure 5-2 The relationship between average makespan and p for Model [ALp] with 5 processors and 25 tasks



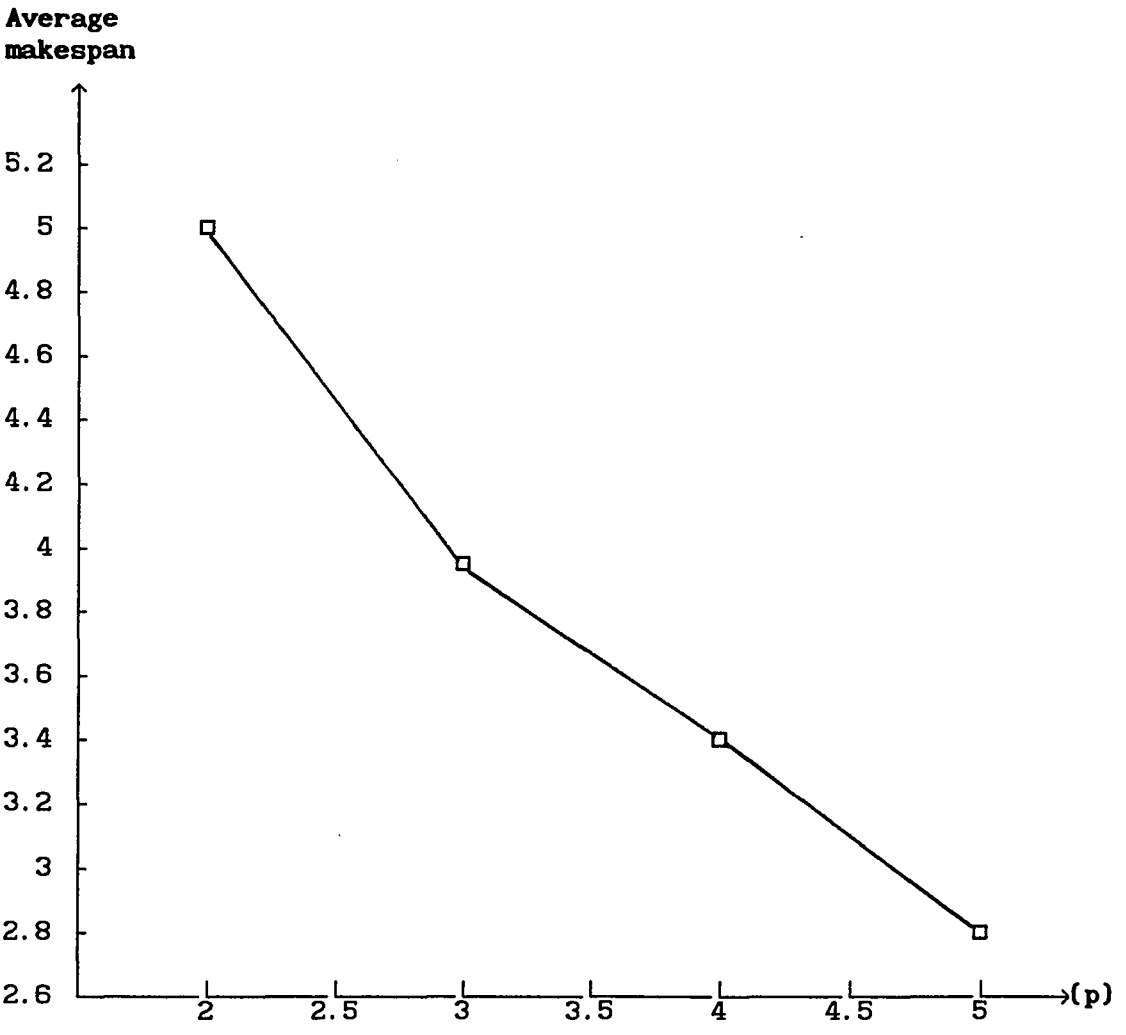
p=1,2,3 of 5 processors and 40 tasks for Models [ALp]

Figure 5-3 The relationship between average makespan and p for Model [ALp] with 5 processors and 40 tasks



p=1,2,3 of 5 processors and 50 tasks for Models [ALp]

Figure 5-4 The relationship between average makespan and p for Model [ALp] with 5 processors and 50 tasks



p=2,3,4,5 of 5 processors and 10 tasks for Models [AMp]

Figure 5-5 The relationship between average makespan and p for Model [AMp] with 5 processors and 10 tasks

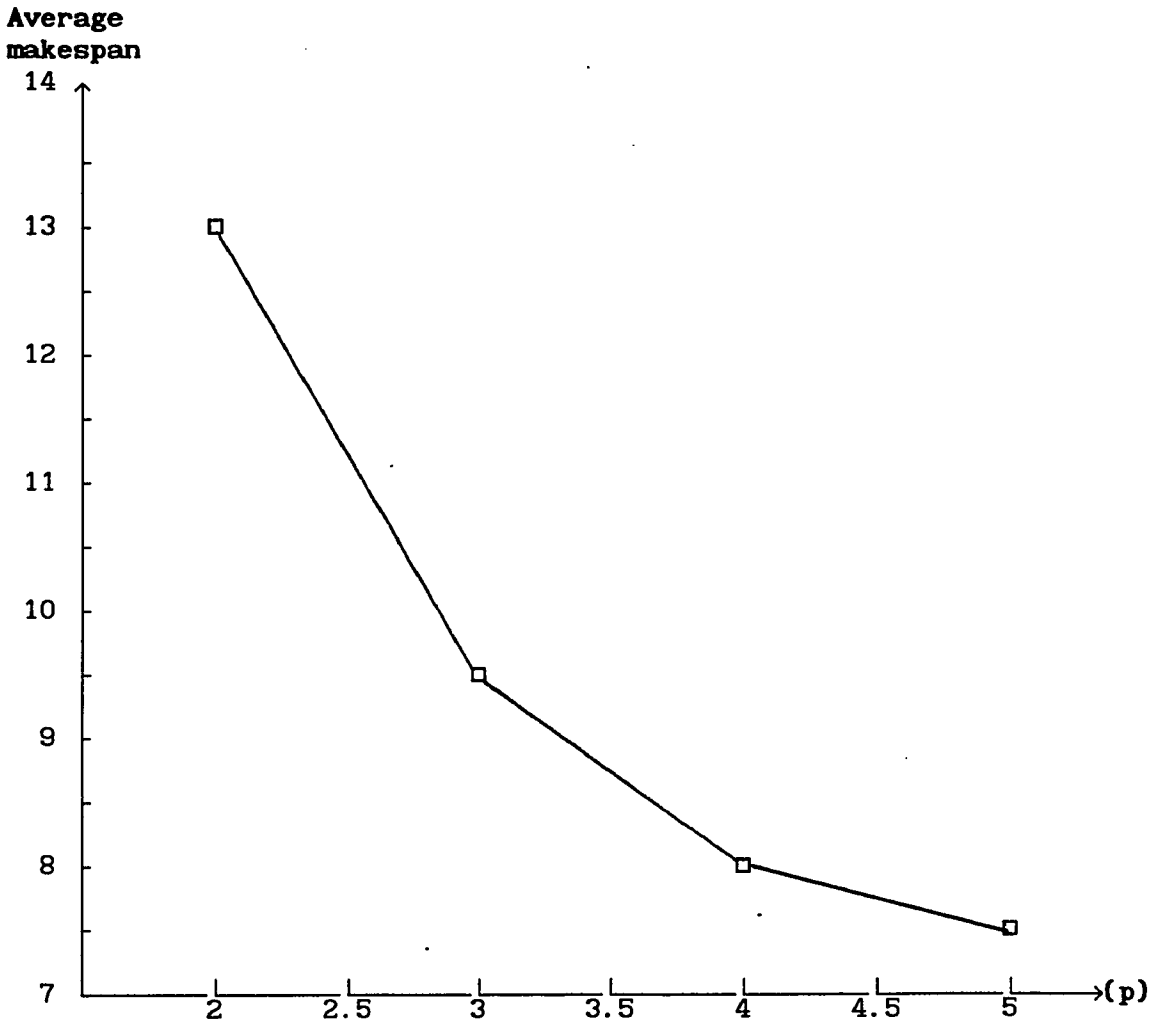
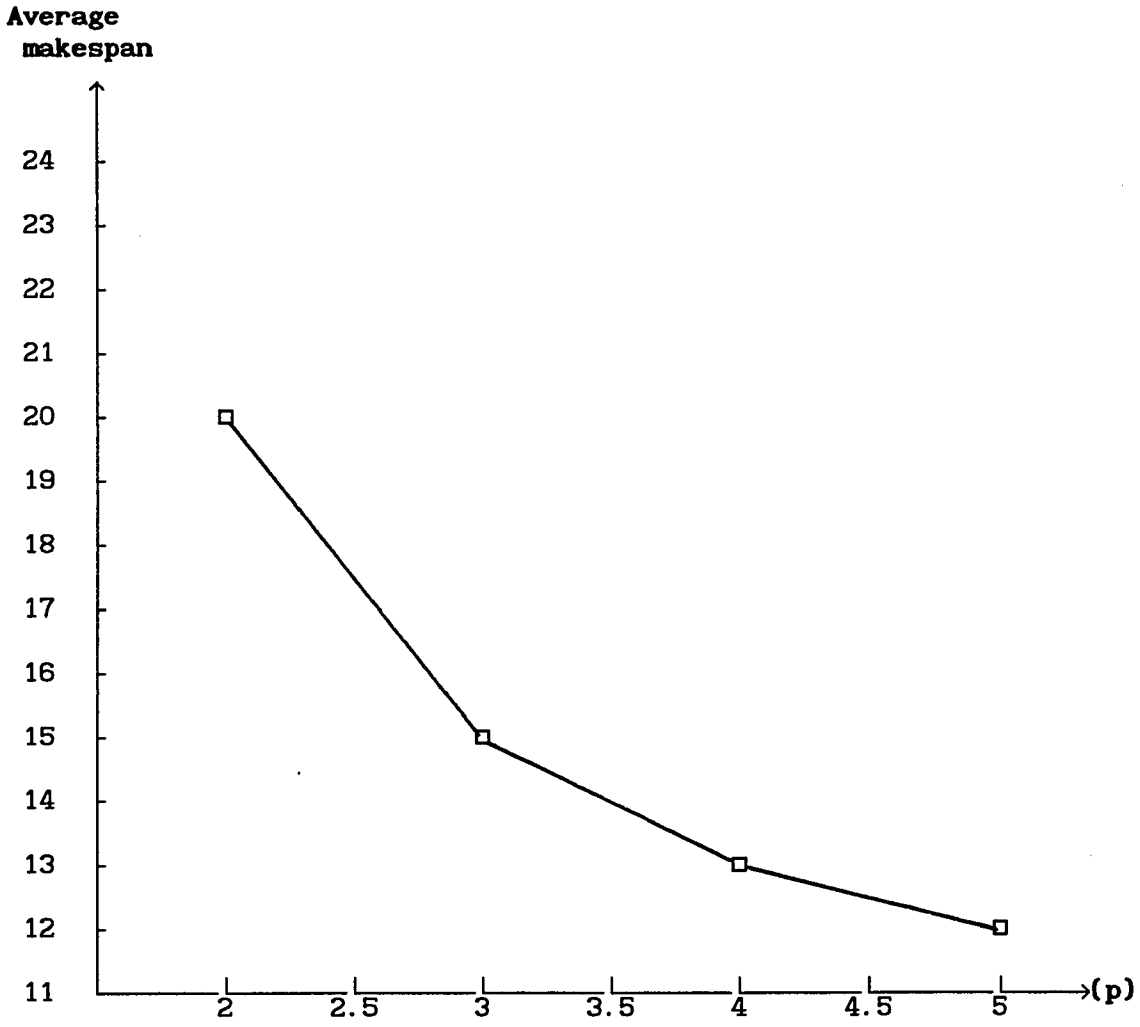
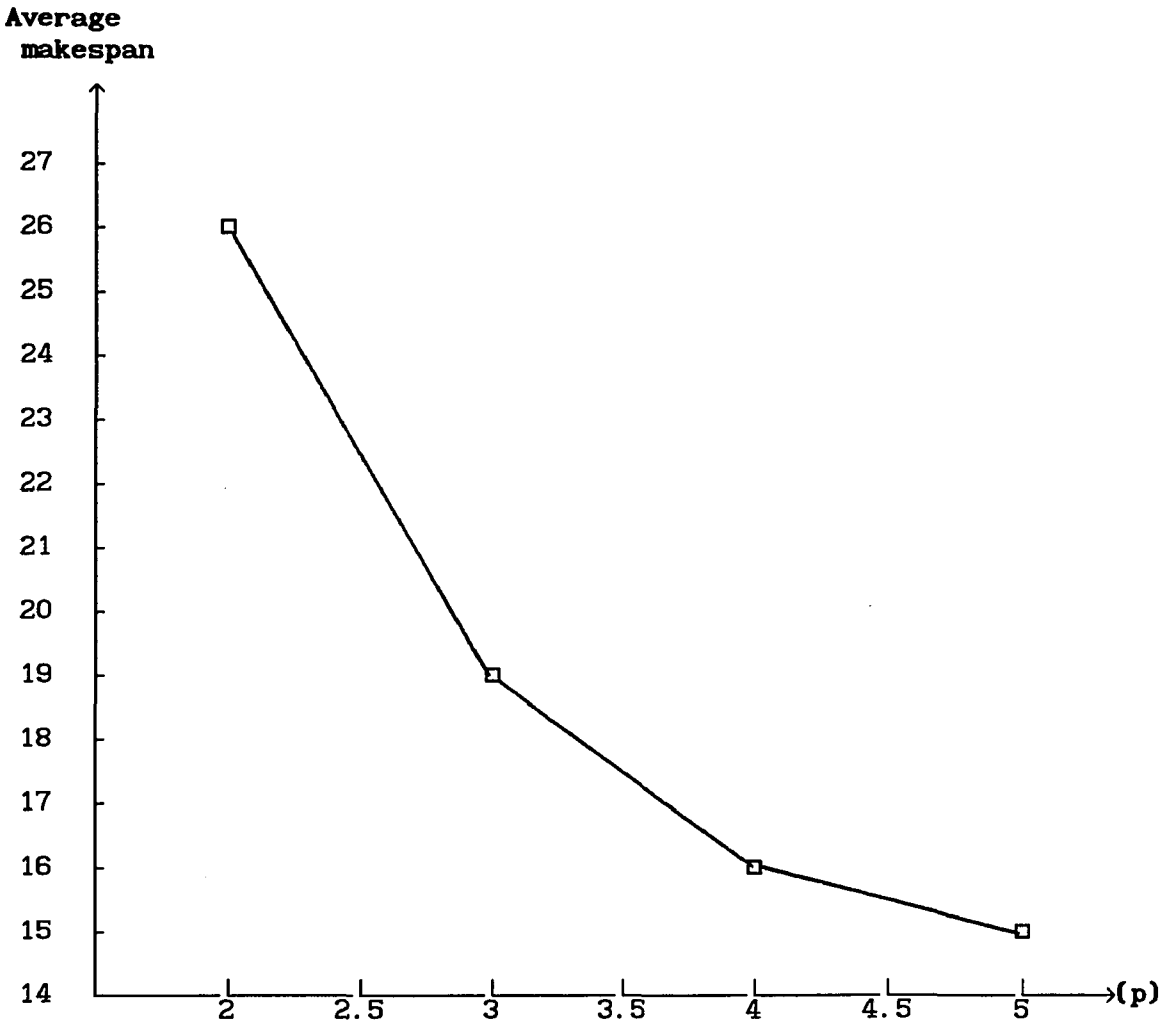


Figure 5-6 The relationship between average makespan and p for Model [AMp] with 5 processors and 25 tasks



p=2,3,4,5 of 5 processors and 40 tasks for Models [AMp]

Figure 5-7 The relationship between average makespan and p for Model [AMp] with 5 processors and 40 tasks



p=2,3,4,5 of 5 processors and 50 tasks for Models [AMp]

Figure 5-8 The relationship between average makespan and p for Model [AMp] with 5 processors and 50 tasks

CHAPTER SIX

SUMMARY, CONCLUSION, AND FUTURE RESEARCH DIRECTIONS

6.1 Thesis Summary

This research has studied the continuous task scheduling problem for unrelated multiprocessing systems. In Chapter 1, we discussed the motivation, problem definition, research questions, potential applications, and the possible solution techniques.

In Chapter 2, we provided an overview of the literature. Basically, the different problem types are based on varying both the task and processor assumptions. Tasks may be partially ordered, preemptive or nonpreemptive, etc. Processors may be identical, uniform, or unrelated. We classified the solution methodologies into two classes: one is the mathematical programming approach, the other is the herustic approach.

In Chapter 3, we showed some important properties of the continuous task scheduling problem and proved it to be NP-complete. We also discussed some previous linear programming formulations for scheduling on an unrelated multiprocessing system. We developed several new models, including an exact integer programming formulation (denoted as Model [AL_1]) for the "at-least-one-processor-working" case. We have also developed three new mathematical programs, two for the "at-least-p-processors-working" case (denoted as the Models [AL_p-Min] and [AL_p-Max]), and one for the "at-most-p-processors-working" case (denoted as the Model [AM_p]), respectively. We have verified the correctness of these three formulations.

We also showed in Lemma 3-1 that the solution set of the "at-least-p-processors-working" case is a subset of the solution set of the "at-least-one-processor-working" case, but not vice versa. This demonstrated the validity and contribution of the Models [AL_p-Min] and [AL_p-Max].

We provided an interesting potential application for the "at-most-p-processors-working" case. By solving the [AM_p] model several times with different p values, we can examine the effect of having at most p processors working in any given period on the minimum makespan. That is, the trade-off between the utilization and the reliability of the system can be captured in the above sensitivity analysis.

We also developed a constraint-relaxation solution approach. The [AL_1] model can be decomposed into two sub-models [Z_{ik}] and [X_{ij}], resulting in two subproblems, one of which is an assignment problem and the other a transportation problem. The assignment problem solution provides the "processor—time period" assignment matrix [Z_{ik}]. This solution is used as an input to the [X_{ij}] model to provide the task assignments to the processors. Note here that the solution may or may not be feasible. If infeasible, the constraint of the [Z_{ik}] model corresponding to makespan is relaxed and a new [Z_{ik}] model is obtained for use with the [X_{ij}] model. The idea is to solve the [Z_{ik}] and [X_{ij}] models sequentially and iteratively until an optimal solution to the original problem is found.

The iterative procedure for models [Z_{ik}] and [X_{ij}] was implemented on GAMS. The maximum number of iterations was

arbitarily set to equal 25, but the procedure always terminated with only a fraction of the limit on the number of iterations. We also provided a survey on the integer programming software packages for our chosen package to solve the models.

Although the iterative model is capable of solving relatively small problems, larger problems prove computationally intractive. For this reason, in chapter 4, we developed a heuristic with two versions of reassignment subroutines, this has been coded in TURBO C programming language and implemented in a PC environment with graphics. In addition to the minimum makespan and the feasible schedule to a given problem instance, we also provided the following information in the output: total execution time on PC, amount and the percentage of the total execution time used in the reassignment subroutines, number of reassignments made, and a graphical display showing the task-to-processor assignments before and after the reassignments.

The experimental design and tests performed are described in Chapter 5. These included 60 parametric combinations of {α, β, (Q, OL)}, where α is the relative load ratio of tasks to processor, β is the processor availability ratio of max w_i/min r_i, Q is the task executable rate by any processor, and OL is the task overlapping level among the task sets. We solved 540 test cases using both the IP solver in GAMS and the heuristics. We compared the total execution time, reassignment time, and the number of reassignments for version A and B. We also verified the execution time and feasible solutions among the output of GAMS, version A, and version B.

6.2 Thesis Conclusion

The first main contribution in this thesis is: We define the continuous task scheduling problem and then prove that the continuous task scheduling problem is NP-complete in the strong sense, which means this problem is at least as hard as the traveling salesman problem or the general integer knapsack problem, and therefore, is an appropriate candidate for heuristic methods.

The second main contribution is to formulate the continuous task scheduling problem as exact integer programming models. The first exact model can be constraint-relaxed and decomposed into two submodels. We showed that solving these submodels iteratively yields a solution to the original problem. This is the third contribution.

The other contribution is the heuristic we developed, which produces optimal solutions to 99% of the test problems with substantially less computation time. The results indicate that the performance of this heuristic is quite good.

For nearly all of the cases tested, both the exact Model [AL_1] and the heuristic achieve the same solutions, but the CPU time of the heuristic is at least 10^4 faster than the exact models solved by GAMS. GAMS is a general-purpose integer programming, while the heuristic takes advantage of special properties of the problem that reduce the search time required to find a solution. We coded this algorithm in the C programming language, while GAMS is written in FORTRAN, and C language has better and faster modules for sorting and searching. The same problem instance size

can be run on a PC using the heuristic, but must be run on a supercomputer when using GAMS. The current practical limit on problem size for GAMS is approximately 5 processors and 50 tasks, but the heuristic is capable of running much larger problems. Of the reassignment subroutines version A and B, version B is about 10% faster than version A. For the test problems examined, there is no difference in accuracy between versions A and B, although it is possible to construct problem instances in which version A will obtain a better solution than version B.

6.3 Future Research Directions

We have presented integer programming formulations and a heuristic for scheduling unit-processing-time tasks on unrelated processors. Some possible future research directions are discussed as follows. For the scheduling of nonpreemptive, general integer processing time tasks, the problem becomes a general integer knapsack problem with side constraints. We can still formulate this problem in an exact integer programming model and find a relatively good heuristic to solve the model approximately. However, this problem will be more challenging than the unit-processing-time continuous task scheduling problem.

For the scheduling of preemptive, general integer processing time tasks units, our models and algorithm can be easily modified to solve the problem. The idea is to replace each task by a set of unit-processing-time tasks (the number of those new tasks is equal to the original processing time of the replaced task). Of course, the size of the problems that can be solved by the heuristic will

inevitably decrease.

The concept of the reassignment subroutine is similar to that of the alternating path in the algorithm for matching (Lawler 1976). It may be interesting to extend this idea to solve other types of scheduling problems, such as parallel array processing.

The working and resting times play important roles in the continuous task scheduling problem. The relatively difficult cases are those with small working time and large resting time. In some cases we may still be able to identify task to processor assignments that meet the working time and resting time restrictions, resulting in a feasible schedule. If we can not find a feasible schedule, then how should we arrange the schdule so that as many as tasks can be executed as early as possible? Observe that the working and resting restrictions (here placed on processors) are somewhat analogous to requirements faced when dealing with the shift scheduling problem. Furthermore, the "at-least-p" constraint was assumed here to apply equally to every time period. In pratice, the value for p may be time-variant, reflecting peak and non-peak demand differences. Allowing p to vary from one time period to the next is a potentially interesting extension to the problem. Note that phase two of the algorithm could be easily modified to handle time-variant values for p.

Another question of interest is whether any computational or analytical approximation procedure for the continuous task scheduling problem is possible. The advantage of a relatively good analytical or computational approximation procedure is to reduce the need for a tree-search thereby saving execution time while

obtaining a solution that is still relatively close to the optimal solution. The filter beam search strategy used in searching the branch-and-bound trees may be useful, because such a search strategy does not go back to search again, which saves execution time but sacrifices optimality.

We discussed earlier how the trade-off between the utilization and the reliability of the system can be captured through sensitivity analysis. We also can consider the case where there is a panalty cost (i.e., increased maintenance cost) if a processor works beyond its max w_i without resting. This might permit a shorter makespan. This maintenance cost would be balanced by a makespan cost and the objective would be to minimize the total cost.

The application of Model [AM_p] with several different p values and penalty costs (i.e., the maintenance cost, overtime working cost) also can be considered. Different combinations between p values and penalty costs will give different makespans. We have to consider the makespan cost under the fixed p and minimize the total cost.

For the continuous task scheduling problem, given w_i and r_i's, can we develop an algorithm to arrange the schedule such that as many tasks can be executed as quickly possible? The Model [AL_p-Max] provides a preliminary answer. Also, the objective for the continuous task scheduling is to minimize the makespan, or the maximum completion time. We can change the objective function to investigate other measures, such as minimizing the total task waiting time.

What we have attempted in this work is to present a unified treatment for this particular scheduling problem. We have pointed out how many problems exist in this domain, and what research opportunities remain. These are challenging applied scheduling problems. What is particular interesting about this problem is that they require a blend of analysis, algebra, topology, computer science, and a knowledge of how the problem arises.

APPENDIX A: DETAIL TABLES FOR MODEL [ALP]

Table A-a.1 Test Results for Model [AL1]

with 5 processors and 10 tasks

$\{\alpha, \beta, (Q, OL)\}$	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	11	10	0	0	7.9	2	2	2
(L, H, RH)	13	12	11	10	0	0	7.9	2	2	2
(L, H, M)	12	12	11	11	0	0	7.9	2	2	2
(L, H, RL)	12	12	11	11	4	4	8.5	3	3	3
(L, H, L)	12	12	11	11	0	0	7.7	2	2	2
(L, M, H)	14	12	11	10	0	0	7.4	2	2	2
(L, M, RH)	12	13	11	10	0	0	6.7	2	2	2
(L, M, M)	13	12	11	10	0	0	7.9	2	2	2
(L, M, RL)	13	13	11	11	4	4	8.9	3	3	3
(L, M, L)	16	13	11	10	0	0	7.9	2	2	2
(L, L, H)	17	15	11	10	0	0	9.6	4	4	4
(L, L, RH)	12	15	11	11	0	0	12.1	4	4	4
(L, L, M)	12	12	11	11	0	0	9.7	4	4	4
(L, L, RL)	12	12	11	11	5	5	12.4	4	4	4
(L, L, L)	13	13	11	10	0	0	6.8	4	4	4

Table A-a.2 Test Results for Model [AL2]

with 5 processors and 10 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	12	12	11	10	0	0	6.6	2	2	2
(L, H, RH)	14	13	11	10	0	0	8.5	2	2	2
(L, H, M)	12	11	11	11	0	0	8.6	2	2	2
(L, H, RL)	12	12	11	11	5	6	9.4	3	3	3
(L, H, L)	12	11	11	11	0	0	8.3	2	2	2
(L, M, H)	14	12	10	10	0	0	7.9	2	2	2
(L, M, RH)	13	13	11	10	0	0	7.3	2	2	2
(L, M, M)	13	12	10	10	0	0	8.4	2	2	2
(L, M, RL)	14	13	11	11	5	6	9.5	3	3	3
(L, M, L)	16	15	10	10	0	0	8.5	2	2	2
(L, L, H)	17	15	10	10	0	0	11.5	4	4	4
(L, L, RH)	16	15	11	11	0	0	13.1	4	4	4
(L, L, M)	13	12	10	11	0	0	9.6	4	4	4
(L, L, RL)	13	12	11	11	6	6	7.4	6	6	6
(L, L, L)	13	13	11	10	0	0	7.6	4	4	4

Table A-a.3 Test Results for Model [AL3]

with 5 processors and 10 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	12	10	10	0	0	8.1	2	2	2
(L, H, RH)	14	12	11	10	0	0	8.3	2	2	2
(L, H, M)	12	12	10	10	0	0	8.4	2	2	2
(L, H, RL)	13	12	10	10	4	6	8.9	3	3	3
(L, H, L)	13	11	11	10	0	0	7.9	2	2	2
(L, M, H)	14	12	10	10	0	0	7.9	2	2	2
(L, M, RH)	13	12	11	10	0	0	7.5	2	2	2
(L, M, M)	13	12	10	10	0	0	8.3	2	2	2
(L, M, RL)	14	12	11	10	4	6	9.5	3	3	3
(L, M, L)	17	15	10	10	0	0	8.5	2	2	2
(L, L, H)	18	15	10	10	0	0	9.7	I	I	I
(L, L, RH)	17	16	11	11	0	0	11.8	I	I	I
(L, L, M)	13	11	11	11	0	0	10.8	I	I	I
(L, L, RL)	13	12	11	11	5	6	13.5	I	I	I
(L, L, L)	14	13	11	10	0	0	18.7	I	I	I

Table A-a.4 Test Results for Model [AL4]

with 5 processors and 10 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	13	11	10	0	0	8.6	2	2	2
(L, H, RH)	15	12	11	10	0	0	9.5	2	2	2
(L, H, M)	13	13	11	10	0	0	10.1	2	2	2
(L, H, RL)	13	12	10	10	5	4	9.2	I	I	I
(L, H, L)	13	11	11	10	0	0	10.8	2	2	2
(L, M, H)	14	12	11	10	0	0	9.5	2	2	2
(L, M, RH)	13	13	11	10	0	0	7.9	2	2	2
(L, M, M)	13	12	10	10	0	0	8.1	2	2	2
(L, M, RL)	14	12	11	10	5	5	9.3	I	I	I
(L, M, L)	17	16	11	10	0	0	8.5	2	2	2
(L, L, H)	18	16	11	10	0	0	10.5	I	I	I
(L, L, RH)	17	16	11	11	0	0	13.5	I	I	I
(L, L, M)	13	12	11	11	0	0	10.6	I	I	I
(L, L, RL)	13	13	11	11	6	6	14.2	I	I	I
(L, L, L)	14	12	11	10	0	0	12.1	I	I	I

Table A-a.5 Test Results for Model [ALs]

with 5 processors and 10 tasks

{α,β,(Q,OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	12	11	11	10	0	0	8.2	2	2	2
(L,H,RH)	12	12	10	10	0	0	8.4	2	2	2
(L,H,M)	12	11	11	10	0	0	8.5	2	2	2
(L,H,RL)	13	12	10	10	6	5	8.9	I	I	I
(L,H,L)	12	11	11	10	0	0	8.5	2	2	2
(L,M,H)	13	12	11	10	0	0	7.5	2	2	2
(L,M,RH)	14	13	11	10	0	0	7.4	2	2	2
(L,M,M)	12	12	10	10	0	0	8.5	2	2	2
(L,M,RL)	12	12	11	10	4	4	9.3	I	I	I
(L,M,L)	17	16	11	11	0	0	10.2	2	2	2
(L,L,H)	16	16	11	10	0	0	10.5	I	I	I
(L,L,RH)	17	16	11	11	0	0	14.2	I	I	I
(L,L,M)	15	13	11	11	0	0	13.2	I	I	I
(L,L,RL)	14	13	11	11	5	5	14.1	I	I	I
(L,L,L)	12	12	11	10	0	0	12.3	I	I	I

Table A-b.1 Test Results for Model [AL1]

with 5 processors and 25 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	10	10	0	0	8.9	5	5	5
(L, H, RH)	12	12	10	10	0	0	9.3	5	5	5
(L, H, M)	12	11	11	10	0	0	9.5	5	5	5
(L, H, RL)	14	12	11	10	2	2	8.6	5	5	5
(L, H, L)	12	11	11	10	0	0	8.3	7	7	7
(L, M, H)	13	12	11	10	0	0	8.2	6	6	6
(L, M, RH)	14	13	11	10	0	0	8.6	6	6	6
(L, M, M)	12	12	11	10	0	0	9.3	6	6	6
(L, M, RL)	12	12	11	10	5	5	9.4	6	6	6
(L, M, L)	13	13	11	11	0	0	9.9	7	7	7
(L, L, H)	14	13	11	10	0	0	9.7	10	10	10
(L, L, RH)	14	14	11	11	0	0	10.4	10	10	10
(L, L, M)	13	13	11	11	0	0	10.6	10	10	10
(L, L, RL)	13	13	11	11	5	5	10.7	10	10	10
(L, L, L)	13	12	11	10	0	0	11.6	12	12	12

Table A-b.2 Test Results for Model [AL2]

with 5 processors and 25 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time *10^{-4}(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	12	11	10	10	0	0	10.9	5	5	5
(L, H, RH)	13	12	11	10	0	0	11.2	5	5	5
(L, H, M)	13	11	11	10	0	0	11.3	5	5	5
(L, H, RL)	13	13	11	10	2	2	10.5	5	5	5
(L, H, L)	15	14	11	10	0	0	10.3	7	7	7
(L, M, H)	14	12	11	10	0	0	10.3	6	6	6
(L, M, RH)	13	13	11	10	0	0	11.1	6	6	6
(L, M, M)	12	12	11	11	0	0	11.3	6	6	6
(L, M, RL)	12	12	11	10	5	5	11.4	6	6	6
(L, M, L)	13	13	11	11	0	0	11.2	7	7	7
(L, L, H)	14	13	11	10	0	0	11.5	10	10	10
(L, L, RH)	14	14	11	11	0	0	12.5	10	10	10
(L, L, M)	13	12	11	11	0	0	12.7	10	10	10
(L, L, RL)	13	12	11	11	5	5	12.8	10	10	10
(L, L, L)	13	13	11	11	0	0	13.7	I	I	I

Table A-b.3 Test Results for Model [AL3]

with 5 processors and 25 tasks

{α,β,(Q,OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	13	11	11	11	0	0	10.8	5	5	5
(L,H,RH)	13	12	11	10	0	0	9.6	5	5	5
(L,H,M)	12	11	11	10	0	0	9.8	5	5	5
(L,H,RL)	13	13	11	10	2	2	9.2	5	5	5
(L,H,L)	13	13	11	10	0	0	8.5	7	7	7
(L,M,H)	14	13	11	11	0	0	8.7	I	I	I
(L,M,RH)	13	13	11	10	0	0	8.9	I	I	I
(L,M,M)	13	13	10	11	0	0	9.5	I	I	I
(L,M,RL)	13	12	11	10	5	5	9.6	I	I	I
(L,M,L)	13	13	11	11	0	0	10.3	7*	8*	8*
(L,L,H)	14	13	10	10	0	0	10.5	I	I	I
(L,L,RH)	13	14	10	10	0	0	11.2	I	I	I
(L,L,M)	13	12	11	11	0	0	11.7	I	I	I
(L,L,RL)	13	12	11	11	5	5	11.8	I	I	I
(L,L,L)	13	13	11	10	0	0	12.1	I	I	I

Table A-b.4 Test Results for Model [AL4]

with 5 processors and 25 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	10	10	0	0	11.2	5	5	5
(L, H, RH)	13	13	11	10	0	0	11.3	5	5	5
(L, H, M)	13	13	11	11	0	0	11.3	5	5	5
(L, H, RL)	13	13	11	10	2	2	10.5	5	5	5
(L, H, L)	13	13	11	10	0	0	11.2	I	I	I
(L, M, H)	14	13	11	11	0	0	9.5	I	I	I
(L, M, RH)	13	13	11	10	0	0	10.5	I	I	I
(L, M, M)	13	13	10	11	0	0	12.1	I	I	I
(L, M, RL)	13	12	11	10	5	5	11.9	6	6	6
(L, M, L)	13	13	11	11	0	0	11.2	I	I	I
(L, L, H)	14	13	10	10	0	0	11.1	I	I	I
(L, L, RH)	13	12	11	10	0	0	11.2	I	I	I
(L, L, M)	13	12	11	11	0	0	11.5	I	I	I
(L, L, RL)	13	12	11	11	5	5	11.8	I	I	I
(L, L, L)	13	13	11	11	0	0	12.9	I	I	I

Table A-b.5 Test Results for Model [ALs]

with 5 processors and 25 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	10	10	0	0	9.2	5	5	5
(L, H, RH)	12	12	11	10	0	0	9.5	5	5	5
(L, H, M)	13	13	11	11	0	0	10.3	5	5	5
(L, H, RL)	13	12	11	10	2	2	10.5	5	5	5
(L, H, L)	13	13	11	11	0	0	10.6	I	I	I
(L, M, H)	13	13	11	11	0	0	11.2	I	I	I
(L, M, RH)	13	13	11	10	0	0	12.1	I	I	I
(L, M, M)	13	13	11	11	0	0	11.2	I	I	I
(L, M, RL)	13	12	11	11	5	5	10.5	I	I	I
(L, M, L)	13	13	11	11	0	0	11.7	I	I	I
(L, L, H)	13	13	10	10	0	0	11.6	I	I	I
(L, L, RH)	14	13	11	10	0	0	11.5	I	I	I
(L, L, M)	13	12	11	11	0	0	11.7	I	I	I
(L, L, RL)	13	12	11	11	5	5	12.3	I	I	I
(L, L, L)	13	13	11	11	0	0	12.9	I	I	I

Table A-c.1 Test Results for Model [AL1]

with 5 processors and 40 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time *10^{-4}(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	13	10	10	0	0	12.3	9	9	9
(L, H, RH)	13	12	10	10	0	0	11.4	9	9	9
(L, H, M)	12	11	11	10	1	1	10.2	9	9	9
(L, H, RL)	12	11	10	10	4	4	9.3	9*	10*	10*
(L, H, L)	12	11	11	10	4	4	9.3	9*	10*	10*
(L, M, H)	13	12	10	10	0	0	13.2	10	10	10
(L, M, RH)	13	13	11	10	0	0	9.1	10	10	10
(L, M, M)	12	12	11	10	1	1	8.9	10	10	10
(L, M, RL)	12	12	11	10	4	4	8.3	12	12	12
(L, M, L)	13	13	11	11	4	4	9.9	12	12	12
(L, L, H)	14	13	11	10	0	0	15.7	16	16	16
(L, L, RH)	14	14	11	10	0	0	11.5	16	16	16
(L, L, M)	13	13	11	10	2	2	10.7	16	16	16
(L, L, RL)	13	13	11	11	4	4	11.9	18	18	18
(L, L, L)	13	12	11	10	2	2	12.3	18	18	18

Table A-c.2 Test Results for Model [AL2]

with 5 processors and 40 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time *10^{-4}(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	14	10	10	0	0	13.8	9	9	9
(L, H, RH)	13	13	10	10	0	0	12.9	9	9	9
(L, H, M)	12	12	11	10	1	1	10.8	9	9	9
(L, H, RL)	12	11	10	10	4	4	10.4	9	9	9
(L, H, L)	12	11	11	10	4	4	14.3	10	10	10
(L, M, H)	13	12	10	10	0	0	8.9	10	10	10
(L, M, RH)	13	13	11	10	6	6	9.8	10	10	10
(L, M, M)	13	12	11	10	0	0	9.7	12	12	12
(L, M, RL)	12	12	11	10	4	4	10.3	12	12	12
(L, M, L)	13	13	11	11	4	4	16.8	12	12	12
(L, L, H)	14	13	11	10	0	0	15.7	I	I	I
(L, L, RH)	14	14	11	10	0	0	14.2	I	I	I
(L, L, M)	13	12	11	10	2	2	13.7	16	16	16
(L, L, RL)	13	13	11	11	4	4	15.2	I	I	I
(L, L, L)	13	12	11	10	4	4	16.5	I	I	I

Table A-c.3 Test Results for Model [AL3]

with 5 processors and 40 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	10	10	0	0	13.6	9	9	9
(L, H, RH)	13	12	10	10	0	0	12.8	9	9	9
(L, H, M)	13	12	11	10	1	1	11.9	9	9	9
(L, H, RL)	12	11	10	10	4	4	9.7	9*	10*	10*
(L, H, L)	12	11	11	10	4	4	10.5	9*	10*	10*
(L, M, H)	13	12	10	10	0	0	14.8	I	I	I
(L, M, RH)	13	13	11	10	6	6	9.7	I	I	I
(L, M, M)	13	12	11	10	0	0	9.5	I	I	I
(L, M, RL)	12	12	11	10	4	4	10.5	12	12	12
(L, M, L)	13	13	10	10	4	4	11.3	I	I	I
(L, L, H)	14	13	11	10	0	0	18.8	I	I	I
(L, L, RH)	14	13	10	10	0	0	12.9	I	I	I
(L, L, M)	13	12	11	10	2	2	14.7	I	I	I
(L, L, RL)	13	13	11	11	4	4	13.5	I	I	I
(L, L, L)	13	12	11	10	4	4	15.3	I	I	I

Table A-c.4 Test Results for Model [AL4]

with 5 processors and 40 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	10	10	0	0	14.7	I	I	I
(L, H, RH)	14	13	11	10	0	0	12.8	I	I	I
(L, H, M)	13	12	11	10	1	1	12.3	I	I	I
(L, H, RL)	12	11	10	10	4	4	9.8	I	I	I
(L, H, L)	12	11	11	10	4	4	10.2	I	I	I
(L, M, H)	13	12	10	10	0	0	14.7	I	I	I
(L, M, RH)	13	13	11	10	6	6	10.2	I	I	I
(L, M, M)	13	12	11	10	0	0	10.6	I	I	I
(L, M, RL)	12	12	11	10	4	4	9.7	I	I	I
(L, M, L)	13	13	10	10	4	4	10.5	I	I	I
(L, L, H)	14	13	11	10	0	0	18.7	I	I	I
(L, L, RH)	14	13	10	10	0	0	16.5	I	I	I
(L, L, M)	13	12	11	10	2	2	17.2	I	I	I
(L, L, RL)	13	13	11	11	4	4	16.2	I	I	I
(L, L, L)	13	12	11	10	4	4	15.3	I	I	I

Table A-c.5 Test Results for Model [ALs]

with 5 processors and 40 tasks

{α,β,(Q,OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	16	13	10	10	0	0	17.3	I	I	I
(L,H,RH)	14	13	11	10	0	0	17.2	I	I	I
(L,H,M)	14	13	11	10	1	1	15.3	I	I	I
(L,H,RL)	13	13	11	11	4	4	14.3	I	I	I
(L,H,L)	12	11	11	10	4	4	13.2	I	I	I
(L,M,H)	13	12	10	10	0	0	19.2	I	I	I
(L,M,RH)	13	12	11	10	6	6	10.3	I	I	I
(L,M,M)	13	12	11	10	0	0	12.5	I	I	I
(L,M,RL)	13	12	11	10	4	4	9.2	I	I	I
(L,M,L)	13	13	10	10	4	4	11.3	I	I	I
(L,L,H)	14	13	11	10	0	0	19.6	I	I	I
(L,L,RH)	14	13	10	10	0	0	15.8	I	I	I
(L,L,M)	13	13	11	10	2	2	14.3	I	I	I
(L,L,RL)	13	13	11	11	4	4	13.7	I	I	I
(L,L,L)	13	12	11	10	4	4	15.2	I	I	I

Table A-d.1 Test Results for Model [AL1]

with 5 processors and 50 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time *10^{-4}(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	10	10	0	0	11.2	12	12	12
(L, H, RH)	13	13	10	10	0	0	13.1	12	12	12
(L, H, M)	13	11	11	10	0	0	12.8	12	12	12
(L, H, RL)	12	11	10	10	2	2	11.5	12	12	12
(L, H, L)	12	11	11	10	0	0	15.5	12	12	12
(L, M, H)	13	12	10	10	0	0	10.6	13	13	12
(L, M, RH)	13	13	11	10	0	0	13.5	13	13	13
(L, M, M)	12	12	11	10	0	0	12.6	13	13	13
(L, M, RL)	12	12	11	10	1	1	11.7	13	13	13
(L, M, L)	13	13	11	11	0	0	12.5	13	13	13
(L, L, H)	14	13	11	10	0	0	14.5	20	20	20
(L, L, RH)	14	14	11	10	0	0	17.4	20	20	20
(L, L, M)	13	13	11	10	0	0	15.2	20	20	20
(L, L, RL)	13	13	11	11	0	0	14.7	20	20	20
(L, L, L)	13	12	11	10	2	2	17.8	20	20	20

Table A-d.2 Test Results for Model [AL2]

with 5 processors and 50 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	11	10	0	0	16.2	12	12	12
(L, H, RH)	13	13	10	11	0	0	19.2	12	12	12
(L, H, M)	13	11	11	10	0	0	17.3	12	12	12
(L, H, RL)	12	11	10	10	2	2	18.5	12	12	12
(L, H, L)	13	12	11	10	0	0	20.5	12	12	12
(L, M, H)	13	12	10	10	0	0	16.5	13	13	12
(L, M, RH)	13	13	11	10	0	0	16.5	13	13	13
(L, M, M)	13	13	11	11	0	0	16.8	13	13	13
(L, M, RL)	12	12	11	10	1	1	16.9	13	13	13
(L, M, L)	13	13	11	11	0	0	16.5	13	13	13
(L, L, H)	14	13	11	10	0	0	20.1	20	20	20
(L, L, RH)	14	13	11	10	0	0	25.2	20	20	20
(L, L, M)	13	13	11	11	0	0	25.3	20	20	20
(L, L, RL)	13	13	11	11	0	0	23.5	20	20	20
(L, L, L)	13	12	11	10	2	2	26.8	20	20	20

Table A-d.3 Test Results for Model [AL3]

with 5 processors and 50 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	11	10	0	0	19.3	12	12	12
(L, H, RH)	13	13	10	11	0	0	22.1	12	12	12
(L, H, M)	13	13	10	10	0	0	21.2	12	12	12
(L, H, RL)	14	11	10	10	2	2	19.6	12	12	12
(L, H, L)	13	12	10	10	0	0	23.7	12	12	12
(L, M, H)	13	12	10	10	0	0	18.9	13	13	12
(L, M, RH)	13	13	11	10	0	0	23.1	13	13	13
(L, M, M)	13	13	11	11	0	0	22.7	13	13	13
(L, M, RL)	14	12	11	10	1	1	19.7	13	13	13
(L, M, L)	13	13	11	11	0	0	21.2	13	13	13
(L, L, H)	14	13	11	10	0	0	23.5	I	I	I
(L, L, RH)	14	13	11	10	0	0	27.8	I	I	I
(L, L, M)	14	13	11	11	0	0	24.1	I	I	I
(L, L, RL)	13	13	11	11	0	0	23.1	I	I	I
(L, L, L)	14	12	11	10	2	2	23.8	I	I	I

Table A-d.4 Test Results for Model [AL4]

with 5 processors and 50 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	13	10	10	0	0	19.2	I	I	I
(L, H, RH)	13	12	11	11	0	0	24.2	I	I	I
(L, H, M)	14	13	11	10	0	0	20.3	I	I	I
(L, H, RL)	14	13	11	10	2	2	20.1	I	I	I
(L, H, L)	13	12	11	10	0	0	24.5	I	I	I
(L, M, H)	13	13	11	10	0	0	20.1	I	I	I
(L, M, RH)	13	13	11	10	0	0	21.4	I	I	I
(L, M, M)	13	13	11	11	0	0	23.1	I	I	I
(L, M, RL)	14	12	11	10	1	1	26.5	I	I	I
(L, M, L)	13	13	11	11	0	0	26.4	I	I	I
(L, L, H)	14	13	11	10	0	0	24.1	I	I	I
(L, L, RH)	14	13	11	10	0	0	26.8	I	I	I
(L, L, M)	14	13	11	11	0	0	24.1	I	I	I
(L, L, RL)	13	13	11	11	0	0	23.1	I	I	I
(L, L, L)	14	12	11	10	2	2	26.8	I	I	I

Table A-d.5 Test Results for Model [ALs]

with 5 processors and 50 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	10	10	0	0	22.1	I	I	I
(L, H, RH)	14	12	11	11	0	0	24.7	I	I	I
(L, H, M)	14	13	10	10	0	0	25.2	I	I	I
(L, H, RL)	14	13	11	10	2	2	13.5	I	I	I
(L, H, L)	13	13	10	10	0	0	27.6	I	I	I
(L, M, H)	13	13	11	10	0	0	21.6	I	I	I
(L, M, RH)	13	13	11	10	0	0	25.5	I	I	I
(L, M, M)	13	13	11	11	0	0	25.6	I	I	I
(L, M, RL)	14	12	11	10	1	1	23.7	I	I	I
(L, M, L)	13	13	11	11	0	0	23.5	I	I	I
(L, L, H)	14	13	11	10	0	0	26.4	I	I	I
(L, L, RH)	14	13	11	10	0	0	29.5	I	I	I
(L, L, M)	14	13	11	11	0	0	28.2	I	I	I
(L, L, RL)	13	13	11	11	0	0	26.7	I	I	I
(L, L, L)	14	12	11	10	2	2	28.5	I	I	I

APPENDIX B: DETAIL TABLES FOR MODEL [AMP]

Table B-a.1 Test results for Model [AM2]

with 5 processors and 10 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	10	10	0	0	11.2	5	5	5
(L, H, RH)	12	12	10	10	0	0	11.3	5	5	5
(L, H, M)	12	11	11	10	0	0	11.2	5	5	5
(L, H, RL)	14	12	11	10	5	2	12.5	5	5	5
(L, H, L)	12	11	11	10	0	0	11.1	5	5	5
(L, M, H)	13	12	11	10	0	0	11.8	5	5	5
(L, M, RH)	14	13	11	10	0	0	10.7	5	5	5
(L, M, M)	12	12	11	10	0	0	11.9	5	5	5
(L, M, RL)	12	12	11	10	5	5	12.3	5	5	5
(L, M, L)	13	13	11	11	0	0	11.9	5	5	5
(L, L, H)	14	13	11	10	0	0	12.3	5	5	5
(L, L, RH)	14	14	11	11	0	0	16.1	5	5	5
(L, L, M)	13	13	11	11	0	0	12.1	5	5	5
(L, L, RL)	13	13	11	11	5	5	12.3	5*	6*	6*
(L, L, L)	13	12	11	10	0	0	11.2	5	5	5

Table B-a.2 Test results for Model [AM3]

with 5 processors and 10 tasks

$\{\alpha, \beta, (Q, OL)\}$	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	13	13	10	10	0	0	11.8	4	4	4
(L,H,RH)	12	12	10	10	0	0	11.7	4	4	4
(L,H,M)	12	11	11	10	0	0	11.8	4	4	4
(L,H,RL)	14	12	11	10	5	2	12.5	4	4	4
(L,H,L)	12	11	11	10	0	0	11.7	4	4	4
(L,M,H)	13	12	11	10	0	0	11.6	4	4	4
(L,M,RH)	14	13	11	10	0	0	11.8	4	4	4
(L,M,M)	12	12	11	10	0	0	11.9	4	4	4
(L,M,RL)	12	12	11	10	5	5	13.1	4	4	4
(L,M,L)	13	13	11	11	0	0	12.9	4	4	4
(L,L,H)	14	13	11	10	0	0	11.2	4	4	4
(L,L,RH)	14	14	11	11	0	0	17.1	4	4	4
(L,L,M)	13	13	11	11	0	0	13.7	4	4	4
(L,L,RL)	13	13	11	11	5	5	16.4	4	4	4
(L,L,L)	13	12	11	10	0	0	11.8	4	4	4

Table B-a.3 Test results for Model [AM4]

with 5 processors and 10 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	13	10	10	0	0	12.6	3	3	3
(L, H, RH)	12	12	11	10	0	0	12.7	3	3	3
(L, H, M)	13	13	11	10	0	0	13.1	3	3	3
(L, H, RL)	14	12	11	10	5	2	14.2	3	3	3
(L, H, L)	12	11	11	10	0	0	13.2	3	3	3
(L, M, H)	13	12	11	10	0	0	12.8	3	3	3
(L, M, RH)	14	13	11	10	0	0	12.1	3	3	3
(L, M, M)	14	13	11	10	0	0	12.9	3	3	3
(L, M, RL)	12	12	11	10	5	5	11.3	3	3	3
(L, M, L)	13	13	11	11	0	0	11.7	3	3	3
(L, L, H)	14	13	11	10	0	0	11.5	4	4	4
(L, L, RH)	14	14	11	11	0	0	17.3	4	4	4
(L, L, M)	13	13	11	11	0	0	15.7	4	4	4
(L, L, RL)	13	13	11	11	5	5	17.5	4	4	4
(L, L, L)	13	12	11	10	0	0	12.7	4	4	4

Table B-a.4 Test results for Model [AMs]

with 5 processors and 10 tasks

{α,β,(Q,OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time *10^{-4}(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	13	13	11	10	0	0	7.9	2	2	2
(L,H,RH)	13	13	10	10	0	0	7.9	2	2	2
(L,H,M)	13	13	11	10	0	0	7.9	2	2	2
(L,H,RL)	14	12	11	10	4	2	8.5	3	3	3
(L,H,L)	12	11	11	10	0	0	7.7	2	2	2
(L,M,H)	13	12	11	10	0	0	7.4	2	2	2
(L,M,RH)	14	13	11	10	0	0	6.7	2	2	2
(L,M,M)	14	13	11	10	0	0	7.9	2	2	2
(L,M,RL)	12	12	11	10	5	5	8.9	3	3	3
(L,M,L)	13	13	11	11	0	0	7.9	2	2	2
(L,L,H)	14	13	11	10	0	0	9.6	4	4	4
(L,L,RH)	14	14	11	11	0	0	12.1	4	4	4
(L,L,M)	13	13	11	11	0	0	9.7	4	4	4
(L,L,RL)	13	13	11	11	5	5	12.4	4	4	4
(L,L,L)	13	12	11	10	0	0	6.8	4	4	4

Table B-b.1 Test results for Model [AM2]

with 5 processors and 25 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	13	10	10	0	0	15.9	13	13	13
(L, H, RH)	12	12	10	10	0	0	16.3	13	13	13
(L, H, M)	12	11	11	10	0	0	16.5	13	13	13
(L, H, RL)	14	12	11	10	2	2	15.6	13	13	13
(L, H, L)	12	11	11	10	0	0	15.3	13	13	13
(L, M, H)	13	12	11	10	0	0	15.5	13	13	13
(L, M, RH)	14	13	11	10	0	0	15.6	13	13	13
(L, M, M)	12	12	11	10	0	0	16.3	13	13	13
(L, M, RL)	12	12	11	10	5	5	16.4	13	13	13
(L, M, L)	13	13	11	11	0	0	14.8	13	13	13
(L, L, H)	14	13	11	10	0	0	15.4	13	13	13
(L, L, RH)	14	14	11	11	0	0	17.4	13	13	13
(L, L, M)	13	13	11	11	0	0	17.6	13	13	13
(L, L, RL)	13	13	11	11	5	5	17.7	13	13	13
(L, L, L)	13	12	11	10	0	0	18.6	13	13	13

Table B-b.2 Test results for Model [AM3]

with 5 processors and 25 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	13	11	10	0	0	13.9	9	9	9
(L, H, RH)	13	12	10	10	0	0	14.3	9	9	9
(L, H, M)	13	11	11	10	0	0	14.5	9	9	9
(L, H, RL)	14	12	11	11	2	2	13.6	9	9	9
(L, H, L)	12	11	11	10	0	0	14.3	9	9	9
(L, M, H)	13	12	11	10	0	0	13.6	9	9	9
(L, M, RH)	14	13	11	10	0	0	14.3	9	9	9
(L, M, M)	12	12	11	10	0	0	14.4	9	9	9
(L, M, RL)	12	12	11	10	5	5	13.2	9	9	9
(L, M, L)	13	13	11	11	0	0	16.6	9	9	9
(L, L, H)	14	13	11	10	0	0	14.4	11	11	11
(L, L, RH)	14	14	11	11	0	0	15.6	11	11	11
(L, L, M)	13	13	11	11	0	0	14.4	11	11	11
(L, L, RL)	13	13	11	11	5	5	5.7	11	11	11
(L, L, L)	13	12	11	10	0	0	16.7	11	11	11

Table B-b.3 Test results for Model [AM4]

with 5 processors and 25 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	13	11	10	0	0	12.8	7	7	7
(L, H, RH)	13	13	11	10	0	0	13.3	7	7	7
(L, H, M)	13	13	11	10	0	0	13.5	7	7	7
(L, H, RL)	14	12	11	11	2	2	12.7	7	7	7
(L, H, L)	12	11	11	10	0	0	12.3	7	7	7
(L, M, H)	13	12	11	10	0	0	13.5	7	7	7
(L, M, RH)	14	13	11	10	0	0	12.6	7	7	7
(L, M, M)	12	12	11	10	0	0	13.3	7	7	7
(L, M, RL)	12	12	11	10	5	5	13.4	7	7	7
(L, M, L)	13	13	11	11	0	0	12.2	7	7	7
(L, L, H)	14	13	11	10	0	0	13.7	10	10	10
(L, L, RH)	14	14	11	11	0	0	14.5	10	10	10
(L, L, M)	13	13	11	11	0	0	15.1	10	10	10
(L, L, RL)	13	13	11	11	5	5	14.8	10	10	10
(L, L, L)	13	12	11	10	0	0	16.6	10	10	10

Table B-b.4 Test results for Model [AMs]

with 5 processors and 25 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	13	11	10	0	0	8.9	5	5	5
(L, H, RH)	13	13	11	10	0	0	9.3	5	5	5
(L, H, M)	13	13	11	10	0	0	9.5	5	5	5
(L, H, RL)	14	12	11	11	2	2	8.6	5	5	5
(L, H, L)	12	11	11	10	0	0	8.3	7	7	7
(L, M, H)	13	12	11	10	0	0	8.2	6	6	6
(L, M, RH)	14	13	11	10	0	0	8.6	6	6	6
(L, M, M)	12	12	11	10	0	0	9.3	6	6	6
(L, M, RL)	12	12	11	10	5	5	9.4	6	6	6
(L, M, L)	13	13	11	11	0	0	9.9	7	7	7
(L, L, H)	14	13	11	10	0	0	9.7	10	10	10
(L, L, RH)	14	14	11	11	0	0	10.4	10	10	10
(L, L, M)	13	13	11	11	0	0	10.6	10	10	10
(L, L, RL)	13	13	11	11	5	5	10.7	10	10	10
(L, L, L)	13	12	11	10	0	0	11.6	12	12	12

Table B-c.1 Test results for Model [AM2]

with 5 processors and 40 tasks

$\{\alpha, \beta, (Q, OL)\}$	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	14	13	11	10	0	0	22.3	20	20	20
(L,H,RH)	13	12	10	10	0	0	21.4	20	20	20
(L,H,M)	13	13	11	11	0	0	20.2	20	20	20
(L,H,RL)	14	12	11	10	2	2	19.3	20	20	20
(L,H,L)	12	11	11	10	0	0	23.1	20	20	20
(L,M,H)	13	12	11	10	0	0	19.1	20	20	20
(L,M,RH)	14	13	11	10	0	0	18.9	20	20	20
(L,M,M)	12	12	11	10	0	0	18.3	20	20	20
(L,M,RL)	12	12	11	10	5	5	19.9	20	20	20
(L,M,L)	13	13	11	11	0	0	25.8	20	20	20
(L,L,H)	14	13	11	10	0	0	21.5	20	20	20
(L,L,RH)	14	14	11	11	0	0	20.7	20	20	20
(L,L,M)	13	13	11	11	0	0	22.1	20	20	20
(L,L,RL)	13	13	11	11	5	5	23.1	20	20	20
(L,L,L)	13	12	11	10	0	0	23.4	20	20	20

Table B-c.2 Test results for Model [AM3]

with 5 processors and 40 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	13	12	10	10	0	0	18.3	14	14	14
(L, H, RH)	13	13	10	10	0	0	17.4	14	14	14
(L, H, M)	13	13	11	11	0	0	16.2	14	14	14
(L, H, RL)	14	12	11	10	2	2	16.3	14	14	14
(L, H, L)	12	11	11	10	0	0	16.4	14	14	14
(L, M, H)	13	12	11	10	0	0	17.9	14	14	14
(L, M, RH)	14	13	11	10	0	0	16.1	14	14	14
(L, M, M)	12	12	11	10	0	0	16.2	14	14	14
(L, M, RL)	12	12	11	10	5	5	15.3	14	14	14
(L, M, L)	13	13	11	11	0	0	16.9	14	14	14
(L, L, H)	14	13	11	10	0	0	22.8	17	17	17
(L, L, RH)	14	14	11	11	0	0	17.5	17	17	17
(L, L, M)	13	13	11	11	0	0	17.7	16	16	16
(L, L, RL)	13	13	11	11	5	5	18.9	18	18	18
(L, L, L)	13	12	11	10	0	0	19.3	18	18	18

Table B-c.3 Test results for Model [AM4]

with 5 processors and 40 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	13	10	10	0	0	16.2	11	11	11
(L, H, RH)	13	13	10	10	0	0	15.3	11	11	11
(L, H, M)	13	13	11	11	0	0	14.3	11	11	11
(L, H, RL)	14	13	11	10	2	2	13.2	11	11	11
(L, H, L)	12	11	11	10	0	0	17.7	11	11	11
(L, M, H)	13	12	11	10	0	0	13.2	12	12	12
(L, M, RH)	14	13	11	10	0	0	13.1	12	12	12
(L, M, M)	12	12	11	10	0	0	12.5	12	12	12
(L, M, RL)	12	12	11	10	5	5	13.9	12	12	12
(L, M, L)	13	13	11	11	0	0	20.1	12	12	12
(L, L, H)	14	13	11	10	0	0	15.6	16	16	16
(L, L, RH)	14	13	11	11	0	0	16.7	16	16	16
(L, L, M)	13	13	11	11	0	0	19.1	16	16	16
(L, L, RL)	13	13	11	11	5	5	19.2	18	18	18
(L, L, L)	13	12	11	10	0	0	19.3	18	18	18

Table B-c.4 Test results for Model [AMs]

with 5 processors and 40 tasks

{α,β,(Q,OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	13	13	11	10	0	0	12.3	9	9	9
(L,H,RH)	13	12	11	10	0	0	11.4	9	9	9
(L,H,M)	13	13	11	11	0	0	10.2	9	9	9
(L,H,RL)	14	13	11	10	2	2	9.3	9	9	9
(L,H,L)	12	11	11	10	0	0	9.3	9	9	9
(L,M,H)	13	12	11	10	0	0	13.2	10	10	10
(L,M,RH)	14	13	11	10	0	0	9.1	10	10	10
(L,M,M)	12	12	11	10	0	0	8.9	10	10	10
(L,M,RL)	12	12	11	10	5	5	8.3	10	10	10
(L,M,L)	13	13	11	11	0	0	9.9	10	10	10
(L,L,H)	14	13	11	10	0	0	15.7	16	16	10
(L,L,RH)	14	13	11	11	0	0	11.5	16	16	16
(L,L,M)	13	13	11	11	0	0	10.7	16	16	16
(L,L,RL)	13	13	11	11	5	5	11.9	16	16	16
(L,L,L)	13	12	11	10	0	0	12.3	16	16	16

Table B-d.1 Test results for Model [AM2]

with 5 processors and 50 tasks

{α,β,(Q,OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time *10^{-4}(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	13	13	11	10	0	0	31.2	26	26	26
(L,H,RH)	14	12	10	10	0	0	33.1	26	26	26
(L,H,M)	13	13	11	11	0	0	32.8	26	26	26
(L,H,RL)	14	13	11	10	2	2	31.5	26	26	26
(L,H,L)	12	11	11	10	0	0	35.6	26	26	26
(L,M,H)	13	12	11	10	0	0	31.5	26	26	26
(L,M,RH)	14	13	11	10	0	0	33.5	26	26	26
(L,M,M)	12	12	11	10	0	0	32.6	26	26	26
(L,M,RL)	12	12	11	10	5	5	31.7	26	26	26
(L,M,L)	13	13	11	11	0	0	32.5	26	26	26
(L,L,H)	14	13	11	10	0	0	34.5	26	26	26
(L,L,RH)	14	14	11	11	0	0	37.4	26	26	26
(L,L,M)	13	13	11	11	0	0	35.2	26	26	26
(L,L,RL)	13	13	11	11	5	5	34.7	26	26	26
(L,L,L)	13	12	11	10	0	0	37.8	26	26	26

Table B-d.2 Test results for Model [AM3]

with 5 processors and 50 tasks

{α, β, (Q, OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time *10^{-4}(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L, H, H)	14	13	11	10	0	0	28.2	17	17	17
(L, H, RH)	14	13	11	10	0	0	30.1	17	17	17
(L, H, M)	13	13	11	10	0	0	29.8	17	17	17
(L, H, RL)	14	13	11	11	2	2	28.5	17	17	17
(L, H, L)	12	11	11	10	0	0	32.6	17	17	17
(L, M, H)	13	12	11	10	0	0	27.4	18	18	18
(L, M, RH)	14	13	11	10	0	0	30.5	18	18	18
(L, M, M)	12	12	11	10	0	0	29.6	18	18	18
(L, M, RL)	12	12	11	10	5	5	28.7	18	18	18
(L, M, L)	13	13	11	11	0	0	29.5	18	18	18
(L, L, H)	14	13	11	10	0	0	33.5	21	21	21
(L, L, RH)	14	14	11	11	0	0	34.5	21	21	21
(L, L, M)	13	13	11	11	0	0	32.2	21	21	21
(L, L, RL)	13	13	11	11	5	5	31.7	21	21	21
(L, L, L)	13	12	11	10	0	0	34.8	21	21	21

Table B-d.3 Test results for Model [AM4]

with 5 processors and 50 tasks

{α,β,(Q,OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time $*10^{-4}$(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	13	13	11	11	0	0	21.2	14	14	14
(L,H,RH)	14	13	11	10	0	0	23.1	14	14	14
(L,H,M)	13	13	11	10	0	0	22.8	14	14	14
(L,H,RL)	14	13	11	11	2	2	21.6	14	14	14
(L,H,L)	13	12	11	10	0	0	25.6	14	14	14
(L,M,H)	13	12	11	10	0	0	21.5	15	15	15
(L,M,RH)	14	13	11	10	0	0	23.5	15	15	15
(L,M,M)	12	12	11	10	0	0	22.6	15	15	15
(L,M,RL)	12	12	11	10	5	5	21.7	15	15	15
(L,M,L)	13	13	11	11	0	0	22.5	15	15	15
(L,L,H)	14	13	11	10	0	0	24.5	20	20	20
(L,L,RH)	14	13	11	11	0	0	27.4	20	20	20
(L,L,M)	14	13	11	11	0	0	25.2	20	20	20
(L,L,RL)	13	13	11	11	5	5	24.7	20	20	20
(L,L,L)	13	12	11	10	0	0	27.8	20	20	20

Table B-d.4 Test results for Model [AMs]

with 5 processors and 50 tasks

{α,β,(Q,OL)}	Total Execution T* 10^{-4} (sec)		Reassignment time *10^{-4}(sec)		Number of successful reassignment		GAMS CPU TIME (seconds)	GAMS make-span	VA make-span	VB make-span
	VA	VB	VA	VB	VA	VB				
(L,H,H)	14	13	11	11	0	0	11.2	12	12	12
(L,H,RH)	14	13	11	10	0	0	13.1	12	12	12
(L,H,M)	14	13	11	10	0	0	12.8	12	12	12
(L,H,RL)	14	13	10	11	2	2	11.5	12	12	12
(L,H,L)	13	13	11	10	0	0	15.6	12	12	12
(L,M,H)	13	12	10	10	0	0	10.4	13	13	13
(L,M,RH)	14	13	11	10	0	0	13.5	13	13	13
(L,M,M)	13	12	10	10	0	0	12.6	13	13	13
(L,M,RL)	13	12	11	10	5	5	14.7	13	13	13
(L,M,L)	13	13	11	11	0	0	17.5	13	13	13
(L,L,H)	14	13	11	10	0	0	15.5	20	20	20
(L,L,RH)	14	13	11	11	0	0	14.5	20	20	20
(L,L,M)	14	13	11	11	0	0	14.2	20	20	20
(L,L,RL)	13	13	11	11	5	5	14.7	20	20	20
(L,L,L)	13	12	11	10	0	0	17.8	20	20	20

BIBLIOGRAPHY

Baker, K. R. (1974), Introduction to Sequencing and Scheduling, Johnson Wiley & Sons, N.Y., 1974

Bannister, J. A. and K. S. Trivedi (1983), "Task allocation in fault-tolerant distributed systems," Acta Information, Vol. 20, 1983.

Bellman, R., A. O. Esogbue, and I. Nabeshima (1982), Mathematical Aspects of Scheduling and Applications, Pergamon Press, 1982

Bertossi, A. A. and M. A. Bonuccelli (1985), "A polynomial feasibility test for preemptive periodic scheduling of unrelated processors", Discrete Applied Math., Vol. 12, 1985, pp. 195-201-201

Brooke A., D. Kendrick, A. Meeraus, GAMS: A User Guide, The Scientific Press, 1988.

Bruno, J., E. G. Coffman Jr. and R. Sethi (1974a), "Algorithm for minimizing mean flow time," Information Processing 74, North-Holland, Amsterdam, 1974, pp. 504-510.

Buten, R. E. and V. Y. Shen (1973), "A scheduling model for computer systems with two classes of processors," in Proc. 1973 Sagamore Computer Conference on Parallel Processing, Springer-Verlag, N. Y., 1973, pp. 130-38.

Bruno, J., E. G. Coffman Jr. and Sethi (1974b), "Scheduling independent tasks to reduce mean finishing time," Communications of ACM, vol. 17, July 1974, pp. 382-387.

Carreno, Jose Juan (1990), "Economic lot scheduling for multiple products on parallel identical processors," Management Science, vol. 36, no. 3, March 1990, pp. 348-358.358.

Chen, Guan-Ing and Ten-Hwang Lai (1988), "Preemptive scheduling of indepdent tasks on a hypercube," Information Processing Letters 28, 1988, North Holland, pp. 201-206.

Coffman, E. G. Jr. (eds.) (1976), Computers and Task/Shop Schduling Theory, John Wiley and Sons, N. Y., 1976.

Conway, R. W., W. L. Maxwell, and L. W. Miller (1967), Theory of Scheduling, Addison-Wesley, Reading, MA., 1976.

Davis, E. and J. M. Jaffe (1981), "Algorithm for Scheduling tasks on unrelated parallel processors," Journal of ACM, Vol. 28, no. 4, Oct. 1981, pp. 721-736.

De, P. and T. E. Morton (1980), "Scheduling to minimize makespan on unequal parallel processors," Decision Science, vol. 11, 1980, pp. 586-602.

Dempster, M. A. H., J. K. Lenstra, and A H. G. Rinnooy Kan (eds.) (1982), Deterministic and Stochastic Scheduling, Reidel, Dordrecht, 1982.

Dror, Mushe, Helman I. Stern and Jan Karel Lenstra (1987). "Parallel machine scheduling: Processing Rates dependent on number of tasks in operations", Management Science, vol. 33, no. 8, Aug. 1987, pp. 1001-1009.

Fisher, M. L. (1980), "Worst case analysis of heuristic algorithms," Management Science, vol. 26, 1980, pp. 1-17.

French, S. (1982), Sequencing and Scheduling: An Introduction to the Mathematics of the Job Shop, Herwood, Chichester, 1982.

Garey, M. R., R. L. Graham and D. S. Johnson (1978), "Performance guarantees for scheduling algorithm," Operations Research, Vol. 26, 1978, pp. 3-21.

Garey, M. R. and D. S. Johnson (1975), "Complexity results for multiprocessor scheduling under resource constraints," SIAM J. Comput., vol. 4, 1975, pp. 397-411.

Garey, M. R. and D. S. Johnson (1978), "Strong NP-completeness results: Motivation, example, and implications," Journal of ACM, vol. 25, no. 3, July 1978, pp. 499-508.

Garey, M. R. and D. S. Johnson (1979), Computers and Intractability: A Guide to the Theory of NP-complete, Freeman, San Francisco, 1979.

Gonzales, M. J. Jr. (1977), "Deterministic processor scheduling," Comput. Survey vol. 9, 1977, pp. 173-204.

Gonzales, T., E. L. Lawler and S. Sahni (1990), "Optimal preemptive scheduling of two unrelated parallel processors," ORSA. Journal of Computing, Summer 1990, pp. 219-224.

Graham, R. L., E. L. lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan (1979), "Optimization and approximation in deterministic sequencing and scheduling: a survey," Ann. Discrete Math, vol. 5, 1979, pp. 287-326.

Graves, S. C., (1981), "A review of production scheduling," Operations Research, vol. 29, 1981, pp. 646-675.

Hansen, J. V. and W. C. Giauque (1986), "Task allocation in distributed processing systems," Operations Research Letters, vol. 5, no. 3, Aug. 1986, pp. 137-143.

Hax, A. C., Dan Candea (1984), Production and Inventory Management, chapter 5, Prentice-Hall.

Herrbach, L. A., J. Y. T. Leung (1990), "Preemptive Scheduling of equal length jobs on two machines to minimize mean flow time," Operations Research, vol. 38, no. 3, pp. 487-494.

Horowitz, E. and S. Sahni (1976), "Exact and Approximate algorithms for scheduling nonidentical processors," Journal of ACM, vol. 23, no. 2, April 1976, pp. 317-327.

Ibarra, O. H. and C. E. Kim (1977), "Heuristic algorithms for scheduling independent tasks on nonidentical processors," Journal of ACM, vol. 24, no. 2, April 1977, pp. 280-289.

Jaffe, J. M. (1980), "Bounds on the scheduling of typed task systems," SIAM Journal on Computing., vol. 15, no. 3, August 1980, pp. 541-551.

Janiak, Adam (1989), "Minimization of resource consumption under a given deadline in the two-processor flow-shop scheduling problem," Information processing letters 32, 1989, North Holland, pp 101-112.

Johnson, D. S. (1983), "The NP-completeness column: an ongoing guide," Journal of Algorithms, vol.4, 1983, pp. 189-203

Kindervater, G. A. P., J. K. Lenstra, A. H. G. Rinnooy Kan (1989), "Perspectives on parallel computing," Operations Research, vol. 4, December 1989, pp 985-990.

Kubale, M. (1987), " The complexity of scheduling independent two-processor tasks on dedicated processors," Information Processing Letters, vol. 24, Feb. 1987, pp. 141-147.

Kunde, Manfred, Michael A. Lanston, and Jing-Ming Lin (1988), "On a special case of a uniform processor scheduling," Journal of Algorithm, vol. 9, 1988, pp 287-296.296.

Lageweg, B. J., J. K. Lenstra, E. L. Lawler and A. H. G. Rinnooy Kan (1982), "Computer-aided complexity classification of combinatorial problems," Communications of ACM, vol. 25, no. 11, Nov. 1982, pp. 817-822.

Lawler, E. L. (1976), Combinatorial Optimazation: Network and Matroids, Holt, Rinehart, and Winston, 1976.

Lawler, E. L. and L. Labetoulle (1978), "On preemptive scheduling of unrelated parallel processors by linear programming," Journal of ACM, vol. 25, no. 4, Oct. 1978, pp. 612-619.

Lawler, E. L., J. K. Lenstra, and A. H. G. Rinnooy Kan (1980), " Generating all maximal independent sets: NP-hardness and polynomial time algorithms," SIAM Journal on Computing, vol. 9, no. 4, Aug. 1980, pp. 558-565.

Lawler, E. L., J. K. Lenstra, and A. H. G. Rinnooy Kan (1982), " Recent developments in deterministic sequencing and scheduling: a survey." in M. A. H. Dempster et al. (eds.) Deterministic and Stochastic Scheduling, 35-73, D. Reidel Publishing Company, 1982.

Lenstra, J. K. (1977), Sequencing by Enumerative Methods, Mathematical Center Tracts 69, Mathematisch Centrum, Amesterdam, 1977.

Lenstra, J. K. and A. H. G. Rinnooy Kan (1983), "Scheduling theory since 1981: an annotated bibliography," Technical Report BW 188/83, Mathematical Center, Amesterdam, 1983.

Lee, Fung F., "Partitioning of regular computation on multiprocessor systems," Journal of parallel and distributed computing," vol. 9, 1990, pp. 312-317.

Liu, J. W. S. and C. L. Liu (1978), "Performance analysis of multiprocessor systems containing functionally dedicated procesors," Acta Informatica, vol. 10, 1978, pp. 95-104.

Martel, C., (1985), "Preemptive Scheduling to minimize maximum completion time on uniform processors with memory constraints," Operations Research, vol. 33, no. 6, pp. 1360-1380.

Martel, C., (1988), "A parallel algorothm for preemptive scheduling of uniform machines," Journal of Parallel and Distributed Computing, vol. 5, pp. 700-715.

Nillson, N. J. (1971), Problem-Solving Methods in Artificial Intelligence, New York, McGraw-Hill, 1971.

Nillson, N. J. (1980), Principles of Artificial Intelligence, Palo Alto, Calif., Tioga, 1980.

OR/MS Today, pp.35-47, Oct. 1990

Parker G., R. Rardin, Discrete Optimization, Academic Press, 1988

Pearl, J. (1984), Heuristics: Intelligent Search Strategies for Computer Problem Solving, The Addison-Wesley Series in Artificial Intelligence, 1984.

Pohl, I. (1970), "First results on the effect of errors in heuristic search," Machine Intelligence, vol. 5, 1970, pp. 219-236.

Potts, C. N. (1985), "Analysis of a linear programming heustic for scheduling unrelated parallel machines," Discrete Applied March., vol. 10, 1985, pp. 155-164.

Price, C. C. and U. W. Pooch (1982), "Search techniques for a nonlinear multiprocessor scheduling problem," Naval Research Logistics Quarterly, vol. 29, no. 2, June 1982, pp. 213-233.

Rinnooy Kan, A. H. G. (1976), Machine Scheduling Problems: Classification, Complexity and Computations, Nijhoff, The Hague, 1976

Sahni, S. (1974) "Computational related problems," SIAM Journal on Computing, vol. 3, no. 4, Dec. 1974, pp. 262-279.

Shen, C. C. and W. H. Tsai (1985), "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," IEEE Trans. on Computers, vol. C-34, no. 3, March 1985, pp. 197-203.

Simons, Barbara B. and Manfred K. Warmuth (1989), "A fast algorithm for multiprocessor scheduling for unit-length task," SIAM Journal of Computing, vol. 18, no. 4, Aug 1989, pp 690-710.

Sinclair, J. B. (1987), "Efficient computation of optimal assignments for distributed tasks," Journal of Parallel and Distributed Computing, vol. 4, 1987, pp. 342-362.

Slagle, J. R. and P. Bursky (1968), "Experiments with a multipoupose theorem-proving heuristic program," Journal of the Association for Computing Machinery, vol. 15, 1968, pp. 85-99.

So, Kut C. (1990), "Some heuristics for scheduling jobs on parallel machines with setup," Management Science, vol. 36, no. 4, April 1990, pp. 467-475.

Spinrad, J. (1985), "Worst-case analysis of a scheduling algorithm," Operations Research Letters, vol. 4, no. 1, May 1985, pp. 9-11.

Winston W, Introduction to Mathematical Programming: Applications and Algorithms, PWS-KENT Publishing Co., 1991

Xu, S. H. (1990),"Minimizing the expected weighted flow time on uniform processors," Penn State U., TIMS/ORSA Las vagas, May 1990.