Εθνικο Μετσοβιο Πολυτεχνειο
Τμημα Ηλεκτρολογων Μηχανικων και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων
Εργαστηριο Λογικης και Επιστημης Υπολογιστων

Προσεγγιστικοί Αλγόριθμοι Δρομολόγησης Παραλληλοποιήσιμων Εργασιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Λεωνίδα Π. Τσεπενέκα

Επιβλέπων: Δημήτρης Φωτάκης
Επίκουρος Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 2016

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΛΟΓΙΚΗΣ ΚΑΙ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

Προσεγγιστικοί Αλγόριθμοι Δρομολόγησης Παραλληλοποιήσιμων Εργασιών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Λεωνίδα Π. Τσεπενέκα

Επιβλέπων: Δημήτρης Φωτάκης
Επίκουρος Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή στις 14 Σεπτεμβρίου 2016.

.....	.....	.....
Δημήτρης Φωτάκης	Ιωάννης Μήλης	Αριστείδης Παγουρτζής
Επίκουρος Καθηγητής Ε.Μ.Π.	Καθηγητής Ο.Π.Α.	Αναπληρωτής Καθηγητής Ε.Μ.Π .

Αθήνα, Σεπτέμβριος 2016

.....

Λεωνίδας Π. Τσεπενέκας
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή αυτής της εργασίας, κ. Δημήτρη Φωτάκη, για την εμπιστοσύνη του, τη συνεχή καθοδήγηση και ενθάρρυνση και για τη δυνατότητα που μου έδωσε να γνωρίσω εκ των έσω την ερευνητική διαδικασία. Ο ενθουσιασμός με τον οποίο αντιμετωπίζει τόσο το διδακτικό όσο και το ερευνητικό κομμάτι της δουλειάς του, υπήρξε ένας από τους βασικότερους παράγοντες που με ενέπνευσαν να ασχοληθώ με το πεδίο των Αλγορίθμων.

Επιπλέον, οφείλω πολλά στον Ορέστη Παπαδιγενόπουλο, με τον οποίο το τελευταίο χρόνο είχαμε μια εξαιρετική συνεργασία. Οι συμβουλές και η βοήθεια που μου προσέφερε όντας εμπειρότερος, ήταν καθοριστικής σημασίας για την περάτωση της εν λόγω εργασίας και τα αποτελέσματα αυτής δεν θα είχαν προκύψει χωρίς τη συμμετοχή του.

Ακόμη, νιώθω την ανάγκη να ευχαριστήσω από καρδιάς τον Αλέξανδρο, το Γιάννη, το Σπύρο και το Πέτρο για τα 6 καταπληκτικά χρόνια που περάσαμε μαζί στη Σχολή. Σε αυτή τη παρέα χρωστάω τις υπέροχες αναμνήσεις των φοιτητικών μου χρόνων.

Ιδιαίτερα θα ήθελα να ευχαριστήσω την οικογενειά μου για την αμέριστη στήριξη που μου έδειξαν σε όλη τη διάρκεια των σπουδών μου. Κάθε φορά που τους χρειάστηκα ήταν δίπλα μου και σίγουρα χωρίς αυτούς δεν θα είχα καταφέρει τίποτα. Η προσφορά τους στη μέχρι τώρα διαδρομή μου είναι ανεκτίμητη.

Τέλος, ένα μεγάλο ευχαριστώ στην αγαπημένη μου Δήμητρα που τόσα χρόνια ήταν πάντα κοντά μου, και στα εύκολα και στα δύσκολα.

Περίληψη

Μια τυπική υπόθεση στα κλασσικά προβλήματα χρονοδρομολόγησης είναι ότι οι εργασίες, χρησιμοποιούν μόνο μια επεξεραστική μονάδα για την εκτέλεσή τους. Παρόλα αυτά, υπάρχει πληθώρα προβλημάτων που εμφανίζονται σε πολλούς και διαφορετικούς τομείς, στα οποία η εν λόγω υπόθεση δεν είναι επαρκής για να μοντελοποιήσει τις ιδιαίτερες απαιτήσεις που παρουσιάζουν. Σε αυτές τις περιπτώσεις θα πρέπει να επεκτείνουμε το κλασικό μοντέλο, επιτρέποντας σε μια εργασία να εκτελείται ταυτόχρονα σε περισσότερους του ενός επεξεργαστές και μάλιστα ενιαία. Αυτό σημαίνει ότι η εργασία θα έχει κοινό χρόνο εκκίνησης και ολοκλήρωσης σε όλους τους επεξεργαστές που της έχουν ανατεθεί και μάλιστα ο χρόνος εκτέλεσης της θα είναι συνάρτηση αυτών. Στην αλγοριθμική βιβλιογραφία προβλήματα της παραπάνω μορφής συναντώνται ως malleable ή multiprocessor ή parallelizable job scheduling.

Όπως και στο κλασικό μοντέλο μιας εργασίας σε μια μηχανή, έτσι και εδώ υπάρχει η ανάγκη σχεδίασης αποδοτικών αλγορίθμων. Στη διπλωματική αυτή, μελετάμε το πρόβλημα της χρονοδρομολόγησης σε τέτοια μοντέλα από τη σκοπιά της θεωρίας δρομολόγησης και των προσεγγιστικών αλγορίθμων. Ξεκινάμε παρουσιάζοντας τα πιο γνωστά αποτελέσματα, επεκτείνοντας παράλληλα μερικά από αυτά. Η βασική συνεισφορά μας είναι η δημιουργία ενός μοντέλου για malleable job scheduling που γενικεύει την ιδέα του uniform machine scheduling και η παρουσίαση προσεγγιστικών αλγορίθμων σταθερού παράγοντα για αυτό.

Λέξεις-Κλειδιά: Χρονοδρομολόγηση, Προσεγγιστικοί αλγόριθμοι, Ανάθεση πόρων, Malleable/Parallelizable/Multiprocessor εργασίες, Μοντέλο μίας εργασίας σε πολλές μηχανές

Abstract

A typical hypothesis in classical scheduling is that a job can be processed by only one machine at a time. However, in many problems arising in a variety of fields, this may not be an adequate model. In this case, we must extend the classical model, allowing a job to use more than one machines for its execution. The processing time of a job is a a function of the machines allocated to it. Also, all the machines allocated to a job are required to execute the job in unison. That is, they are required to start and finish the job at the same time. In algorithmic literature, the above are mentioned as malleable or multiprocessor or parallelizable job scheduling problems.

In this diploma thesis, we study scheduling problems of that nature from the viewpoint of approximation algorithms and the typical scheduling theory. We present several known results and we also slightly extend some of them. Our main contribution is a novel malleable job scheduling model, that generalizes the idea of the well known uniform machine scheduling, and some constant factor approximation algorithms for it.

Keywords: Scheduling, Approximation Algorithms, Resource allocation, Malleable/-Parallelizable/Multiprocessor jobs, One-job-multiple-machines model

Contents

List of Figures

Chapter 1

Introduction

In this diploma thesis we are dealing with scheduling problems. But what exactly is a scheduling problem? Informally, one may think of scheduling as a family of problems, in which a set of jobs must be assigned for execution in a set of processing resources, with the jobs competing over time for the use of these resources. The objective is to allocate the available resources to jobs, in such a way that a particular criterion is met and also all kinds of different constraints, unique to each problem, are satisfied.

Scheduling problems are of great importance. First of all, they have way too many applications in real life situations. Perhaps, their most important application is in computer systems. Nowadays, all computer systems, from personal computers to supercomputer clusters, consist of many individual processing units and also have to satisfy thousands of applications. So the need for efficient schedulers is crucial, in order for these systems to meet their performance standards. Apart from computer systems, scheduling problems are also relevant in many other different fields, such as human resource planning, production planing, transportation and even formal language theory [1]. Beside their many applications, scheduling problems are also important from a purely algorithmic viewpoint. They have a rich and intriguing combinatorial structure and their study has led to some central, as well as mathematically elegant results for the field.

Although the algorithmic literature concerning scheduling problems is vast, the majority of the studied models are based on the following hypothesis. Every job that needs to be executed, can only use one processing unit at a time. This is the so-called one-job-one-machine model. However, in many settings arising in a variety of fields, this may not be an adequate model. There are many real life situations in which a job may use or even demand multiple processors for its execution. Thus, the classical model must be extended to this direction, in order to capture the special requirements and characteristics of such cases. This is the so-called one-job-multiple-machines model.

The aforementioned model, although equally important, is not very well studied in the existing literature and that drove us to work on it.

We now present some of the most typical examples of problems, in which jobs may require several resources at once for their processing. To begin with, there are many programming frameworks for modern multicore computer systems that allow the parallelization of particular tasks, in order to exploit the inherent parallel capabilities of the underlying architecture. Parallelizing a job simply stands for using more than one computational cores for its execution and of course in parallel. In such systems, the more processors a job uses simultaneously, the smaller its processing time. That happens because the workload of the job is distributed among the processors alloted to it. Parallel architectures and frameworks that support this type of job parallelization, are very popular lately, because of the better performance guarantees they offer. Some examples of such frameworks that take into account the underlying parallel architecture, and try to exploit it via job parallelization are MPI [2], OpenMP [3] and Cilk [4]. It is quite clear, that the classical scheduling algorithms that have been developed for parallel machines of the nineties are not well adapted to these new execution supports. Thus, the study of the one-job-multiple-machines model is necessary for providing better schedulers for such systems.

The real life applications of the one-job-multiple-machines model are not limited only in multiprocessor computer systems. Another typical example is the berth allocation or berth scheduling problem, which is a well known $\mathcal{NP}$-complete problem in operations research. In that problem we have large vessels that need to be loaded or unloaded and they may occupy several berths at once for their processing. Here, the berths play the role of the machines, and it may very easily be the case that a large vessel cannot be satisfied by only one berth. A similar problem involves human workforce planning and is the following. There is a group of workers, each with a different set of skills. There is also a set of tasks that the group should complete before they leave work. But each task demands for its execution specific skills, that not all workers posses. Thus, the workers must find the best way to form teams, which will work on different tasks, such that each team possesses in total the skills required for the task assigned to it. And of course the final goal is to complete all tasks as soon as possible in order to leave work early. In this problem, every member of the group corresponds to a computational unit, and a task may need more than one workers for its completion, because a single worker may not master all the appropriate skills.

Finally, we have to mention that the classical one-job-one-machine scheduling model is just a special case of the one-job-multiple-machines model that we are interested in. Thus, the study of the latter is more difficult and more demanding, but nonetheless necessary, as well as fascinating and thought-provoking.

1.1 Notation and Definitions

In this section, we are going to present all the basic notation and definitions, needed for understanding the rest of this thesis. We begin by examining the above for the typical one-job-one-machine scheduling model, and then continue by extending it to the more complex one-job-multiple-machines case.

1.1.1 Formal Definition of a Scheduling Problem

In every classical scheduling problem we have a set of jobs $\mathcal{J}$, and a set of machines $\mathcal{M}$. We use $n = |\mathcal{J}|$ and $m = |\mathcal{M}|$ to denote the number of jobs and machines respectively. In the rest of the thesis we are going to use the terms "jobs" and "tasks", as well as the terms "machines" and "processors" interchangeably. Every job $j \in \mathcal{J}$ must be assigned for execution to some machine $i \in \mathcal{M}$, and its processing time there is denoted by $p_{i,j}$. That is, if $j \in \mathcal{J}$ ends up running on machine $i \in \mathcal{M}$, it will burden that machine for $p_{i,j} > 0$ units of time.

A schedule is a function $f : \mathcal{J} \to \mathcal{M} \times \mathbb{R}$, that maps every job to a machine-time pair. That is, the job will use that machine for its processing, starting at the time indicated in the pair. A schedule is called **feasible**, if at any point in time every machine executes at most one job.

In many scheduling problems there are also some other special constraints that need to be satisfied, apart from the feasibility described above. We are going to mention only one type of such constraints, which will be later encountered in our study and is very common in many problems. That is the precedence constraints. In a few words, we say that a job i precedes job j, denoted by $i \prec j$, if the execution of the latter can start only after the completion of the former job. Usually the precedence relations of jobs are depicted as a *Directed Acyclic Graph*. An example is following.

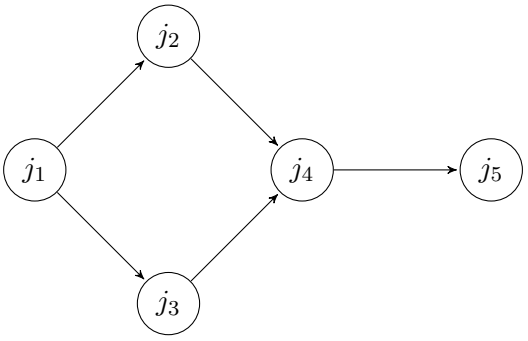


FIGURE 1.1: A graph representing the precedence constraints among jobs.

As we can see, there is a node for each job and an edge $i \to j$ suggests that, job j cannot start its execution, unless i is done processing.

Another important characteristic of some scheduling problems is the notion of preemption. In problems that do not allow preemption, once a job $j \in \mathcal{J}$ begins processing on some machine $i \in \mathcal{M}$, it will occupy this machine consecutively for $p_{i,j}$ units of time. On the other hand, in problems with the preemptive feature, the execution of a job can be interrupted at any time and resumed later. For example, suppose that a job has to run for 7 time units on the machine that it is assigned to. In a preemptive problem initially it can occupy the machine for 4 time units, then be interrupted so that another job can use the machine, and finally occupy it again for another 3 time units until its completion.

1.1.2 Machine Environment

Perhaps, the most important characteristic of a scheduling problem is the machine environment, in which it takes place. The machine environment defines the fundamental properties of the available machines. In the previous section, we defined the processing time of job $j \in \mathcal{J}$, when executed on machine $i \in \mathcal{M}$ to be $p_{i,j}$. We are now going to see, that this quantity is actually determined by the intrinsic characteristics of $\mathcal{M}$. In the scheduling literature there are three well studied types of environments:

- **Identical machines**, denoted by P. In this case we assume that all m machines have exactly the same computational capabilities. In other worlds, each job perceives all machines in the same way, and thus its processing time is independent of the machine it will use. So, $\forall i \in \mathcal{M} : \ p_{i,j} = p_j$ and we can simplify the notation by just writing p_j.
- **Uniform machines**, denoted by Q. In this case we assume that every machine has its own computational speed, say s_i. That is, each machine can process at a different rate. If the workload of a job is w_j, then the processing time of j in machine $i \in \mathcal{M}$ is defined as $p_{i,j} = \frac{w_j}{s_i}$. Observe that identical machines are just a special case of uniform machines, in which all processors are of the same speed.
- **Unrelated machines**, denoted by R. This is the most general machine environment and constitutes an extension of both the two mentioned earlier. In this case, we assume no relation at all between the processing time of a job and the machine it is scheduled on. Thus we simply write $p_{i,j}$. Any algorithm designed for this case, will provide results for the above environments as well.

1.1.3 Objective Functions and Metrics

All scheduling problems that we study in this work are optimization problems. That means that any algorithm concerning them, apart from satisfying the required feasibility

constraints, also tries to either maximize or minimize a desired objective function. To be more precise, the vast majority of scheduling problems are minimization problems, and now we will provide a short review of some of the most usual metrics considered in the literature. Before we begin, we need a bit more notation. Let us denote by C_j the completion time of job $j \in \mathcal{J}$ in a given feasible schedule.

- **The Makespan objective**. Formally defined as $C_{max} = \max_{j \in \mathcal{J}} C_j$. The makespan is the length of the produced schedule. It can also be seen, as the latest time that a machine is still active, or the latest time that a job is still undergoing execution. Historically it was the first metric considered for scheduling problems. Makespan is an ideal metric for situations that we care a lot about the total length of the whole execution process. One such case is when we have a single user system and we want to service that user as soon as possible.
- **The Sum of Completion Times** or $\sum_{j \in \mathcal{J}} C_j$. Observe that minimizing this objective is equivalent to the problem of minimizing the average completion time, since the average completion time just rescales the objective function for each feasible solution by a factor of $1/n$. This metric is suitable for multi-user systems with tasks from many different agents. In that case we would like to minimize the average completion time and not just satisfy particular users.
- **The Sum of Weighted Completion Times**. Suppose that each job $j \in \mathcal{J}$ has a positive weight w_j, indicating its significance. Then we aim at minimizing $\sum_{j \in \mathcal{J}} w_j \cdot C_j$. The purpose of this metric is almost the same as the one described above, with just a slight difference. In this case, jobs with higher priority ought to be scheduled earlier.

The objective functions described above are just the ones we are going to use throughout this thesis. There are actually many more considered in the literature [5].

1.1.4 The Standard Three Field Notation

In the beginning of the eighties Graham, Lawler, Lenstra and Kan [6] developed a very concrete and elegant notation for describing scheduling problems. This notation is now widely accepted and used. It consists of three independent fields, of the following form $A|B|C$, each referring to a different aspect of the problem.

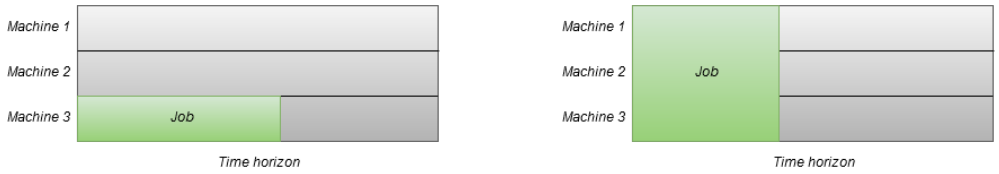
- The first field A, describes the machine environment. For example, if the first field contains R, then we are dealing with an unrelated machines problem.

- The second field B, describes all types of special constraints concerning the problem. For example, if we are dealing with a problem that allows preemption, we have to state that in the second field.

- The third field C, contains the objective function we want to minimize.

1.1.5 Formal Definition of the One-Job-Multiple-Machines Model

We are now going to see how all the above notation and definitions adjust, in the case of the one-job-multiple-machines model. First of all, again we have a set of jobs $\mathcal{J}$ that need to be scheduled using a set of machines $\mathcal{M}$, with $n = |\mathcal{J}|$, $m = |\mathcal{M}|$. The only difference is that now, a job $j \in \mathcal{J}$ can be assigned for execution to some set $S \subseteq \mathcal{M}$ of machines, with processing time denoted by $p_j(S)$ and $S \neq \emptyset$. When $j \in \mathcal{J}$ runs on $S \subseteq \mathcal{M}$, it occupies all machines of S in unison. That is, j has the same starting and finishing time in all of those machines. For better understanding, we present a visual example.



(A) One-job-one-machine model (B) One-job-multiple-machines model

FIGURE 1.2: Job execution under the new model.

A schedule in this case is a function $f : \mathcal{J} \to 2^{\mathcal{M}} \times \mathbb{R}$, that maps every job to a set of machines-time pair. Again, a schedule is called **feasible**, if at any point in time every machine executes at most one job. The notion of precedence constraints is relevant for this model too, maintaining exactly the same form. Also, problems with preemption can still arise in this new model. If a problem allows preemption, then after a job $j \in \mathcal{J}$ is assigned for execution to a set $S \subseteq \mathcal{M}$, it can be interrupted at any time, leaving the machines of S available for other jobs, and resume at a later point possibly occupying a different set, for the rest of its processing. Finally, the aforementioned objective functions hold in this case too, and our study is totally based on them.

The most important and interesting fact is to see how the three machine environments presented earlier, are generalized for this model. The intrinsic aspects of the machine environment again define the difficulty and the unique structure of the problem.

- **The identical machines setting**. In this case, all machines of $\mathcal{M}$ have the same computational capabilities. So, the processing time of each task is a known function of just the number of processors alloted to it. For simplicity, we denote

by $p_j(n)$ the processing time function of job $j \in \mathcal{J}$, where n is the variable used for the number of machines j uses during its execution. In order for any problem to make sense in this kind of setting, $p_j(n)$ ought to be a non-increasing function of n. The more processors a job occupies, the more its workload is distributed among them and thus the smaller its processing time. The goal is to find for each task $j \in \mathcal{J}$, an allotment of processors n_j and an overall schedule assigning the tasks to the machines, which minimizes the desired objective function. Usually in the literature, problems in such settings are referenced as *Malleable Job Scheduling*. Finally observe, that an algorithm designed for this case, can provide results for the classical identical machine scheduling problem too, by assuming $\forall n : p_j(n) = p_j$.

- **The uniform machines setting**. In this case, each machine processes at a different rate and so has its own speed s_i. The execution time of a task depends on the amount of speed allocated to it and thus is represented as a function $p_j(s)$, where s denotes the total speed it utilizes. For example, if job $j \in \mathcal{J}$ runs on two machines, one with speed 4 and the other with speed 6, its processing time will be $p_j(10)$. Again, we assume that $\forall j \in \mathcal{J}$, $p_j(s)$ is a non-increasing function in s. Observe, that again the identical setting is just a special case of this environment, because we can assume that identical machines are just uniform machines with the same speed. Problems of this kind have never been studied in the literature, and to the extend of our knowledge we are the first to deal with them.

- **The unrelated machines setting**. This case is the more general, as it constitutes an extension of both the previous mentioned. In problems of such nature, we assume no relation at all between the processing time of a job j and the set of processors it will occupy. So, $\forall S \subseteq \mathcal{M}, \forall j \in \mathcal{J}$ there is a value $p_j(S)$ representing the corresponding processing time. Such problems appear in the literature as *General Parallel Task Scheduling* or *Multiprocessor Job Scheduling*. Observe again, that an algorithm designed for this case, can provide results for the classical unrelated machine scheduling problem too, by assuming that $\forall j \in \mathcal{J}$:

$$p_j(S) = \begin{cases} p_{i,j} & \text{if } S = \{s_i\} \\ +\infty & \text{otherwise} \end{cases}.$$

Finally we must mention, that there is no globally accepted and widespread used generalization of the three field notation we presented earlier, for the one-job-multiple-machines model. Each problem has to be presented analytically here.

1.2 Overview of Approximation Algorithms

All optimization problems considered in this thesis are known to be $\mathcal{NP}$-hard. Perhaps, the most common approach in dealing with $\mathcal{NP}$-hardness is via approximation algorithms. Approximation algorithms aim at finding a near optimal solution in polynomial time. The quality of a given solution and consequently the quality of an approximation algorithm is measured via the *approximation ratio* or *guarantee*. We formally define the approximation ratio of an algorithm for a minimization problem as follows.

Definition 1.1. An algorithm A for a minimization problem Π is called an ρ-approximation, with $\rho > 1$, if for every instance I of Π we have $OPT(I) \leq SOL_A(I) \leq \rho \cdot OPT(I)$. $OPT(I)$ is the value of the optimal solution for instance I and $SOL_A(I)$ is the value returned by the algorithm for the same instance.

There is a very similar definition for approximating maximization problems. In general ρ can be a function of the size of the input, that is $\rho = f(|I|)$. The goal when designing an approximation algorithm is to ensure the lowest possible ratio. A constant factor is the best we can wish for, although sometimes this is not possible and we have to settle for logarithmic, linear or other function of the input size ratios.

1.3 Organization of the Thesis

In *Chapter 2*, we study the *Malleable Job Scheduling Problem*. We begin by presenting a 3/2-approximation algorithm for the makespan objective. Furthermore, we briefly discuss the same problem under the assumption of identical jobs. We continue by surveying a 2-approximation algorithm for the problem of minimizing the average completion time. Next, we examine a problem that at first sight doesn't seem to have a connection with our model. We present some results concerning the minimization of makespan for it, and show that it can actually stand as a generalization of the malleable setting. Finally, as a contribution we extend these existing results in the case of the average weighted completion time objective.

The purpose of *Chapter 3* is to introduce the more general *Multiprocessor Job Scheduling Model*. All problems considered in this chapter are dealing with the makespan objective. We begin by presenting an inapproximability result for this model, and two algorithms for some special cases. Finally, we study a central result of the field, which consists of a $PTAS$ in the case where the number of machines is fixed.

Chapter 4 contains our research work. Initially, we introduce a new model, regarding malleable scheduling in uniform machines. We then concentrate on some special cases of the model and provide approximation algorithms for them.

Chapter 2

Malleable Job Scheduling

The purpose of this chapter is to introduce the reader to the problem of scheduling malleable jobs. For this reason, we present a collection of the most important algorithms and results, concerning all different types of objective functions. We begin by formally defining the problem.

2.1 Problem Definition

In the *Malleable Job Scheduling Problem* we have a set $\mathcal{M}$ of $m = |\mathcal{M}|$ identical processors, as well as a set $\mathcal{J}$ of $n = |\mathcal{J}|$ independent malleable tasks. A malleable task is a computational unit that may be executed using several processors, and its processing time depends only on the number of them. A set of jobs is called independent, if there are no precedence constraints among them. More formally, a task $j \in \mathcal{J}$ has an execution time $p_j(n_j)$, which is a known function of the number of machines n_j, with $1 \leq n_j \leq m$, that it uses during its processing. All processors alloted to a job are required to execute that job in unison. That is, the n_j processors should start the job at the same time st_j, and process it non-preemptively until time $st_j + p_j(n_j)$. Such a job can also be observed as a malleable rectangle, whose width $p_j(n_j)$ stretches along a time-axis and whose height n_j stretches along a machine-axis, although the n_j machines are not required to be consecutive with respect to some ordering.

Computing a feasible schedule for such a problem, consists in finding for each job $j \in \mathcal{J}$ a machine allotment n_j and a starting time st_j satisfying the following constraints:

- At any time t in the schedule, all machines execute at most one job.
- The total processor consumption at any moment t doesn't exceed m. That is:

$$\sum_{\{j \in \mathcal{J} \mid st_j \leq t \leq st_j + p_j(n_j)\}} n_j \leq m$$

Before we continue, we need to define a very important function and then state some very crucial assumptions used in most of the literature, which we will take into account throughout this chapter.

Definition 2.1 (Work function)**.** For every job $j \in \mathcal{J}$, we define its *work function* as, $w_j(n_j) = n_j \cdot p_j(n_j)$. Geometrically, this function corresponds to the area of the aforementioned malleable rectangle.

Definition 2.2 (Monotony Hypothesis)**.** A set of jobs $\mathcal{J}$ is said to be monotonic, if for each job $j \in \mathcal{J}$, $p_j(n_j)$ is a non-increasing and $w_j(n_j)$ is a non-decreasing function of n_j.

Observe, that the assumption concerning the execution time function of a job is quite reasonable to make. The more processors a job utilizes, the more its workload is distributed and thus the smaller its processing time. However, this not actually a restriction, since every such function $p_j(n_j)$ can be transformed to fulfil the monotony hypothesis by setting:

$$p'_j(n_j) = \min_{1 \leq j \leq n_j} p_j(n_j)$$

This new function is clearly non-increasing. We simply make the optimal number of machines do the work and leave the rest idle. Note also, that this transformation doesn't affect the optimal solution of any problem.

Furthermore, the assumption about the non-decreasing work is very meaningful, since it depicts the typical behaviour of parallel applications. From that point of view, this monotonic hypothesis may be interpreted by the well-know Brent's lemma [7], which states that although the parallel execution of a job achieves some speedup, that cannot be superlinear due to the unavoidable communication overhead.

2.2 Minimizing the Makespan

We begin by examining the problem under the makespan minimization objective. This form of the problem is actually well studied in the literature with some very interesting results regarding it. Perhaps, the most important is a *PTAS* by Jansen and Porkolab [8]. Their result cannot be further improved to a fully polynomial time scheme since the problem is strongly NP-hard [9]. The complexity of their scheme, although polynomial in the size of the input contains some large exponential factors, which render it useless for any practical application. Thus, low complexity, constant factor approximation algorithms are still relevant. One of the first such algorithms was a 2-approximation proposed by Turek et al. [10], based on some simple discrete resource allocation techniques, as well as a classical list scheduling argument. In our study, we

prefer to present more extensively the best known, in terms of minimum ratio, constant factor approximation, which is a $\frac{3}{2} + \epsilon$ due to Mounie et al.[11].

Before we continue with the analysis of the algorithm, we must give another useful definition.

Definition 2.3 (Canonical number of processors)**.** For every job $j \in \mathcal{J}$, we define its canonical number of processors $\gamma(j, h)$, as the minimum number of machines needed, in order for j to have $p_j(\gamma(j, h)) \leq h$. Even if by using all m machines, j cannot have an execution time less or equal to h, then by convention we write $\gamma(j, h) = +\infty$.

In addition, observe that $w_j(\gamma(j, h))$ is also the minimum work needed to execute j in time at most h, if of course $\gamma(j, h) < +\infty$.

2.2.1 Description of the Algorithm

The algorithm of [11] is based on a dual approximation technique [12]. A λ-dual approximation procedure for a scheduling problem can be described as follows:

A λ-approximation procedure

1: Take as input a target makespan d.
2: Deliver a schedule with makespan $C \leq \lambda d$, or answer correctly that no feasible schedule of length at most d exists for the problem.

Using the above procedure and a binary search on the target makespan, we can get a $\lambda(1 + \epsilon)$ approximation for every $\epsilon > 0$. If this procedure runs in polynomial time, and $log(C_{max})$ is also polynomial (where C_{max} the upper bound of d used in the binary search), then obviously the approximate schedule is also computed in polynomial time. The factor ϵ has to do with the precision of the binary search, and defines a trade-off between running time and good approximation ratio.

The algorithm presented in this section uses a dual approximation technique in the following way. For a given d, if the oracle doesn't answer **"NO"**, it returns a schedule partitioned into two shelves. The first shelf is of length d and contains jobs with processing time $\frac{d}{2} \leq p \leq d$. All these jobs can actually run in parallel and so the first shelf is a legal schedule. In the second shelf, of length $\frac{d}{2}$, we have jobs with processing time at most $\frac{d}{2}$. Ideally, we would like the second shelf to be a legal schedule too. Unfortunately, this is not the case, and some local transformations, which should maintain the length of both shelves, should take place. After that, the whole 2-shelves schedule of makespan at most $\frac{3d}{2}$ will be feasible, and that concludes the design of a $\frac{3}{2}$-dual approximation procedure for the problem.

Next, we sketch the main structure of the algorithm. In latter sections, we study separately all of its subroutines, providing both intuition and in-depth mathematical

analysis. Note that the properties of step 3 of the below algorithm define the construction of the desired oracle.

A $\frac{3}{2}$-dual approximation

1: Take as input a target makespan d, the number of machines m and the set of monotonic malleable jobs $\mathcal{J}$.
2: Let $\mathcal{J}_\mathcal{S} = \{j \in \mathcal{J} \mid p_j(1) \leq \frac{d}{2}\}$ and $W_\mathcal{S} = \sum_{j \in \mathcal{J}_\mathcal{S}} p_j(1)$.
3: For the rest of the tasks, $\mathcal{J} - \mathcal{J}_\mathcal{S}$, find an allotment of machines to each of them, with the following properties:

- The total work is at most $md - W_\mathcal{S}$.
- The processing time of all jobs is at most d.
- The jobs with processing time greater than $\frac{d}{2}$ require in total at most m processors, and thus can run in parallel.

If you fail to do so, answer **NO** and stop.
4: Use the *BuildFeasible* subroutine, in order to transform the above schedule into a feasible one.
5: Use the *InsertSmall* subroutine to schedule the remained tasks of $\mathcal{J}_\mathcal{S}$, and return the final solution.

2.2.2 The *InsertSmall* Subroutine

It is clear form the above description, that the algorithm temporarily doesn't consider the tasks of $\mathcal{J}_\mathcal{S}$. These are the ones with sequential time at most $\frac{d}{2}$. That happens, because every $j \in \mathcal{J}_\mathcal{S}$ is quite "small" in a way, and scheduling $\mathcal{J}_\mathcal{S}$ doesn't affect substantially the optimal solution. Hence, given a **YES** answer of the oracle and the corresponding computed allotment, we must legally insert the tasks of $\mathcal{J}_\mathcal{S}$ in the constructed schedule for $\mathcal{J} - \mathcal{J}_\mathcal{S}$. So suppose that after step 4 of the algorithm we have a feasible 2-Shelves schedule for the jobs of $\mathcal{J} - \mathcal{J}_\mathcal{S}$, with the following property. The jobs of each self can run in parallel. In a latter section we will see how this is achieved by the *BuildFeasible* subroutine.

Lemma 2.4. *If there exists a 2-Shelves schedule of length $\frac{3d}{2}$ for $\mathcal{J} - \mathcal{J}_\mathcal{S}$ with work at most $md - W_\mathcal{S}$, then a schedule of length $\frac{3d}{2}$ can be derived for $\mathcal{J}$ in polynomial time.*

Proof. Consider the 2-Shelves schedule of figure 2.1(A), which is the one produced by the *BuildFeasible* subroutine. Initially, modify the starting time of all tasks of the second shelf, so that they complete at time exactly $\frac{3d}{2}$ (Figure 2.2(B)). Next, define the load of processor $i \in \mathcal{M}$, as $load(i) = \frac{3d}{2} - idle(i)$, where $idle(i)$ is the idle time of that processor.

Now schedule the jobs of $\mathcal{J}_\mathcal{S}$ using the following naive rule. Order them in an arbitrary way, and consider one task at a time. Allocate that task to the least loaded processor,

at the earliest possible time. The only problem with this approach is that a job of $\mathcal{J}_S$ may not be completed before the jobs of the second shelf start execution. But at each step, every processor has a load of at most d. Suppose otherwise. Then:

$$\forall i \in \mathcal{M} : \; load(i) > d \implies W_{total} > md$$

, where W_{total} the total work area of the schedule. However we know that:

$$W_{total} = W_{\mathcal{J}-\mathcal{J}_S} + W_S$$

$$W_{total} \leq md - W_S + W_S$$

$$W_{total} \leq md$$

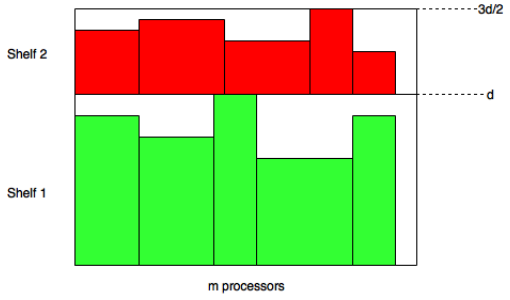
, and so we reached a contradiction. Thus $load(i) \leq d$, in each iteration of scheduling tasks of $\mathcal{J}_S$.

$$load(i) = \frac{3d}{2} - idle(i)$$

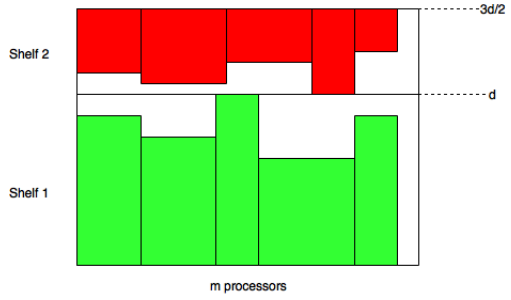
$$\frac{3d}{2} - idle(i) \leq d \implies idle(i) \geq \frac{d}{2}$$

Hence, the idle time interval on the least loaded machine has enough length to contain the sequential execution of a job $j \in \mathcal{J}_S$. For better understanding of the whole process, we present a visual example.

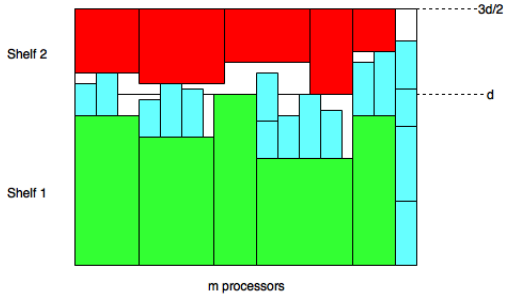
□



(A) After the *BuildFeasible* subroutine



(B) After moving the jobs of the second shelf



(C) After the *InsertSmall* subroutine

FIGURE 2.1: The *InsertSmall* subroutine.

2.2.3 Partitioning the Jobs into 2 Shelves

As mentioned earlier, the most vital part of the algorithm is the design of the dual approximation oracle. This oracle tries to partition the jobs of $\mathcal{J} - \mathcal{J}_\mathcal{S}$ into two shelves, with a machine allotment that satisfies the properties of step 3. Apart from presenting its construction in detail, we are also going to prove that if such an allotment doesn't exist then there is no schedule of makespan at most d.

In the first shelf, call it S_1, we must place jobs with processing time in the interval $[\frac{d}{2}, d]$, that can all run in parallel. In the second shelf, call it S_2, we must place jobs with running time at most $\frac{d}{2}$. We would also like $md - W_\mathcal{S}$ to be an upper bound of the total work area. Due to the monotonic hypothesis, when it comes to deciding in which shelf we will place a job, we have only two choices to consider regarding its allotment. We either place it in S_1 with $\gamma(j, d)$ processors or in S_2 with $\gamma(j, \frac{d}{2})$ processors. So constructing the oracle relies on solving the following optimization problem.

$$\text{Find } W^\star = \min_{S_1 \subseteq \mathcal{J}} \Big\{ \sum_{j \in S_1} w(j, \gamma(j, d)) + \sum_{j \notin S_1} w(j, \gamma(j, \frac{d}{2})) \Big\}$$
$$\text{Such that } \sum_{j \in S_1} \gamma(j, d) \leq m$$

In the end, by checking if $W^\star \leq md - W_\mathcal{S}$ we can answer properly.

We can solve the above problem by a simple reduction to knapsack. Suppose that initially all jobs are alloted $\gamma(j, \frac{d}{2})$ processors and are all required to run in S_2. By choosing some of the jobs to run in S_1 we decrease the total work. That too follows directly from the monotony assumption. Hence, we can interpret $v_j = w(j, \gamma(j, \frac{d}{2})) - w(j, \gamma(j, d))$, as the profit of a job, when it enters the knapsack. Also, $\omega_j = \gamma(j, d)$ stands for its weight and the total capacity W is naturally m. Thus, the partition problem is transformed in the following way.

$$\text{Find } W^\star = \sum_{j \in \mathcal{J}} w(j, \gamma(j, \frac{d}{2})) - \max_{S_1 \subseteq \mathcal{J}} \sum_{j \in S_1} v_j$$
$$\text{Such that } \sum_{j \in S_1} \omega_j \leq m$$

By using the well-known pseudopolynomial algorithm for knapsack [13], we can get a solution in time $\mathcal{O}(nW) = \mathcal{O}(nm)$. The only thing left to prove is that if $W^\star > md - W_\mathcal{S}$, then there in no schedule of makespan at most d.

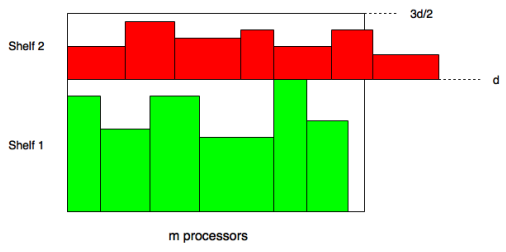
Lemma 2.5. *If there exists a schedule of length at most d, then the knapsack formulation of the problem delivers a desired allotment.*

Proof. Due to the monotony hypothesis it is clear that $W_{\mathcal{J}_\mathcal{S}} \leq W_\mathcal{S}$. Also, because the optimal makespan is at most d we have $W_{OPT} \leq md$, where W_{OPT} the total work in the optimal schedule. The above imply, that the tasks of $\mathcal{J} - \mathcal{J}_\mathcal{S}$ in the optimal solution occupy an area $W^{OPT}_{\mathcal{J}-\mathcal{J}_\mathcal{S}} \leq md - W_\mathcal{S}$. We denote by S_1^{OPT} the set of jobs that have processing time at least $\frac{d}{2}$ in that schedule. Obviously, their machine allotment is at least $\gamma(j, d)$. Now observe that the jobs of S_1^{OPT} must run in parallel, since the makespan of the schedule is bounded by d. From all the above, we conclude that S_1^{OPT} is a feasible solution for our knapsack formulation and thus $W^\star \leq W^{OPT}_{\mathcal{J}-\mathcal{J}_\mathcal{S}}$. □

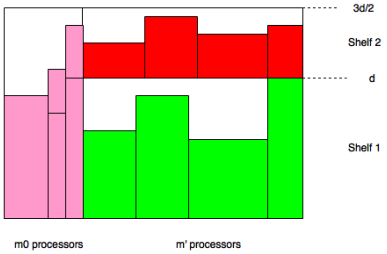
2.2.4 The *BuildFeasible* Subroutine

As we have already mentioned, after the partitioning done by the oracle, the tasks of the first shelf constitute a feasible schedule, in the sense that they can all run in parallel. The same doesn't hold for the tasks of the second shelf. An example is shown in figure 2.2(A). In order to finally deliver a feasible schedule, Mounie et al. provide in their paper a subroutine that takes as input the allotment given by the oracle, and by applying some local transformations achieves feasibility.

A detailed analysis of this subroutine is not within the scope of this thesis, as it requires a great deal of mathematical work, which is also not that insightful. Instead we prefer to provide a more abstract description of it. The goal of this procedure is to partition the set of machines into two groups, S_0 with cardinality m_0 and S' with cardinality m', such that $m = m_0 + m'$. In the m_0 processors will run the jobs that are relocated in the transformation process, and their total makespan will not exceed $\frac{3d}{2}$. The 2-shelves schedule produced by the oracle is now restricted in the m' processors. The only difference is that after *BuildFeasible* the tasks can run in parallel in both shelves. Figure 2.2(B) shows the final result. So in general, this transformation phase is just a way of choosing some jobs, change their initial allotment and schedule them in the machines of S_0, while trying to ensure that the jobs left in the second shelf can run in parallel too. Finally, this procedure also ensures that the total work of tasks in S' is bounded by $m'd - W_\mathcal{S}$ and so the conditions of application of Lemma 2.4 are verified.



(A) Orcale's Output



(B) *BuildFeasible* Output

FIGURE 2.2: Structure of the result.

2.3 Minimizing the Makespan for Identical Jobs

In this section, we are focusing our attention on a special case of the previously studied problem. In particular, we suppose that the tasks under consideration are all identical, in the sense that they have the same processing time function, say $p(x)$. We denote by x the variable indicating the number of alloted machines. This problem has some practical interest, as many applications generate at each step a set of identical tasks to be computed on a parallel platform. Another motivation for studying this case comes from some well-known algorithmic techniques, such as *Divide and Conquer*, or *Branch and Bound*. Using these techniques to solve a problem, the problem is split into smaller independent subproblems, which can be processed in parallel and are almost identical.

We begin by presenting a fairly simple 2-approximation proposed in [14]. The algorithm consists of a straightforward case analysis, which follows in full detail. Before we continue, we need one more definition.

Definition 2.6 (Phase Schedule)**.** A set of tasks is said to be executed in a phase, when all the tasks in the set start at the same time, and no other task starts processing before the completion of all the tasks in the phase.

- **Case 1:** $n \geq m$. Let us denote by $C^\star$, $W^\star$ the makespan and the total work in the optimal schedule respectively. We know that $\frac{W^\star}{m} \leq C^\star$. Suppose otherwise. Then $W^\star > mC^\star$. This is clearly a contradiction, because it states that the total work of the optimal solution cannot fit inside the area of the rectangle representing the whole schedule. Thus, due to the work monotonicity, we can conclude that $\frac{np(1)}{m} \leq m$. Now, we are going to build a schedule of $\lfloor \frac{n}{m} \rfloor$ phases, each consisting of m jobs. Every job inside the phase uses only one processor. The remaining $n \ mod \ m$ jobs will be scheduled last, on a phase of their own, sharing uniformly the m machines. Hence, the makespan of the constructed schedule will be:

$$C \leq \Big\lfloor \frac{n}{m} \Big\rfloor p(1) + p(\Big\lfloor \frac{m}{n \ mod \ m} \Big\rfloor)$$

$$C \leq \frac{n}{m}p(1) + p(1) \leq (\frac{n}{m} + 1)p(1)$$

$$C \leq 2\frac{n}{m}p(1) \leq 2C^\star$$

  The second inequality follows since $p(\lfloor \frac{m}{n \ mod \ m} \rfloor) \leq p(1)$ and the last since $\frac{n}{m} > 1$. See figure 2.3 for a graphical representation.

- **Case 2:** $n < m$. Here, the resulting schedule will consist of only one phase, where each task utilizes $\lfloor \frac{m}{n} \rfloor$ processors. Obviously, the makespan of this solution

is $C = p(\lfloor \frac{m}{n} \rfloor)$. Since this is not the optimal solution, we can deduce that every job must use there at least $\lfloor \frac{m}{n} \rfloor$ processors for its execution. This means that $np(\lfloor \frac{m}{n} \rfloor) \lfloor \frac{m}{n} \rfloor \leq W^\star$ and consequently $\frac{n}{m} p(\lfloor \frac{m}{n} \rfloor) \lfloor \frac{m}{n} \rfloor \leq C^\star$. To conclude:

$$\frac{C}{C^\star} \leq \frac{p(\lfloor \frac{m}{n} \rfloor)}{\frac{n}{m} p(\lfloor \frac{m}{n} \rfloor) \lfloor \frac{m}{n} \rfloor} = \frac{m}{n \lfloor \frac{m}{n} \rfloor} \leq \frac{\frac{m}{n}}{\lfloor \frac{m}{n} \rfloor} \leq 2$$

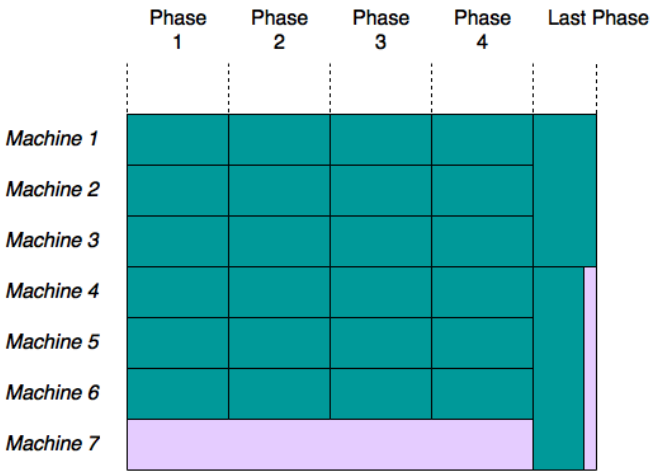


FIGURE 2.3: A phase schedule for identical malleable jobs.

The best result regarding the problem of minimizing the makespan in a system of identical malleable jobs is due to Decker et al. [15]. In their paper, they present a more sophisticated case analysis, based more or less on the ideas shown above and they achieve a $\frac{5}{4}$ approximation. Their result clearly beats the $\frac{3}{2}$ factor of [11], when we are restricted in this special case of the problem.

2.4 Minimizing the Sum of Completion Times

In this section, we study the problem under the objective of minimizing the sum of completion times. As we have already mentioned, in every feasible schedule each job $j \in \mathcal{J}$ has a starting time st_j, and a machine allotment n_j. Its completion time is then $C_j = st_j + p_j(n_j)$, and thus we wish to minimize $\sum_{j \in \mathcal{J}} C_j = \sum_{j \in \mathcal{J}} (st_j + p_j(n_j))$. The monotony assumptions will hold in this case too. This problem is also known to be NP-hard in the strong sense [16]. We are going to present a 2-approximation algorithm, proposed by Turek et al. [17]. Before we continue with the description of the algorithm, we need to define a restricted version of the *Malleable Job Scheduling Problem* used in the analysis.

Definition 2.7 (*Rigid Job Scheduling Problem*)**.** The *Rigid Job Scheduling Problem* [5] is almost identical with the *Malleable Job Scheduling Problem*. The only difference lies

in the machine allotment. In this case, the number of machines that a job $j \in \mathcal{J}$ will use during its processing is determined before the execution. In other words, the allotments are fixed parameters of the input.

It is obvious that the *Rigid Job Scheduling Problem* is indeed a special case of *Malleable Job Scheduling*. If the fixed allotment for every job $j \in \mathcal{J}$ is n_j, we can define an execution time function for the malleable version as follows:

$$p'_j(x) = \begin{cases} +\infty & x < n_j \\ p_j(n_j) & x \geq n_j \end{cases}$$

The algorithm of Turek et al. is based on a 2-phase approach. In the first phase an allotment is computed, satisfying some particular properties. In the second phase, a schedule is constructed for the *Rigid Job Scheduling Problem*, with fixed allotment the one found in the previous phase. That schedule is a 2-approximation for the rigid problem, and due to the properties of the chosen allotment this ratio is also maintained for the malleable case.

2.4.1 Lower Bounds for the Rigid Problem

First of all, we need some lower bounds on the optimal solution of the rigid problem. Let us denote by $F^{\star}_{\mathcal{J}}$ the optimal value for a monotonic task set $\mathcal{J}$, with fixed allotments n_j. We denote by $\overrightarrow{n} = \{n_1, n_2, \ldots, n_n\}$ the vector of all machine allotments.

Definition 2.8 (Squashed Area Bound)**.** Let $\mathcal{J}$ be a set of n monotonic rigid jobs. Arrange the jobs in order of increasing work, so that $n_j p_j(n_j) \leq n_{j+1} p_{j+1}(n_{j+1})$, $\forall j \in [1, n)$. The squashed area bound is defined as:

$$A_{\mathcal{J}}(\overrightarrow{n}) = \frac{1}{m} \sum_{j \in \mathcal{J}} n_j p_j(n_j)(n - j + 1) \leq F^{\star}_{\mathcal{J}} \quad [18]$$

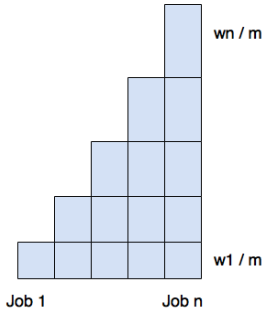


FIGURE 2.4: The Squashed Area Bound

Figure 2.4 illustrates a graphic representation of the squashed area bound. Note that in the above definition $A_{\mathcal{J}}(\overrightarrow{n})$ is evaluated as a horizontal sum. Vertical summation yields the alternative expression:

$$A_{\mathcal{J}}(\overrightarrow{n}) = \frac{1}{m} \sum_{j \in \mathcal{J}} \sum_{k=1}^{j} n_k p_k(n_k)$$

Definition 2.9 (Height Bound)**.** Let $\mathcal{J}$ be a set of n monotonic rigid jobs. The height bound for an allotment vector $\overrightarrow{n}$ is defined to be $H_{\mathcal{J}}(\overrightarrow{n}) = \sum_{j \in \mathcal{J}} p_j(n_j) \leq F^{\star}_{\mathcal{J}}$.

Definition 2.10 (Normalized Work Bound)**.** Let $\mathcal{J}$ be a set of n monotonic rigid jobs. The normalized work bound for an allotment vector $\overrightarrow{n}$ is defined to be $W_{\mathcal{J}}(\overrightarrow{n}) = \frac{1}{m} \sum_{j \in \mathcal{J}} n_j p_j(n_j) \leq F^{\star}_{\mathcal{J}}$.

The above two quantities are obvious lower bounds. The height bound corresponds to the second sum of $\sum_{j \in \mathcal{J}} C_j = \sum_{j \in \mathcal{J}} (st_j + p_j(n_j))$, and since $1 \leq n_j \leq m$ we have $W_{\mathcal{J}}(\overrightarrow{n}) \leq H_{\mathcal{J}}(\overrightarrow{n})$. In the following, we are going to combine all the above lower bounds, in order to provide a tighter one.

Lemma 2.11. *Let $\mathcal{J}$ be a set of n monotonic rigid jobs, with an allotment vector $\overrightarrow{n}$. Then $A_{\mathcal{J}}(\overrightarrow{n}) + \frac{1}{2}(H_{\mathcal{J}}(\overrightarrow{n})) - W_{\mathcal{J}}(\overrightarrow{n})) \leq F^{\star}_{\mathcal{J}}$.*

Proof. For a given schedule let $E_{\mathcal{J}}(t) = \{j \in \mathcal{J} \mid st_j \leq t < st_j + p_j(n_j)\}$ be the set of jobs active at time t. Also, $C_{\mathcal{J}}(t) = \{j \in \mathcal{J} \mid t < st_j + p_j(n_j)\}$ the set of jobs that have not yet completed by time t. For each job, we define a function corresponding to the fraction of it, that is left to be completed at time t.

$$c^j_{\mathcal{J}}(t) = \begin{cases} 1 & t \leq st_j \\ 1 - \frac{t - st_j}{p_j(n_j)} & st_j \leq t < st_j + p_j(n_j) \\ 0 & st_j + p_j(n_j) < t \end{cases}$$

The total fractional number of jobs to be completed at time t is:

$$c_{\mathcal{J}}(t) = \sum_{j \in \mathcal{J}} c^j_{\mathcal{J}}(t) = |C_{\mathcal{J}}(t)| - \sum_{j \in E_{\mathcal{J}}(t)} \frac{t - st_j}{p_j(n_j)}$$

$$\int_0^{\infty} c_{\mathcal{J}}(t)\, \mathrm{d}t = \sum_{j \in \mathcal{J}} \int_0^{\infty} c^j_{\mathcal{J}}(t)\, \mathrm{d}t = \sum_{j \in \mathcal{J}} \Big(\int_0^{st_j + p_j(n_j)} \mathrm{d}t - \int_{st_j}^{st_j + p_j(n_j)} \frac{t - st_j}{p_j(n_j)}\, \mathrm{d}t \Big)$$

$$\int_0^{\infty} c_{\mathcal{J}}(t)\, \mathrm{d}t = \sum_{j \in \mathcal{J}} (st_j + p_j(n_j)) - \sum_{j \in \mathcal{J}} \frac{p_j(n_j)}{2} = F_{\mathcal{J}} - \frac{1}{2} H_{\mathcal{J}}(\overrightarrow{n})$$

, where $F_{\mathcal{J}}$ is the solution cost of the given schedule.

Now, we need to talk a bit about the *squashed area construction*. From the given set $\mathcal{J}$ with jobs ordered by increasing work, we create a new set of n jobs $\mathcal{S}$. Job $j \in \mathcal{S}$ has a fixed allotment of $n_{\mathcal{S},j} = m$ processors and an execution time $\frac{1}{m} n_j p_j(n_j)$. It is clear that:

$$A_{\mathcal{J}}(\overrightarrow{n}) = A_{\mathcal{S}}(\overrightarrow{n_{\mathcal{S}}}) = F^{\star}_{\mathcal{S}}$$

$$W_{\mathcal{J}}(\overrightarrow{n}) = H_{\mathcal{S}}(\overrightarrow{n_{\mathcal{S}}})$$

Furthermore, given any feasible schedule for $\mathcal{J}$ it can be shown by a simple convexity argument that the optimal schedule for $\mathcal{S}$ obeys $c_{\mathcal{J}}(t) \geq c_{\mathcal{S}}(t)$ for every t. Thus, we have $\int_0^{\infty} c_{\mathcal{J}}(t) \geq \int_0^{\infty} c_{\mathcal{S}}(t)$. Using the resulted form of the integral and the above two equations we get:

$$\int_0^{\infty} c_{\mathcal{J}}(t) \geq \int_0^{\infty} c_{\mathcal{S}}(t)$$

$$F_{\mathcal{J}} - \frac{1}{2} H_{\mathcal{J}}(\overrightarrow{n}) \geq F^{\star}_{\mathcal{S}} - \frac{1}{2} H_{\mathcal{S}}(\overrightarrow{n_{\mathcal{S}}})$$

$$F_{\mathcal{J}} - \frac{1}{2} H_{\mathcal{J}}(\overrightarrow{n}) \geq A_{\mathcal{J}}(\overrightarrow{n}) - \frac{1}{2} W_{\mathcal{J}}(\overrightarrow{n})$$

$$F_{\mathcal{J}} \geq A_{\mathcal{J}}(\overrightarrow{n}) + \frac{1}{2}(H_{\mathcal{J}}(\overrightarrow{n})) - W_{\mathcal{J}}(\overrightarrow{n}))$$

The above holds for every feasible schedule of $\mathcal{J}$, and so it holds for the optimal too.

□

2.4.2 A Scheduling Algorithm for the Rigid Problem

Let $\mathcal{J}$ be a set of n monotonic rigid jobs, with an allotment vector $\overrightarrow{n}$, such that $\forall j \in \mathcal{J} : \; n_j < \lceil \frac{m}{2} \rceil$. We present a 2-approximation algorithm for this case, which is a simple extension of the classical list scheduling algorithms of the literature [19].

The **LIST** algorithm

1: Arrange the jobs in order of increasing work.
2: In this order, schedule job $j \in \mathcal{J}$ at the earliest possible time that at least n_j processors are available.

Lemma 2.12. ***LIST*** *is a* 2*-approximation algorithm for the* Rigid Job Scheduling Problem*, when for every job* $n_j < \lceil \frac{m}{2} \rceil$.

Proof. Let us denote by $F_{\mathcal{J}}$ the solution cost of **LIST**. It suffices to prove that $F_{\mathcal{J}} \leq 2A_{\mathcal{J}}(\overrightarrow{n}) + H_{\mathcal{J}}(\overrightarrow{n})) - 2W_{\mathcal{J}}(\overrightarrow{n})$, since by **Lemma 2.11** we get the desired result. We already now that $F_{\mathcal{J}} = \sum_{j \in \mathcal{J}} st_j + \sum_{j \in \mathcal{J}} p_j(n_j) = \sum_{j \in \mathcal{J}} st_j + H_{\mathcal{J}}(\overrightarrow{n})$

Therefore, to prove the bound in the lemma we can show that:

$$\sum_{j\in\mathcal{J}} st_j \leq 2A_{\mathcal{J}}(\overrightarrow{n}) + -2W_{\mathcal{J}}(\overrightarrow{n})$$

Using the vertical summation for the squashed area bound and the definition of the work normalized bound, this is equivalent to showing:

$$\sum_{j\in\mathcal{J}} st_j \leq \frac{2}{m}\sum_{j\in\mathcal{J}}\sum_{k=1}^{j} n_k p_k(n_k) - \frac{2}{m}\sum_{j\in\mathcal{J}} n_j p_j(n_j)$$

To prove the above, it suffices to show the inequality for the jth summand.

$$st_j \leq \frac{2}{m}\sum_{k=1}^{j} n_k p_k(n_k) - \frac{2}{m} n_j p_j(n_j) = \frac{2}{m}\sum_{k=1}^{j-1} n_k p_k(n_k)$$

The above equality follows since we consider the jobs in order of increasing work. Now, all that is left to do is prove the inequality for each st_j. Consider the moment that job $j \in \mathcal{J}$ starts processing in the schedule constructed by **LIST**. Up until that point, the whole schedule area consists of useful work, denoted by W, and idle periods, with total area denoted by I. Observe that the number of idle processors in any time prior to st_j must be less than $\lceil \frac{m}{2} \rceil$. The opposite would contradict the scheduling rule. Thus, I cannot exceed W. But it is easy to see that $W \leq \sum_{k=1}^{j-1} n_k p_k(n_k)$. So, because $I \leq W$, we also have $I \leq \sum_{k=1}^{j-1} n_k p_k(n_k)$. Finally we get:

$$m \cdot st_j = I + W \leq 2\sum_{k=1}^{j-1} n_k p_k(n_k) \implies st_j \leq \frac{2}{m}\sum_{k=1}^{j-1} n_k p_k(n_k)$$

□

2.4.3 Restricting an Allotment

Now we come back to the malleable version of the problem. The result of the previous section obviously applies for the more general case too. That is, if we have a set of malleable jobs $\mathcal{J}$ together with an arbitrary allotment $\overrightarrow{n}$ for which $\forall j : n_j < \lceil \frac{m}{2} \rceil$, and we apply **LIST**, then we get a schedule with cost $F \leq 2A_{\mathcal{J}}(\overrightarrow{n}) + H_{\mathcal{J}}(\overrightarrow{n})) - 2W_{\mathcal{J}}(\overrightarrow{n})$. In this section, we show how to get a similar result for every possible initial allotment, even if it does not obey the above requirement. So, for a given allotment $\overrightarrow{n}$, we define a new allotment $\overrightarrow{z}$, called the *skinny allotment*, as follows:

$$z_j = \begin{cases} n_j & \text{if } n_j \leq \lceil \frac{m}{2} \rceil \\ \lceil \frac{m}{2} \rceil & \text{otherwise} \end{cases}$$

Lemma 2.13. *Applying **LIST** to $\overrightarrow{z}$ yields a schedule with solution cost:*

$$F \leq 2A_{\mathcal{J}}(\overrightarrow{n}) + H_{\mathcal{J}}(\overrightarrow{n})) - W_{\mathcal{J}}(\overrightarrow{n})$$

Proof. Due to **Lemma 2.12** we have $F \leq 2A_{\mathcal{J}}(\overrightarrow{z}) + H_{\mathcal{J}}(\overrightarrow{z})) - 2W_{\mathcal{J}}(\overrightarrow{z})$. Also, because of the work monotonicity hypothesis and the fact that $z_j \leq n_j$ for all jobs, we have $A_{\mathcal{J}}(\overrightarrow{z}) \leq A_{\mathcal{J}}(\overrightarrow{n})$. Hence, it suffices to prove:

$$H_{\mathcal{J}}(\overrightarrow{z})) - 2W_{\mathcal{J}}(\overrightarrow{z}) \leq H_{\mathcal{J}}(\overrightarrow{n})) - W_{\mathcal{J}}(\overrightarrow{n})$$

Using the definitions of the height and the normalized work bound, we need to show that for all j:

$$p_j(z_j)(1 - \frac{2z_j}{m}) \leq p_j(n_j)(1 - \frac{n_j}{m})$$

To prove the above consider the following two cases:

- $n_j \leq \lceil \frac{m}{2} \rceil \implies z_j = n_j$. The inequality holds trivially.
- $n_j > \lceil \frac{m}{2} \rceil \implies z_j = \lceil \frac{m}{2} \rceil$. The left-hand side of the inequality becomes $p_j(\lceil \frac{m}{2} \rceil)(1 - \frac{2\lceil \frac{m}{2} \rceil}{m})$, which is clearly non-positive. For the right-hand side of the inequality we have $p_j(n_j)(1 - \frac{n_j}{m}) > p_j(n_j)(1 - \frac{\lceil \frac{m}{2} \rceil}{m}) \geq 0$. The result follows.

□

2.4.4 Finding an Initial Allotment

Given lemmas **2.11**, **2.12**, **2.13**, we can understand that the final step of a 2-approximation is just a matter of finding a proper initial allotment. Let $\overrightarrow{n^\star}$ be the allotment vector of the optimal solution. We would like to compute an initial allotment $\overrightarrow{n}$ such that:

$$2A_{\mathcal{J}}(\overrightarrow{n}) + H_{\mathcal{J}}(\overrightarrow{n})) - W_{\mathcal{J}}(\overrightarrow{n}) \leq 2A_{\mathcal{J}}(\overrightarrow{n^\star}) + H_{\mathcal{J}}(\overrightarrow{n^\star})) - W_{\mathcal{J}}(\overrightarrow{n^\star})$$

In their paper, Turek et al. provide a method of minimizing $2A_{\mathcal{J}}(\overrightarrow{n}) + H_{\mathcal{J}}(\overrightarrow{n})) - W_{\mathcal{J}}(\overrightarrow{n})$, over all possible allotment vectors $\overrightarrow{n}$. We will describe it briefly.

- For all jobs $j \in \mathcal{J}$ and for all possible positions k, that j may have in the relative work ordering of tasks in the desired allotment $\overrightarrow{n}$, compute:

$$n_j^k = arg \min_{1 \leq i \leq m} \{p_j(i) + (2(n-s)+1)\frac{i}{m}p_j(i)\}$$

- For all jobs $j \in \mathcal{J}$ and for all possible positions k, compute

$$F_j^k = p_j(n_j^k) + (2(n-s)+1)\frac{n_j^k}{m}p_j(n_j^k)$$

  This quantity is the minimum cost attributed to job j, in the lower bound of **Lemma 2.11**, if it is assigned the k^{th} position in the relative work ordering.

- Construct a complete bipartite graph $G(A, B, E)$. In A there is a node for every job of $\mathcal{J}$. In B there is node representing all possible n positions that a job may have in the work-based ordering of an allotment. The weight of edge (j, k) is F_j^k.

- Find a minimum weight matching in G. This can be done in polynomial time [20]. If edge (j, k) is included in the matching, then $n_j = n_j^k$.

Although, the above procedure seems intuitively correct, some technical mathematical analysis is still needed. To be more precise, it must be shown that the optimal allotment corresponds to a matching in G, as well as a matching produced in such a way is legal. In other words, the positions assigned to jobs reflect the true work ordering. In their paper, Turek et al. provide detailed proofs for the above.

2.5 Scheduling with Resource Dependent Processing Times

In this section, we address a scheduling model, which at first sight seems irrelevant to the malleable problem considered earlier. This particular model was first introduced and studied by Grigoriev et al.[21–23]. Its connection to the *Malleable Job Scheduling* problem will be made clear by the end of the section.

2.5.1 Problem Definition

Let $\mathcal{V} = \{1, \ldots, \}$ be a set of jobs that need to be scheduled non-preemptively on a set of unrelated machines $\mathcal{M} = \{1, \ldots, m\}$. There is also a pool of k additional identical renewable resources, that can be distributed over the jobs in process, in order to speedup their execution. Thus, if job j uses machine i for its processing and also has

been assigned $s \in [0, k]$ of the addition resources, then it has an execution time $p_{i,j,s}$. We assume that for all pairs (i, j) we have

$$p_{i,j,0} \geq p_{i,j,1} \geq \ldots \geq p_{i,j,k}$$

Without loss of generality, we also assume that all $p_{i,j,s}$ are integral and hence we can restrict to feasible schedules where jobs start and stop at integral points in time. Regarding the allocation of resources to jobs, we have the following restrictions. In any feasible schedule, the total resource consumption at any time cannot exceed k. Furthermore, the amount of the additional resources assigned to a job cannot change during its execution. It is obvious, that this model is actually an extension of the classical $R||C_{max}$ problem, as well as a natural variant of the generalized assignment problem studied by Shmoys and Tardos [24].

2.5.2 A 4-Approximation Algorithm

We begin by presenting a 4-approximation algorithm, for the problem of minimizing the makespan in this model, proposed in [23].

2.5.2.1 Relaxing the Problem

Initially, Grigoriev et al. consider an integer programming formulation that defines a relaxation of the problem. Let $x_{i,j,s}$ be binary variables, indicating that job j uses machine i, while an amount of s resources is allocated to it. Moreover, let $S_{i,j} = \{0\} \cup \{s \mid s \leq k, p_{i,j,s} < p_{i,j,s-1}\}$ be the set of relevant indices for job j on machine i. Considering only this index sets obviously suffices, as the feasibility and the makespan of any solution is not violated. We then have the following program.

$$\sum_{i \in \mathcal{M}} \sum_{s \in S_{i,j}} x_{i,j,s} = 1, \ \forall j \in \mathcal{V} \tag{2.1}$$

$$\sum_{j \in \mathcal{V}} \sum_{s \in S_{i,j}} x_{i,j,s} p_{i,j,s} \leq C, \ \forall i \in \mathcal{M} \tag{2.2}$$

$$\sum_{j \in \mathcal{V}} \sum_{i \in \mathcal{M}} \sum_{s \in S_{i,j}} x_{i,j,s} s p_{i,j,s} \leq kC \tag{2.3}$$

$$x_{i,j,s} = 0, \text{ if } p_{i,j,s} > C \tag{2.4}$$

$$x_{i,j,s} \in \{0, 1\}, \ \forall i, j, s \tag{2.5}$$

Here, C represents the schedule makespan. Equalities (2.20) ensure that every job runs on some machine, using an amount of s resources. Inequalities (2.21) imply that the load of each machine is a lower bound on the makespan. The left-hand side of

inequality (2.22) is the total resource consumption over the whole schedule, and (2.22) simply expresses the fact that this quantity cannot exceed kC. Finally, constraints (2.23) make sure we do not use machine-resource pairs such that the processing time of a job is greater than C.

From all the above, it is clear that if a feasible schedule of makespan C exists, then there is a solution (C, x) for program (2.20)-(2.24). Hence, there is also a solution for the linear relaxation of (2.20)-(2.24). By using binary search on the target makespan C (we care only for integral values), we can find in polynomial time a solution $(C^\star, x^{LP})$ for the resulted linear program. Certainly, $C^\star$ will be a lower bound of the optimal makespan.

2.5.2.2 The Rounding Procedure

Given a feasible solution $(C^\star, x^{LP})$ for the linear relaxation of (2.20)-(2.24), Grigoriev et al. aim at rounding this fractional solution to an integer one without violating too much constraints (2.21) and (2.22). They have proven the following lemma by giving a rounding method with the desired result.

Lemma 2.14. *Let $C^\star$ be the minimal integer for which the following linear program has a feasible solution.*

$$\sum_{i\in\mathcal{M}}\sum_{s\in S_{i,j}} x_{i,j,s} = 1,\ \forall j \in \mathcal{V} \tag{2.6}$$

$$\sum_{j\in\mathcal{V}}\sum_{s\in S_{i,j}} x_{i,j,s}p_{i,j,s} \le C^\star,\ \forall i \in \mathcal{M} \tag{2.7}$$

$$\sum_{j\in\mathcal{V}}\sum_{i\in\mathcal{M}}\sum_{s\in S_{i,j}} x_{i,j,s}c_{i,j,s} \le kC^\star \tag{2.8}$$

$$x_{i,j,s} = 0,\ \textit{if } p_{i,j,s} > C^\star \tag{2.9}$$

$$x_{i,j,s} \ge 0,\ \forall i,j,s \tag{2.10}$$

and let $(C^\star, x^{LP})$ be the corresponding solution, then we can find in polynomial time a solution $x^\star$ for the following integer program:

$$\sum_{i\in\mathcal{M}}\sum_{s\in S_{i,j}} x_{i,j,s} = 1,\ \forall j \in \mathcal{V} \tag{2.11}$$

$$\sum_{j\in\mathcal{V}}\sum_{s\in S_{i,j}} x_{i,j,s}p_{i,j,s} \le C^\star + p_{max},\ \forall i \in \mathcal{M} \tag{2.12}$$

$$\sum_{j\in\mathcal{V}}\sum_{i\in\mathcal{M}}\sum_{s\in S_{i,j}} x_{i,j,s}c_{i,j,s} \le kC^\star \tag{2.13}$$

$$x_{i,j,s} \in \{0,1\},\ \forall i,j,s \tag{2.14}$$

where $p_{max} = max\{p_{i,j,s} \mid x^{LP}_{i,j,s} > 0\}$ and $c_{i,j,s}$ non-negative fixed values.

The above lemma can be viewed as an extension of the famous Shmoys and Tardos rounding theorem [24] for the generalized assignment problem. The extension lies in the

fact that the rounding is done in a bipartite multigraph, instead of a simple bipartite graph. Moreover, the rounding method that Grigoriev et al. propose in their paper, can also be viewed as the derandomization of the randomized rounding algorithm of Kumar et al. [25], by applying the technique of conditional probabilities.

2.5.2.3 The Scheduling Algorithm

The approach for obtaining the final constant approximation result consists of the following. In the first place, the rounding procedure described earlier must be used, in order to decide the resource allocations and the machine assignments. To be more precise, job j will run on machine i, using s additional resources iff $x^{\star}_{i,j,s} = 1$. In the application of **Lemma 2.14** we set $c_{i,j,s} = s \cdot p_{i,j,s}$. Afterwards, the jobs are scheduled via a greedy list scheduling algorithm.

Algorithm **LP-GREEDY**

1: The machine assignments and the resource allocations are determined by the rounding procedure.
2: Consider the jobs in an arbitrary order starting with $t = 0$:
- If there are enough resources available, so that some yet unscheduled job can start running on its predetermined machine at time t, then schedule it there.
- If there is not such an unscheduled job available, update t to the next smallest completion time in the currently constructed schedule.

Theorem 2.15. *Algorithm **LP-GREEDY** is a 4-approximation for the problem at hand.*

Proof. Denote by C^{LPG} the makespan of the schedule produce by **LP-GREEDY** and by C^{OPT} the optimal one. Let $t(\beta)$ the earliest point in time after which only big jobs are scheduled in the constructed solution. We define as big jobs the ones with resource consumption greater than $\frac{k}{2}$. The part of the schedule after time $t(\beta)$ has length $\beta = C^{LPG} - t(\beta)$. During that period only big jobs are getting processed.

Furthermore, there must be a machine i on which some job completes at time $t(\beta)$, that it is not a big job. Suppose otherwise. Then, due to the definition of $t(\beta)$ all machines must be idle right before this point in time. But that clearly contradicts the greedy scheduling rule. We now focus our attention on this specific machine i. In the time interval $[0, t(\beta)]$ there must be periods that i is either busy or idle. Let us denote be α the total length of the busy periods. From constraint (2.12) we know that:

$$\alpha = \sum_{j \in \mathcal{V}} \sum_{s \in S_{i,j}} x_{i,j,s} p_{i,j,s} \leq C^{\star} + p_{max} \leq 2C^{\star}$$

The last inequality follows directly from (2.28). If we denote by γ the length of the idle periods then we have:

$$C^{LPG} \leq \alpha + \beta + \gamma$$

The next step provides an upper bound for the quantity $\beta + \gamma$.

Observe that the total resource consumption of the constructed schedule is at least $\beta\frac{k}{2} + \gamma\frac{k}{2}$. During the time interval $[t(\beta), C^{LPG}]$ only big jobs are undergoing processing and thus, the total resource consumption there is at least $\beta\frac{k}{2}$, since at any point in time at least $\frac{k}{2}$ of the additional resources are used. It is also the case, that during the idle periods of machine i in the time interval $[0, t(\beta)]$, at least $\frac{k}{2}$ resources are used at any time. Suppose otherwise. Then there was an idle time in i, with at least $\frac{k}{2}$ available resources. But this contradicts the scheduling rule. The job that finishes at time $t(\beta)$ in i could have been scheduled earlier at that idle time. Finally, recall that $kC^\star$ is an upper bound of the total resource consumption of the schedule (2.13). Hence, we have

$$kC^\star \geq \beta\frac{k}{2} + \gamma\frac{k}{2} \implies \gamma + \beta \leq 2C^\star$$

By combining the above, we get the lower bound of 4. Remember also that $C^\star \leq C^{OPT}$.

□

2.5.3 A 3.75-Approximation Algorithm

In their paper[23] Grigoriev et al., provide a slightly improved approximation algorithm compared to the one presented earlier. Their main approach is just a better tuning of the techniques already used. Initially, they formulate an alternative integer programming relaxation for the problem. Let $B_{i,j} = \{s \in S_{i,j} \mid \frac{k}{2} < s \leq k\}$ be the set of indices that lie in the interval $(\frac{k}{2}, k]$.

$$\sum_{i\in\mathcal{M}} \sum_{s\in S_{i,j}} x_{i,j,s} = 1, \ \forall j \in \mathcal{V} \tag{2.15}$$

$$\sum_{j\in\mathcal{V}} \sum_{s\in S_{i,j}} x_{i,j,s} p_{i,j,s} \leq C, \ \forall i \in \mathcal{M} \tag{2.16}$$

$$\sum_{j\in\mathcal{V}} \sum_{i\in\mathcal{M}} \Big(1.5 \sum_{s\in S_{i,j}} x_{i,j,s} \frac{s}{k} p_{i,j,s} + 0.25 \sum_{s\in B_{i,j}} x_{i,j,s} p_{i,j,s}\Big) \leq 1.75C \tag{2.17}$$

$$x_{i,j,s} = 0, \text{ if } p_{i,j,s} > C \tag{2.18}$$

$$x_{i,j,s} \in \{0, 1\}, \ \forall i, j, s \tag{2.19}$$

The above integer program defines a relaxation of the original problem. To prove this, we only need to verify validity of constraint (2.17), since the rest stay the same as in program (2.20)-(2.24). Considering any feasible schedule, any two jobs with resource

consumption in $B_{i,j}$ cannot be processed in parallel. Thus:

$$\sum_{j\in\mathcal{V}}\sum_{i\in\mathcal{M}}\sum_{s\in B_{i,j}} x_{i,j,s}p_{i,j,s} \leq C$$

Combining the above with valid inequality (2.22) yields (2.17).

As before, the linear relaxation of (2.15)-(2.19) is also a relaxation of the original problem, and by using the same binary search we can acquire a fractional solution $(C^\star, x^{LP})$, such that $C^\star \leq C^{OPT}$. The rounding procedure of **Lemma 2.14** with

$$c_{i,j,s} = \begin{cases} (1.5\frac{s}{k} + 0.25)\frac{p_{i,j,s}}{1.75} & \text{if } s \in B_{i,j} \\ 1.5\frac{s}{k}\frac{p_{i,j,s}}{1.75} & \text{if } s \in S_{i,j}\backslash B_{i,j} \end{cases}$$

gives an integral solution that violates only constraint (2.17) by a factor at most $C^\star$.

The scheduling algorithm used after the determination of machine assignments and resource allocations is also modified. In particular, it is a more sophisticated version of list scheduling that can be interpreted as a restricted version of the harmonic algorithm for bin packing. The concluding result is an approximation factor of 3.75.

2.5.4 Connection to the Malleable Problem

We are going to prove that the *Malleable Job Scheduling* problem is just a special case of the model consider by Grigoriev et al.[21–23]. This means that all algorithms for scheduling with resource dependent processing times apply directly to the malleable setting. The importance of this observation will be evident in the next section.

We now present the reduction between the two problems. Suppose we have an instance of the *Malleable Job Scheduling* problem, with n jobs, m identical machines and processing time functions $p_j(x)$. We build an instance for the resource dependent problem as follows:

- For each malleable job j we have a corresponding "classical" job j.
- For each malleable job j we have a corresponding unrelated machine j. Thus, in total we have n machines.
- $\forall i, j \in \{1, \ldots, n\}: \quad p_{i,j,0} = +\infty$. No job can run without any resources.
- $p_{i,j,s} = \begin{cases} p_{i,j,s} = p_j(s), \text{ if } i = j \\ p_{i,j,s} = +\infty, \text{ otherwise} \end{cases}$

A solution for the above instance can easily be transformed to a feasible schedule for the malleable case, maintaining the completion times of all jobs. If st_j is the starting time of j in its dedicated machine using s_j additional resources, then we can start that job at st_j in the malleable schedule, since there will be s_j available machines at that time. That holds because the machines of the malleable problem are interpreted as the additional resource. Obviously, the processing time of each job is the same in both situations. Finally, via the same transformation, though in the other direction, we can prove that any malleable schedule (and so the optimal too) corresponds to a solution of the resource dependent problem for the constructed instance.

2.6 Minimizing the Sum of Weighted Completion Times

In this section, we address the non-preemptive *Malleable Job Scheduling* problem under the objective of minimizing the sum of weighted completion times. To provide a result for this case, we utilize the unrelated machine scheduling with resource dependent processing times problem. As we have already mentioned, each algorithm for that model applies directly to the malleable problem as well. Thus, our aim is to extend the results of Grigoriev et al.[21–23]. presented earlier, in order to capture the unique characteristics of the alternative objective function considered here.

2.6.1 A 25.55-Approximation Algorithm

In order to schedule the jobs on the machines of $\mathcal{M}$, using the ideas of [26], we create the following interval-indexed LP-relaxation for minimizing the total weighted completion time. We define $(0, t_{\max} = \min_{i\in\mathcal{M}} \sum_{j\in\mathcal{V}} p_{i,j,k}]$ to be the time horizon of potential completion times, where $t_{\max}$ is an upper bound on the makespan of any optimal schedule. We discretize the time horizon into intervals $[1,1], (1,(1+\delta)], ((1+\delta),(1+\delta)^2], \ldots, ((1+\delta)^{L-1}, (1+\delta)^L]$, where $\delta \in (0,1)$ is a small constant, and L is the smallest integer such that $(1+\delta)^{L-1} \geq t_{\max}$. Let $I_\ell = ((1+\delta)^{\ell-1}, (1+\delta)^\ell]$, for $1 \leq \ell \leq L$, and $\mathcal{L} = \{1, 2, \ldots, L\}$. Note that, interval $[1,1]$ implies that no job finishes its execution before time 1; in fact, we can assume, w.l.o.g., that all processing times are positive integers. Obviously, the number of intervals is polynomial in the size of the instance and in $\frac{1}{\delta}$. We denote by $x_{i,j,s,\ell}$ the binary indicator variables denoting that job

j has completed its execution on machine i with s additional resources within I_ℓ.

$$\textbf{LP} : \text{minimize} \sum_{j \in \mathcal{V}} w_j C_j$$

$$\text{subject to} : \sum_{i \in \mathcal{M}} \sum_{\ell \in \mathcal{L}} \sum_{s \in S_{i,j}} x_{i,j,s,\ell} = 1, \qquad \forall j \in \mathcal{V} \tag{2.20}$$

$$\sum_{i \in \mathcal{M}} \sum_{s \in S_{i,j}} \sum_{\ell \in \mathcal{L}} (1+\delta)^{\ell-1} x_{i,j,s,\ell} \leq C_j, \qquad \forall j \in \mathcal{V} \tag{2.21}$$

$$\sum_{j \in \mathcal{V}} \sum_{s \in S_{i,j}} p_{i,j,s} \sum_{t \leq \ell} x_{i,j,s,t} \leq (1+\delta)^{\ell}, \qquad \forall i \in \mathcal{M}, \ell \in \mathcal{L} \tag{2.22}$$

$$\sum_{j \in \mathcal{V}} \sum_{s \in S_{i,j}} \sum_{i \in \mathcal{M}} s p_{i,j,s} \sum_{t \leq \ell} x_{i,j,s,t} \leq k(1+\delta)^{\ell}, \qquad \forall \ell \in \mathcal{L} \tag{2.23}$$

$$p_{i,j,s} > (1+\delta)^{\ell} \Rightarrow x_{i,j,s,\ell} = 0, \ \forall i \in \mathcal{M}, j \in \mathcal{V}, s \in S_{i,j}, \ell \in \mathcal{L} \tag{2.24}$$

$$x_{i,j,s,\ell} \geq 0, \qquad \forall i \in \mathcal{M}, j \in \mathcal{V}, s \in S_{i,j}, \ell \in \mathcal{L}$$

Our objective is to minimize the sum of weighted completion times of all jobs. Constraints (2.20) ensure that every job is completed on some machine using some number of additional resources in some time interval. Constraints (2.21) impose a lower bound on the completion time of each job. Constraints (2.22) are validity constraints which state that the total processing time of jobs executed up to an interval I_ℓ on a processor $i \in \mathcal{M}$ is at most $(1+\delta)^\ell$. Constraints (2.23) impose upper bounds for the total resource usage until the time $(1+\delta)^\ell$. For each $\ell \in \mathcal{L}$, constraints (2.24) indicate that if processing time of a job j on a processor i takes more than $(1+\delta)^\ell$, then j cannot not be scheduled on i and complete its execution within I_ℓ. Note that even the corresponding integer program of the LP-formulation is a $(1+\delta)$-relaxation of the original problem.

Our algorithm, begins from a fractional solution $(\bar{x}_{i,j,s,\ell}, \bar{C}_j)$ of the LP and rounds it to an integral schedule keeping a constant approximation factor. In order to succeed it, the algorithm separates the jobs into sets $S(\ell) = \{j \in \mathcal{V} \mid (1+\delta)^{\ell-1} \leq \alpha \bar{C}_j \leq (1+\delta)^\ell\}$, where $\alpha > 1$ is a fixed constant. Then, it schedules integrally the jobs of each $S(\ell)$ on processors of $\mathcal{M}$(using the techniques of Grigoriev et al. [23]) and greedily places the produced schedules, one after the other, in an increasing order of ℓ.

Rounding Routine

1: Find a fractional solution to the LP $(\bar{x}_{i,j,s,\ell}, \bar{C}_j)$.
2: Partition the tasks into sets $S(\ell) = \{j \in \mathcal{V} \mid (1+\delta)^{\ell-1} \leq a\bar{C}_j \leq (1+\delta)^\ell\}$, where $\alpha > 1$ a fixed constant.
3: For each $\ell = 1 \ldots L$ in an increasing order, find an integral assignment and schedule of the jobs in $S(\ell)$ on $\mathcal{M}$ using the rounding theorem of [23].

In the following, we present the analysis of this algorithm.

Lemma 2.16. *For each $\ell = 1 \dots L$, the fractional assignment of jobs in $S(\ell)$ can be scheduled alone on the machines of $\mathcal{M}$ with a makespan at most $3\frac{\alpha}{\alpha-1}(1+\delta)^{\ell} + (1+\delta)^{\ell}$.*

Proof. We are going to use the well-known filtering technique of [27]. We can see that for a job $j \in S(\ell)$ it is the case that:

$$\sum_{t>\ell} \sum_{i\in\mathcal{M}} \sum_{s\in S_{i,j}} x_{i,j,s,t} \leq \frac{1}{\alpha}$$

Assuming the contrary yields that by constraints (2.21):

$$\begin{aligned}
C_j &\geq \sum_{i\in\mathcal{M}} \sum_{\ell\in\mathcal{L}} \sum_{s\in S_{i,j}} (1+\delta)^{\ell-1} x_{i,j,s,\ell} \\
&\geq \sum_{i\in\mathcal{M}} \sum_{t>\ell} \sum_{s\in S_{i,j}} (1+\delta)^{t-1} x_{i,j,s,t} \\
&+ \sum_{i\in\mathcal{M}} \sum_{t\leq\ell} \sum_{s\in S_{i,j}} (1+\delta)^{t-1} x_{i,j,s,t} \\
&\geq \sum_{i\in\mathcal{M}} \sum_{t>\ell} \sum_{s\in S_{i,j}} (1+\delta)^{t-1} x_{i,j,s,t} \\
&\geq (1+\delta)^{\ell} \sum_{i\in\mathcal{M}} \sum_{t>\ell} \sum_{s\in S_{i,j}} x_{i,j,s,t} \\
&> (1+\delta)^{\ell} \frac{1}{\alpha},
\end{aligned}$$

which is a contradiction to the definition of $S(\ell)$:

Therefore, using constraints (2.20) we can see that:

$$\sum_{t\leq\ell} \sum_{i\in\mathcal{M}} \sum_{s\in S_{i,j}} x_{i,j,s,t} \geq \frac{\alpha-1}{\alpha}$$

Now, for each job $j \in S(\ell)$ if we set $x'_{i,j,s,t} = 0$ when $t > \ell$ and

$$x'_{i,j,s,t} = \frac{x_{i,j,s,t}}{\sum_{i\in\mathcal{M},s} \sum_{t\leq\ell} x_{i,j,s,t}}$$

when $t \leq \ell$, the solution $< x' >$ would satisfy the constraints (2.20) and (2.24) of the LP. Moreover, the solution $< x' >$ would satisfy the constraints (2.22) and (2.23) if we multiply the right hand of the equation by the factor $\frac{\alpha}{\alpha-1}$. Given this, if we set

$$y_{i,j,s} = \sum_{t\leq\ell} x'_{i,j,s,t}$$

then $< y >$ consists a feasible solution to the following linear formulation for each $\ell \in \mathcal{L}$.

$$\sum_{i\in\mathcal{M}}\sum_{s\in S_{i,j}} y_{i,j,s} = 1 \qquad ,\forall j \in S(l) \tag{2.25}$$

$$\sum_{j\in S(l)}\sum_{s\in S_{i,j}} y_{i,j,s}p_{i,j,s} \leq \frac{\alpha}{\alpha-1}(1+\delta)^{\ell} \qquad ,\forall i \in \mathcal{M} \tag{2.26}$$

$$\sum_{j\in S(l)}\sum_{i\in\mathcal{M}}\sum_{s\in S_{i,j}} y_{i,j,s}sp_{i,j,s} \leq k\frac{\alpha}{\alpha-1}(1+\delta)^{\ell} \tag{2.27}$$

$$y_{i,j,s} = 0 \qquad ,\text{if } p_{i,j,s} > (1+\delta)^{\ell} \tag{2.28}$$

$$y_{i,j,s} \geq 0 \qquad ,\forall i,j,s$$

Note that the above formulation is the same as in Subsection 2.5.2.1. Therefore, using the algorithm for the makespan minimization, with target makespan $C^{\star} = \frac{\alpha}{\alpha-1}(1+\delta)^{\ell}$ and $p_{max} = (1+\delta)^{\ell}$, it follows that the jobs of $S(\ell)$ can be integrally scheduled alone on the processors of $\mathcal{P}$ with makespan at most $3C^{\star} + p_{max} = 3\frac{\alpha}{\alpha-1}(1+\delta)^{\ell} + (1+\delta)^{\ell}$. □

Theorem 2.17. *There is a* 25.55*-approximation algorithm for the problem of scheduling jobs on unrelated machines with resource dependent processing times.*

Proof. Let $j \in \mathcal{V}$ be a job that belongs to the set $S(\ell)$ and is scheduled by our algorithm on a processor i using s additional resources. For the completion time of j, taking the union of the schedules $S(t)$, for $t \leq \ell$, it must hold:

$$\begin{aligned}
C_j &\leq \sum_{t\leq\ell}(3x_{i,j,s,t} + (1+\delta)^t) \\
&\leq 3\sum_{t\leq\ell} x_{i,j,s,t} + \sum_{t\leq\ell}(1+\delta)^t \\
&\leq 3\sum_{t\leq\ell} x_{i,j,s,t} + \sum_{t\leq\ell}(1+\delta)^t \\
&\leq 3\frac{\alpha}{\alpha-1}(1+\delta)^{\ell} + \frac{1}{\delta}(1+\delta)^{\ell+1} \\
&\leq (3\frac{\alpha}{\alpha-1} + 1 + \frac{1}{\delta})(1+\delta)^{\ell} \\
&\leq \alpha(3\frac{\alpha}{\alpha-1} + 1 + \frac{1}{\delta})(1+\delta)\bar{C}_j
\end{aligned}$$

Recal that $\sum_{j\in\mathcal{V}} w_j\bar{C}_j \leq OPT$. Therefore, by choosing $(\alpha,\delta) = (1.65817, 0.341831)$ we get a 25.55 approximation algorithm. That is the value that minimizes the function representing the approximation guarantee. □

As we have proved in the previous section, the approximation ratio of 25.55 applies directly to the malleable case. To the extend of our knowledge this is the first result regarding the $\sum_{j\in\mathcal{J}} w_jC_j$ metric. Previous results, concern either the rigid problem [28], or the preemptive setting [29].

Chapter 3

General Multiprocessor Job Scheduling

In this chapter, we focus our attention on the most general version of the one-job-multiple-machines model. In this kind of setting a job can be processed in parallel on any subset of processors, and its execution time depends only on the particular subset it is assigned to. So, if $\mathcal{J}$ the set of tasks ($n = |\mathcal{J}|$) and $\mathcal{M}$ the set of machines ($m = |\mathcal{M}|$), then $\forall S \subseteq \mathcal{M}$, with $S \neq \emptyset$, and $\forall j \in \mathcal{J}$ there is a value $p_j(S)$ representing the corresponding processing time. In total there are $2^m - 1$ alternative set assignments for each job. In many situations a job is not allowed to run on some of these sets, thus there is a different, although equivalent formal definition for the problem. For each job $j \in \mathcal{J}$ there is a set of l_j available processing modes:

$$M_j = \{M_{j,k} = (Q_{j,k}, t_{j,k}) \mid 1 \leq k \leq l_j\}$$

$Q_{j,k}$ is a non-empty subset of $\mathcal{M}$ on which the processing time of of j is $t_{j,k} < \infty$. If a processor subset S is not contained in any mode, then we assume $p_j(S) = \infty$. Throughout the whole chapter, we will study the problem only under the makespan objective.

Furthermore, we are going to present results for both the preemptive and the non-preemptive case. In a non-preemptive problem, a job undergoing execution will never be suspended until its completion. On the other hand, in the preemptive setting one may suspend a job before its completion and resume its execution at a later point, possibly on a different mode. The strict meaning of the preemptive execution of a job j is the following. If j runs on mode $M_{j,k}$ for $p_{j,k}$ units of time, then it is necessary that:

$$\sum_{1 \leq k \leq l_j} \frac{p_{j,k}}{t_{j,k}} = 1$$

.

Finally, a special case of the problem will come in handy in our study. In this special case there is only one available mode for each job, i.e. $|M_j| = 1$. In other worlds, the assignment of sets to jobs is fixed beforehand and what remains is only the scheduling decisions. The *fixed* setting has been extensively studied in the literature and we will use some of these results in our advantage.

3.1 Inapproximability Results

In this section, we present inapproximability results for both the preemptive and the non-preemptive case. These results suggest that probably there is no efficient approximation, and thus we have to focus on restricted versions of the original problem.

3.1.1 The Non-Preemptive Case

We provide an approximation factor preserving reduction from the **Minimum Vertex Coloring Problem** [30]. We are going to use the *fixed* version of the scheduling problem, where additionally the processing time of all jobs in their only available mode is the same, say a constant $c > 0$. First of all, note that because all jobs have the same processing time in their available machine allocation, we can discretize the time horizon into intervals of length c. The maximum number of such intervals necessary for scheduling all jobs is obviously n (one for each job). Every job will run on one of these intervals in the optimal solution. So the scheduling problem is equivalent to finding the minimum number of intervals in which all jobs can be legally scheduled.

Theorem 3.1. *If there is an $\rho-$approximation algorithm for **General Multiprocessor Job Scheduling** then there is an $\rho-$approximation algorithm for the problem of **Minimum Vertex Coloring**.*

Proof. Suppose we have a graph $G(V, E)$ as an instance I of the coloring problem. We can create in polynomial time an instance I' of the multiprocessor problem.

- First for every $v \in V$ create a job $j_v \in \mathcal{J}$. Every job $j_v \in \mathcal{J}$ will have only one set of machines where it can be run, call it s_v (initially empty). Also, $p_{j_v}(s_v) = c > 0$, a constant common for every job.
- For every $\{u, v\} \in E$ create a machine $i_{uv} \in \mathcal{M}$ and $s_u \leftarrow s_u \cup \{i_{uv}\}$, $s_v \leftarrow s_v \cup \{i_{uv}\}$.

We will now prove that $OPT(I) = OPT(I')$ and given a solution of I' we can translate it to a solution of I with the same cost.

- Suppose we are given the optimal solution for the coloring problem. That is the minimum number of different colors needed for graph G. We can schedule all jobs (vertices) of the same color in the same time interval because these vertices don't share an edge in the graph and thus they don't have a common machine in the scheduling instance I'. So $OPT(I') \leq OPT(I)$.

- Suppose we are given the optimal solution for the scheduling problem. We know that all jobs are scheduled inside the time intervals of length c. So the jobs inside any time interval don't share a common machine. Furthermore, the corresponding vertices of the graph don't share edges and thus can have the same color. So $OPT(I) \leq OPT(I')$.

The procedure of the second bullet shows how to translate a solution of I' to a solution of I with the same cost. So, if we have an $\rho-$approximation algorithm for **Multiprocessor Job Scheduling**, then given any graph $G(V, E)$ we can built an instance of the scheduling problem as shown above and use our algorithm there to obtain:

$$SOL_{scheduling} \leq \rho \cdot OPT(I') = \rho \cdot OPT(I) \xrightarrow{\text{translation}} SOL_{coloring} \leq \rho \cdot OPT(I)$$

□

Finally, it is known that no polynomial time algorithm for vertex coloring can achieve an approximation guarantee of $\Omega(n^{\frac{1}{10}})$ unless $NP = co\text{-}NP$ [31]. Due to the above theorem this result will also hold for our problem.

3.1.2 The Peemptive Case

We provide a reduction from the **3-SAT Problem** [30], which was first proposed in [32]. We use the general version of the preemptive problem, where the processing time of every job in each of its modes is 1. In [33, 34] a method was proposed, that constructs from every $3CNF$ formula ϕ with m variables, a graph G_ϕ of $n = m^{\mathcal{O}(1)}$ nodes with the following properties:

- The n nodes are partitioned into r cliques $C_1, \ldots, C_r$.

- If ϕ is satisfiable, then $\alpha(G_\phi) = r$ (the cardinality of the maximum independent set is denoted by $\alpha(G_\phi)$). In other words, G_ϕ has an independent set with exactly one node from each clique.

- If ϕ is not satisfiable, then $\alpha(G_\phi) \leq \frac{r}{g}$, where $g = n^\epsilon$ for some positive constant ϵ.

Given such a graph G_ϕ, let $u_{j,1}, \ldots, u_{j,c_j}$ be the vertices of clique C_j. We then construct an instance I of the scheduling problem as follows:

- For every clique C_j we have a corresponding job j.
- For every node $u_{j,k} \in C_j$ we have a corresponding processing mode for job j, $M_{j,k} = (Q_{j,k}, 1)$.
- Each edge of the graph corresponds to a machine i of our scheduling problem.
- $Q_{j,k}$ contains all processors corresponding to edges incident to $u_{j,k}$.

Theorem 3.2. *There is no polynomial time ρ-approximation algorithm for* ***Preemptive Multiprocessor Scheduling*** *with $\rho < g$ unless $P = NP$.*

Proof. For a given $3CNF$ formula ϕ we construct G_ϕ and subsequently the corresponding instance I of the scheduling problem. If ϕ is satisfiable we know that there is an independent set $\{u_{1,k_1}, \ldots, u_{r,k_r}\}$ containing a vertex from each clique. Furthermore, due to the way of building I, there exists a schedule with length 1 by assigning j to Q_{j,k_j}. In this particular schedule, all jobs can run without preemption in parallel. Obviously, this is an optimal solution. Instead, if ϕ is not satisfiable, then $\alpha(G_\phi) \leq \frac{r}{g}$. That means that any time instance of any schedule no more than $\frac{r}{g}$ jobs can be executed simultaneously. Therefore, the makespan of any schedule will be greater than g, since there are r jobs in total. Given the existence of a polynomial time ρ-approximation algorithm, with $\rho < g$, we can determine in polynomial time the satisfiability of ϕ.

- ϕ satisfiable $\implies OPT_{scheduling} = 1$. The approximation algorithm will return a solution of cost $1 \leq SOL \leq \rho$.
- ϕ not satisfiable $\implies OPT_{scheduling} > g > \rho$. Thus, the approximation algorithm will return a solution of cost $SOL \geq \rho$.

However, 3-SAT is known to be NP-complete. □

3.2 Linear Array Networks

In this section, we study a restricted version of the problem, introduced in [32]. This version is motivated by actual computer networks. In practice, the parallel processors are connected together by a network of a specific topology, and in those systems jobs require that the machines they utilize, satisfy certain topological properties. In linear array networks, the machines form a one dimensional array. Jobs in such networks can only run in sets of consecutive machines, regarding the ordering of the array. We present algorithms for both the preemptive and the non-preemptive case [32].

3.2.1 The Non-Preemptive Problem

Suppose we have jobs $\mathcal{J}$, machines $\mathcal{M}$ and processing modes M_j as presented earlier. In this case, $\forall j \in \mathcal{J},\ \forall k \in [1, l_j] : Q_{j,k}$ is a set of consecutive elements of $\mathcal{M}$. This problem is clearly NP-hard, since it generalizes $R||C_{max}$. We give the following integer programming relaxation of the problem, where $x_{j,k}$ are binary variables indicating that job j runs on its k-th mode. Also, the variable C is the target makespan.

$$\sum_{k\in[1,l_j]} x_{j,k} = 1,\ \forall j \in \mathcal{J} \tag{3.1}$$

$$\sum_{j\in\mathcal{J}} \sum_{\forall k:\ i\in Q_{j,k}} x_{j,k} \cdot t_{j,k} \leq C,\ \forall i \in \mathcal{M} \tag{3.2}$$

$$x_{j,k} = 0,\ \text{if } p_{j,k} > C \tag{3.3}$$

$$x_{j,k} \in \{0, 1\},\ \forall j, k \tag{3.4}$$

Following the same reasoning of section **2.5.2.1** it is easy to verify that for C equal the optimal value of the problem, integer program (3.1)-(3.4) is a relaxation of the original problem, and hence so is the linear relaxation of (3.1)-(3.4). Again, by using binary search on the target makespan C, we can find in polynomial time a solution (x^{LP}, C^{LP}) of the linear program, such that $C^{LP} \leq OPT$.

From the fractional solution x^{LP} we construct a bipartite multigraph $B(x^{LP}) = (U, V, E)$ as follows. In U there is a vertex u_j corresponding to each job $j \in \mathcal{J}$ and similarly, in V there is a vertex v_i for every machine $i \in \mathcal{M}$. For each $x^{LP}_{j,k} > 0$, find the maximum integer $h \in [0, \lceil logm \rceil - 1]$, such that there exists an integer c with $c \cdot 2^h \in Q_{j,k}$. Then add $e = (u_i, v_{c2^h})$ in E. $c \cdot 2^h$ works as a representative for $Q_{j,k}$. Let $level(e) = h$, $w_1(e) = x^{LP}_{j,k}$ and $w_2(e) = t_{j,k}$. Note also that c must always be an odd number. If not, we can increment h by choosing $\frac{c}{2}$.

By using the multigraph rounding method of Grigoriev et al.[23] we acquire a graph B', in which each job node is incident to only one edge and the total load of every machine node (calculated as the sum of $w_2(e)$ for edges incident to it) is at most $2C^{LP}$. Let B'_h be the subgraph of B' induced by edges of level h. Observe that the sets U'_h partition U.

Lemma 3.3. *For each h, the jobs of U'_h can be scheduled with makespan at most $2C^{LP}$.*

Proof. We will show that for any two nodes $u_j,\ u_z \in U'_h$ that do not share an end vertex, the corresponding processing modes of their incident edges are disjoint. Let $e_j = (u_j, v_{c_j \cdot 2^h}),\ e_z = (u_z, v_{c_z \cdot 2^h})$ the edges incident to the two nodes. Let also $Q_j,\ Q_z$ the processing modes that resulted in these two edges. Suppose that without loss of generality $c_j < c_z$, and also $Q_j \cap Q_z \neq \emptyset$ for contradiction.

Since c_j, c_z are odd numbers and Q_j, Q_z contain consecutive elements of $\mathcal{M}$, we can conclude that there exists an even number c, such that $c_j < c < c_z$ and $c \cdot 2^h \in Q_j \ \vee \ c \cdot 2^h \in Q_z$. But this means that $\frac{c}{2} \cdot 2^h \in Q_j \ \vee \ \frac{c}{2} \cdot 2^h \in Q_z$, which is clearly a contradiction because of the method of selecting h.

Since $Q_j \cap Q_z = \emptyset$ for any pair of edges that do not share an end vertex, we do not need to concern about the ordering of the tasks of $U_h^{'}$ in their schedule. By assigning each job to the mode specified by its incident edge, we get a feasible schedule with makespan at most $2C^{LP} \leq 2OPT$. □

Theorem 3.4. *There is an $\mathcal{O}(logm)$-approximation algorithm for* Multiprocessor Job Scheduling *in linear array networks.*

Proof. The previous lemma states that for each $h \in [0, \lceil logm \rceil - 1]$, we can schedule the jobs of $U_h^{'}$ with total makespan at most $2OPT$. Due to the fact that $U_h^{'}$ partition the job set, we can simply merge all the resulting schedules serially and obtain an $\mathcal{O}(logm)$-approximation. □

Regarding the approach of solving the linear relaxation and rounding the fractional solution, the authors of [32] also provide a result concerning the integrality gap[35] of program (3.1)-(3.4).

Theorem 3.5. *There is an instance of the problem such that the linear program has a feasible solution, but no schedule of makespan less than $\Omega(C^{LP} \cdot \frac{logm}{loglogm})$ exists.*

3.2.2 The Preemptive Problem

In this case everything is similar to the previous section's problem, except the fact that we allow preemption of jobs. This version it is also NP-hard, since it embeds in itself the **Subset Sum** problem [30]. Consider the following linear programming relaxation of the problem. Again C stands for the target makespan.

$$\sum_{k \in [1, l_j]} x_{j,k} = 1, \ \forall j \in \mathcal{J} \tag{3.5}$$

$$\sum_{j \in \mathcal{J}} \sum_{\forall k: \ i \in Q_{j,k}} x_{j,k} \cdot t_{j,k} \leq C, \ \forall i \in \mathcal{M} \tag{3.6}$$

$$\sum_{k \in [1, l_j]} x_{j,k} \cdot t_{j,k} \leq C, \ \forall j \in \mathcal{J} \tag{3.7}$$

$$x_{j,k} = 0, \text{ if } p_{j,k} > C \tag{3.8}$$

$$x_{j,k} \geq 0, \ \forall j, k \tag{3.9}$$

It is easy to verify that for C equal the optimal value of the problem, integer program (3.5)-(3.9) is a relaxation of the original problem. So, again by using binary search on the target makespan C, we can find in polynomial time a solution (x^{LP}, C^{LP}), such that $C^{LP} \leq OPT$. This particular solution partitions each job $j \in \mathcal{J}$ into subjobs $T_{j,k}$ for $1 \leq k \leq l_j$. Subjob $T_{j,k}$ requires the set $Q_{j,k}$ for $x^{LP}_{j,k} \cdot t_{j,k}$ units of time. Obviously, we do not mind having a fractional solution now. Moreover, no two subtasks of the same job can run in parallel, and that is expressed via constrained (3.7).

A 2-approximation greedy algorithm

1: Let $t_{cur} = 0$ and T the set of all subtasks created after solving the previous linear program.
2: Sort the subtasks in T according to the leftmost processor they require, with respect to the machine ordering. Consider the subtasks of T in this order one by one. For $T_{j,k}$, if all processors in $Q_{j,k}$ are idle at time t_{cur} and also no other subtask of the initial job j is already scheduled to start at t_{cur}, then start $T_{j,k}$.
3: Let t_{min} the minimum time that a subtask which started in the previous step completed its processing. Suspend all running subtasks at that time, define their remaining portions and add them again in T. Set $t_{cur} \leftarrow t_{min}$ and go back to Step 2, until $T = \emptyset$.

First of all, observe that the above algorithm is polynomial in terms of its running time. In each iteration at least one of the original subtasks is completed, and their total number f is bounded by the size of the input, $f \leq \sum_{j \in \mathcal{J}} l_j$.

Theorem 3.6. *Our algorithm is a 2-approximation for the problem at hand.*

Proof. Let $T_{p,q}$ the subtask that finishes last in the constructed schedule. Let $st_{p,q}$ be its starting time and $f_{p,q}$ its finishing time. Furthermore, let r be the leftmost processor of $Q_{p,q}$. It is clear, that at any time instance $t < s_{p,q}$ either r is busy or another subtask of the original job p is undergoing execution. Because we consider the subtasks in an increasing order of their leftmost machine, if r is not busy at some time t, then all processors of $Q_{p,q}$ are not busy either. That means that if the above two conditions do not hold, then $T_{p,q}$ can start its execution earlier, something that contradicts the scheduling rule.

Therefore, we conclude that $f_{p,q}$ is at most the total execution time of the subtasks occupying r plus the total execution time of the subtasks created from p. From constraints (3.6) and (3.7) we get that:

$$f_{p,q} \leq 2C^{LP} \leq 2OPT$$

. □

3.3 Fixed Number of Machines

As we have already mentioned, the hardness results of section *3.1* dictate that good approximation algorithms can only be found for restricted versions of the problem. In this section, we study the problem assuming that the number of machines m is fixed and not part of the input. More precisely, we present some fairly simple algorithms for the $m = 2$ and $m = 3$ case, as well as a sophisticated $PTAS$ for general fixed m.

3.3.1 The $m = 2$ Case

First of all, observe that the problem at hand is actually an extension of the classical $R_2||C_{max}$ problem, and thus it is NP-hard. We present an optimal pseudopolynomial algorithm for the problem, proposed in [36].

In problems of such nature, there are actually two kinds of decisions that need to be made. At first, there is the **assignment**, where for each job $j \in \mathcal{J}$ we have to choose the proper processing mode. Then comes the **scheduling**, where we decide when to start the execution of each job. We will deal separately with these decisions, starting from finding a desirable assignment. We would like to find an assignment $A^\star$ of modes to jobs, such that the maximum load of a machine under this assignment is the minimum possible. The load of a machine under a given assignment $A^\star$ is calculated as the sum of processing times of jobs, that will occupy that machine in the mode chosen for them. Let us denote by $T_{A^\star}$ the maximum machine load under the desired assignment. Obviously:

$$T_{A^\star} \leq OPT$$

Let $\mathcal{M} = \{1, 2\}$ and $\forall j \in \mathcal{J} :\ p_{min}(j) = min\{p_j(\{1\}), p_j(\{2\}), p_j(\{1, 2\})\}$. Define

$$T_0 = \sum_{j \in \mathcal{J}} p_{min}(j)$$

T_0 is just an upper bound estimation of the optimal makespan.

We present a dynamic programming approach that calculates the value $T_{A^\star}$ for a two machine system. Define the value $f(j, x)$ for $j \geq 1$, $0 \leq x < \infty$ to be the minimum y such that there exists an assignment A for jobs $1, \ldots, j$, for which the total processing time on machine 1 is x, and the total processing time on machine 2 is y, where x, y finite numbers. If there does not exist an assignment for these jobs, with the load of machine 1 to be x and the load of machine 2 to be y, then we set $f(j, x) = \infty$. The dynamic program is based on the fact that each new job has only three mode alternatives. That means that we know exactly how it will affect the load of each machine, and thus we can incorporate that knowledge into the recursive equations.

Initial conditions:

$$f(1,0) = p_1(\{2\})$$

$$f(1, p_1(\{1\})) = 0$$

$$f(1, p_1(\{1,2\})) = p_1(\{1,2\})$$

$$f(1,x) = \infty \text{ for all other } x$$

Recursive Equations:

$$f(j,x) = min\{\ f(j-1, x-p_j(\{1\})),\ f(j-1,x)+p_j(\{2\}),\ f(j-1, x-p_j(\{1,2\}))+p_j(\{1,2\})\ \}$$

,where $2 \leq j \leq n$ and $0 \leq x \leq T_0$

The time complexity of solving the above dynamic program is $\mathcal{O}(n \cdot T_0)$. It is also clear that $min\{max(x, f(n,x)) :\ 0 \leq x \leq T_0\} = T_{A^\star}$ and that concludes our search for the optimal value for the assignment problem. Now, all that is left to do is the scheduling. This is actually fairly simple for the two machines case. We use the following algorithm.

Pseudopolynomial Algorithm for $m = 2$

1: Solve the aforementioned DP to find the proper assignment $A^\star$.
2: Schedule the jobs that are assigned to set $\{1,2\}$ in arbitrary order.
3: Schedule the jobs that are assigned to sets $\{1\}$ and $\{2\}$ in parallel, and in any arbitrary order.

Figure **3.1** provides a graphical representation of such a schedule. Observe, that under this schedule there is no idle time between jobs on each machine. Hence, the completion time will equal $T_{A^\star}$ and thus it is optimal.

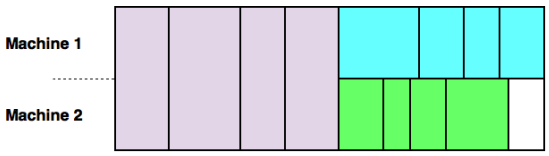


FIGURE 3.1: Scheduling for the $m = 2$ case.

3.3.2 The $m = 3$ Case

Hoogeveen et al. [37] showed that even the *fixed* version of this problem is strongly NP-hard. The lowest approximation ratio achieved so far for the general case of the $m = 3$ problem is a $\frac{7}{6} + \epsilon$, proposed by Miranda [38] and for the *fixed* version a $\frac{7}{6}$, due to Goemans [39]. In this section, we tackle the general $m = 3$ problem, and we present a slightly simpler approximation algorithm of ratio $\frac{3}{2} + \epsilon$, found in [36].

Following the reasoning of the previous section we first find an assignment $A^\star$ that minimizes the maximum load of a machine, $T_{A^\star}$. We use a modified dynamic programming approach. Define the function $f(j, x, y)$ as the minimum value of the total processing time assigned to machine 3, given that there exists an assignment for jobs $1, \ldots, j$, where x is the total processing time of machine 1 and y the total processing time of machine 2. In case there is no such assignment we set $f(j, x, y) = \infty$.

Initial conditions:

$$f(1, p_1(\{1\}), 0) = 0$$

$$f(1, p_1(\{1,2\}), p_1(\{1,2\})) = 0$$

$$f(1, p_1(\{1,3\}), 0) = \begin{cases} p_1(\{1,3\}), \text{ if } p_1(\{1,3\}) \neq p_1(\{1\}) \\ 0, \text{ otherwise} \end{cases}$$

$$f(1, 0, p_1(\{1,2\})) = 0$$

$$f(1, 0, p_1(\{2,3\})) = \begin{cases} p_1(\{2,3\}), \text{ if } p_1(\{2,3\}) \neq p_1(\{2\}) \\ 0, \text{ otherwise} \end{cases}$$

$$f(1, 0, 0) = p_1(\{3\})$$

$$f(1, p_1(\{1,2,3\}), p_1(\{1,2,3\})) = \begin{cases} p_1(\{1,2,3\}), \text{ if } p_1(\{1,2,3\}) \neq p_1(\{1,2\}) \\ 0, \text{ otherwise} \end{cases}$$

$f(1, x, y) = \infty$ for all other x

Recursive Equations:

$$f(j,x,y) = \begin{cases} f(j-1, x - p_j(\{1\}), y) \\ f(j-1, x, y - p_j(\{2\})) \\ f(j-1, x, y)) + p_j(\{3\} \\ f(j-1, x - p_j(\{1,2\}, y - p_j(\{1,2\})) \\ f(j-1, x - p_j(\{1,3\}), y) + p_j(\{1,3\}) \\ f(j-1, x, y - p_j(\{2,3\})) + p_j(\{2,3\}) \\ f(j-1, x - p_j(\{1,2,3\}), y - p_j(\{1,2,3\})) + p_j(\{1,2,3\}) \end{cases}$$

,where $2 \leq j \leq n$ and $0 \leq x, y \leq T_0$

The time complexity of solving the above dynamic program is $\mathcal{O}(n \cdot T_0^2)$. It is also clear that $min\{max(x, y, f(n, x, y)) : \ 0 \leq x, y \leq T_0\} = T_{A^\star}$ and that concludes our search for the optimal value for the assignment problem.

The authors of [36] also provide a fully polynomial time approximation scheme for the assignment problem, which is based on the above pseudopolynomial algorithm. More precisely, they use the standard scaling technique by dividing all processing times by a constant $K = \frac{\epsilon T_0}{3n}$, for any $\epsilon > 0$. In the job system that results from this transformation, they apply the aforementioned pseudopolynomial algorithm that constructs an assignment A^{apx} for the new system. Keeping the same job-processor assignments, A^{apx} is also an assignment for the original job system, with maximum machine load $T_{A^{apx}}$. They prove the following theorem.

Theorem 3.7. *The assignment A^{apx} for the original job set can be constructed in time $\mathcal{O}(\frac{n^3}{\epsilon^2})$ and also $T_{A^{apx}} \leq (1+\epsilon)T_{A^\star}$.*

Given the assignment A^{apx} the only thing left is the scheduling algorithm. Let us denote by $l(1)$ the sum of the processing times of the jobs assigned to $\{1\}$ under A^{apx}. Similarly, we define $l(2)$, $l(3)$. Without loss of generality, assume that $l(1) \geq l(2) \geq l(3)$. Otherwise, simply reindex the machines. We present an algorithm that schedules the jobs of $\mathcal{J}$, whilst respecting A^{apx}.

A $\frac{3}{2} + \epsilon$-approximation greedy algorithm

1: Starting from time 0, execute the jobs assigned to $\{1,2,3\}$ on all three machines, in any arbitrary order. Suppose that the last of them finishes at time $f_{1,2,3}$.
2: Starting from time $f_{1,2,3}$, execute the jobs assigned to $\{1,2\}$ on these two machines, in any arbitrary order. Suppose that the last of them finishes at time $f_{1,2}$.
3: Starting from time $f_{1,2}$, execute the jobs assigned to $\{1,3\}$ on these two machines, in any arbitrary order. Suppose that the last of them finishes at time $f_{1,3}$.
4: Starting from time $f_{1,3}$, execute the jobs assigned to $\{2,3\}$ on these two machines, in any arbitrary order. Suppose that the last of them finishes at time $f_{2,3}$.
5: Starting from time $f_{1,3}$, execute the jobs assigned to $\{1\}$, in any arbitrary order. Starting from time $f_{2,3}$, execute the jobs assigned to $\{2\}$, in any arbitrary order. Starting from time $f_{2,3}$, execute the jobs assigned to $\{3\}$, in any arbitrary order.

Theorem 3.8. *Let C be the makespan of the schedule constructed by the above algorithm. Then $C \leq \frac{3}{2}(1+\epsilon)T_{A^\star}$.*

Proof. Since $T_{A^{apx}} \leq (1+\epsilon)T_{A^\star}$, it suffices to show that $C \leq \frac{3}{2}T_{A^{apx}}$. Let f_1, f_2, f_3 the completion times of the three machines, respectively. Obviously, $C = max\{f_1, f_2, f_3\}$. As a result of the scheduling algorithm and the assumption made about $l(1)$, $l(2)$, $l(3)$, we always have $f_2 \geq f_3$. In order to complete the proof, we need to distinguish between two cases.

- $f_1 \geq f_2$. Hence $C = f_1$. Clearly, $C = T_{A^{apx}}$, since there is no idle time on processor 1. See figure **3.2(A)**.

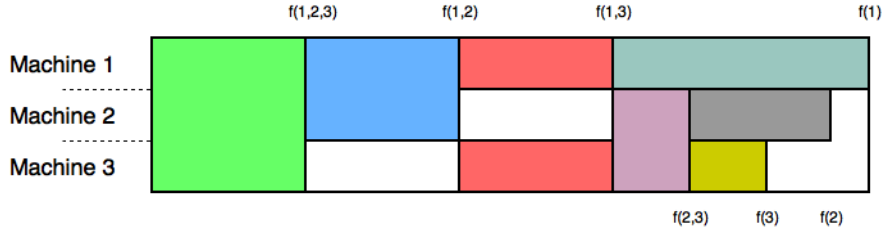
- $f_1 < f_2$. Hence $C = f_2$. In this case, the total sum of processing times on all three machines can be expressed as follows:

$$\begin{aligned} T_{total} &= 3f_{1,2,3} + 2(f_{1,3} - f_{1,2}) + 2(f_{2,3} - f_{1,3}) \\ &\quad + (f_1 - f_{1,3}) + (f_2 - f_{2,3}) + (f_3 - f_{2,3}) \\ &\leq 2f_{2,3} + 2(f_2 - f_{2,3}) \\ &= 2f_2 = 2C \end{aligned}$$

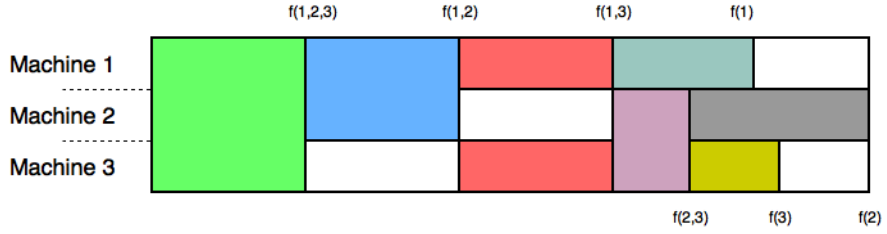
The inequality follows because $(f_1 - f_{1,3}) = l(1) \geq l(2) = (f_2 - f_{2,3})$. For a better understanding see **Figure 3.2(B)**. Observe also that $T_{A^{apx}} \geq \frac{T_{total}}{3}$. Suppose otherwise. Then $3T_{A^{apx}} < T_{total}$, which is clearly a contradiction since $T_{A^{apx}}$ is the greatest load among the three machines. Combining all the above we get

$$C \leq \frac{3}{2} T_{A^{apx}}$$

□



(A) $f_1 \geq f_2$



(B) $f_1 < f_2$

FIGURE 3.2: Scheduling for the $m = 3$ case.

To conclude, the algorithm described here is tight in terms of its approximation ratio. There exists an instance in which the ratio is arbitrarily close to $\frac{3}{2}$.

3.3.3 The General Fixed m Case

This particular version of the problem is still NP-hard in the strong sense, as it constitutes a generalization of the $m = 3$ case. Regarding the fixed modes setting, the best known result is a $PTAS$ developed by Amoura et al. [40]. However, in this section we are going to present a $PTAS$ for the most general case, in which every job may have many alternative processing modes. This specific result is due to Chen et al. [41].

The aforementioned $PTAS$ is based on a special class of schedules, namely the (m, ϵ)-canonical schedules, which will be formally defined later. The authors prove that for any instance of the problem, there is an (m, ϵ)-canonical schedule, whose makespan is very close to that of the optimal. Subsequently, they provide a scheme that approximates that particular schedule and thus get a good approximation for the initial problem.

Before we continue with the analysis we need to introduce some combinatorial facts. First, the number of different partitions of a set of m elements is called the m-th Bell number [42], and it is denoted by B_m. It can be proved easily by induction that $B_m \leq m!$. Furthermore:

Lemma 3.9. *Let $T = \{t_1, t_2, \ldots, t_n\}$ be a non-increasing sequence of integers, with $m \geq 2$ a fixed integer and $\epsilon > 0$ an arbitrary real number. Then there is an index j_0 such that:*

- $j_0 = (3mB_m + 1)^k$, *where* $k \leq \lfloor \frac{m}{\epsilon} \rfloor$.
- *For any subset T' of at most $3j_0mB_m$ integers t_q in T with $q > j_0$, we have:*

$$\sum_{t_q \in T'} t_q \leq \frac{\epsilon}{m} \sum_{i=1}^{n} t_i$$

We will denote by $j_{m,\epsilon}$ the smallest index that satisfies the above conditions and we will call it the *cut-index* for the sequence T. Observe also that given the existential guarantee of **Lemma 3.9**, $j_{m,\epsilon}$ can be found in polynomial time, since m and ϵ are fixed constants.

3.3.3.1 The (m, ϵ)-Canonical Schedules

This special class of schedules is only defined for the version of the problem in which the processing modes are fixed. Suppose then that each job $j \in \mathcal{J}$ can run only in $Q_j \subseteq \mathcal{M}$, with processing time t_j. Now define the sequence $T = \{t_1, t_2, \ldots, t_n\}$ and without loss of generality assume that it is non-increasing. For the fixed number m of

processors and for any arbitrary $\epsilon > 0$ find the cut-index $j_{m,\epsilon}$. We then split the job set $\mathcal{J}$ into two subsets:

$$\mathcal{J}_L = \{j \in \mathcal{J} \mid j \leq j_{m,\epsilon}\} \text{ ,the large jobs}$$

$$\mathcal{J}_S = \{j \in \mathcal{J} \mid j > j_{m,\epsilon}\} \text{ ,the small jobs}$$

Consider any schedule for the job set $\mathcal{J}$. Let $\{y_1, y_2, \ldots, y_h\}$ be the non-decreasing sequence of the starting and finishing times of the $j_{m,\epsilon}$ large jobs. A *small job block* χ consists of the subset $\mathcal{M}' \subseteq \mathcal{M}$ of processors that are not used by large jobs in the time interval $[y_p, y_{p+1}]$. Therefore, the small job blocks in the studied schedules are those "areas" that small jobs actually run. At any time moment τ in $[y_p, y_{p+1}]$ some small jobs run, each utilizing a processor subset Q_j. The collection $[Q_1, \ldots, Q_s]$ of all the processor subsets of small jobs that run at time τ in the small job block χ will be called the type of τ. A *layer* in χ is a maximal interval in $[y_p, y_{p+1}]$ such that all moments have the same type. The type of any moment τ of a layer is defined as the type of the layer.

Let L_1 and L_2 be two layers of the same small job block χ, with types $[Q_1, \ldots, Q_s]$ and $[R_1, \ldots, R_l]$ respectively. L_1 covers L_2 if $[R_1, \ldots, R_l] \subseteq [Q_1, \ldots, Q_s]$. In particular, if L_1 and L_2 are two consecutive layers in χ such that L_2 starts right after L_1 finishes and L_1 covers L_2, then L_2 is actually a continuation of L_1 with some of the small jobs finished.

Definition 3.10. A floor σ in the small job block χ is a sequence of consecutive layers $\{L_1, L_2, \ldots, L_z\}$. Moreover, all jobs interlacing L_1 start in L_1 and all jobs interlacing L_z finish in L_z.

An example of a floor follows in **Figure 3.3**. Note that a small job block may not have any non-empty floor at all. Furthermore, it is easy to see that any greedy list-scheduling argument can construct from the jobs $\mathcal{J}_\sigma$ of a floor σ, another floor σ' with the same height.

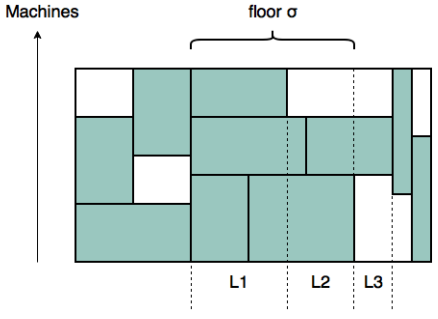


FIGURE 3.3: Example of a floor.

Definition 3.11. Let $[Q_1, \ldots, Q_s]$ be a partition of $\mathcal{M}'$. We say that we can assign the type $[Q_1, \ldots, Q_s]$ to a floor $\sigma = \{L_1, L_2, \ldots, L_z\}$ if the type of layer L_1 is a subcollection of $\{Q_1, \ldots, Q_s\}$.

Definition 3.12. A small job block χ is a tower if it is constituted by a sequence of floors each with a unique type.

Note that since each floor corresponds to a different partition of $\mathcal{M}'$, in a tower inside a small job block we can have at most $B_m \leq m!$ floors. In our discussion, we will concentrate only on schedules of the following form.

Definition 3.13. Let $\mathcal{J}$ be the job set of an instance to the fixed modes problem. For fixed $m \geq 2$ and $\epsilon > 0$ we divide the jobs into the two sets $\mathcal{J}_L$, $\mathcal{J}_S$. A schedule for $\mathcal{J}$ is called (m, ϵ)-canonical if every small job block is a tower.

Concerning the above described class of schedules, the authors of [41] have proved two very crucial theorems. We are going to present these two results without their proofs, as they are quite detailed and their analysis goes beyond the scope of our thesis.

Theorem 3.14. *Let $C(\mathcal{J})$ be an (m, ϵ)-canonical schedule for the job set $\mathcal{J}$. Let π be the sequence of large jobs and towers (regarded as a single job) in $C(\mathcal{J})$, ordered in terms of their starting times. The the classical list-scheduling algorithm based on the order π constructs a schedule with makespan not larger than that of $C(\mathcal{J})$.*

Theorem 3.15. *Let $\mathcal{J}$ be an instance of the problem with the fixed mode assignments, with optimal solution OPT. For any $\epsilon > 0$, there is an (m, ϵ)-canonical schedule of $\mathcal{J}$ whose makespan is upper bounded by $(1 + \epsilon)OPT$.*

Before we continue with the presentation of the approximation scheme we need another useful definition.

Definition 3.16. Let σ be a floor of type $[Q_1, Q_2, \ldots, Q_s]$ and height l. Then subset Q_j plus the height l is called a room of type Q_j in σ.

3.3.3.2 The Approximation Scheme

Now we come back to the original problem, in which each job $j \in \mathcal{J}$ may have many alternative processing modes $M_{j,k} = (Q_{j,k}, t_{j,k})$. The main idea behind this scheme is that the fixed m allows us to use a quite extensive brute force approach. Initially, we define $min_j = \min_{1 \leq k \leq l_j} t_{j,k}$ and $T_0 = \sum_{j \in \mathcal{J}} min_j$. The value T_0 is an obvious upper bound for the optimal makespan. Furthermore, for the time being let us assume that we know the partition of large and small jobs in the optimal solution and the order of

large jobs and towers there. We would then like to place the small jobs inside the known towers and arrange large jobs and towers in a proper way.

To do so, we are going to use a dynamic program based on an boolean array D. For each tower χ_q, where $1 \leq q \leq 2j_{m,\epsilon}+1$, associated with the processor subset M_q, the total number of floors is $f_q = B_{|M_q|} \leq B_m \leq m!$. Let $\sigma_{q,1}, \sigma_{q,2}, \ldots, \sigma_{q,f_q}$ all the different floors of tower χ_q. For each floor $\sigma_{q,f}$, let $\gamma_{k,f,1}, \gamma_{k,f,2}, \ldots, \gamma_{k,f,r_{k,f}}$ the $r_{k,f} \leq m$ rooms of that floor. Therefore, the configuration of small jobs in the towers, when we try to obtain an (m,ϵ)-canonical schedule, is specified by a $((2j_{m,\epsilon}+1)mB_m)$ tuple indicating the running time of all possible different rooms. Finally, the needed array element $D[j, t_{1,1,1}, \ldots, t_{q,f,r}, \ldots, t_{2j_{m,\epsilon}+1,B_m,m}]$ has the value TRUE if and only if there is a schedule of the first j small jobs, such that the running time of room $\gamma_{q,f,r}$ is $t_{q,f,r}$. By assuming that all elements of D have initially the value false we use the following DP algorithm.

SCHEDULE-SMALL

1: **Input**: The set $\mathcal{J}_S$ of small jobs and the order π of large jobs and towers.
2: $D[0,0,\ldots,0] \leftarrow$ TRUE
3: **for** $j = 1$ to n_S **do**
4: **for** each mode $Q_{j,k}$ of small job j **do**
5: **for** each $D[j-1,\ldots,t_{q,f,r},\ldots] =$ TRUE such that j can be added to room $\gamma_{q,f,r}$ under mode $Q_{j,k}$ without exceeding T_0 **do**
6: $D[j,\ldots,t_{q,f,r}+t_{j,k},\ldots] \leftarrow$ TRUE
7: **for** each $D[n_S,\ldots,t_{q,f,r},\ldots] =$ TRUE **do**
8: Call the list scheduling algorithm based on the order π.
9: Return the schedule with the minimum makespan.

Combining the results of **Theorem 3.14**, **Theorem 3.15** and the fact the the brute force search described above will eventually find the jobs to rooms assignments of the near optimal (m,ϵ)-canonical schedule, we get that the result of **SCHEDULE-SMALL** will have a makespan bounded by OPT. Moreover, with careful analysis we can see that its running time is $\mathcal{O}(n \cdot 2^m \cdot T_0^{(2j_{m,\epsilon}+1)mB_m} \cdot ((2j_{m,\epsilon}+1)mB_m))$, which is pseudopolynomial due to the factor T_0. Fortunately, that can be corrected via the standard scaling technique.

The only thing left now is to determine the partition of jobs into $\mathcal{J}_S$ and $\mathcal{J}_L$, the modes of large jobs, the modes of towers and the sequence of towers and large jobs. Again a brute force approach can guarantee all the above. Furthermore, the processing times will be scaled down by a factor K, which will be defined later. So the final $PTAS$ follows.

Obviously, if we forget about the scaling down the above algorithm will indeed find a schedule with makespan bounded by $(1+\epsilon)OPT$. However, the authors of [41] prove that the scaling down and then the expansion procedure actually burden the final makespan

APPROX-SCHEME

1: $K \leftarrow \frac{\epsilon T_0}{nm}$
2: Let $\mathcal{J}'$ be the job set obtained by scaling down all processing times by K.
3: **for** $k = 0$ to $\lfloor \frac{m}{\epsilon} \rfloor$ **do**
4: $\quad j_0 \leftarrow (3mB_m + 1)^k$
5: $\quad$ **for** each subset ${\mathcal{J}'}_L$ of j_0 large jobs in $\mathcal{J}'$ **do**
6: $\quad\quad$ **for** each mode assignment to the jobs in ${\mathcal{J}'}_L$ **do**
7: $\quad\quad\quad$ **for** each mode assignment to the $2j_0 + 1$ towers **do**
8: $\quad\quad\quad\quad$ **for** each possible sequence π of towers and large jobs **do**
9: $\quad\quad\quad\quad\quad$ Call **SCHEDULE-SMALL**
10: In the schedule with the minimum makespan resulted form step 9, expand the processing time of all jobs in order to avoid the effects of the scaling down and return that schedule.

by a factor nK. Furthermore, it is quite clear that $\frac{T_0}{m} \leq OPT$. Thus:

$$\begin{aligned} C &\leq (1+\epsilon)OPT + nK \\ &= (1+\epsilon)OPT + \frac{\epsilon T_0}{m} \\ &\leq (1+2\epsilon)OPT \end{aligned}$$

To conclude, the running time of the described approximation scheme is proved to be

$$\mathcal{O}(\left\lfloor \frac{m}{\epsilon} \right\rfloor \cdot n^{j_{m,\epsilon}} \cdot 2^{mj_{m,\epsilon}} \cdot 2^{2mj_{m,\epsilon}+m} \cdot (3j_{m,\epsilon}+1)!)$$

Chapter 4

Malleable Job Scheduling In Uniform Machines

In this chapter, we introduce a new model regarding malleable jobs. Again we have a set of tasks $\mathcal{J}$, $n = |\mathcal{J}|$, that need to be scheduled using a set of *uniform* or *related* machines $\mathcal{M}$, $m = |\mathcal{M}|$. What makes the difference is that in this case the machines may have different processing rates, and that is represented by a speed factor. Thus, each machine $i \in \mathcal{M}$ has a speed $s_i \geq 1$. Furthermore, following the malleable paradigm, each job can run using several machines simultaneously and in unison, as in all previous models. However, the processing time of a job will now depend on the total speed it utilizes. To be more precise, suppose that $j \in \mathcal{J}$ is processed on $S \subseteq \mathcal{M}$, with $S \neq \emptyset$. Then its execution time is $p_j(\sum_{i \in S} s_i)$.

To the extent of our knowledge this particular model have never been studied in the literature, despite its practical significance. From a practical viewpoint, there are many parallel malleable processing applications, in which the processing units have different computational capacities. Moreover, since this particular model constitutes a generalization of *Malleable Job Scheduling* in identical machines, its study is quite interesting and thought-provoking from an algorithmic viewpoint as well. Observe also, that due to that relation all problems regarding our model remain NP-hard.

Finally, in this diploma thesis we are focusing our attention in a restricted version of the above described setting. Specifically, we study processing time functions with a predetermined structure, rather than the totally abstract case. This simplification, which has actually some practical value, helped us comprehend the unique difficulties and characteristics of the problem at hand, and so produce some approximation results. In addition, this assumption about the $p_j(s)$ functions gave us an insight into how to deal with the general problem too.

4.1 The Model

We begin by formally presenting the problem we are interested in.

- We have a set $\mathcal{J} = \{1, \ldots, n\}$ of jobs, a set $\mathcal{M} = \{1, \ldots, m\}$ of related processors and for every machine $i \in \mathcal{M}$ a speed $s_i \geq 1$. Every job $j \in \mathcal{J}$ has a fixed amount of work $w_j > 0$ and a fixed delay $d_j > 0$. When job $j \in \mathcal{J}$ runs on $S \subseteq \mathcal{M}$, with $S \neq \emptyset$, its processing time is $p_j(S) = d_j + \frac{w_j}{\sum_{i \in S} s_i}$.
- We want to find a feasible non-preemptive schedule with the minimum makespan. In any feasible schedule every job $j \in \mathcal{J}$ will run on a set $S \subseteq M$ of machines, with $S \neq \emptyset$, and at any point in time every machine must execute only one job.

The general intuition behind the above model is the following. Firstly, every job requires a fixed amount of processing d_j, independent of its machine allocation. In actual systems, this factor can be interpreted as a predetermined set-up time that the job has to undergo before its parallelization. Furthermore, the other factor that influences the execution of a job is its workload, namely the quantity w_j. Inspired by the classical $Q||C_{max}$ problem, we suppose that the amount of processing time resulting from w_j is derived by a division with the total speed utilized. To conclude, one may say that the execution time of a job has two parts. An *identical* part d_j, and a *related* part $\frac{w_j}{\sum_{i \in S} s_i}$.

In order to denote the processing time functions we are also going to use the following notation. For $j \in \mathcal{J}$, $p_j(s) = d_j + \frac{w_j}{s}$, where s is the variable indicating the total speed assigned to j.

Observe also, that this particular setting satisfies the monotonic assumptions mentioned in **Section 2.1**. For one thing, the more speed a job utilizes the less its processing time. Thus, $p_j(s)$ is a decreasing function of s. Furthermore, let us define the work function for a job j as $wrk_j(s) = s \cdot p_j(s) = w_j + d_j \cdot s$. It is then obvious that $wrk_j(s)$ is increasing in s.

Finally, someone may wonder about the necessity of the d_j factor in the definition of the processing time function, claiming that a straight-forward generalization of the $Q||C_{max}$ problem for the malleable case would suggest a function of the following form, $p_j(s) = \frac{w_j}{s}$. The thing is that such a model is actually trivial. We prove this right away.

Theorem 4.1. $\frac{\sum_{j \in \mathcal{J}} w_j}{\sum_{i \in \mathcal{M}} s_i} \leq OPT$

Proof. Suppose $\frac{\sum_{j \in \mathcal{J}} w_j}{\sum_{i \in \mathcal{M}} s_i} > OPT$. Denote by $\Pi_i^\star$ the last point in time where machine i is active in the optimal solution. Because of our hypothesis we have:

$$\forall\, i \in \mathcal{M} :\ \Pi_i^\star \leq OPT < \frac{\sum_{j \in \mathcal{J}} w_j}{\sum_{i \in \mathcal{M}} s_i}$$

The total work performed by machine i is at most $s_i \Pi_i^\star$. Thus, the total work performed by all the machines is:

$$\sum_{k \in \mathcal{M}} s_k \Pi_k^\star < \frac{\sum_{j \in \mathcal{J}} w_j}{\sum_{i \in \mathcal{M}} s_i} \sum_{k \in \mathcal{M}} s_k = \sum_{j \in \mathcal{J}} w_j$$

Therefore, the total work performed by all the machines is strictly less than the total work needed for executing all jobs. Contradiction. □

Based on the above result we present a fairly simple optimal algorithm.

An optimal trivial algorithm

1: Let $L = \{1, \ldots, n\}$ be an arbitrary list of the n jobs.
2: Schedule one job at a time from L using all m processors. Thus, jobs run sequentially.

Theorem 4.2. *The above algorithm is optimal for the problem where $d_j = 0$, $\forall j \in \mathcal{J}$.*

Proof. Because jobs run sequentially the makespan of the schedule is:

$$C = \frac{w_1}{\sum_{i \in M} s_i} + \ldots + \frac{w_n}{\sum_{i \in M} s_i} = \frac{\sum_{j \in J} w_j}{\sum_{i \in M} s_i} \leq OPT$$

where the inequality follows from **Theorem 4.1**. □

4.2 NP-Hardness

Before we continue with our approximation algorithms, we have to be sure that the problem at hand is indeed NP-hard. In this section, we present a fairly simple reduction from the famous **Partition** problem, which is already known to be NP-hard.

An instance of **Partition** consists of a set $A = \{a_1, \ldots, a_m\}$ of m positive integers. We define $S_A = \sum_{i=1}^{m} a_i$, and the question is whether there exist two non-empty subsets $A_1, A_2 \subseteq A$ such that:

- $A_1 \cap A_2 = \emptyset$ and $A_1 \cup A_2 = A$.
- $\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i = \frac{S_A}{2}$

The reduction goes as follows. From a given instance of **Partition** we construct an instance of our scheduling problem. First, for every item $a_i \in A$ we have a machine $i \in \mathcal{M}$ with speed $s_i = a_i$. Regarding the jobs, we are going to have only two of those, which will also be identical. Their common processing time function is defined to be:

$$p(s) = d + d \cdot \frac{S_A}{2s}$$

Let us denote by OPT the optimal solution of the scheduling problem. Observe that in any case $OPT \geq 2d$. If the two jobs are to be sequenced then definitely that holds. If they are going to run in parallel, then they cannot both have speed allocation strictly greater than $\frac{S_A}{2}$, as the total amount of speed is S_A. Therefore, at least one of them will have speed consumption $s' \leq \frac{S_A}{2}$ and so:

$$d + \frac{S_A}{2s'} \geq d + \frac{S_A}{2 \cdot \frac{S_A}{2}} = 2d$$

Finally, observe that if there exists a *YES* answer solution to the **Partition** problem, then by assigning the machines corresponding to subset A_1 to the first job, and the machines corresponding to A_2 to the second job, executing them in parallel, we get a schedule of makespan $2d$. Actually, this is the only situation in which the constructed instance has $OPT = 2d$. Hence, solving optimally the scheduling problem implies correctly deciding about **Partition**.

4.3 The Identical Jobs Case

In this section, we are studying the problem, under the assumption that all jobs in $\mathcal{J}$ are identical. That means that they share a common processing time function $p(s) = d + \frac{w}{s}$. This version of the problem has some practical merit, since many applications generate at each step a set of identical tasks to be computed on a parallel platform. Finally, observe that the NP-hardness reduction presented in the previous section, ensures that even this special case of the problem is NP-hard, since the reduction only utilizes identical jobs.

We continue with the analysis of a constant factor approximation algorithm for the problem. Initially, we partition the machines into two sets. $M_f = \{i \in \mathcal{M} \mid s_i > \frac{w}{d}\}$, the set of "fast" machines and $M_s = \{i \in \mathcal{M} \mid s_i \leq \frac{w}{d}\}$ the set of "slow" machines. The intuition behind this separation is that a job using a "fast" machine has always a processing time of at most $2d$, regardless of all the other processors it utilizes.

Suppose now that n_f jobs use at least one machine of M_f in the optimal solution. We can use exhaustive search (**for** $n_f \leftarrow$ **1 to n**) to determine n_f. The rest of the jobs, $n_s = n - n_f$, use only machines of M_s in the optimal solution. This brute force approach, with time complexity $\mathcal{O}(n)$, guarantees that at some iteration n_f, n_s will be the actual values of the optimal schedule. Given the correct n_f, n_s we proceed with the rest of our scheduling algorithm.

At first we must construct a schedule of n_f jobs on M_f. Suppose that we have the optimal solution of our initial problem, and we isolate the n_f jobs that use at least one

machine of M_f. We then assign each of them to only one machine of M_f, which they already use in the optimal schedule. By their definition this can always be achieved, and their processing time becomes at most $2d$. The last claim holds because in either case $d > \frac{w}{sp}$, where sp is the total speed assigned to each of them. W.l.o.g we may also assume that their processing time becomes exactly $2d$. This newly constructed schedule obviously has makespan $C \leq 2 \cdot OPT$, when restricted only to the n_f jobs. But this schedule, is actually a solution of $P|p = 2d|C_{max}$ with $|P| = |M_f|$, since our last assumption totally ignores the machine speeds.

Because of all the above, solving $P|p = 2d|C_{max}$ can give us an approximation of the schedule of the n_f jobs on M_f. But $P|p = 2d|C_{max}$ can be solved optimally in polynomial time through a greedy algorithm, and by using that we get a schedule with makespan $C^\star \leq C \leq 2 \cdot OPT$. Finally, since $C^\star$ isn't feasible for our problem (actually is a solution of $P|p = 2d|C_{max}$), we simply assign to each of the $|M_f|$ identical machines a unique speed from M_f and the processing time of each job becomes $d + \frac{w}{x} < 2d$ instead of $2d$, because $x > \frac{w}{d}$. So we get a schedule for the n_f jobs on M_f with makespan $C_f \leq C^\star \leq 2 \cdot OPT$.

Now we are concerned with scheduling the n_s jobs, which in the optimal solution use only machines of M_s. To do so, we must first provide a lower bound on their optimal makespan. Let us define $S = \sum_{i \in M_s} s_i$. Also, we denote by $q_j^\star$ the speed allocation of a job in the optimal schedule and by $\mathcal{J}_s$ the set of the n_s jobs under consideration. Each of these jobs occupies $w + d \cdot q_j^\star$ units of area (work) in the optimal solution. Furthermore, since all jobs of $\mathcal{J}_s$ utilize only machines of M_s, the total area provided for them is at most $S \cdot OPT$. Thus:

$$\frac{n_s w + d \sum_{j \in \mathcal{J}_s} q_j^\star}{S} \leq OPT \tag{4.1}$$

Supposing otherwise would mean:

$$n_s w + d \sum_{j \in \mathcal{J}_s} q_j^\star > S \cdot OPT$$

This is clearly a contradiction, since it implies that the total area occupied by the n_s jobs is greater than the total available area for them.

To construct a schedule for the jobs in $\mathcal{J}_s$ we are going to use parametrized analysis, based on an parameter $\alpha \leq 1$. In the end, the resulted approximation ratio will be expressed as a function of α and obviously we are going to minimize this function over all values of α. Furthermore, we distinguish two cases based on the value of S and give distinct ways of scheduling in each situation. The first case consists of a straight-forward algorithm, whilst the second one demands a more thorough and careful approach.

First of all, if $S \leq \frac{w}{\alpha d}$, where $\alpha \leq 1$, we can serialize the n_s jobs by giving all S to each of them. This schedule has a makespan:

$$\begin{aligned} C_{s,1} &= n_s(d + \frac{w}{S}) \\ &\leq n_s(\frac{w}{\alpha S} + \frac{w}{S}) \\ &= \frac{n_s w}{S}(1 + \frac{1}{\alpha}) \\ &\leq (1 + \frac{1}{\alpha})OPT \end{aligned} \tag{4.2}$$

The first inequality results from the assumption about S. For the second inequality observe that $\frac{n_s w}{S}$ is also a lower bound, as indicated by 4.1.

Now, if $S > \frac{w}{\alpha d}$ we are going to construct disjoint groups of machines in M_s, where each of these groups will act as a new "virtual" machine. The jobs will no longer see $\mathcal{M}_s$ as their processing environment, rather they could be only executed on one of these groups, utilizing simultaneously all machines it contains. The intuition behind this approach is to make these groups resemble a "fast" machine, in terms of total speed.

To do so, we are going to use the famous First Fit algorithm for **Bin Packing**, with bins of size $\frac{w}{\alpha d}$. The bins represent the groups, the items placed in the bins are obviously the machines of M_s and the size of each item is the speed of the corresponding machine. However, unlike the **Bin Packing** problem we are not interested in the final number of used bins. In our case, we take advantage of another special property of First Fit, namely that given the correct conditions, it always fills at least half of all the bins, except perhaps the last one. These conditions are satisfied in our case, since $\forall i \in M_s : s_i \leq \frac{w}{d}$, $\alpha \leq 1$ and $S > \frac{w}{\alpha d}$. $S > \frac{w}{\alpha d}$ implies that there is enough speed to fill at least two bins and $\forall i \in M_s : s_i \leq \frac{w}{d}$, $\alpha \leq 1$ ensures that all item sizes are less than the size of the bin.

Construct-Groups Algorithm

1: Consider the machines of M_s in any arbitrary order.
2: Construct the first group and place in it the first machine regarding that order.
3: Place the next machine in the first group that can fit without violating the total size $\frac{w}{\alpha d}$. If it cannot fit anywhere construct a new group and place it there.
4: Continue with the above until you are out of machines.

It is pretty obvious that at the end of this procedure every group is at least half full. If this is not the case we could have merged the contents of two groups. However, this may not hold for the last group. To sum up, this algorithm will build h groups with the following properties:

- $s_{g,1} \leq s_{g,2} \leq \ldots \leq s_{g,h}$
- $\forall i \in [1, h] : s_{g,i} \leq \frac{w}{\alpha d}$

- $\forall i \in [2,h] : s_{g,i} \geq \frac{w}{2\alpha d}$. Perhaps, the smallest group will not be at least half full. But because $S > \frac{w}{\alpha d}$, the algorithm cannot construct only one group which will also be not half full.

After the the above procedure, if $s_{g,1} < \frac{w}{2\alpha d}$ we redistribute (in any possible way) its speed to the other groups, even if we violate the $\frac{w}{\alpha d}$ size. Finally, we obtain a machine grouping with the following properties:

$$\begin{aligned} & s'_{g,1} \leq s'_{g,2} \leq \ldots \leq s'_{g,h'} \\ & \forall i \in [1,h'] : \frac{w}{2\alpha d} \leq s'_{g,i} \leq \frac{3w}{2\alpha d} \end{aligned} \tag{4.3}$$

The second property results, since:

$$\forall i \in [2,h] : s_{g,i} + s_{g,1} \leq \frac{w}{\alpha d} + \frac{w}{2\alpha d} = \frac{3w}{2\alpha d}$$

Now that we have the machine groups, we use the following simple scheduling rule. Every group gets $\left\lceil \frac{n_s \cdot s'_{g,i}}{S} \right\rceil$ jobs. This is an attempt to uniformly distribute work, depending on the group speed. The first thing that needs to be done is to prove that the aforementioned rule guarantees that all jobs will be scheduled. This is fairly simple.

$$\sum_{i=1}^{h'} \left\lceil \frac{n_s \cdot s'_{g,i}}{S} \right\rceil \geq \sum_{i=1}^{h'} \frac{n_s \cdot s'_{g,i}}{S} = \frac{n_s}{S} \sum_{i=1}^{h'} s'_{g,i} = \frac{n_s}{S} S = n_s$$

We begin using the above rule, scheduling the groups in an increasing order of speed. At some point, when scheduling for group i, we may not have enough jobs to fulfil the $\left\lceil \frac{n_s \cdot s'_{g,i}}{S} \right\rceil$ requirement. This actually does not pose any problem, due to the fact the total machine load, and thus the makespan will not be affected. And that leads us to the next step of our analysis, namely computing the resulted makespan of this rule. The load of each group, l_i, will be the following:

$$\begin{aligned} l_i &= \left\lceil \frac{n_s \cdot s'_{g,i}}{S} \right\rceil (d + \frac{w}{s'_{g,i}}) \\ &\leq (\frac{n_s \cdot s'_{g,i}}{S} + 1)(d + \frac{w}{s'_{g,i}}) \\ &= \frac{n_s}{S}(w + s'_{g,i} d) + (d + \frac{w}{s'_{g,i}}) \\ &\leq \frac{n_s}{S}(w + \frac{3w}{2\alpha}) + (d + \frac{w}{s'_{g,i}}) \\ &= \frac{n_s w}{S}(1 + \frac{3}{2\alpha}) + (d + \frac{w}{s'_{g,i}}) \end{aligned} \tag{4.4}$$

The last inequality follows directly from 4.3.

Let us denote by $C_{s,2}$ the makespan of such a schedule. Obviously,

$$C_{s,2} = \max_{1 \le i \le h'} \{l_i\} \le \frac{n_s w}{S}(1 + \frac{3}{2\alpha}) + (d + \frac{w}{s'_{g,i}})$$

, for some group i. Hence, to conclude our proof we need to analyse quantity 4.4, in terms of relating it to OPT. To do so, we proceed with a case analysis on the optimal solution, which will result in different lower bounds. Furthermore, observe that the two cases mentioned below are mutually exclusive.

- **Case A)** $\forall j \in \mathcal{J}_s : q_j^\star > \frac{w}{2\alpha d}$. In this case we have the following:

$$\frac{n_s w + d n_s \frac{w}{2\alpha d}}{S} \le \frac{n_s w + d \sum_{j \in \mathcal{J}_s} q_j^\star}{S} \implies \frac{n_s w}{S}(1 + \frac{1}{2\alpha}) \le OPT \tag{4.5}$$

  The implication follows from 4.1.

- **Case B)** $\exists j \in \mathcal{J}_s : q_j^\star \le \frac{w}{2\alpha d}$. Because $s'_{g,i} \ge \frac{w}{2\alpha d}$ for all groups, we conclude that:

$$d + \frac{w}{s'_{g,i}} \le d + \frac{w}{q_j^\star} \le OPT \tag{4.6}$$

Suppose that **Case A** holds. Then:

$$\begin{aligned} C_{s,2} &\le \frac{n_s w}{S}(1 + \frac{3}{2\alpha}) + (d + \frac{w}{s'_{g,i}}) \\ &\le \frac{1 + \frac{3}{2\alpha}}{1 + \frac{1}{2\alpha}} OPT + (1 + 2\alpha)d \\ &\le \Big(\frac{1 + \frac{3}{2\alpha}}{1 + \frac{1}{2\alpha}} + (1 + 2\alpha)\Big) OPT \end{aligned}$$

The second inequality results from 4.3 and from comparing $\frac{n_s w}{S}(1 + \frac{3}{2\alpha})$ with the lower bound of 4.5. The third inequality is straight-forward since obviously $d \le OPT$. Finally, we say that this case results in an approximation ratio $\rho_A(\alpha) = \frac{1 + \frac{3}{2\alpha}}{1 + \frac{1}{2\alpha}} + (1 + 2\alpha)$.

Now suppose that **Case B** holds. Then:

$$\begin{aligned} C_{s,2} &\le \frac{n_s w}{S}(1 + \frac{3}{2\alpha}) + (d + \frac{w}{s'_{g,i}}) \\ &\le (1 + \frac{3}{2\alpha}) OPT + OPT \end{aligned}$$

The second inequality results by utilizing 4.1 and 4.6. So, we say that this case gives an approximation ratio $\rho_B(\alpha) = 2 + \frac{3}{2\alpha}$.

The total approximation ratio, when scheduling with $S > \frac{w}{\alpha d}$, is $\rho(\alpha) = max\{\rho_A(\alpha), \rho_B(\alpha)\}$. In order to minimize $\rho(\alpha)$ we need to find an α_m such that $\rho_A(\alpha_m) = \rho_B(\alpha_m)$. By solving the latter equation we get $\alpha_m \approx 0.67965204$. The approximation ratio at that

value is $\rho_s = 4.20701$. Thus, we constructed a schedule for the n_s jobs with makespan $C_s \leq 4.20701 \cdot OPT$. Observe also that ρ_B always dominates $(1 + \frac{1}{\alpha})$ from $C_{s,1}$ and so we don't need to take it into account at all.

Finally, because $M_f \cap M_s = \emptyset$ the two schedules we have constructed can run in parallel. So, $C = max\{C_s, C_f\} \leq 4.20701 \cdot OPT$ is the overall schedule makespan. We also provide a total synopsis of the algorithm.

Identical-Jobs Algorithm

1: **for** $n_f = 1$ to n **do**
2: $n_s \leftarrow n - n_f$
3: Schedule n_f jobs on M_f using the greedy algorithm for $P|p = 2d|C_{max}$.
4: **if** $S \leq \frac{w}{\alpha_m d}$ **then**
5: Schedule all n_s jobs sequentially giving all of S to each of them.
6: **else**
7: Use the **Construct-Groups** algorithm.
8: If necessary redistribute the speed of the slowest group.
9: In each group i with speed $s'_{g,i}$ schedule $\left\lceil \frac{n_s \cdot s'_{g,i}}{S} \right\rceil$ jobs.
10: Return the schedule with the minimum makespan.

Theorem 4.3. *The above algorithm is a* 4.20701*-approximation for our problem.*

Theorem 4.4. *The running time of the above algorithm is* $\mathcal{O}(n \cdot m^2)$

Proof. Steps 5 and 8 can actually be implemented in constant time. Step 3 takes time $\mathcal{O}(m)$. Moreover, the algorithm of step 7, which is actually **First Fit**, has running time $\mathcal{O}(m^2)$, while step 9 can be completed in time $\mathcal{O}(m)$. Combined with the initial **for** iteration we get the final result. □

4.4 Jobs With Different Workloads

In this section, we tackle another special version of the general model. More precisely, we suppose that again all jobs of $\mathcal{J}$ share a common delay d, however each of them has a workload, w_j, of its own. The problem at hand is clearly a general case of the identical jobs problem, and hence it remains NP-hard. In order to solve this problem, we try to exploit the results of the previous section. To do so, we need a rather restricting assumption, namely:

$$\forall i \in \mathcal{M} : \ \forall j \in \mathcal{J} : \ s_i \leq \frac{w_j}{d} \tag{4.7}$$

What the above hypothesis actually claims, is that all machines are "slow", as defined earlier, for every job.

To begin with, we need again a lower bound on the optimal makespan. Following the reasoning behind acquiring 4.1, we can prove that:

$$\frac{\sum_{j\in\mathcal{J}} w_j}{S} \leq OPT \tag{4.8}$$

where $S = \sum_{i\in\mathcal{M}} s_i$.

Initially, we get rid of the heaviest jobs, that is the ones with sufficiently big w_j. So, we schedule all jobs with $S \leq \frac{w_j}{d}$ sequentially, giving each of them all machines of $\mathcal{M}$. The resulted schedule has makespan:

$$\begin{aligned} C_1 &= \sum_{\{j\in\mathcal{J} \mid S\leq\frac{w_j}{d}\}} (d + \frac{w_j}{S}) \\ &\leq \sum_{\{j\in\mathcal{J} \mid S\leq\frac{w_j}{d}\}} 2\frac{w_j}{S} \\ &\leq 2\cdot OPT \end{aligned} \tag{4.9}$$

The first inequality follows from the definition of these jobs, whilst the second from 4.8.

Subsequently, we need to deal with the rest of the jobs, $\mathcal{J}_s = \{j \in \mathcal{J} \mid S > \frac{w_j}{d}\}$. In order to schedule them, we will take advantage of the identical jobs algorithm. To do so, we must partition $\mathcal{J}_s$ into classes of jobs that have similar workloads. The algorithm for achieving this follows.

Create-Classes Algorithm

1: Arrange the jobs of $\mathcal{J}_s$ in order of increasing w_j.
2: Take as pivot the first job in the list, with workload w_1. Place in the first class, $\mathcal{J}_1$, all jobs $j \in \mathcal{J}$ with $\frac{w_1}{2} \leq w_j \leq w_1$. Remove $\mathcal{J}_1$ from the list.
3: Continue the above procedure by taking as pivot the first job in the remaining list.
4: In the end, we have l classes, $\mathcal{J}_1$, ..., $\mathcal{J}_c$, ..., $\mathcal{J}_l$, with their pivoting elements satisfying $w_1 \geq w_2 \geq \ldots \geq w_c \geq \ldots \geq w_l$.

Observe, that for the first class $\mathcal{J}_1$, that is the one with the heaviest pivoting element, we can use the **Construct-Groups** algorithm based on w_1, since $S > \frac{w_1}{d}$ and $\forall i \in \mathcal{M}: \; s_i \leq \frac{w_1}{d}$. The parameter α of the previous section is now set to be 1. Therefore, we create h_1 machine groups with total speed each:

$$\frac{w_1}{2d} \leq s^1_{g,i} \leq \frac{3w_1}{2d} \tag{4.10}$$

Consider now an arbitrary group from the ones that are built specifically for $\mathcal{J}_1$. The machines inside it can be also used in order to construct groups for the second class $\mathcal{J}_2$. From the way we partition the jobs into classes we know that the relation between the pivoting elements of the first and the second class is $w_2 < \frac{w_1}{2}$. Thus, for the group of

$\mathcal{J}_1$ considered we have:

$$s^1_{g,i} \geq \frac{w_1}{2d} > \frac{w_2}{d}$$

The first inequality follows from 4.10, whilst the second is a result of the relation between w_1 and w_2. Furthermore, because the machines of that group are "slow" for w_2 as well 4.7, we can indeed build the groups for $\mathcal{J}_2$ inside the initial group for $\mathcal{J}_1$. The above procedure can be extended in every pair of consecutive classes $\mathcal{J}_c$, $\mathcal{J}_{c+1}$. To sum up, we have achieved the following:

- For the class $\mathcal{J}_1$, with pivoting element w_1 we have constructed groups of speed $\frac{w_1}{2d} \leq s^1_{g,i} \leq \frac{3w_1}{2d}$.
- Every group of the class $\mathcal{J}_c$, can be further partitioned into groups for the class $\mathcal{J}_{c+1}$, each with speed $\frac{w_{c+1}}{2d} \leq s^{c+1}_{g,i} \leq \frac{3w_{c+1}}{2d}$, where w_{c+1} the pivoting element of $\mathcal{J}_{c+1}$.

A visual representation of this structure follows.

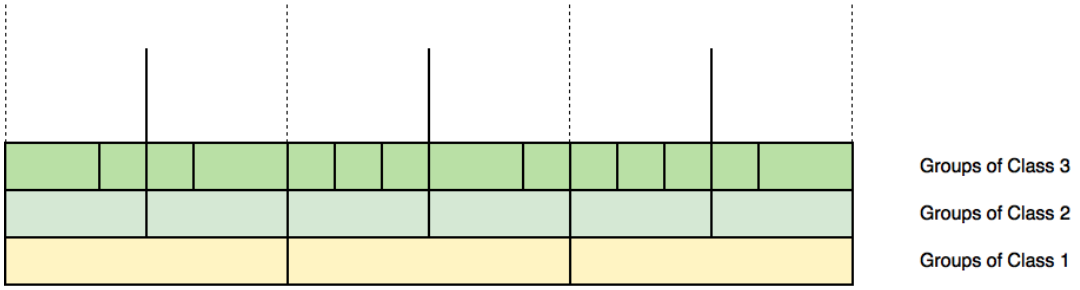


FIGURE 4.1: The grouping structure.

The only thing left to do now, is to schedule the jobs of $\mathcal{J}_s$ using the above presented structure. The algorithm to do this is fairly simple and is based on a classical list scheduling argument.

Schedule Unequal Workloads Algorithm

1: Consider the job classes in decreasing order of their pivoting element, $\mathcal{J}_1, \ldots, \mathcal{J}_l$.
2: Start with the jobs of $\mathcal{J}_1$, scheduling them on the groups corresponding to that class. Consider on job at a time, and place it in the group such that the resulting total makespan is as small as possible.
3: Continue with this procedure until you finish with all classes.

Notice that due to the specific structure presented in **Figure 4.1**, there is no idle time before the job finishing last starts processing. This is a result of the greedy scheduling rule and the fact that the groups of class $c+1$ are formed within the groups of class c. A visual example can clarify this property.

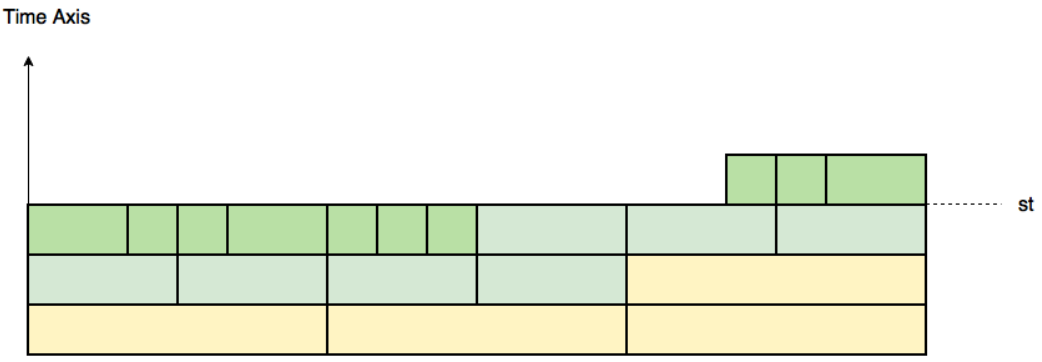


FIGURE 4.2: A schedule for unequal workload jobs.

Let us denote by st the starting time of the job finishing last, by sp_j the speed allocated to job j in the schedule and by $w_c(j)$ the pivoting element of the class containing j. Then, because there is no idle time before st we have:

$$\begin{aligned} st &\leq \frac{\sum_{j\in\mathcal{J}_s}(w_j + d\cdot sp_j)}{S} \\ &\leq \frac{\sum_{j\in\mathcal{J}_s}(w_j + d\cdot \frac{3w_c(j)}{2d})}{S} \\ &\leq \frac{\sum_{j\in\mathcal{J}_s} 4w_j}{S} \\ &\leq 4\cdot OPT \end{aligned}$$

The second inequality follows, since each job is assigned to a group of its class. The third results from the relation between w_c and w_{c+1}.

The final makespan of such a schedule is obviously st plus the processing time of the last job, which is bounded by $3\cdot OPT$. Hence, $C_2 \leq 7\cdot OPT$. The total schedule is obtained via sequencing the large jobs and the jobs of $\mathcal{J}_s$. Thus, $C = C_1 + C_2 \leq 9\cdot OPT$.

4.5 Future Work

Regarding the new model we introduce in this thesis, there are many intriguing future research directions. First of all, one may try to improve the approximation factor of the identical jobs case. Since we do not know if this particular problem is NP-hard in the strong sense, there may even be a $FPTAS$ for it. Furthermore, the hypothesis about the machine speeds 4.7, for the version with the unequal workloads should be removed, in order to provide a general solution to that problem. Moreover, it would be interesting to study the model even when the job delays are not identical, namely assuming processing time functions of the form $p_j(s) = d_j + \frac{w_j}{s}$. Finally, the most interesting direction to follow, is studying malleable job scheduling with general processing time functions, without assuming any particular structure for them.

Bibliography

[1] Jakob Gonczarowski and Manfred K. Warmuth. Applications of scheduling theory to formal language theory. *Theoretical Computer Science*, 37:217 – 243, 1985. ISSN 0304-3975. doi: http://dx.doi.org/10.1016/0304-3975(85)90092-1. URL `http://www.sciencedirect.com/science/article/pii/0304397585900921`.

[2] open-mpi.org. `https://www.open-mpi.org/`. Accessed: 2016-05-26.

[3] openmp.org. `http://openmp.org/wp/`. Accessed: 2016-05-26.

[4] cilkplus.org. `https://www.cilkplus.org/`. Accessed: 2016-05-26.

[5] Joseph Leung, Laurie Kelly, and James H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004. ISBN 1584883979.

[6] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. In *Discrete Optimization II Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*, volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979. URL `http://www.sciencedirect.com/science/article/pii/S016750600870356X`.

[7] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, April 1974. ISSN 0004-5411. doi: 10.1145/321812.321815. URL `http://doi.acm.org/10.1145/321812.321815`.

[8] Klaus Jansen and Lorant Porkolab. Linear-time approximation schemes for scheduling malleable parallel tasks. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 490–498, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics. ISBN 0-89871-434-6. URL `http://dl.acm.org/citation.cfm?id=314500.314870`.

[9] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM J. Discret. Math.*, 2(4):473–487, November 1989. ISSN 0895-4801. doi: 10.1137/0402042. URL `http://dx.doi.org/10.1137/0402042`.

[10] John Turek, Joel L. Wolf, and Philip S. Yu. Approximate algorithms scheduling parallelizable tasks. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 323–332, New York, NY, USA, 1992. ACM. ISBN 0-89791-483-X. doi: 10.1145/140901.141909. URL `http://doi.acm.org/10.1145/140901.141909`.

[11] Gregory Mounie, Christophe Rapine, and Denis Trystram. A 3/2-approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM Journal on Computing*, 37(2):401–412, 2007. doi: 10.1137/S0097539701385995. URL `http://dx.doi.org/10.1137/S0097539701385995`.

[12] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, January 1987. ISSN 0004-5411. doi: 10.1145/7531.7535. URL `http://doi.acm.org/10.1145/7531.7535`.

[13] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990. ISBN 0-471-92420-2.

[14] T. Decker. *Ein universelles Lastverteilungssystem und seine Anwendung bei der Isolierung reeller Nullstellen*. PhD thesis, 2000.

[15] T. Decker, T. Lücking, and B. Monien. A 5/4-approximation algorithm for scheduling identical malleable tasks. *Theor. Comput. Sci.*, 361(2):226–240, September 2006. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.05.012. URL `http://dx.doi.org/10.1016/j.tcs.2006.05.012`.

[16] Jason Glasgow and Hadas Shachnai. Minimizing the flow time for parallelizable task systems. Technical report, 1993.

[17] John Turek, Walter Ludwig, Joel L. Wolf, Lisa Fleischer, Prasoon Tiwari, Jason Glasgow, Uwe Schwiegelshohn, and Philip S. Yu. Scheduling parallelizable tasks to minimize average response time. In *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '94, pages 200–209, New York, NY, USA, 1994. ACM. ISBN 0-89791-671-9. doi: 10.1145/181014.181331. URL `http://doi.acm.org/10.1145/181014.181331`.

[18] John Turek, Uwe Schwiegelshohn, Joel L. Wolf, and Philip S. Yu. Scheduling parallel tasks to minimize average response time. In *Proceedings of the Fifth*

Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '94, pages 112–121, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics. ISBN 0-89871-329-3. URL http://dl.acm.org/citation.cfm?id=314464.314485.

[19] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, 1975. doi: 10.1137/0204015. URL http://dx.doi.org/10.1137/0204015.

[20] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955. ISSN 1931-9193. doi: 10.1002/nav.3800020109. URL http://dx.doi.org/10.1002/nav.3800020109.

[21] Alexander Grigoriev, Maxim Sviridenko, and Marc Uetz. *Unrelated Parallel Machine Scheduling with Resource Dependent Processing Times*, pages 182–195. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32102-6. doi: 10.1007/11496915_14. URL http://dx.doi.org/10.1007/11496915_14.

[22] Alexander Grigoriev, Maxim Sviridenko, and Marc Uetz. *LP Rounding and an Almost Harmonic Algorithm for Scheduling with Resource Dependent Processing Times*, pages 140–151. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-38045-0. doi: 10.1007/11830924_15. URL http://dx.doi.org/10.1007/11830924_15.

[23] Alexander Grigoriev, Maxim Sviridenko, and Marc Uetz. Machine scheduling with resource dependent processing times. *Mathematical Programming*, 110(1):209–228, 2007. ISSN 1436-4646. doi: 10.1007/s10107-006-0059-3. URL http://dx.doi.org/10.1007/s10107-006-0059-3.

[24] David B. Shmoys and Éva Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1):461–474, 1993. doi: 10.1007/BF01585178. URL http://dx.doi.org/10.1007/BF01585178.

[25] V. S. Anil Kumar and M. V. Marathe. Approximation algorithms for scheduling on multiple machines. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 254–263, Oct 2005. doi: 10.1109/SFCS.2005.21.

[26] Leslie A. Hall, Andreas S. Schulz, David B. Shmoys, and Joel Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Math. Oper. Res.*, 22(3):513–544, August 1997. ISSN 0364-765X. doi: 10.1287/moor.22.3.513. URL http://dx.doi.org/10.1287/moor.22.3.513.

[27] Jyh-Han Lin and Jeffrey Scott Vitter. e-approximations with minimum packing constraint violation (extended abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 771–782, New York, NY, USA, 1992. ACM. ISBN 0-89791-511-9. doi: 10.1145/129712.129787. URL `http://doi.acm.org/10.1145/129712.129787`.

[28] Uwe Schwiegelshohn, Walter Ludwig, Joel L. Wolf, John Turek, and Philip S. Yu. Smart smart bounds for weighted response time scheduling. *SIAM J. Comput.*, 28 (1):237–253, February 1999. ISSN 0097-5397. doi: 10.1137/S0097539795286831. URL `http://dx.doi.org/10.1137/S0097539795286831`.

[29] O. Beaumont, N. Bonichon, L. Eyraud-Dubois, and L. Marchal. Minimizing weighted mean completion time for malleable tasks scheduling. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 273–284, May 2012. doi: 10.1109/IPDPS.2012.34.

[30] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[31] Mihir Bellare and Madhu Sudan. Improved non-approximability results. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94, pages 184–193, New York, NY, USA, 1994. ACM. ISBN 0-89791-663-8. doi: 10.1145/195058.195129. URL `http://doi.acm.org/10.1145/195058.195129`.

[32] Oh-Heum Kwon and Kyung-Yong Chwa. Approximation algorithms for general parallel task scheduling. *Inf. Process. Lett.*, 81(3):143–150, February 2002. ISSN 0020-0190. doi: 10.1016/S0020-0190(01)00210-1. URL `http://dx.doi.org/10.1016/S0020-0190(01)00210-1`.

[33] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *J. ACM*, 45(3): 501–555, May 1998. ISSN 0004-5411. doi: 10.1145/278298.278306. URL `http://doi.acm.org/10.1145/278298.278306`.

[34] Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, September 1994. ISSN 0004-5411. doi: 10.1145/185675.306789. URL `http://doi.acm.org/10.1145/185675.306789`.

[35] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2011. ISBN 0521195276, 9780521195270.

[36] Jianer Chen and Chung-Yee Lee. General multiprocessor task scheduling. *Naval Research Logistics (NRL)*, 46(1):57–74, 1999. ISSN 1520-6750. doi: 10.1002/(SICI)1520-6750(199902)46:1⟨57::AID-NAV4⟩3.0.CO;2-H. URL http://dx.doi.org/10.1002/(SICI)1520-6750(199902)46:1<57::AID-NAV4>3.0.CO;2-H.

[37] J.A. Hoogeveen, S.L. van de Velde, and B. Veltman. Complexity of scheduling multiprocessor tasks with prespecified processor allocations. *Discrete Applied Mathematics*, 55(3):259 – 272, 1994. ISSN 0166-218X. doi: http://dx.doi.org/10.1016/0166-218X(94)90012-4. URL http://www.sciencedirect.com/science/article/pii/0166218X94900124.

[38] A. Miranda-Garcia. *Approximation Algorithms for Multiprocessor Task Scheduling*. Texas A & M University, 1998. URL https://books.google.gr/books?id=T--aNwAACAAJ.

[39] Michel X. Goemans. An approximation algorithm for scheduling on three dedicated machines. *Discrete Appl. Math.*, 61(1):49–59, July 1995. ISSN 0166-218X. doi: 10.1016/0166-218X(94)00160-F. URL http://dx.doi.org/10.1016/0166-218X(94)00160-F.

[40] Amoura, Bampis, Kenyon, and Manoussakis. Scheduling independent multiprocessor tasks. *Algorithmica*, 32(2):247–261, 2002. doi: 10.1007/s00453-001-0076-9. URL http://dx.doi.org/10.1007/s00453-001-0076-9.

[41] Jianer Chen and Antonio Miranda. A polynomial time approximation scheme for general multiprocessor job scheduling. *SIAM Journal on Computing*, 31(1):1–17, 2001. doi: 10.1137/S0097539798348110. URL http://dx.doi.org/10.1137/S0097539798348110.

[42] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1994. ISBN 0201558025.