APPROXIMATION ALGORITHMS FOR SCHEDULING PROBLEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Chandra Chekuri
August 1998

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Rajeev Motwani
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Serge Plotkin

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Clifford Stein

Approved for the University Committee on Graduate Studies:

Abstract

This thesis describes efficient approximation algorithms for some NP-Hard deterministic machine scheduling and related problems. An approximation algorithm for an NP-Hard optimization problem is a polynomial time algorithm which, given any instance of the problem, returns a solution whose value is within some guaranteed multiplicative factor α of the optimal solution value for that instance. The quantity α is called the approximation ratio of the algorithm. A typical problem in machine scheduling consists of a set of jobs that are to be executed in either preemptive or non-preemptive fashion on a set of available machines subject to a variety of constraints. Two common objectives are minimizing makespan (the time to complete all jobs) and minimizing average completion time. Constraints that we study include precedence constraints between jobs and release dates on jobs. Brief descriptions of the problems we study and highlights of our results follow.

We study single machine and parallel machine scheduling problems with the objective of minimizing average completion time and its weighted generalization. We introduce new techniques that either improve earlier results and/or result in simple and efficient algorithms. For the single machine problem with jobs having release dates only we obtain an $\frac{e}{e-1} \simeq 1.58$ approximation. For the parallel machine case we obtain a 2.85 approximation. We then focus on the case when jobs have precedence constraints. For the single machine problem we obtain an 2-approximation algorithm that in contrast to earlier algorithms does not rely on linear programming relaxations. We also give a general algorithm that converts an α-approximate single machine schedule into a $(2\alpha+2)$-approximate parallel machine schedule. The conversion algorithm is simple and yields efficient and combinatorial constant factor algorithms for several variants.

We then consider the problem of minimizing makespan on machines with different speeds when jobs have precedence constraints. We obtain an $O(\log m)$ approximation (m is the number of machines) in $O(n^3)$ time. Our approximation ratio matches the best known ratio

up to constant factors. However, our algorithm is efficient and easy to implement and is based on a natural heuristic.

We introduce a new class of scheduling problems that arise from query optimization in parallel databases. The novel aspect is in modeling communication costs between operators in a task system that represents a query execution plan. We address one of the problems that we introduce, namely, the pipelined operator tree problem. An instance of the problem consists of a tree with node weights that represent processing times of the associated operators, and edge weights that represent communication costs. Scheduling two nodes connected by an edge on different processors adds communication cost equal to the edge weight to both the nodes. The objective is to schedule the nodes on parallel processors to minimize makespan. This is a generalization of the well known multi-processor scheduling problem. We obtain a polynomial time approximation scheme for this problem. We also obtain efficient $O(n \log n)$ time algorithms that have ratios of 3.56 and 2.58 respectively.

Finally we study multi-dimensional generalizations of three well known problems in combinatorial optimization: multi-processor scheduling, bin packing, and knapsack. We study versions of these problems when items are multi-dimensional vectors instead of real numbers. We obtain several approximability and inapproximability results. For the vector scheduling problem we obtain a PTAS when d, the dimension of the vectors, is fixed. For d arbitrary we obtain a $O(\min\{\log dm, \log^2 d\})$ approximation and also show that no constant factor approximation is possible unless NP = ZPP. For the vector bin packing problem we obtain a $(1 + \epsilon d + O(\ln(1/\epsilon)))$ approximation for all d and show APX-Hardness even for $d = 2$. The vector version of knapsack captures a broad class of hard integer programming problems calling packing integer programs. The only general technique known to solve these problems is randomized rounding. We show that results obtained by randomized rounding are the best possible for a large class of these problems.

Practical applications motivating some of our problems include instruction scheduling in compilers and query optimization in parallel databases.

Acknowledgements

This thesis and the growth in my knowledge and maturity over the last few years owe a great deal to many teachers, colleagues, and friends. First among them is my adviser Rajeev Motwani. He accepted me as his student at a time when I was unsure of my interests and capabilities, and gave me direction by suggesting concrete interesting problems. The many hours he spent with me in my first few years here, when I needed his time most, have contributed much to my scholarship. I express my sincere thanks to him for his support, guidance, and wisdom. His persistence in tackling problems, confidence, and great teaching will always be an inspiration.

Cliff Stein's sabbatical at Stanford has been instrumental in shaping this thesis. Collaboration with him on parts of Chapter 2 gave me impetus to finish my thesis on scheduling. His calm and diligent approach to research and its responsibilities are worth emulating. I thank him for carefully reading my thesis and for coming to my defense all the way from Dartmouth. He has been like a second adviser to me and I am grateful for his kindness.

Serge Plotkin's classes have taught me many new ideas and approaches for which I am most thankful. In several ways he has made the the theory community at Stanford a nicer place. Thanks to him also for being on my reading committee.

My co-authors, colleagues, and many others from the community have given me their valuable time and insights, and I am grateful to all of them. In addition to Rajeev and Cliff, I would especially like to thank Sanjeev Khanna and Moses Charikar for their influence on my work. Sanjeev has been a mentor to me since my first year. His encouragement and constant advice about working on hard and meaningful problems have guided me through out. I am extremely happy to have a chapter in this thesis that is joint work with him. I could always count on sound and intelligent advice from Moses, and his clear thinking helped untangle many of my ideas. Though no papers written jointly with him (and others) are part of this thesis, they have had a substantial influence in opening new areas of research

to me.

I had five wonderful years at Stanford during which time I have not only matured academically but also as an individual. I have been lucky to have many beautiful and cherished friendships, and it is with more than a tinge of sadness that I leave this place. I would like to thank Harish in particular for adding a great deal to my life. He has lifted my spirits many a time and memories of our trip to South America will always be with me. Julien and Sanjeev with their worldly advice have been more than generous friends. The long hours spent in my office would not have been possible but for the company of wonderful students around me. I will miss the many enjoyable hours spent talking about sundry topics to Ashish, Sudipto, Piotr, Shiva, and Suresh, Craig's sense of humour, Donald's generosity, Jeffrey's personality, lunch with Scott, coffee with Julien, Harish's cookies, tennis with Ashish, Shiva, and Suresh (all credit for my improved game goes to them), theory lunch, and many such seemingly small but important things that will be remembered for a long time. My sincere thanks also to Eric, Luca, Michael, Nikolai, Oliver, Ram, and Tomas for their company. Many others are not mentioned here, but know my affection for them. Stanford University and the Computer Science Department have provided an incredible academic and cultural environment. I am thankful to both institutions and not least the fabulous weather of the bay area.

Finally, I thank my parents and my brother for their unconditional love and support through ups and downs. Their faith and confidence in my abilities, though sometimes irrational, have kept me going. I dedicate this thesis to my brother for the many happy times we shared.

Chandra Chekuri
Stanford, California
August 1998

I gratefully acknowledge funding support from a Stanford University School of Engineering Fellowship, an IBM Cooperative Fellowship, and from NSF under the Award CCR-9357849 to Rajeev Motwani with matching support from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

Contents

Chapter 1

Introduction

Sequencing and scheduling problems are motivated by allocation of limited resources over time. The goal is to find an *optimal* allocation where optimality is defined by some problem dependent objective. Given the broad definition above it is no surprise that scheduling is a vast sub-area of optimization. The work in this thesis falls under the category of *deterministic machine scheduling*. Resources are modeled by *machines* and activities are modeled by *jobs* that can be executed by the machines. Deterministic refers to the scenario in which all parameters of the problem instance are known in advance. In contrast, stochastic scheduling is concerned with the case when some or all of the input data is assumed to be from a known probability distribution (see [88] for details). Another approach to model the uncertainty in the input data is by considering various adversarial online scenarios [8]. Applications dictate the most accurate set of assumptions for the problem at hand. Problems in deterministic machine scheduling are combinatorial optimization problems. Consequently, techniques from the rich field of combinatorial optimization find motivation in, and are extensively applied to scheduling problems.

Most of the early work on scheduling, starting from early 1950's, was motivated by production planning and manufacturing, and was primarily done in the operations research and management science community. The advent of computers and their widespread use had a considerable impact both on scheduling problems and solution strategies. Computers and related devices are, in a very concrete sense, resources that need to be allocated to various processes. Many problems in scheduling found new applications in computer science. In addition, a number of new problems and variations have been motivated by application areas in computer science such as parallel computing, databases, compilers, and time sharing. The advent of computers also initiated the formal study of efficiency of computation that led to the notion of NP-Completeness. Karp's seminal work [68] established the pervasive nature of NP-Completeness by showing that decision versions of several naturally occurring problems in combinatorial optimization are NP-Complete, and thus are unlikely to have *efficient* (polynomial time) exact algorithms. Following Karp's work, many problems, including scheduling problems, were shown to be NP-Complete. It is widely believed that P (the set of languages that can be recognized by Turing machines in deterministic polynomial time) is a proper subset of NP (the set of languages that can be recognized by Turing machines in non-deterministic polynomial time). Proving that P $\neq$ NP is the most outstanding problem in theoretical computer science today.

The practical importance of NP-Hard optimization problems necessitates tractable relaxations. By tractable we mean efficient solvability, and polynomial time is a robust theoretical notion of efficiency. A very fruitful approach has been to relax the notion of optimality and settle for *near-optimal* solutions. A near-optimal solution is one whose objective function value is within some small *multiplicative*[1] factor of the optimal value. Approximation algorithms are heuristics that provide *provably good* guarantees on the quality of the solutions they return. This approach was pioneered by the influential paper of Johnson [63] in which he showed the existence of good approximation algorithms for several NP-Hard optimization problems. He also remarked that the optimization problems that are all indistinguishable in the theory of NP-Completeness behave very differently when it comes to approximability. Remarkable work in the last couple of decades in both the design of approximation algorithms and proving inapproximability results has validated Johnson's remarks. The book on approximation algorithms edited by Hochbaum [54] gives a good glimpse of the current knowledge on the subject. The methodology of evaluating algorithms by the quality of their solutions is useful in comparing commonly used heuristics, and often the analysis suggests new and improved heuristics.

Approximation algorithms for several problems in scheduling have been developed in the last three decades. In fact it is widely believed that the first NP optimization problem for which an approximation algorithm was formally designed and analyzed is multiprocessor scheduling, by Graham in 1966 [42]. This precedes the development of the theory of NP-Completeness. Despite the progress made on many problems, the approximability of several

[1] It is possible to define other notions of near-optimality such as additive error approximations. However the multiplicative notion is the most robust. See [81] for examples and discussion.

fundamental problems is still open (see the many surveys available [73, 46, 65]).

In this thesis we develop approximation algorithms for some NP-Hard deterministic scheduling problems. The problems we consider are primarily motivated by applications in computer science such as parallel query optimization, code scheduling in compilers, and task scheduling. Some of the problems we consider are particular to the application at hand and thus are "new" problems or generalizations of existing problems. Most of the focus in designing approximation algorithms for optimization problems is in improving the approximation guarantee, and less so on obtaining the best running time. Many algorithms are based on rounding linear programming based relaxations. Partly motivated by the practical applications that gave rise to some of the problems considered here, we focus on obtaining efficient and "combinatorial" algorithms(we use the word combinatorial in an admittedly imprecise manner here). Combinatorial algorithms have two useful properties. They are typically simpler and efficient to implement. Further they often provide structural insights that can be exploited for special cases, as we shall see in Chapters 2 and 3.

In the next section we review relevant background on scheduling theory and approximation theory, and Section 1.2 outlines the contributions of this thesis.

1.1 Background and Notation

1.1.1 Scheduling Theory

Scheduling theory encompasses a large and diverse set of models, algorithms, and results. Even a succinct overview of the field would take many pages. Hence we review only those concepts that are directly relevant to this thesis and refer the reader to many excellent books and surveys available [88, 27, 73]. The problems we consider involve scheduling or allocating jobs to machines under various constraints. Unless otherwise mentioned n denotes the number of jobs and m denotes the number of machines. A schedule is *non-preemptive* if each job runs uninterrupted on one machine from start to finish. In a *preemptive* schedule, a job may be interrupted or may switch machines at any time. A schedule specifies for each time instant, the set of jobs executing at that instant, and the machines on which they are executing. In this thesis we focus mostly on non-preemptive schedules. However preemptive schedules play an important role, as relaxations, that can be used in the design and analysis of algorithms for non-preemptive schedules. We use C_j to denote the completion time of job j in a schedule. The two objective functions we will be concerned with are *makespan* and *average completion time*, and our goal will be to find schedules that *minimize* one of these two objectives. Makespan, denoted by $C_{\max}$, is simply the maximum over all jobs of the completion time of the job ($C_{\max} = \max_j C_j$). Average completion time as the name suggests is simply $(\sum_j C_j)/n$. Jobs could have positive weights w_j associated with them in which case we take the average weighted completion time $(\sum_j w_j C_j)/n$. Without loss of generality we can ignore the $1/n$ factor in the objective function. This is frequently done in the literature and the objective function is also referred to as *sum of (weighted) completion times* and we shall use both interchangeably. Jobs can have several constraints on them; in our work we consider *release dates* and *precedence constraints.* If job j has a release date r_j, then it cannot be started before time r_j. Precedence constraints, specified in the form of a directed acyclic graph, model dependencies between jobs. If job j precedes job i, denoted by $j \prec i$, then i cannot be started until j has been finished. We consider both single machine scheduling and parallel machine scheduling. For the most part we assume that machines are identical, except in Chapter 3, where we consider machines with different speeds. Graham et al. [44] introduced the compact $\alpha \mid \beta \mid \gamma$ notation to classify the many different scheduling problems that arise depending on the machine environment, job types, and objective function. Not all problems we consider can be described using the above notation, hence we use it only in Chapters 2 and 3. We give an example that explains some of the above definitions.

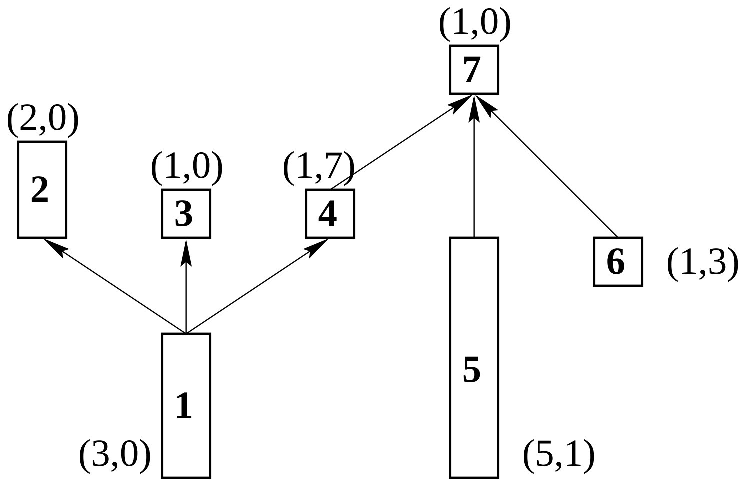

Figure 1.1: A problem instance with precedence constraints and release dates.

Figure 1.1 illustrates a problem instance with seven jobs. Jobs $J_1, \ldots, J_7$ are represented by a rectangle each and the number in bold inside indicates the job number. A job's processing time p_j, and its release date r_j, are indicated as a pair (p_j, r_j) next to it. Precedence constraints are shown by directed arrows. Figure 1.2 illustrates a non-preemptive schedule for the above instance on two identical parallel machines M1 and M2.

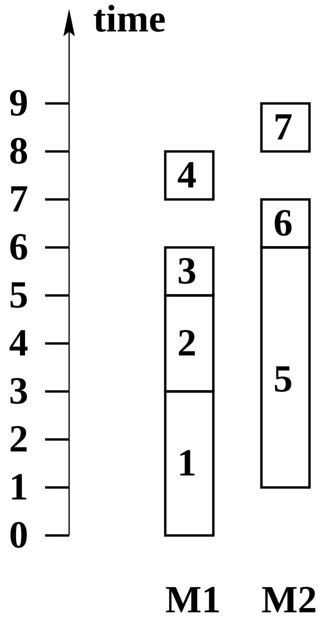

Figure 1.2: A schedule on two machines for the instance in Fig 1.1.

In the schedule shown, the completion times $C_1, \ldots, C_7$ are $3, 5, 6, 8, 6, 7$, and 9 respectively. The makespan of the schedule, $C_{\max}$, is 9 and the sum of completion times, $\sum_j C_j$, is 44. Release date constraints force jobs J_4 and J_5 to wait even though a machine is available. Since there is a precedence constraint from J_4 to J_7, job J_7 has to wait until J_4 completes.

List Scheduling: A popular heuristic in machine scheduling, that is used extensively both in theory and practice, is list scheduling. Given a set of jobs, a list is simply a total ordering on them that prioritizes the jobs, and by convention, jobs that come earlier in the list (order) have higher priority. In the literature, the name list scheduling is used to refer to different algorithms that have similarities, yet important differences as well. Almost all list scheduling algorithms have the following structure. First, a list or an ordering of the jobs is computed, typically using a relaxation such as a preemptive schedule or a linear programming relaxation. Jobs are considered according to the list order and scheduled on the available machines. Precedence constraints, release dates, and other constraints complicate the algorithm since jobs that come later in the list might be ready to execute before those that are earlier in the list. List scheduling algorithms differ in the rule used to pick the next job from the list when a machine is free. We describe two variants that are most common and useful. They represent two extreme cases. The first variant picks the earliest job in the list among those jobs that are currently ready to execute. Graham [42, 43] analyzed this rule and showed its effectiveness in minimizing makespan on parallel machines both with and without precedence constraints on the jobs. The second variant schedules the jobs strictly in the order of the list. In other words, it delays all jobs until the first job on the list has been scheduled. This variant has been used for minimizing average completion time [47] (and others) in conjunction with orderings produced by preemptive schedules and linear programming relaxations. We will use both variants in this thesis. In Chapter 2 we develop a new list scheduling variant called Delay List that combines the two variants in an interesting way to obtain algorithms for a variety of problems.

1.1.2 Approximation Algorithms and Inapproximability

We briefly but formally review notions of NP optimization problems and their approximability. An NP optimization problem (abbreviated by NPO henceforth) is defined below.

Definition 1.1.1 *An NPO problem Π is a four-tuple $(I, sol, obj, goal)$ such that*

- *I is the set of input instances of Π and is recognizable in polynomial time.*
- *For any instance $x \in I$, $sol(x)$ is the set of feasible solutions for I. Solutions for an instance x are constrained to be polynomially bounded in $|x|$ the size of x, that is there exists a polynomial p such that $y \in sol(x)$ implies that $|y| \leq p(|x|)$. Further, just as for NP languages, there should exist a polynomial time computable boolean function f such that $f(x, y)$ is true if and only if $y \in sol(x)$.*
- *For every instance x, the objective function obj assigns a positive value to each solution $y \in sol(x)$ and should be polynomial time computable.*
- *The parameter $goal \in \{\min, \max\}$ specifies whether the objective function should be minimized or maximized.*

Informally, feasible solutions correspond to those solutions that satisfy the constraints of the given problem. The objective function assigns to each feasible solution a positive real value. The goal specifies whether the problem is a minimization or a maximization problem. Decision versions of NPO problems are languages in NP. Decision versions are obtained by considering instances whose objective function values are constrained in some way. For example, if an optimization problem involves finding a schedule of minimum makespan, a decision version is obtained by considering all instances that have a makespan of at most 3. An NP-Hard optimization problem, for us, is an NPO problem for which there is an NP-Complete decision version. Thus, tractability of NPO problems is predicated on whether P = NP or not. Many years of substantial effort have failed to resolve the above question. It is widely believed now that P $\neq$ NP.

Approximation algorithms that provide near optimal solutions while running in polynomial time are clearly very well motivated by the assumed intractability of NPO problems. Given an instance x of Π, let $\text{OPT}(x)$ denote the the objective function value of an optimal feasible solution to x. A *ρ-approximation algorithm* $\mathcal{A}$, for a *minimization* problem Π, is a *polynomial* time algorithm that, on every instance x of Π, computes a solution $y \in sol(x)$ such that $obj(y) \leq \rho \cdot \text{OPT}(x)$. For *maximization* problems a *ρ-approximation algorithm* satisfies $obj(y) \geq \frac{1}{\rho} \cdot \text{OPT}(x)$. The asymmetry in the definition is to ensure that $\rho \geq 1$ for all approximation algorithms. The value ρ is called the *approximation ratio* or the *performance ratio* of $\mathcal{A}$ and in general could be a function of $|x|$, the input size. A *polynomial time approximation scheme* (PTAS) for an NPO problem is an algorithm which, for each fixed $\epsilon > 0$, provides a $(1+\epsilon)$ approximation while running in time polynomial (depending on ϵ) in the size of the input instance. Assuming P $\neq$ NP, a PTAS is the best result we can obtain for a strongly NP-Hard problem[2].

Understanding the approximability of an NPO problem Π involves finding the minimum value of ρ for which there is a ρ-approximation for Π. We denote this minimum value by γ, the approximability threshold of Π. If a problem has a PTAS, ρ can be made arbitrarily close to 1. However for most NPO problems γ is strictly greater than 1 and could also depend on the size of the instance. A problem is said to be *ρ-hard* if there is no ρ-approximation algorithm unless P = NP[3]. Hardness of approximation results provide lower bounds on γ, in contrast to approximation algorithms that provide upper bounds. We now proceed to

[2]An NPO problem is strongly NP-Hard if it is NP-Hard even if the input instance is specified in unary representation.

[3]Sometimes the hardness is based on weaker assumptions. For example it is "believed" that RP (the set of languages that can be recognized in randomized polynomial time) is a proper subset of NP. Thus hardness results can be based on the assumption that RP $\neq$ NP.

discuss some approaches to showing inapproximability results. Surprisingly, NPO problems, whose decision versions are all polynomial time reducible to each other (since they are NP-Complete), behave very differently in their approximability. For example, the multiprocessor scheduling problem has a PTAS while it is known that there is no polynomial factor approximation for the well known traveling salesman problem. This seeming anomaly is explained by the fact that reductions between NP-Complete problems, that preserve polynomial time computability, do not preserve the quality of the approximate solutions between the corresponding NPO problems. To study relationships between NPO problems, we need a more refined notion called an *approximation preserving reduction*. We will define one such reduction later that will illustrate the concept. Another useful notion in showing hardness results is that of *complete* problems. Completeness is a natural notion associated with complexity classes. A problem is complete for a class if it is a member of the class, and all other problems in the class reduce to it under an appropriate notion of reducibility. A problem is *hard* for a complexity class if all problems in the class reduce to it (it need not be a member of the class). For NPO classes we are interested in approximation preserving reductions. We emphasize that the notion of completeness is strongly tied to the notion of reducibility used. Complete problems are canonical hard problems in the class. For example, as their name suggests, NP-Complete problems are complete for the class NP under the notion of polynomial time reducibility. To show proper containment of a complexity class $\mathcal{A}$ in another larger class $\mathcal{B}$, it is sufficient to show that some problem in $\mathcal{B}$ is not contained in $\mathcal{A}$. Once this is accomplished, to show that a problem Π in $\mathcal{B}$ is not contained in $\mathcal{A}$, it is sufficient to show that Π is complete for $\mathcal{B}$. However, for such an approach to work, the reduction used to define completeness should satisfy the following property. If a problem Π reduces to a problem Π', then $\Pi' \in \mathcal{A}$ implies that $\Pi \in \mathcal{A}$.

Based on known results about the approximability thresholds of various problems, researchers have classified problems into a small number of classes (see [4] for a survey). This classification is by no means exhaustive but is useful in organizing known results, for proving new results via reductions, and to understand the underlying structure of NPO problems. We will be interested in two classes, namely PTAS[4], the class of problems that have a polynomial time approximation scheme, and APX, the class of problems problems that have a constant factor approximation ($\gamma = O(1)$). Other classes are similarly defined with larger

[4]We abuse notation by using PTAS to denote both an approximation scheme and the class of problems that have an approximation scheme.

thresholds for γ. Clearly PTAS is a subset of APX. It is natural to question whether the containment is proper. This is indeed the case for there are problems such as the k-center problem [34] that are in APX but have no PTAS unless P = NP. However, a complete problem for APX was not known until very recently. Papadimitriou and Yannakakis [85] defined a syntactic class called Max-SNP that is a subset of APX[5]. They also established that the well known Max 3-SAT problem, among several other problems, is complete for Max-SNP under a notion of reduction called L-reduction. In a remarkable line of work that culminated in [5] (see Arora's thesis [3] for details of the result in [5] and pointers to prior work), it was shown that Max 3-SAT has no PTAS unless P = NP. Thus, Max 3-SAT is a complete problem that, via L-reductions, can be used to show that other problems do not have a PTAS. We formally define an L-reduction below.

Definition 1.1.2 *A problem Π L-reduces to a problem Π' if there exist two polynomial time computable functions f and g, and positive constants α and β such that the following two conditions hold.*

- *The function f maps instances x of Π to instances x' of Π' such that*

$$\text{OPT}(x) \leq \alpha \cdot \text{OPT}(x').$$

- *The function g is a mapping from solutions for x', $sol'(x')$, to solutions for x, $sol(x)$, such that for every $y' \in sol(x')$*

$$|\text{OPT}(x) - obj(g(y'))| \leq \beta \cdot |\text{OPT}(x') - obj'(y')|.$$

An L-reduction is an approximation preserving reduction that relates the error in the solutions of the two problems by a constant factor that depends on α and β. It is not hard to show that if Π_1 L-reduces to Π_2, then a PTAS for Π_2 implies a PTAS for Π_1. Therefore, from the result of [5] it immediately follows that if Max 3-SAT L-reduces to a problem Π, then Π has no PTAS unless P = NP. In fact we can use any other complete problem for Max-SNP in place of Max 3-SAT. It was later established by Khanna et al. [70] and Crescenzi et al. [19, 20] that Max 3-SAT is complete for the larger class of APX under subtler notions of

[5]It is in fact a proper subset since only maximization problems are considered. The more interesting aspect is the fact that every problem in Max-SNP, by its syntactic characterization, has a constant factor approximation.

reductions. Hence Max 3-SAT is APX-Complete. Using L-reductions or other reductions from [70, 20], many problems have been shown to be either APX-Complete or APX-Hard, and thus do not have a PTAS unless P = NP. In this thesis we use only L-reductions. Hardness of approximation results for other problems (classes) are similarly shown using appropriate approximation preserving reductions from a canonical hard problem such as Max 3-SAT.

In addition to using approximation preserving reductions from known hard problems, there is a more direct approach to showing hardness. This approach has been successful for some scheduling problems and we briefly describe it below. Given a minimization problem Π, suppose we can find an NP-Complete language A, a polynomial time computable function f, and two constants α and β with $\alpha < \beta$ such that the following conditions hold. The function f maps *yes* instances of A (the strings that belong to A) to instances x of Π such that $\text{OPT}(x) \leq \alpha$. In addition, f maps *no* instances of A (the strings that do not belong to A) to instances x of Π such that $\text{OPT}(x) > \beta$. Then it is clear that unless P = NP, there is no β/α approximation for Π. This approach is more problem-specific than using generic approximation preserving reductions outlined earlier, but has been effective for several scheduling problems. See the survey in [75] for examples.

See Papadimitriou's book [86] for more details and related aspects of computational complexity.

1.2 Organization and Overview of Contributions

We next describe the problems considered in this thesis and our results.

1.2.1 Scheduling to Minimize Average Completion Time

In Chapter 2 we consider the problem of scheduling jobs on machines to minimize the sum of weighted completion times, that is $\sum_j w_j C_j$, and the special case when all $w_j = 1$. This measure has been of interest in operations research since the work of Smith in 1956 [104] and has several applications. Our motivation for studying sum of weighted completion times, aside from its intrinsic theoretical interest, comes from applications to compiler optimizations. Compile-time instruction scheduling is essential for effectively exploiting the fine-grained parallelism offered in pipelined, superscalar, and VLIW architectures (for example, see [53, 112]). Current research is addressing the issue of profile-based compiler optimization. In a recent paper, Chekuri et al. [12] show that sum of weighted completion times is the measure of interest in profile-driven code optimization; some of our results are related to the heuristics described and empirically tested therein.

Despite interest in the objective function for many years, progress on several basic problems in terms of approximation algorithms has been made only recently. Approximation algorithms have been developed for several variants [87, 47, 9, 15, 39, 100, 17] in the last couple of years. Several of these algorithms are based on solving a preemptive or linear programming relaxation and then using the solution to get an ordering on the jobs.

We introduce new techniques that generalize this basic paradigm. We use these ideas to obtain an improved $e/(e-1) \simeq 1.58$ approximation algorithm for one-machine scheduling to minimize average completion time ($w_j = 1$ for all j) with release dates. In the process we also obtain an *optimal* randomized on-line algorithm for the same problem that beats a lower bound for deterministic on-line algorithms. We consider extensions to the case of parallel machine scheduling, and for this we introduce two new ideas: first, we show that a preemptive one-machine relaxation is a powerful tool for designing parallel machine scheduling algorithms that simultaneously produce good approximations and have small running times; second, we show that a non-greedy "rounding" of the relaxation yields better approximations than a greedy one. This yields a $(2.89+\epsilon)$ approximation algorithm for minimizing average completion time with release dates in the parallel setting.

When jobs have precedence constraints we obtain a 2 approximation algorithm for the single machine case that matches the best known ratio for this problem. Our algorithm however is not based on solving a linear programming relaxation and is currently the most efficient constant factor approximation algorithm for the above problem. We also give a general algorithm that obtains an m-machine schedule using a one-machine schedule when jobs have release dates and precedence constraints. In particular, given a schedule that provides an α-approximation to the optimal one machine schedule, we obtain a $(2+2\alpha)$-approximation[6] to the optimal m-machine schedule. In the process we obtain a surprising generalization of Graham's list scheduling result [42]. Our conversion algorithm is simple and efficient. Combined with our 2 approximation result for the single machine problem we obtain constant factor approximation algorithms for several variants of the problems that are competitive with the best ratios known while having the advantage of efficiency

[6]In fact we obtain a slightly stronger result. See Chapter 2 for details.

and simplicity. We also apply the conversion algorithm to obtain improved approximation ratios for restricted precedence graphs such as in-trees, out-trees, and series-parallel graphs.

1.2.2 Minimizing Makespan on Machines with Different Speeds

The problem of scheduling precedence constrained jobs on a set of identical parallel machines to minimize makespan is one of the oldest problems for which approximation algorithms have been devised. Graham [42] showed that a simple list scheduling algorithm gives a ratio of 2 and it is the best known algorithm to date. In Chapter 3 we consider a generalization of this model in which machines have different speeds. In the scheduling literature such machines are called *uniformly related*. We formalize the problem below. We are given a set of n jobs with precedence constraints between them that are to be scheduled on a set of m machines. Machine i has a speed factor s_i and job j has a processing requirement p_j. The uniformity assumption means that job j takes p_j/s_i time units to execute on machine i. The objective is to find a schedule to minimize $C_{\max}$, the makespan of the schedule. We are interested in non-preemptive schedules though our results are valid for the preemptive case as well. In the scheduling literature [44] where problems are classified in the $\alpha|\beta|\gamma$ notation, this problem is referred to as $Q|prec|C_{\max}$.

Jaffe [62] gave an approximation algorithm achieving a ratio of $O(\sqrt{m})$ and this was the first algorithm with a ratio independent of the speeds of the processors. Chudak and Shmoys [17] recently obtained a much improved ratio of $O(\log m)$. Their algorithm is based on rounding a linear programming relaxation of the problem. We provide an alternative algorithm with provably more efficient running time while achieving a similar approximation ratio of $O(\log m)$. Our algorithm, in addition to being efficient, is simple, intuitive, and combinatorial. Using the results of Shmoys, Wein, and Williamson [103], we can extend our algorithm to obtain an $O(\log m)$ approximation ratio even if jobs have release dates. We also show that our algorithm achieves an approximation ratio of 4 when the precedence constraints between the jobs are induced by a collection of chains.

1.2.3 Scheduling Problems in Parallel Query Optimization

In Chapter 4 we introduce a class of novel multiprocessor scheduling problems that arise in the optimization of SQL queries for parallel machines. The scheduling model is developed to reflect the communication costs inherent in a shared-nothing parallel environment where independent machines communicate via message passing. The overall goal is to schedule a tree of inter-dependent communicating operators while exploiting both inter-operator and intra-operator parallelism. The tree represents the computation of a query in the database system. We obtain several scheduling variants depending on the constraints allowed. We concentrate on the specific problem of scheduling a *pipelined operator tree* (POT) in which all operators run in parallel using inter-operator parallelism. In the POT scheduling problem the input instance is a tree with weights on both nodes and edges. Node weights represent cost of operators and edge weights represent cost of remote communication between the operators. Communication cost between two operators connected by an edge is incurred only if they are assigned different machines. The scheduling problem is to assign operators to machines so as to minimize the maximum machine load. We obtain a polynomial time approximation scheme for this NP-Hard problem. We also develop two other approximation algorithms that are simple and efficient. The first of these has an approximation ratio of 3.56 and has a running time of $O(n \log n)$ while the other has a ratio of 2.87 and the same running time (albeit with a larger hidden constant).

1.2.4 Approximability of Vector Packing Problems

Chapter 5 examines several variants of vector packing problems. Our main motivation for studying these problems comes from multi-dimensional resource scheduling problems encountered in several systems. Consider a task in a system where each task requires CPU time, disk I/O, and network bandwidth (through a network controller for example). The requirements of such a task are typically reduced to a single "work" measure that captures to a certain extent the aggregate requirements. This simplification is done in most cases to make the problem tractable by reducing the search space complexity. However, for large tasks, especially those that might have skewed resource requirements, it is advantageous to model their resource requirements as multi-dimensional vectors. The work in [35, 36, 37, 38] demonstrates the effectiveness of this approach for parallel query processing and scheduling continuous media databases. We examine the following basic scheduling problem that arises in the applications mentioned above.

Let J be a set of n d-dimensional real vectors $p_1, \ldots, p_n$. Each vector p_j represents a job that has d distinct resource requirements. The quantity p_j^i indicates the resource requirement of job j for resource i. The goal is to schedule these jobs on a set of m parallel machines where each machine has each of the d resources required by the jobs. For our purposes a schedule is simply a partition of the jobs into m sets, one for each of the machines. Given a schedule S, the *load* on resource i of machine k is simply the sum of the ith coordinates of all the jobs scheduled on machine k. Our goal is to minimize the load of the most loaded resource over all machines. The objective function is based on the assumption that the load, as defined above, is a good measure of the schedule length. This assumption provides a good approximation for large tasks that are typically encountered in applications such as database systems [36].

For the scheduling problem above we obtain a polynomial time approximation scheme for every fixed d. We generalize the ideas of Hochbaum and Shmoys [55] for multi-processor scheduling in a non-trivial way to obtain our algorithm. For arbitrary d we obtain an $O(\min(\log dm, \log^2 d))$ approximation. Further we show that for arbitrary d unless NP = ZPP there is no constant factor approximation algorithm.

Our vector scheduling problem is closely related to other vector packing problems, namely vector bin packing (or multi-dimensional bin packing) [33, 23] and packing integer programs (PIPs) [93, 105]. We obtain several interesting approximation algorithms and inapproximability results for these variants as well.

The work described in this thesis is mostly contained in papers published earlier [11, 12, 15, 10] and some others in preparation [13, 14].

Chapter 2

Minimizing Average (Weighted) Completion Time

2.1 Introduction

In this chapter[1] we present new approximation results for scheduling to minimize average (weighted) completion time (equivalently sum of (weighted) completion times). We are given n jobs $J_1, \ldots, J_n$, where job J_j has processing time p_j and a positive weight w_j. We consider several variants but the objective in all of them is to find schedules to minimize either $\sum_j C_j$ (average completion time) or $\sum_j w_j C_j$ (average weighted completion time) where C_j is the completion time of job j in the schedule. We are primarily interested in non-preemptive schedules. A simple variant is when jobs have no other constraints and the goal is to schedule them on a single machine. For this case, the weighted completion time problem can be solved optimally in polynomial time [104] using what is known as Smith's rule. Smith's rule states that scheduling the jobs in non-increasing order of w_j/p_j is optimal. We are interested in a more general setting with release dates, precedence constraints, and multiple machines any of which make the problem $\mathcal{NP}$-hard [73]. Thus, we will consider approximation algorithms, or, in an on-line setting, competitive ratios.

Recent work has led to constant-factor approximations for sum of weighted completion times for a variety of these $\mathcal{NP}$-hard scheduling problems [87, 47, 9, 39, 17, 100]. Most of

[1] This chapter is joint work with Rajeev Motwani, Balas Natarajan, and Cliff Stein and appears in [15] and [14].

these algorithms work by first constructing a relaxed solution, either a preemptive schedule or a linear-programming relaxation. These relaxations are used to obtain an ordering of the jobs, and then the jobs are list scheduled as per this ordering.

We contribute in two ways. First we introduce several new techniques which generalize the above mentioned basic paradigm. Second we also give the first constant factor algorithms for several variants that do not rely on solving a linear programming relaxation. We are thus able to obtain simple, efficient, and competitive algorithms.

We use our techniques to obtain an improved approximation algorithm for one-machine scheduling to minimize average completion time with release dates. Our main result here is a $\frac{e}{e-1} \approx 1.58$-approximation algorithm. This algorithm can be turned into a randomized on-line algorithm with the same bound, where an algorithm is *on-line* if before time r_j it is unaware of J_j, but at time r_j it learns all the parameters of J_j. This randomized on-line algorithm is particularly interesting as it beats a lower bound for deterministic on-line algorithms [60], and matches a recent lower bound for randomized on-line algorithms [108]. We then consider the case when jobs have precedence constraints. We give a 2 approximation algorithm for this problem that matches the best ratio prior to our work. In contrast to earlier work based on linear programming relaxations our algorithm is combinatorial. We use our algorithm to obtain combinatorial algorithms for the parallel machine case as well.

We then consider extensions to parallel machine scheduling, and introduce two new ideas: first, we show that a preemptive *one-machine* relaxation is a powerful tool for designing parallel machine scheduling algorithms that simultaneously produce good approximations and have small running times; second, we show that a non-greedy "rounding" of the relaxation produces better approximations than simple list scheduling. In fact, we prove a general theorem relating the value of one-machine relaxations to that of the schedules obtained for the original m-machine problems. This theorem applies even when there are precedence constraints and release dates. Using this theorem, we can obtain simple combinatorial algorithms for many variants while achieving constant factor approximation ratios that are competitive with those obtained by less efficient algorithms. We also obtain improved approximations for precedence graphs such as in-trees, out-trees, and series-parallel graphs, which are of interest in compiler applications that partly motivated our work.

We begin with a more detailed discussion of our results and their relation to earlier work.

One-Machine Scheduling with Release Dates

The first constant-factor approximation algorithm for minimizing the average completion time on one machine with release dates was the following 2-approximation algorithm of Phillips, Stein and Wein [87]. First, an optimal preemptive schedule P is found using the shortest remaining processing time (SRPT) algorithm [73] which, at any time, runs an available job that has the least processing time left; note that this is an on-line algorithm. Given P, the jobs are ordered by increasing completion times, C_j^P, and are scheduled non-preemptively according to that ordering, introducing idle time as necessary to account for release dates. A simple proof shows that each job J_j completes at time no later than $2C_j^P$, implying a 2-approximation. Two other 2-approximation algorithms have been discovered subsequently [60, 107], and it is also known that no deterministic on-line algorithm has approximation ratio better than 2 [60, 107]. This approximation technique has been generalized to many other scheduling problems, and hence finding better approximations for this basic problem is believed to be an important step towards improved approximations for more general problems.

One of the main results of this chapter is a deterministic off-line algorithm for this basic problem that gives an $\frac{e}{e-1}$-approximation ($\frac{e}{e-1} \approx 1.58$). We also obtain an *optimal* randomized on-line algorithm(in the *oblivious* adversary model) with expected competitive ratio $\frac{e}{e-1}$. This beats the deterministic on-line lower bound. Our approach is based on what we call *α-schedules* (this notion was also used by [87] and [48] in a somewhat different manner). Given a preemptive schedule P and $\alpha \in [0,1]$, we define $C_j^P(\alpha)$ to be the time at which αp_j, an α-fraction of J_j, is completed. An *α-schedule* is a non-preemptive schedule obtained by list scheduling jobs in order of increasing $C_j^P(\alpha)$, possibly introducing idle time to account for release dates. Clearly, an α-schedule is an on-line algorithm; moreover, for $\alpha = 1$, the α-scheduler is exactly the algorithm of Phillips et al. [87] and hence a 2-approximation. We show that for arbitrary α, an α-scheduler has a *tight* approximation ratio of $1 + 1/\alpha$. Given that $1 + 1/\alpha \geq 2$ for $\alpha \in [0,1]$, it may appear that this notion of α-schedulers is useless for obtaining ratios better than 2.

A key insight is that a worst-case instance with performance ratio $1+1/\alpha$ for one value of α is not a worst-case instance for many other values of α. This suggests that a randomized algorithm which picks α at random, and then behaves like an α-scheduler, may lead to an approximation better than 2. Unfortunately, we show that choosing α uniformly at random gives an expected approximation ratio of 2, and that this is tight. However, this leaves open the possibility that for any given instance I, there exists a choice $\alpha(I)$ for which the $\alpha(I)$-schedule yields an approximation better than 2. We refer to the resulting deterministic off-line algorithm which, given I, chooses α to minimize $\alpha(I)$, as Best-α.

It turns out, however, that the randomized on-line algorithm which chooses α to be 1 with probability 3/5 and 1/2 with probability 2/5 has competitive ratio 1.8; consequently, for any input I, either the 1-schedule or the $\frac{1}{2}$-schedule is no worse than an 1.8-approximation. The non-uniform choice in the randomized version suggests the possibility of defining randomized choices of α that may perform better than 1.8 while being easy to analyze. In fact, our result here is that it is possible to define a distribution for α that yields a randomized on-line $\frac{e}{e-1}$-approximation algorithm implying that Best-α is an $\frac{e}{e-1}$-approximation algorithm. It should be noted that Best-α can be implemented in $O(n^2 \log n)$ time. We also establish some lower bounds on the performance ratios of the algorithms described above. Torng and Uthaisombut [110] recently showed that our analysis of Best-α is tight by giving instances on which the approximation ratio achieved by Best-α is arbitrarily close to $e/(e-1)$.

Our bounds are actually job-by-job, i.e., we produce a schedule N in which $E[C_j^N] \leq \frac{e}{e-1} C_j^P$ for all j where $E[C_j^N]$ is the expected completion time of J_j in the non-preemptive schedule. Thus, our conversion results generalize to *weighted* completion time. However, since for the weighted case even the preemptive scheduling problem is $\mathcal{NP}$-hard (given release dates), we must use an approximation for the relaxation. There is a 2-approximation for the preemptive case [47], which yields a 3.16-approximation for the non-preemptive case. However this does not improve earlier results.

Independently, Goemans [39] has used some related ideas to design a 2-approximation algorithm for the problem of non-preemptive scheduling on one machine so as to minimize the average weighted completion time. His algorithm is also a Best-α algorithm, but it works off of a linear programming relaxation rather than a preemptive schedule. Schulz and Skutella [100] improved the results of Goemans for the weighted case to obtain a ratio of 1.693. Their improvement is obtained by extending the idea of using a single random α for all jobs to using independent random α values for each job.

One-Machine Scheduling with Precedence Constraints

Scheduling with precedence constraints is NP-Hard even without release dates [72] if all $w_j = 1$ and p_j are allowed to be arbitrary or if all $p_j = 1$ and w_j are allowed to be arbitrary.

Polynomial time solvable cases include precedence constraints induced by forests (in and out forests) and generalized series-parallel graphs [72, 1, 109]. The first approximation algorithm for this problem was given by Ravi et al. [94] and had a guarantee of $O(\log n \log L)$ where $L = \sum_j p_j$. They in fact solve a more general problem called the storage time product problem. Hall et al. and Schulz [48, 98] gave the first constant factor approximations using linear programming relaxations. It is interesting to note that several different formulations [47] give the same bound of 2.

We give an efficient combinatorial 2 approximation algorithm for the single machine problem which matches the ratio achieved in [47]. Our algorithm has two advantages. First, the algorithms in [47] are based on solving linear programming relaxations while ours is based on solving a minimum cut, and hence is simpler and more efficient. Second, combining this new algorithm with the conversion algorithm to be discussed below, we get an efficient combinatorial algorithm for the multiple machine case as well. Margot, Queyranne, and Wang [79] have independently obtained the same 2 approximation algorithm that we present here.

In a recent paper, Chudak and Hochbaum [16] show a half integral linear programming relaxation for the same problem. They also achieve an approximation ratio of 2. The half-integral program can be solved using minimum cut computations, and thus also yields a combinatorial algorithm. However the running time obtained is worse than that of our algorithm by a factor of n. Their relaxation is a slight modification of Potts's linear ordering relaxation [90]. Hall et al. [47] showed that Potts's relaxation is feasible for their completion time relaxation, and hence also provides a 2 approximation. Though factor 2 integrality gaps have been demonstrated for both the completion time relaxation and the time indexed relaxation [47], no such gap had been shown for the linear ordering relaxation. We show a factor 2 integrality gap for the linear ordering relaxation thus establishing that it is no stronger than the other relaxations from a worst case analysis point of view. Surprisingly, the instance on which we show the gap uses expander graphs. Our proof also establishes the same integrality gap for the parallel and series inequality formulation of Queyranne and Wang [91] via the results of Schulz [97].

Scheduling Parallel Machines with Release Dates

We consider the generalizations of the single machine problems to the case of m identical parallel machines. We first consider the problem of minimizing average completion time with release dates and no precedence constraints. Extending the techniques to the m-machine problem gives rise to two complications: the problem of computing an optimal m-machine preemptive schedule is $\mathcal{NP}$-hard [26], and the best known approximation ratio is 2 [87]; further, the conversion bounds from preemptive to non-preemptive schedules are not as good. Chakrabarti et al. [9] obtain a bound of 7/3 on the conversion from preemptive to non-preemptive case, yielding a 14/3-approximation for scheduling on m machines with release dates. Several other algorithms don't use the preemptive schedule but use a linear programming relaxation. Phillips et al. [87] gave the first such algorithm, a 24-approximation algorithm. This has been greatly improved to $4 + \epsilon$ [48], $4 - \frac{1}{m}$ [47], and 3.5 [9]. Using a general on-line framework [9], one can obtain an algorithm with an approximation ratio of $2.89 + \epsilon$. Unfortunately, the algorithm with the best approximation is inefficient, as it uses the polynomial approximation scheme for makespan due to Hochbaum and Shmoys [56].

We give a new algorithm for this problem. First we introduce a different relaxation – a *preemptive one-machine relaxation*. More precisely, we maintain the original release dates and allow preemptions, but divide all the processing times by m. We then compute a one-machine schedule. The resulting completion time ordering is then used to generate a non-preemptive m-machine schedule that is a 3-approximation. Our algorithm is not only on-line but is also efficient and runs in $O(n \log n)$ time. We then show that the approximation ratio can be improved to 2.85, using the general conversion algorithm described below. This improves on the approximation bounds of previous algorithms and gives a much smaller running time of $O(n \log n)$. Subsequent to our work, Schulz and Skutella [100] using some of our ideas have obtained an approximation ratio of 2 for the more general case of sum of weighted completion times.

A General Conversion Algorithm for Scheduling with Precedence Constraints

We now consider the weighted completion time problem with precedence constraints *and* release dates on parallel machines; for the time being, we assume that all release dates are 0, but our results apply unchanged for arbitrary release dates. For arbitrary m, the problem is $\mathcal{NP}$-hard even without precedence constraints, unless the weights are all equal (without release dates); on the other hand, the problem is strongly $\mathcal{NP}$-hard even when all weights are identical and the precedence graph is a union of chains [26]. An approximation ratio of $5.33 + \epsilon$ is achievable with release dates and precedence constraints [9]. This has been recently improved to 4 in [82].

We obtain a fairly general algorithm for m-machine problems with precedence constraints, in addition to release dates and job weights. To do so, we first solve a one-machine preemptive relaxation, and apply an algorithm we call DELAY LIST to get an m-machine non-preemptive schedule. Since, in general, the one-machine preemptive relaxation is also $\mathcal{NP}$-hard, we would have to settle for a ρ-approximation for it; then, our algorithm would give a $(2\rho + 2)$-approximation for the m-machine case. Using the above algorithm we obtain improved approximation algorithms for special cases of precedence constraints induced by trees and series parallel graphs. While we do not, at present, improve the best known bounds for the most general version, we feel that our general theorem is of independent interest and likely to find applications in the future. Further, our conversion algorithm has the advantage of being extremely simple and combinatorial. Note that in applications such as compilers [12], speed and simplicity are sometimes more important than getting the best possible ratio. Finally, our algorithm has a surprising property: it gives schedules that are good for both makespan and average completion time (Chakrabarti et al. [9], and Stein and Wein [106] also have shown the existence of such schedules).

We also show that a similar but simpler technique gives a 2-approximation if the precedence graph is an in-tree; note that the optimal one-machine schedule for in-trees is polynomial-time computable. The idea is to use list scheduling with a one-machine schedule as the "list," which is a natural heuristic for *makespan* [43].

The rest of this chapter is organized in two sections. In Section 2.2 we give the algorithms for the one-machine case and in Section 2.3 we treat the parallel machine case.

2.2 Scheduling on a Single Machine

2.2.1 One-Machine Scheduling with Release Dates

In this section, we present our results for one-machine scheduling with release dates to minimize average completion time. Let P be a preemptive schedule, and let C_i^P and C_i^α denote the completion time of J_i in P and in the non-preemptive α-schedule derived from P, respectively. We begin by analyzing simple α-schedules. Techniques from [87, 47] are easily generalized to yield the following:

Theorem 2.2.1 *Given an instance of one-machine scheduling with release dates, for any $\alpha \in [0, 1]$, an α-schedule has $\sum_j C_j^\alpha \leq (1 + 1/\alpha) \sum_j C_j^P$, and this is tight.*

Proof. Index the jobs by the order of their α-points in the preemptive schedule P. Let $r_j^{\max} = \max_{1 \le k \le j} r_j$ be the latest release date among jobs with α points no greater than that of j. By time $r_j^{\max}$, jobs 1 through j have all been released, and hence

$$C_j^\alpha \le r_j^{\max} + \sum_{k=1}^{j} p_j. \tag{2.1}$$

We know that $C_j^P \ge r_j^{\max}$, by definition j cannot finish earlier than $r_j^{\max}$. We also know that $C_j^P \ge \alpha \sum_{k=1}^{j} p_j$, since the α fractions of jobs $1, \ldots, j$ must run before time C_j^P. Plugging these last two inequalities into (2.1) yields $C_j^\alpha \le (1 + \frac{1}{\alpha})C_j^P$. Summing over all j yields the lemma.

To see that this is tight, consider the following class of instances. Let ϵ be a small positive number. We will also allow jobs with processing time 0, although the proof can be modified even if these are not allowed. At time 0, we release a job with processing time 1. At time $\alpha - \epsilon$, we release a job with processing time ϵ and at time $\alpha + \epsilon$, we release x jobs of processing time 0. The optimal preemptive completion time is $x(\alpha + \epsilon) + \alpha + 1 + \epsilon$, while the completion time of the α-schedule is $1 + 2\alpha + x(1 + \alpha)$. As x gets large and ϵ goes to 0, the ratio between the two goes to $1 + \frac{1}{\alpha}$. The same instance also shows that the approximation ratio is tight. $\square$

This theorem, in and of itself, always yields approximation bounds that are worse than 2.

We thus introduce a new fact that ultimately yields better algorithms. We will show that the *makespan* of an α-schedule is within a $(1 + \alpha)$-factor of the makespan of the corresponding preemptive schedule; in fact, we will prove a stronger result in Lemma 2.2.3 below. Thus, the idle time introduced in the non-preemptive schedules decreases as α is reduced from 1 to 0. On the other hand, the (worst-case) bound on the completion time of any specific job increases as α goes from 1 to 0. It is the balancing of these two effects that leads to better approximations. In the following discussion we do not assume that the preemptive schedule is the optimal preemptive schedule found using SRPT. In fact our results on converting preemptive schedules to non-preemptive schedules apply in general but when we want to prove upper bounds on the performance ratio for sum of completion times, we assume that the preemptive schedule is the optimal preemptive schedule which is a lower bound on the optimal non-preemptive schedule.

Let $S_i^P(\beta)$ denote the set of jobs which complete exactly β fraction of their processing time before C_i^P in the schedule P (note that J_i is included in $S_i^P(1)$). We overload notation by using $S_i^P(\beta)$ to also denote the sum of processing times of all jobs in the set $S_i^P(\beta)$; the meaning should be clear from the context. Let T_i be the total idle time in P before J_i completes.

The preemptive completion time of J_i can be written as the sum of the idle times and processing times of jobs that ran between C_i^P. This yields the following lemma:

Lemma 2.2.2 $C_i^P = T_i + \sum_{0<\beta\le 1} \beta S_i^P(\beta)$.

We next upper bound the completion time of a job J_i in the α-schedule.

Lemma 2.2.3

$$C_i^\alpha \le T_i + (1+\alpha)\sum_{\beta\ge\alpha} S_i^P(\beta) + \sum_{\beta<\alpha} \beta S_i^P(\beta).$$

Proof. Let $J_1, \ldots, J_{i-1}$ be the jobs that run before J_i in the α schedule. We will give a procedure which converts the preemptive schedule into a schedule in which

(C1) jobs $J_1, \ldots, J_i$ run non-preemptively in that order,

(C2) the remaining jobs run preemptively, and

(C3) the completion time of J_i obeys the bound given in the lemma.

Since the actual C_i^α is no greater than the completion time of J_i in this schedule, the lemma will be proven.

Splitting up the second term in the bound from Lemma 2.2.2, we get the following equation:

$$C_i^P = T_i + \sum_{\beta<\alpha} \beta S_i^P(\beta) + \sum_{\beta\ge\alpha} \alpha S_i^P(\beta) + \sum_{\beta\ge\alpha} (\beta-\alpha) S_i^P(\beta).$$

Let $J^B = \bigcup_{\beta\ge\alpha} S_i^P(\beta)$ and $J^A = J - J^B$. We can interpret the four terms in the above equation as (1) the idle time in the preemptive schedule before C_i^P, (2) the pieces of jobs in J^A that run before C_i^P, (3) for each job $J_j \in J^B$, the pieces of J_j that run before $C_j^P(\alpha)$, and (4) for each job $J_j \in J^B$, the pieces of J_j that run between $C_j^P(\alpha)$ and C_i^P. Let x_j be the β for which $J_j \in S_i^P(\beta)$. Then $\sum_{\beta\ge\alpha}(\beta-\alpha)S_i^P(\beta)$ can be rewritten as $\sum_{J_j\in J^B}(x_j-\alpha)p_j$ and observe that $(x_j-\alpha)p_j$ is the amount of processing of job J_j that occurs between $C_j^P(\alpha)$ and C_i^P.

Let $J^C = \bigcup_{j \leq i}\{J_j\}$. Clearly J^C is a subset of J^B. Now think of schedule P as an ordered list of pieces of jobs (with sizes). For each $J_j \in J^C$ modify the list by (1) removing all pieces of jobs that run between $C_j^P(\alpha)$ and C_i^P and (2) inserting a piece of size $(x_j - \alpha)p_j$ at the point corresponding to $C_j^P(\alpha)$. In this schedule, we have pieces of size $(x_j - \alpha)p_j$ of jobs $J_1, \ldots, J_i$ in the correct order (plus other pieces of jobs). Now convert this ordered list back into a schedule by scheduling the pieces in the order of the list, respecting release dates. We claim that job i still completes at time C_i^P. To see this observe that the total processing time before C_i^P remains unchanged and that other than the pieces of size $(x_j - \alpha)p_j$, we only moved pieces later in time, so no additional idle time need be introduced.

Now, for each job $J_j \in J^C$, extend the piece of size $(x_j - \alpha)p_j$ to one of size p_j by adding $p_j - (x_j - \alpha)p_j$ units of processing and replace the pieces of J_j that occur earlier, of total size αp_j, by idle time. We now have a schedule in which $J_1, \ldots J_i$ are each scheduled non-preemptively for p_j units of time and in which the completion time of J_i is

$$
\begin{aligned}
C_i^P + \sum_{J_j \in J^C} (p_j - (x_j - \alpha)p_j) &\leq C_i^P + \sum_{J_j \in J^B} (p_j - (x_j - \alpha)p_j) \\
&= C_i^P + \sum_{\beta \geq \alpha} (1 - \beta + \alpha) S_i^P(\beta) \\
&= T_i + (1 + \alpha) \sum_{\beta \geq \alpha} S_i^P(\beta) + \sum_{\beta < \alpha} \beta S_i^P(\beta),
\end{aligned}
$$

where the second equality just comes from re-indexing terms by β instead of j, and the third comes from plugging in the value of C_i^P from Lemma 2.2.2. To complete the proof, we observe that the remaining pieces in the schedule are all from jobs in $J - J^C$, and we have thus met the conditions (C1), (C2) and (C3) above. □

Although we will not use it directly, applying Lemma 2.2.3 to the last job yields:

Corollary 2.2.4 *The makespan of the α-schedule is at most $(1 + \alpha)$ times the makespan of the corresponding preemptive schedule, and there are instances for which this bound is tight.*

Having analyzed completion times as in Lemma 2.2.3, we see that the approximation ratio is going to depend on the distribution of the different sets $S_i^P(\beta)$. To avoid the worst case α, we choose α randomly, according to some probability distribution. We now give a very general bound on this algorithm, which we call RANDOM-α.

Lemma 2.2.5 *Suppose α is chosen from a probability distribution over $[0,1]$ with a density function f. Then, the $E[\sum_i C_i^\alpha] \leq (1+\delta)\sum_i C_i^P$, where*

$$\delta = \max_{0<\beta\leq 1} \int_0^\beta \frac{1+\alpha-\beta}{\beta} f(\alpha)\,\mathrm{d}\alpha.$$

Proof. We will show that the expected completion time of any job J_i is within $(1+\delta)$ of its preemptive completion time. From Lemma 2.2.3 it follows that for any given α,

$$C_i^\alpha \quad \leq \quad T_i + (1+\alpha)\sum_{\beta\geq\alpha} S_i^P(\beta) + \sum_{\beta<\alpha} \beta S_i^P(\beta).$$

Therefore, when α is chosen according to f, the expected completion time of J_i, $E[C_i^\alpha] = \int_0^1 f(\alpha) C_i^\alpha \,\mathrm{d}\alpha$, is bounded by

$$T_i + \int_0^1 f(\alpha)\left((1+\alpha)\sum_{\beta\geq\alpha} S_i^P(\beta) + \sum_{\beta<\alpha}\beta S_i^P(\beta)\right)\,\mathrm{d}\alpha$$

since T_i is independent of α. We now bound the second term in the above expression

$$\begin{aligned}
& \int_0^1 f(\alpha)\left((1+\alpha)\sum_{\beta\geq\alpha} S_i^P(\beta) + \sum_{\beta<\alpha}\beta S_i^P(\beta)\right)\,\mathrm{d}\alpha \\
= & \sum_{0<\beta\leq 1} S_i^P(\beta)\left(\int_0^\beta (1+\alpha) f(\alpha)\,\mathrm{d}\alpha + \int_\beta^1 \beta f(\alpha)\,\mathrm{d}\alpha\right) \\
= & \sum_{0<\beta\leq 1} \beta S_i^P(\beta)\left(1 + \int_0^\beta \frac{1+\alpha-\beta}{\beta} f(\alpha)\,\mathrm{d}\alpha\right) \\
\leq & \left(1 + \max_{0<\beta\leq 1}\int_0^\beta \frac{1+\alpha-\beta}{\beta} f(\alpha)\,\mathrm{d}\alpha\right)\sum_{0<\beta\leq 1} \beta S_i^P(\beta) \\
\leq & (1+\delta)\sum_{0<\beta\leq 1} \beta S_i^P(\beta).
\end{aligned}$$

It follows that

$$E[C_i^\alpha] \leq T_i + (1+\delta)\sum_{0<\beta\leq 1}\beta S_i^P(\beta) \leq (1+\delta) C_i^P.$$

Using linearity of expectations, it is easy to show that the expected sum of completion times of the schedule is within $(1+\delta)$ of the preemptive schedule's sum of completion times.

□

With the above lemma in place, we can simply choose different pdf's to establish different bounds.

Theorem 2.2.6 *For the problem of scheduling to minimize weighted completion time with release dates,* Random-α *performs as follows:*

1. *If α is chosen uniformly, the expected approximation ratio is at most* 2.
2. *If α chosen to be* 1 *with probability* 3/5 *and* 1/2 *with probability* 2/5, *the expected approximation ratio is at most* 1.8.
3. *If α chosen with the density function $f(\alpha) = \frac{e^\alpha}{e-1}$, the expected approximation ratio is at most $\frac{e}{e-1} \approx 1.58$.*

Proof.

1. Choosing α uniformly corresponds to the pdf $f(\alpha) = 1$. Plugging into the bound from Lemma 2.2.5, we get an approximation ratio of

$$\begin{aligned} 1 + \max_{0<\beta\le 1} \int_0^\beta \frac{1+\alpha-\beta}{\beta} \mathrm{d}\alpha &= 1 + \max_{0<\beta\le 1} \frac{1}{\beta}\left((1-\beta)\beta + \frac{\beta^2}{2}\right) \\ &= 1 + \max_{0<\beta\le 1}\left(1 - \frac{\beta}{2}\right) \\ &\le 2. \end{aligned}$$

2. Omitted.
3. If $f(\alpha) = \frac{e^\alpha}{e-1}$ then

$$\begin{aligned} 1 + \max_{0<\beta\le 1} \int_0^\beta \left(\frac{1+\alpha-\beta}{\beta}\right)\left(\frac{e^\alpha}{e-1}\right) \mathrm{d}\alpha &= 1 + \max_{0<\beta\le 1} \frac{1}{\beta(e-1)}((1-\beta)e^\beta \\ &\quad +(\beta-1)e^\beta - ((1-\beta)-1)) \\ &= 1 + \max_{0<\beta\le 1} \frac{1}{e-1} \\ &= \frac{e}{e-1}. \end{aligned}$$

□

In the off-line setting, rather than choosing α randomly, we can try different values of α and choose the one that yields the best schedule. We call the algorithm which computes the schedule of value $\min_\alpha \sum_j C_j^\alpha$, Best-$\alpha$.

Theorem 2.2.7 *Algorithm* BEST-α *is an* $e/(e-1)$*-approximation algorithm for nonpreemptive scheduling to minimize average completion time on one machine with release dates. It runs in* $O(n^2 \log n)$ *time.*

Proof. The approximation bound follows from Theorem 2.2.6. For the running time, we observe that given a preemptive SRPT schedule we can efficiently determine the best possible choice of α. The SRPT schedule only preempts at release dates. Thus it has at most $n-1$ preemptions and including 0 and 1 there are are at most $n+1$ "combinatorially distinct" values of α for a given preemptive schedule. □

In the on-line setting, we cannot implement BEST-α. However, if we choose α randomly we get the following:

Theorem 2.2.8 *There is a polynomial-time randomized on-line algorithm with an expected competitive ratio* $e/(e-1)$ *for the problem of non-preemptive scheduling to minimize average completion time on one machine in the presence of release dates.*

Proof. The randomized on-line algorithm is the following. The algorithm picks an $\alpha \in (0,1]$ at random according to the density function $f(x) = \frac{e^x}{e-1}$ before receiving any input (this is the only randomness used in the algorithm). The algorithm simulates the off-line preemptive SRPT schedule and a job is scheduled in the non-preemptive schedule at the exact time when it finishes α fraction of its processing time in the simulated SRPT schedule. Observe that this rule leads to a valid on-line non-preemptive schedule and that in fact the order of the jobs scheduled is exactly the same as in the α-schedule. To show the bound on the expected competitive ratio, we claim that the bounds in Lemma 2.2.3 (and hence Theorem 2.2.6 also) hold for the non-preemptive schedule created by the on-line algorithm. A careful examination of the proof of Lemma 2.2.3 makes this clear. □

We give some negative results in the following theorem.

Theorem 2.2.9 *For the problem of scheduling to minimize average completion time with release dates,* RANDOM-α *performs as follows:*

1. *If α is chosen uniformly, the expected approximation ratio is at least* 2.
2. *For the* BEST-α *algorithm, the approximation ratio is at least* $4/3$.

Proof. We will use a set of parameterized instances to show all the above bounds. We define an instance $I(\delta, \epsilon, n)$ as follows. At time 0 a job of size 1 is released and at time

$\delta < 1$, n jobs of size $\epsilon = o(1/n^2)$ each are released. The optimal preemptive schedule for this instance has a total completion time of $1 + n\epsilon + n(\delta + \frac{n+1}{2}\epsilon)$. The optimal non-preemptive schedule for this instance can be obtained by first completing all the small jobs and running the large job after them for a total completion time of $1 + \delta + n\epsilon + n(\delta + \frac{n+1}{2}\epsilon)$. It is easy to see that there are only two combinatorially distinct values for α, $\alpha \leq \delta$ and $\alpha > \delta$ and we can restrict our attention to the two corresponding schedules and the probability with which they are chosen. Let $S1$ and $S2$ be the two schedules and $C1$ and $C2$ be their total completion times respectively. It is easy to see that $C1 = 1 + n(1 + (n+1)\epsilon/2)$ and $C2 = n(\delta + (n+1)\epsilon/2) + 1 + \delta + n\epsilon$.

1. If α is chosen uniformly at random, $S1$ is chosen with probability δ and $S2$ is chosen with probability $(1 - \delta)$ and a simple calculation shows that if we choose $n \gg 1$ and $1 \gg \delta \gg \epsilon$, the expected approximation ratio approaches 2.

2. Consider an instance I in which in addition to the jobs of $I(1/2, \epsilon, n)$ we release n more jobs of size ϵ at time $1 + n\epsilon$. The optimal preemptive schedule for I consists of the preemptive schedule for $I(1/2, \epsilon, n)$ followed by the additional n small jobs. The completion time of the optimal preemptive schedule is dominated by the term $3n/2$. An optimal non-preemptive schedule which runs the size 1 job last after the small jobs has a total completion time $3n/2 + 2$. It is easy to see that there are only two combinatorially distinct α-schedules one corresponding to $\alpha \leq 1/2$ and the other corresponding to $\alpha > 1/2$. In both cases it is easy to verify that the completion time of the schedule is $2n$ plus smaller order terms. Thus the approximation ratio of the Best-α cannot be better than $4/3$.

□

After learning of our results, Stougie and Vestjens [108] improved the lower bound for randomized on-line algorithms to $e/(e-1)$. This implies that our randomized on-line algorithm is *optimal*. Torng and Uthaisombut [110] have shown that for the Best-α algorithm, there are instances on which the approximation ratio is arbitrarily close to $\frac{e}{e-1}$. This implies that our analysis is tight up to lower order terms.

2.2.2 One-Machine Scheduling with Precedence Constraints

We now look at the case when jobs have precedence constraints between them and all release dates are 0. Let $G = (V, E)$ denote the precedence graph where V is the set of jobs. We will use jobs and vertices interchangeably. We say that i precedes j, denoted by $i \prec j$, if and only if there is a path from i to j in G. For any vertex $i \in V$, let G_i denote the subgraph of G induced by the set of vertices preceding i.

Definition 2.2.10 *The* rank *of a job J_i, denoted by q_i, is defined as $q_i = p_i/w_i$. Similarly, the* rank *of a set of jobs A denoted by $q(A)$ is defined as $q(A) = p(A)/w(A)$, where $p(A) = \sum_{J_i \in A} p_i$ and $w(A) = \sum_{J_i \in A} w_i$.*

Definition 2.2.11 *A subdag G' of G is said to be* precedence closed *if for every job $J_i \in G'$, G_i is a subgraph of G'.*

The rank of a graph is simply the rank of its node set.

Definition 2.2.12 *We define G^* to be a precedence-closed subgraph of G of minimum rank, i.e., among all precedence-closed subgraphs of G, G^* is of minimum rank.*

Note that G^* could be the entire graph G.

A Characterization of the Optimal Schedule

Smith's rule for a set of independent jobs states that there is an optimal schedule that schedules jobs in non-decreasing order of their ranks. We generalize this rule for the case of precedence constraints in a natural way. A version of the following theorem was proved by Sydney in 1975 [109] but we rediscovered it. We present our own proof for the sake of completeness.

Definition 2.2.13 *A* segment *in a schedule S is any set of jobs that are scheduled consecutively in S.*

Theorem 2.2.14 *There exists an optimal sequential schedule where the optimal schedule for G^* occurs as a segment that starts at time zero.*

Proof. The theorem is trivially true if G^* is the same as G. We consider the case when G^* is a proper subdag of G. Suppose the statement of the theorem is not true. Let S be some optimal schedule for G in which G^* does not occur as a segment that starts at time zero. For $k \geq 1$, let $A_1, A_2, \ldots, A_k$ be the segments of G^* in S, in increasing order of starting times. For $i \geq 2$ let the segment between A_{i-1} and A_i be denoted by B_i and let B_1

be the segment before A_1 that starts at time zero. For convenience we assume that B_1 is non-empty (we can always use a dummy segment with $p(B_1) = w(B_1) = 0$). Let $q(G^*) = \alpha$ and for $1 \leq j$ let B^j denote the union of the segments $B_1, B_2, \ldots, B_j$. For $1 \leq j$ it follows, from the definition of G^*, that $q(B^j) \geq \alpha$, for otherwise the union of B^j and G^* would be precedence closed and have rank less than α. Let A^j similarly denote the union of the segments $A_1, A_2, \ldots, A_j$. We also claim that $q(A^k - A^j) \leq \alpha$ for otherwise $q(A^j) < \alpha$.

Let S' be the schedule formed from S by moving all the A_i's ahead of B_j's while preserving their order within themselves. The schedule S' is legal since G^* is precedence closed. Let Δ denote the difference in the sum of weighted completion times of S and S'. We will show that $\Delta \geq 0$ which will complete the proof. While comparing the two schedules, we can ignore the contribution of the jobs that come after A_k since their status remains the same in S'. Let $\Delta(A_j)$ and $\Delta(B_j)$ denote the difference in weighted completion time of A_j and B_j respectively in S and S'. Therefore $\Delta = \sum_{j \leq k} \Delta(A_j) + \sum_{j \leq k} \Delta(B_j)$. It is easy to see that

$$\Delta(A_j) = w(A_j)p(B^j)$$

and

$$\Delta(B_j) = -w(B_j)p(A^k - A^{j-1}).$$

From our earlier observations on $q(B^j)$ and $q(A^k - A^j)$ we have $p(B^j) \geq \alpha\, w(B^j)$ and $p(A^k - A^j) \leq \alpha\, w(A^k - A^j)$. Therefore

$$\begin{aligned}
\Delta &= \sum_{j \leq k} \Delta(A_j) + \sum_{j \leq k} \Delta(B_j) \\
&= \sum_{j \leq k} w(A_j)p(B^j) - \sum_{j \leq k} w(B_j)p(A^k - A^{j-1}) \\
&\geq \alpha \sum_{j \leq k} w(A_j)w(B^j) - \alpha \sum_{j \leq k} w(B_j)w(A^k - A^{j-1}) \\
&= \alpha \sum_{j \leq k} w(A_j) \sum_{i \leq j} w(B_j) - \alpha \sum_{j \leq k} w(B_j) \sum_{i \geq j} w(A_i) \\
&= 0.
\end{aligned}$$

The third inequality above follows from our observations about $q(B^j)$ and $q(A - A^j)$ and the last equality follows from a simple change in the order of summation. $\square$

Note that when G^* is the same as G this theorem does not help in reducing the problem.

A 2 approximation

Theorem 2.2.14 suggests the following natural algorithm. Given G, compute G^* and schedule G^* and $G - G^*$ recursively. It is not apriori clear that G^* can be computed in polynomial time, however we will reduce this problem to that of computing a maximum flow in an associated graph. The second and more important problem that needs to be solved is to take care of the case when G^* is the same as G. We have to settle for an approximation in this case, for otherwise we would have a polynomial time algorithm for to compute the optimal schedule.

The following lemma establishes a strong lower bound on the optimal schedule value when $G^* = G$.

Lemma 2.2.15 *If G^* is the same as G, $\text{OPT} \geq w(G) \cdot p(G)/2$.*

Proof. Let $\alpha = q(G)$. Let S be an optimal schedule for G. Without loss of generality assume that the ordering of the jobs in S is $J_1, J_2, \ldots, J_n$. For any j, $1 \leq j \leq n$, observe that $C_j = \sum_{1 \leq i \leq j} p_i \geq \alpha \sum_{1 \leq i \leq j} w_i$. This is because the set of jobs $J_1, J_2, \ldots, J_j$ form a precedence closed subdag, and from our assumption on G^* it follows that $\sum_{i \leq j} p_j / \sum_{i \leq j} w_i \geq \alpha$. We bound the value of the optimal schedule as follows.

$$
\begin{aligned}
\text{OPT} &= \sum_j w_j C_j \\
&\geq \sum_j w_j \sum_{i \leq j} \alpha w_i \\
&= \alpha \left(\sum_j w_j^2 + \sum_{i<j} w_i w_j \right) \\
&= \alpha \left((\sum_j w_j)^2 - \sum_{i<j} w_i w_j \right) \\
&\geq \alpha \left(w(G)^2 - w(G)^2/2 \right) \\
&= \alpha w(G)^2/2 \\
&= w(G) p(G)/2
\end{aligned}
$$

The last equality is true because $q(G^*) = q(G) = p(G)/w(G) = \alpha$. □

Lemma 2.2.16 *Any feasible schedule with no idle time has a weighted completion time of at most $w(G) \cdot p(G)$.*

Proof. Obvious. □

Theorem 2.2.17 *If G^* for a graph can be computed in time $O(T(n))$, then there is a 2 approximation algorithm for computing the minimum weighted completion time schedule that runs in time $O(nT(n))$.*

Proof. Given G, we compute G^* in time $O(T(n))$. If G^* is the same as G we schedule G arbitrarily and Lemmas 2.2.15 and 2.2.16 guarantee that we have a 2 approximation. If G^* is a proper subdag we recurse on G^* and $G - G^*$. From Theorem 2.2.14 we have $\text{OPT}(G) = \text{OPT}(G^*) + p(G^*) \cdot w(G - G^*) + \text{OPT}(G - G^*)$. Inductively if we have 2 approximations for G^* and $G - G^*$ it is clear that we have a 2 approximation of the overall schedule. Now we establish the running time bound. We observe that $G^{**} = G^*$, therefore it suffices to recurse only on $G - G^*$. It follows that we make at most n calls to the routine to compute G^* and the bound follows. □

All that remains is to show how to compute G^* in polynomial time.

Computing G^*

An algorithm to compute G^* using a maximum flow computation is presented in Lawler's book [71]. We describe the algorithm, its proof, and some recent running time improvements for the sake of completeness. To compute G^* we consider the more general problem of finding a subdag of rank at most $\lambda > 0$, if one exists. We reduce the latter problem to the problem of computing an s-t mincut in an associated graph. The following defines the associated graph.

Definition 2.2.18 *Given a dag $G = (V, E)$, and a real number $\lambda > 0$, we define a capacitated directed graph $G_\lambda = (V \cup \{s,t\}, E', c)$ where the edge set E' is defined by $E' = \{(s,i),(i,t) \mid 1 \le i \le n\} \cup \{(i,j) \mid j \prec i\}$ and the capacities are defined by*

$$c(e) = \begin{cases} p_i & \text{if } e = (i,t) \\ \lambda w_i & \text{if } e = (s,i) \\ \infty & \text{otherwise} \end{cases}$$

Lemma 2.2.19 *Given a dag G, there is a subdag of rank less than or equal to λ if and only if the s-t mincut in G_λ is at most $\lambda w(G)$. If (A, B) is a cut whose value is bounded by $\lambda w(G)$, $q(A - \{s\}) \le \lambda$ and $A - \{s\}$ is precedence closed in G.*

Proof. Let (A, B) be an s-t cut in G_λ whose value is bounded by $\lambda w(G)$. We first claim that $A - \{s\}$ is precedence closed in G. Suppose not. Then there is a pair of vertices (i, j) such that $i \prec j$ and $j \in A$ and $i \notin A$. But then $c(j, i) = \infty$ which is a contradiction since $c(A, B) \leq \lambda w(G)$. From this fact and the definition of G_λ, it follows that

$$\begin{aligned} c(A, B) &= \sum_{i \in A} p_i + \sum_{i \notin A} \lambda w_i \\ &= \sum_{i \in A} (p_i - \lambda w_i) + \sum_{i \in V} \lambda w_i \\ &= \sum_{i \in A} (p_i - \lambda w_i) + \lambda w(G) \end{aligned}$$

Since $c(A, B) \leq \lambda w(G)$ it follows that $\sum_{i \in A}(p_i - \lambda w_i) \leq 0$ which implies that $q(A - \{s\}) \leq \lambda$. Using similar arguments as above, a precedence closed subdag A in G whose rank is less than λ induces a cut of value at most $\lambda w(G)$ in G_λ. □

Lemma 2.2.20 *G^* can be computed in strongly polynomial time $O(n^3)$, or in $O(n^{8/3} \log U)$ time where $U = \max_i(p_i + w_i)$.*

Proof. Computing the minimum cut in G_λ for each $\lambda > 0$ can be viewed as a parametric maxflow computation. There are at most n values of λ for which the minimum cut changes in the graph. Gallo et al. [30] show that all the distinct values of λ can be computed in asymptotic time equal to that needed for one maximum flow computation using the push-relabel algorithm. Goldberg and Tarjan's [41] push-relabel algorithm runs in $O(nm \log(n^2/m))$ time. Recently, Goldberg and Rao [40] improved the maximum flow running time to $O(\min\{n^{2/3}, m^{1/2}\} m \log(n^2/m) \log U)$ where U is the maximum capacity, and also showed that their bound applies for the parametric flow techniques of Gallo et al. [30]. The associated graph we construct has $\Omega(n^2)$ edges, therefore the claimed bounds follow. □

Integrality Gap of the Linear Ordering Relaxation

In this section we show that the linear ordering relaxation of Potts [90] has a factor 2 integrality gap. The gap also applies to the half integral formulation of Chudak and Hochbaum [16] which is a slight modification of the Potts's formulation, and the formulation of Queyranne and Wang [91]. We first describe the linear ordering relaxation of Potts. For each pair of jobs i and j there is a $\{0,1\}$-variable δ_{ij} that is 1, if i is scheduled before j, and 0 otherwise. Either i is scheduled before j or vice versa, therefore

$$\delta_{ij} + \delta_{ji} = 1, \qquad 1 \leq i < j \leq n. \tag{2.2}$$

Precedence constraints imply that

$$\delta_{ij} = 1, \qquad i \prec j. \tag{2.3}$$

Transitive relations in a feasible schedule are captured by the following set of inequalities. They state that if i is scheduled before j and j is scheduled before k, then i is scheduled before k.

$$1 + \delta_{ik} \geq \delta_{ij} + \delta_{jk}, \qquad 1 \leq i,j,k \leq n, i \neq j \neq k \neq i. \tag{2.4}$$

The completion time of job j, indicated by the variable C_j, is given by

$$C_j = p_j + \sum_{k \neq j} \delta_{kj}\, p_k, \qquad 1 \leq j \leq n. \tag{2.5}$$

The linear ordering relaxation is simply

$$\begin{gathered} \min \sum_j w_j C_j \\ \text{subject to (2.2)–(2.5)} \\ \delta_{ij} \in \{0,1\}. \end{gathered}$$

Potts showed that the above system of inequalities is a complete formulation for the single machine scheduling problem. We obtain a linear relaxation by replacing the integrality constraints on δ_{ij} by the following inequalities.

$$\delta_{ij} \geq 0, \qquad 1 \leq i \neq j \leq n. \tag{2.6}$$

Chudak and Hochbaum obtained a half integral formulation by replacing the transitive inequalities (2.4) by the following inequalities,

$$\delta_{ki} \leq \delta_{kj}, \qquad \text{if } i \prec j, k \neq j, k \neq i. \tag{2.7}$$

Now we give an instance of the scheduling problem for which the integrality gap of Potts's formulation is a factor of 2. We use a certain family of strongly expanding graphs. For n sufficiently large there exists an undirected bipartite graph $G = (L, R, E)$ such that the following three conditions hold.

- $|L| = |R| = n$.
- Every vertex in R has degree $n^{3/4}$, and every vertex in L has degree at most $3n^{3/4}$.
- Every subset of $n^{3/4}$ vertices in R has at least $n - n^{3/4}$ neighbors in L.

The existence of such a graph can be shown by the probabilistic method (see Problem 5.5 in [80]). Let the vertices in L be numbered 1 to n, and those in R, $(n+1)$ to $2n$. We construct an instance of the scheduling problem as follows. For each vertex i in G, we have a corresponding job J_i, for a total of $2n$ jobs. For each edge $(i, j) \in E$ where $i \in L$ and $j \in R$, we add the precedence constraint $i \prec j$. We set $p_i = 1$ for $1 \leq i \leq n$ and $p_i = 0$ for $(n+1) \leq i \leq 2n$. The weights are defined by $w_i = 1 - p_i$. Let I be the instance obtained. Two jobs i and j are unrelated, denoted by $i \prec\succ j$, if neither $i \prec j$ nor $j \prec i$ is true.

Lemma 2.2.21 *Setting $\delta_{ij} = 1$ if $i \prec j$ and $\delta_{ij} = 1/2$ if $i \prec\succ j$ gives a feasible solution to the linear ordering relaxation for I.*

Proof. It is easy to verify that equations (2.2)-(2.4) are satisfied by the stated assignment. $\square$

Lemma 2.2.22 *The optimum value of the linear ordering relaxation for I is at most $(n^2 + n^{7/4})/2$.*

Proof. We will show that the value of the relaxation for the feasible schedule obtained by setting the δ_{ij} as in Lemma 2.2.21 has a value at most $(n^2 + n^{7/4})/2$. We can ignore jobs $1, \ldots, n$ since their weights are 0. Consider a job $j > n$. We have

$$\begin{aligned}
C_j &= p_j + \sum_{k \neq j} \delta_{kj}\, p_k \\
&= \sum_{k \prec j, k \leq n} 1 + \sum_{k \prec\succ j, k \leq n} 1/2 \\
&= n^{3/4} + (n - n^{3/4})/2 \\
&= (n + n^{3/4})/2.
\end{aligned}$$

Summing up gives

$$\begin{aligned}\sum_j w_j C_j &= \sum_{j>n} C_j \\ &= \sum_{j>n} (n+n^{3/4})/2 \\ &= (n^2+n^{7/4})/2.\end{aligned}$$

This proves the lemma. □

Lemma 2.2.23 *Any valid schedule for I has a weighted completion time of at least $(n-n^{3/4})^2$.*

Proof. Let S be any valid schedule for I. Assume with out loss of generality that the jobs $(n+1)$ to $2n$ are ordered in increasing order of completion times in S. We claim that $C_{(n+n^{3/4})} \geq (n-n^{3/4})$. To prove this consider the jobs $(n+1)$ to $(n+n^{3/4})$. Let $A = \{i \mid i \prec j \text{ for some } (n+1) \leq j \leq (n+n^{3/4})\}$. A is the set of all predecessors of jobs $(n+1), \ldots, (n+n^{3/4})$. From the properties of the expander graph from which I was constructed, $|A| \geq (n-n^{3/4})$. Since each job in A is completed before $C_{(n+n^{3/4})}$ the claim follows. Therefore $C_j \geq (n-n^{3/4})$ for all $(n+n^{3/4}) \leq j \leq 2n$ and the lemma is proved. □

The following theorem follows from Lemmas 2.2.22 and 2.2.23.

Theorem 2.2.24 *The integrality gap of the linear ordering relaxation on I is $2-o(1)$.*

2.3 Scheduling on Parallel Machines

2.3.1 Parallel Machine Scheduling with Release Dates

We now turn to the problem of minimizing average completion time on parallel machines, in the presence of release dates. In this section, we give a 3-approximation algorithm for the problem. Our algorithm is simple and does not use linear programming, or slow dynamic programming. It introduces the notion of a one-machine preemptive relaxation. The algorithm is also on-line. We will show later in the chapter how to improve this to a 2.85-approximation algorithm, using more involved techniques.

Given an instance I for non-preemptive scheduling on m machines, we define a relaxation $I1$ in which we have processing times $p'_j = p_j/m$ and release dates $r'_j = r_j$. We call a relaxation *valid* if it maps feasible m-machine schedules for I to feasible one-machine schedules for $I1$ such that the average completion time of a schedule for $I1$ is no more than that of the corresponding schedule for I.

Lemma 2.3.1 *Instance $I1$ is a valid one-machine preemptive relaxation of the m-machine problem I.*

Proof. We will show how to convert a feasible schedule N, for input I, to a feasible schedule $P1$, for input $I1$, without increasing the average completion time. Take any schedule N and consider a particular time unit t. Let the $k \leq m$ jobs that are running during that time be $J_1, \ldots, J_k$. In $P1$, at time t, run $1/m$ units of each of jobs $J_1, \ldots, J_k$, in arbitrary order. The completion time of job J_j in $P1$, C_j^{P1} is clearly no greater than C_j^N, the completion time of J_j in N. □

Given a solution to $P1$, we form N by ordering jobs by C_j^{P1} and then scheduling them non-preemptively in that order, respecting release dates. Let C_j^* be the completion time of J_j in an optimal schedule for I. $I1$ may be a bad relaxation, in the sense that $\sum C_j^{I1}$ may be much less than $\sum C_j^*$. However, we can still use this relaxation to obtain a good non-preemptive schedule.

Lemma 2.3.2 $\sum_j C_j^N \leq (3 - \frac{1}{m}) \sum_j C_j^*$.

Proof. We first focus on a particular job J_j. For convenience, we assume that the jobs are ordered according to their completion times in $P1$. Thus J_j is the jth job to complete in $P1$. We now derive three lower bounds on C_j^{P1}. First, we have the trivial bound $C_j^{P1} \geq r'_j + p'_j$. Further, C_j^{P1} is at least as big as the processing times of the jobs that precede it. Therefore,

$$C_j^{P1} \geq \sum_{k=1}^{j} p'_k = \sum_{k=1}^{j} \frac{p_k}{m}. \tag{2.8}$$

Let $r_j^{\max} = \max_{1 \leq k \leq j} r'_k$ be the latest release date among jobs that complete before j; then, $C_j^{P1} \geq r_j^{\max}$.

Now consider the list schedule N. Clearly by time $r_j^{\max}$ all jobs $J_1, \ldots J_j$ have been released. Even if no job starts before time $r_j^{\max}$, by standard makespan arguments J_j will complete by

$$C_j^N \leq r_j^{\max} + \sum_{k=1}^{j-1} \frac{p_k}{m} + p_j$$

$$\leq \quad C_j^{P1} + C_j^{P1} + p_j\left(1 - \frac{1}{m}\right), \tag{2.9}$$

where the second inequality follows from (2.8) and $C_j^{P1} \geq r_j^{\max}$ above. Summing (2.9) over all jobs, we get a total completion time of

$$\sum_j C_j^N \leq 2\sum_j C_j^{P1} + \left(1 - \frac{1}{m}\right)\sum_j p_j. \tag{2.10}$$

By Lemma 2.3.1, $\sum_j C_j^{P1} \leq \sum_j C_j^*$, and trivially the optimal solution to I must have total completion time $\sum_j C_j^* \geq \sum p_j$, so this algorithm is a $(3 - \frac{1}{m})$-approximation.

□

The running time is just the time to run SRPT on the one machine relaxation and the time to list schedule, for a total of $O(n \log n)$. This algorithm can be made on-line by simulating the preemptive schedule and adding a job to the list when it completes in the preemptive schedule.

2.3.2 Precedence Constraints and a Generic Conversion Algorithm

We now give a very general conversion technique to obtain parallel machine schedules from single machine schedules. Our algorithm works in the presence of precedence constraints and release dates. Given an average weighted completion time scheduling problem, we will show that if we can approximate the one-machine preemptive variant, then we can also approximate the m-machine non-preemptive variant, with a slight degradation in the quality of approximation.

Precedence constraints will be represented in the usual way by a directed acyclic graph (DAG) whose vertices correspond to jobs, and whose edges represent precedence constraints.

In this section, we use a slightly different one-machine relaxation from the previous section, namely, we do not divide the processing times by m. We use the superscript m to denote the number of machines; thus, S^m denotes a schedule for m machines, C^m denotes the completion time of S^m, and C_j^m denotes the completion time of job J_j under schedule S^m. The subscript OPT refers to an optimal schedule; thus, an optimal schedule is denoted by S^m_{OPT}, and its weighted completion time is denoted by C^m_{OPT}. For a set of jobs A, $p(A)$ denotes the sum of processing times of jobs in A.

Definition 2.3.3 *For any vertex j, recursively define the quantity κ_j as follows. For a* *vertex j with no predecessors $\kappa_j = p_j + r_j$. Otherwise define $\kappa_j = p_j + \max\{\max_{i \prec j} \kappa_i, r_j\}$. Any path P_{ij} where $p(P_{ij}) = \kappa_j$ is referred to as a* critical path *to j.*

Conversion algorithm Delay List

We now describe the Delay List algorithm. Given a one-machine schedule which is a ρ-approximation, Delay List produces a schedule for $m \geq 2$ machines whose value is within a factor $(k_1\rho + k_2)$ of the optimal m-machine schedule, where k_1 and k_2 are small constants. We will describe a variant of this scheduling algorithm which yields $k_1 = (1+\beta)$ and $k_2 = (1 + 1/\beta)$ for any $\beta > 0$. Therefore, for cases where we can find optimal one-machine schedules (trees, series-parallel), we obtain a 4-approximation for m-machines by setting $\beta = 1$. To our knowledge, these are the best results for these special cases[2]

The main idea is as follows. The one-machine schedule taken as a list provides some priority information on which jobs to schedule earlier [3]. Unlike with makespan, the completion time of every job is important for weighted completion time. When trying to convert the one-machine schedule into an m-machine one, precedence constraints prevent complete parallelization. Thus, we may have to execute jobs out-of-order from the list to benefit from parallelism. If all p_i are identical (say 1), we can afford to use naive list scheduling [4]. If there is an idle machine and we schedule some available job on it, it is not going to delay jobs which become available soon, since it completes in one time unit. On the other hand if p_i's are different, a job could keep a machine busy, delaying more profitable jobs that become available soon. At the same time, we cannot afford to keep machines idle for too long if there are jobs ready to be scheduled. We strike a balance between the two extremes: schedule a job out-of-order only if there has been enough idleness already to justify scheduling it. To measure whether there has been enough idleness, we introduce a charging scheme.

Assume, for ease of exposition, that all processing times are integers and time is discrete. This restriction can be removed without much difficulty and we use it only in the interests of clarity and intuition. With this assumption we can use discrete time units. A job is *ready*

[2] Very recent work [82] has matched these results using LP relaxations.

[3] In the rest of the paper we assume without loss of generality that a *list* obeys the precedence constraints, that is, if $i \prec j$ then i comes earlier in the list than j.

[4] In this section, by *list scheduling* we mean the algorithm which schedules the first available job in the list if a machine is free. This is in contrast to another variant considered in earlier sections in which jobs are scheduled strictly in the order of the list.

if it has been released and all its predecessors have finished. Let $\beta > 0$ be some constant. At each time step t, the algorithm applies one of the following three cases:

1. *there is an idle machine M and the first job J_j in the list is ready at time t* — schedule J_j on M;

2. *there is an idle machine and the first job J_j in the list is not ready at t but there is another ready job on the list* — focusing on the job J_k which is the first in the list among the ready jobs, schedule it if there is at least βp_k *uncharged* idle time among all machines, and charge βp_k idle time to J_k;

3. *there is no idle time or the above two cases do not apply* — do not schedule any job, merely increment t.

Definition 2.3.4 *A job is said to be scheduled* in order *if it is scheduled when it is at the head of the list (that is using Case 1 in the algorithm). Otherwise it is said to be scheduled* out of order. *The set of jobs which are scheduled before a job J_i but which come later in the list than J_i is denoted by O_i. The set of jobs which come after J_i in the list is denoted by A_i and those which come before J_i by B_i (includes J_i).*

Definition 2.3.5 *The time at which job J_i is ready in a schedule S is denoted by q_i and time at which it starts is denoted by s_i.*

Definition 2.3.6 *For each job J_i, define a path $P_i' = J_{j_1}, J_{j_2}, \ldots, J_{j_\ell}$, with $J_{j_\ell} = J_i$ with respect to the schedule S^m as follows. The job J_{j_k} is the predecessor of $J_{j_{k+1}}$ with the largest completion time (in S^m) among all the predecessors of $J_{j_{k+1}}$ such that $C_{j_k}^m \geq r_{j_{k+1}}$; ties are broken arbitrarily. J_{j_1} is the job where this process terminates when there are no predecessors which satisfy the above condition. The jobs in P_i' define a disjoint set of time intervals $(0, r_{j_1}], (s_{j_1}^m, C_{j_1}^m], \ldots, (s_{j_\ell}^m, C_{j_\ell}^m]$ in the schedule. Let κ_i' denote the sum of the lengths of the intervals.*

The two facts below follow easily from the above definitions.

Fact 2.3.7 $\kappa_i' \leq \kappa_i$.

Fact 2.3.8 *The idle time charged to each job J_i is less than or equal to βp_i.*

A crucial feature of the algorithm is that when it schedules jobs, it considers only the first job in the list that is ready, even if there is enough idle time for other ready jobs that are later in the list.

Lemma 2.3.9 *For every job J_i, there is no uncharged idle time in the interval (q_i^m, s_i^m), and furthermore all the idle time is charged only to jobs in B_i.*

Proof. By the preceding remarks, it is clear that no job in A_i is scheduled in the time interval (q_i^m, s_i^m) since J_i was ready at q_i^m. From this we can conclude that there is is no idle time charged to jobs in A_i in that time interval. Since J_i is ready in that interval and was not scheduled, there cannot be any uncharged idle time.

□

The following lemma shows that for any job J_i, the algorithm does not schedule too many jobs from A_i before scheduling J_i itself.

Lemma 2.3.10 *For every job J_i, the total idle time charged to jobs in A_i, in the interval $(0, s_i^m)$, is bounded by $(\kappa_i' - p_i)m$. It follows that $p(O_i) \leq (\kappa_i' - p_i)m/\beta \leq (\kappa_i - p_i)m/\beta$.*

Proof. Consider a job J_{j_k} in P_i'. The job $J_{j_{k+1}}$ is ready to be scheduled at the completion of J_{j_k}, that is $q_{i_{k+1}}^m = C_{i_k}^m$. From Lemma 2.3.9, it follows that in the time interval between $(C_{j_k}^m, s_{j_{k+1}}^m)$ there is no idle time charged to jobs in $A_{j_{k+1}}$. Since $A_{j_{k+1}} \supset A_i$ it follows that all the idle time for jobs in A_i has to be accumulated in the intersection between $(0, s_i^m)$ and the time intervals defined by P_i'. This quantity is clearly bounded by $(\kappa_i' - p_i)m$. The second part follows since the total processing time of the jobs in O_i is bounded by $1/\beta$ times the total idle time that can be charged to jobs in A_i (recall that $O_i \subseteq A_i$). □

Theorem 2.3.11 *Let S^m be the schedule produced by the algorithm* Delay List *using a list S^1. Then for each job J_i, $C_i^m \leq (1+\beta)p(B_i)/m + (1+1/\beta)\kappa_i' - p_i/\beta$.*

Proof. Consider a job J_i. We can split the time interval $(0, C_i^m)$ into two disjoint sets of time intervals T_1 and T_2 as follows. The set T_1 consists of all the disjoint time intervals defined by P_i'. The set T_2 consists of the time intervals obtained by removing the intervals in T_1 from $(0, C_i^m)$. Let t_1 and t_2 be the sum of the times of the intervals in T_1 and T_2 respectively. From the definition of T_1, it follows that $t_1 = \kappa_i' \leq \kappa_i$. From Lemma 2.3.9, in the time intervals of T_2, all the idle time is either charged to jobs in B_i and, the only jobs which run are from $B_i \cup O_i$. From Fact 2.3.8, the idle time charged to jobs in B_i bounded by $\beta p(B_i)$. Therefore the time t_2 is bounded by $(\beta p(B_i) + p(B_i) + p(O_i))/m$. Using Lemma 2.3.10 we see that $t_1 + t_2$ is bounded by $(1+\beta)p(B_i)/m + (1+1/\beta)\kappa_i^{'} - p_i/\beta$. □

Remark 2.3.12 *In the above theorem the only requirement is that S^1 be a list that obeyed the precedence constraints. In particular we can ignore release dates in computing the one machine schedule.*

One Machine Relaxation

In order to use Delay List, we will need to start with a one machine schedule. The following two lemmas provide lower bounds on the optimal m-machine schedule in terms of the optimal one-machine schedule. This one-machine schedule can be either preemptive or non-preemptive, the bounds hold in either case.

Lemma 2.3.13 $C^m_{\mathrm{OPT}} \geq C^1_{\mathrm{OPT}}/m$.

Proof. Given a schedule S^m on m machines with sum of weighted completion times C^m, we will construct a one-machine schedule S^1 with sum of weighted completion times at most mC^m as follows. Order the jobs according to their completion times in S^m with the jobs completing early coming earlier in the ordering. This ordering is our schedule S^1. Note that there could be idle time in the schedule due to release dates. If $i \prec j$ then $C_i^m \leq s_j^m < C_j^m$ (assuming that $p_j > 0$) which implies that there will be no precedence violations in S^1. We claim that $C_i^1 \leq mC_i^m$ for every job J_i. Let P be the sum of the processing times of all the jobs which finish before C_i^m (including J_i) in S^m. Let I be the total idle time in the schedule S^m before C_i^m. It is easy to see that $mC_i^m \geq P + I$. We claim that $C_i^1 \leq P + I$. The idle time in the schedule S^1 can be charged to idle time in the schedule S^m and P is the sum of all jobs which come before J_i in S^1. This implies the desired result. □

Lemma 2.3.14 $C^m_{\mathrm{OPT}} \geq \sum_i w_i \kappa_i = C^\infty_{\mathrm{OPT}}$.

Proof. The length of the critical path κ_i, is an obvious lower bound on the completion time C_i^m of job J_i. Summing up over all jobs gives the first inequality. It is also easy to see that if the number of machines is unbounded that every job J_i can be scheduled at the earliest time it is available and will finish by κ_i. Thus we obtain the equality. □

Obtaining generic m-machine schedules

In this section we derive our main theorem relating m-machine schedules to one-machine schedules.

We begin with a corollary to Theorem 2.3.11.

Corollary 2.3.15 *Let S^m be the schedule produced by the algorithm* Delay List *using a one-machine schedule S^1 as the list. Then for each job J_i, $C_i^m \le (1+\beta)C_i^1/m + (1+1/\beta)\kappa_i$.*

Proof. Since all jobs in B_i come before J_i in the one-machine schedule, it follows that $p(B_i) \le C_i^1$. Plugging this and Fact 2.3.7 into the bound in Theorem 2.3.11, we conclude that $C_i^m \le (1+\beta)C_i^1/m + (1+1/\beta)\kappa_i$. □

Theorem 2.3.16 *Given a one-machine schedule that is within a factor ρ of an optimal one-machine schedule,* Delay List *gives a m-machine schedule that is within a factor $(1+\beta)\rho + (1+1/\beta)$ of an optimal m-machine schedule.*

Proof. Let S^1 be a schedule which is within a factor ρ of the optimal one-machine schedule. Then $C^1 = \sum_i w_i C_i^1 \le \rho C^1_{\mathrm{OPT}}$. By Theorem 2.3.11, the schedule created by the algorithm Delay List satisfies,

$$\begin{aligned} C^m &= \sum_i w_i C_i^m \\ &\le \sum_i w_i \left((1+\beta)\frac{C_i^1}{m} + \left(1 + \frac{1}{\beta}\right)\kappa_i \right) \\ &\le \frac{1+\beta}{m}\sum_i w_i C_i^1 + \left(1+\frac{1}{\beta}\right)\sum_i w_i \kappa_i. \end{aligned}$$

From Lemmas 2.3.13 and 2.3.14 it follows that

$$\begin{aligned} C^m &\le \frac{(1+\beta)\rho\, C^1_{\mathrm{OPT}}}{m} + \left(1+\frac{1}{\beta}\right) C^\infty_{\mathrm{OPT}} \\ &\le \left((1+\beta)\rho + \left(1+\frac{1}{\beta}\right)\right) C^m_{\mathrm{OPT}}. \end{aligned}$$

□

Corollary 2.3.17 *There is a simple $O(n\log n)$ time 4-approximation algorithm for sum of weighted completion times on parallel machines when the precedence graphs are restricted to be trees or series-parallel graphs and the jobs have release dates.*

Proof. Applying the DELAY LIST algorithm with $\beta = 1$ to the optimal single machine schedule that ignores release dates (which can be computed in $O(n \log n)$ time [1]) gives the desired result. By Remark 2.3.12 we can ignore the release dates in computing the single machine schedule. $\square$

Remark 2.3.18 *Since the bounds in our conversion algorithm are job-by-job, the algorithm is applicable to other objective functions that are linear combinations of functions of individual completion times.*

Applying conversion to in-tree precedence

We obtain stronger results for in-tree precedence *without* release dates. This restricted problem is already strongly NP-Hard. We analyze the standard *list scheduling* algorithm which starts with an ordering on the jobs (the list), and greedily schedules each successive job in the list at the earliest possible time. We use the optimal one-machine schedule as the list. We show that this algorithm gives a 2-approximation for in-trees. Recall that s_i^m is the start time of J_i in the schedule S^m.

Lemma 2.3.19 *If S^m is the list schedule using a one-machine schedule S^1 as the list, then for any job J_i, $C_i^m \leq \kappa_i + C_i^1/m$.*

Proof. Since there are no release dates, we can assume that the schedule S^1 has no idle time. Without loss of generality assume that $J_1, \ldots, J_n$ is the ordering of the jobs ordered according to their *start* times s_i^m in S^m (we break ties arbitrarily). We will prove the lemma by induction on i. We strengthen the hypothesis by adding the following invariant. If $C_i^m > C_j^m + p_i$ where J_j is the last predecessor of J_i to finish in S^m, then all the jobs scheduled before s_i^m in S^m are ahead of J_i in the list S^1 and there is no idle time in the schedule, before time s_i^m. In this case it follows that $C_i^m \leq p_i + C_i^1/m$. The base case is trivial since $\kappa_1 = p_1$ and the first job finishes at time p_1. Suppose that the hypothesis holds for all jobs J_k, $k < i$, we will prove it holds for J_i. Let J_j, $j < i$ be the last predecessor of J_i to finish in the schedule S^m. We consider two cases.

1. $C_i^m = C_j^m + p_i$. By the hypothesis, $C_j^m \leq \kappa_j + C_j^1/m$. It follows that $C_i^m \leq \kappa_i + C_i^1/m$ since $\kappa_i \geq \kappa_j + p_i$ and $C_j^1 < C_i^1$.

2. $C_i^m > C_j^m + p_i$. Let $t = C_j^m$. Let P be the set of jobs which finish exactly at time t and P' be the set of jobs which had their last predecessor running until time t. Note

that $J_i \in P'$, $J_j \in P$, and all the jobs in P' are ready to be run at time t. In an in-tree a node has at most one immediate successor therefore $|P'| \leq |P|$. Therefore the number of jobs that are ready at t but were not ready at t^- is at most $|P|$. If J_i was not scheduled at t there must exist a job $J_l \notin P'$ which is scheduled at t. This implies that J_l occurs before J_i in the list S^1. Since no immediate predecessor of J_l finished at t, by the induction hypothesis we conclude that there was no idle time and no job which comes later than J_l in S^1 is scheduled before time t. Since J_i was ready at time t, it follows that there is no idle time and no job later than J_i in S^1 is scheduled between time t and the s_i^m. From these observations it follows that $C_i^m \leq p_i + C_i^1/m \leq \kappa_i + C_i^1/m$.

In both cases we see that the induction hypothesis is established for J_i and this finishes the proof. □

Theorem 2.3.20 *There is an $O(n \log n)$-time algorithm with approximation ratio 2 for minimizing sum of weighted completion times on m machines for in-tree precedence without release dates.*

Proof. The proof is similar to that of Theorem 2.3.16 except that we use the stronger bounds from Lemma 2.3.19. The running time is dominated by the time to compute the optimal one machine schedule which can be done in $O(n \log n)$ time [1]. □

A 2.85-approximation for scheduling without precedence constraints

We now improve the approximation bound for parallel machine scheduling with release dates to 2.85 using the ideas developed in this section, which improves the the earlier ratio of $2.89+\epsilon$ [9]. We combine the ideas of the one machine relaxation developed in Section 2.3.1 and the idea of using delay based list scheduling to derive an alternate algorithm which has worse ratio than the algorithm in Section 2.3.1. But we observe that the bounds we get from the analysis of these two algorithms can be combined to get an improved lower bound on the optimal which leads to the improvement.

Lemma 2.3.21 *If we apply* DELAY LIST *to P1 with parameter β, the resulting schedule D has total completion time*

$$\sum C_j^D \leq (2+\beta)C_j^* + \frac{1}{\beta}\sum_j r_j.$$

Proof. We focus on a particular job J_j. From Theorem 2.3.11 we conclude that $C_j^D \leq (1+\beta)t(B_j)/m + (1+1/\beta)\kappa_j - p_j/\beta$. Since we do not have precedence constraints on the jobs, $\kappa_j = r_j + p_j$. From the definition of B_j and the fact that the list is the order in which jobs finish in $P1$, it follows that $t(B_i)/m \leq C_j^{P1}$. We therefore conclude that $C_j^D \leq (1+\beta)C_j^{P1} + p_j + r_j/\beta$. Summing this over all jobs we obtain

$$\sum_j C_j^D \leq (1+\beta)\sum_j C_j^{P1} + \sum_j p_j + \frac{1}{\beta}\sum_j r_j$$

Since both $\sum_j C_j^{P1}$ and $\sum_j p_j$ are lower bounds on the optimal schedule value, it follows that

$$\sum_j C_j^D \leq (2+\beta)\sum_j C_j^* + \frac{1}{\beta}\sum_j r_j.$$

□

We now show that we can balance the two algorithms, list scheduling and Delay List to achieve an approximation ratio of 2.85.

Lemma 2.3.22 *For any input I, either list scheduling from $P1$ or using* Delay List *on $P1$ produces a schedule with $\sum_j C_j \leq 2.85\sum_j C_j^*$ and runs in $O(n\log n)$ time.*

Proof. By (2.10), we know that

$$\sum_j C_j^N \leq 2\sum_j C_j^{P1} + \sum_j p_j. \tag{2.11}$$

If, for some α, we know that $\sum_j p_j \leq \alpha\sum_j C_j^*$, then the list scheduling algorithm is a $2+\alpha$ approximation algorithm.

Now consider the case when $\sum_j p_j > \alpha\sum_j C_j^*$. If we combine this equation with the simple bound that $\sum_j C_j^* \geq \sum_j (p_j + r_j)$, we get that $\sum_j p_j \geq \alpha\sum_j (p_j + r_j)$, which implies that

$$\sum_j r_j \leq \frac{1-\alpha}{\alpha}\sum_j p_j \leq \frac{1-\alpha}{\alpha}\sum C_j^*. \tag{2.12}$$

We can now plug (2.12) into the upper bound on C_j^D from Lemma 2.3.21 to get

$$\sum_j C_j^D \leq \left(2 + \beta + \frac{1-\alpha}{\alpha\beta}\right)\sum_j C_j^*.$$

We do not know the value of α, but for each possible α we can choose the β that minimizes the two terms. Simple algebra and calculus shows that given α, we can choose β to be $\sqrt{\frac{1-\alpha}{\alpha}}$. Thus, trying all possible α yields the result.

We can obtain a running time of $O(n \log n)$ by observing that the analysis of the 3-approximation algorithm allows us to estimate C_j^* to within a factor 3. We can then perform a binary search on α.

□

2.4 Concluding Remarks

We considered several scheduling variants with the goal of minimizing sum of completion times and provided new techniques and algorithms. Subsequent to our work improved results [100, 99, 82] for some of the problems considered here have been obtained. Much of the work on this topic has focused on obtaining improved approximation algorithms but not much is known on the hardness of approximation of several variants. Recent work by Hoogeveen et al. [59] shows APX hardness for some parallel machine variants. The results in [59] are based on the hardness of approximation results that are known for minimizing makespan. However it is surprising that while minimizing sum of completion times seems harder than minimizing makespan (the approximation ratios are typically larger than those for the makespan counterparts), the hardness of approximation results are weaker.

The techniques in [59] do not apply to the one machine problems. For the problem of minimizing average completion time on a single machine with release dates we conjecture the following.

Conjecture 2.4.1 *There is a PTAS for non-preemptive scheduling to minimize average completion time on a single machine with jobs having release dates and no precedence constraints.*

The weighted version seems harder since even the preemptive case in NP-Hard.

For single machine scheduling with precedence constraints, several different LP formulations and algorithms give a tight 2 approximation. An obvious open problem is to obtain an algorithm with an improved approximation ratio. We have the following conjecture regarding the hardness of approximation for this problem.

Conjecture 2.4.2 *The problem of minimizing sum of weighted completion times on a single machine when jobs have precedence constraints is APX-Complete.*

Our results in this chapter might help in addressing the above questions. Theorem 2.2.14 shows that it is sufficient to improve the approximation ratio for the case when $G^* = G$. This is a more structured and smaller set of instances to look at. In Section 2.2.2 we used expanders to construct specific instances with a factor of 2 integrality gap for the linear ordering relaxation of Potts [90]. We believe that understanding the role of expanders in the instances created as above is crucial for both improving the approximation ratio and proving hardness of approximation results.

Recent work of Munier et al. [82] has improved the approximation ratio for parallel machine scheduling with precedence constraints and release dates form 5.33 [9] to 4. The improved algorithm uses list scheduling based on the ordering provided by a linear programming relaxation. The interesting aspect of the algorithm is that the list scheduling is based strictly on the ordering provided by the LP while we use artificial delays in our conversion algorithm. Using LP relaxations directly for the parallel machine problem gives stronger results than that are obtained by us using Delay List. However our algorithm is more efficient. We also believe that Delay List and its analysis can be improved to obtain improved results. Munier et al. [82] show that their algorithm gives the same ratio even when there are *delays* between jobs. A delay d_{ij} between a pair of jobs i and j constrains job j to start d_{ij} units after the completion of i. Delay constraints can be thought of latencies on the precedence edges. We remark that our conversion algorithm Delay List also gives the same approximation ratio even in the presence of delays. The modifications are straightforward.

Chapter 3

Makespan on Machines with Different Speeds

3.1 Introduction

In this chapter we[1] consider the problem of scheduling precedence constrained jobs on machines that have different speeds. We formalize the problem below. We are given a set of n jobs $1, \ldots, n$, with job j requiring p_j units of processing time. The jobs are to be scheduled on a set of m machines. Machine i has a speed factor s_i. Job j with a processing requirement p_j takes p_j/s_i time units to run on machine i. In the scheduling literature such machines are called *uniformly related* and the problem we consider is referred to as $Q|prec|C_{\max}$ [44]. Let C_j denote the completion time of job j. The objective is to find a schedule to minimize $C_{\max} = \max_j C_j$, the makespan of the schedule. We restrict ourselves to non-preemptive schedules where a job once started on a machine has to run to completion on the same machine. Our results carry over to the preemptive case as well.

Liu and Liu [77] analyzed the performance of Graham's list scheduling algorithm for the case of different speeds and showed that it has an approximation ratio of $(1+\max_i s_i/\min_i s_i - \max_i s_i/\sum_i s_i)$. This ratio depends on the ratio of the largest to the smallest speed and could be arbitrarily large even for a small number of machines. The first algorithm to have an approximation ratio independent of the speeds was given by Jaffe [62]. By generalizing the analysis of Liu and Liu he showed that list scheduling restricted to the set of machines

[1]Parts of this chapter are joint work with Michael Bender and appeared in [10].

with speeds that are within a factor of $1/\sqrt{m}$ of the fastest machine speed results in an $O(\sqrt{m})$ approximation ratio. More recently, Chudak and Shymoys [17] improved the ratio considerably and gave an algorithm with an approximation ratio of $O(\log m)$. At a more basic level their algorithm has an approximation ratio of $O(K)$ where K is the number of *distinct* speeds. Their algorithm relies on solving a linear programming relaxation and uses the information obtained from the solution to allocate jobs to machines. We obtain a new algorithm that finds an allocation without solving a linear program. The approximation ratio of our algorithm is also $O(\log m)$ but is advantageous for the following reason. Our algorithm runs in $O(n^3)$ time and is combinatorial, hence is more efficient than the algorithm in [17]. Further, the analysis of our algorithm relies on a new lower bound which is very natural, and might be useful in other contexts. In addition we show that our algorithm achieves a constant factor approximation when the precedence constraints are induced by a collection of chains. We also obtain a similar ratio even if jobs have release dates by the result in [103]. Our work uses several of the basic ideas from [17]. A linear programming relaxation that gives an $O(\log m)$ approximation ratio for the more general problem of minimizing sum of weighted completion times ($Q|prec|\sum_j w_j C_j$) is also presented in[17]. Our ideas do not generalize for that problem.

The rest of this chapter is organized as follows. Section 3.2 contains ideas from the paper of Chudak and Shmoys [17] that are useful to us. We present our lower bound in Section 3.3, and give the approximation algorithm and its analysis in Section 3.4.

3.2 Preliminaries

We summarize below the basic ideas in the work of Chudak and Shmoys [17]. Their main result is an algorithm that gives an approximation ratio of $O(K)$ for the problem of $Q|prec|C_{\max}$ where K is the number of distinct speeds. They also show how to reduce the general case with arbitrary speeds to one in which there are only $O(\log m)$ distinct speeds, as follows.

- Ignore all machines with speed less than $1/m$ times the speed of the fastest machine.
- Round down all speeds to the nearest power of 2.

They observe that the above transformation can be done while losing only a constant factor in the approximation ratio. We will therefore restrict ourselves to instances with K distinct speeds with the implicit assumption that $K = O(\log m)$. The above ideas also imply an $O(\log(\max_i s_i / \min_i s_i))$ approximation ratio instead of an $O(\log m)$ ratio. For some instances this is a large improvement.

When all machines have the same speed ($K = 1$), Graham [42] showed that list scheduling gives a 2 approximation. His analysis shows that in any schedule produced by list scheduling, we can identify a chain of jobs $j_1 \prec j_2 \ldots \prec j_r$ such that a machine is idle only when one of the jobs in the above chain is being processed. The time spent processing the chain is clearly a lower bound for the optimal makespan. In addition, the sum total of time intervals during which all machines are busy is also a lower bound via arguments about the average load. These two bounds provide an upper bound of 2 on the approximation ratio of list scheduling. One can apply a similar analysis for the multiple speed case. As observed in [17], the difficulty is that the time spent in processing the chain, identified from the list scheduling analysis, is not a lower bound for the optimal makespan value. The only claim that can be made is that the processing time of any chain on the fastest machine is a lower bound. However the jobs in the chain guaranteed by the list scheduling analysis do not necessarily run on the fastest machine. Based on this observation, the algorithm in [17] tries to find an assignment of jobs to speeds (machines) that ensures that the processing time of any chain is bounded by some factor of the optimal.

We will follow the notation of [17] for sake of continuity and convenience. Recall that we have K distinct speeds. Let m_k be the number of machines with speed $\overline{s}_k$, $k = 1, \ldots, K$, where $\overline{s}_1 > \ldots > \overline{s}_K$. Let M_u^v denote the sum $\sum_{k=u}^{v} m_k$. In the sequel we will be interested in assigning jobs to speeds. For a given assignment, let $k(j)$ denote the speed at which job j is assigned to be processed. The average processing allocated to a machine of a specific speed k, denoted by D_k, is the following.

$$D_k = \frac{1}{m_k \overline{s}_k} \sum_{j:k(j)=k} p_j.$$

A chain is simply a subset of jobs that are totally ordered by the precedence constraints. Let $\mathcal{P}$ be the set of all chains induced by the precedence constraints. For a given job assignment we can compute a quantity C defined by the following equation.

$$C = \max_{P \in \mathcal{P}} \sum_{j \in P} \frac{p_j}{\overline{s}_{k(j)}}$$

For a chain P the quantity $\sum_{j \in P} \frac{p_j}{\overline{s}_{k(j)}}$ denotes the minimum time required to finish the jobs in the chain if they are scheduled according to the given assignment of jobs to speeds. Thus C represents a lower bound on the makespan for the given assignment of jobs to speeds.

A natural variant of list scheduling called speed based list scheduling developed in [17] is constrained to schedule according to the speed assignments of the jobs. In classical list scheduling, the first available job from the list is scheduled as soon as a machine is free. In speed based list scheduling, an available job is scheduled on a free machine provided the speed of free machine matches the speed assignment of the job. The proof of the following theorem follows from a simple generalization of Graham's analysis of list scheduling.

Theorem 3.2.1 (Chudak & Shmoys [17]) *For any job assignment $k(j)$, $j = 1, \ldots, n$, the speed-based list scheduling algorithm produces a schedule of length*

$$C_{\max} \leq C + \sum_{k=1}^{K} D_k.$$

In [17] a linear programming relaxation of the problem is used to obtain a job assignment that satisfies the following two conditions: $\sum_{k=1}^{K} D_k \leq (K + \sqrt{K})C^*_{\max}$, and $C \leq (\sqrt{K} + 1)C^*_{\max}$, where $C^*_{\max}$ is the optimal makespan. Plugging these in Theorem 3.2.1 results in an $O(K)$ approximation ratio. We use an alternative method based on chain decompositions to obtain an assignment satisfying similar properties.

3.3 A New Lower Bound

In this section we develop a simple and natural lower bound that will be used in the analysis of our algorithm. Before formally stating the lower bound we provide some intuition. The two lower bounds used in Graham's analysis for identical parallel machines are the maximum chain length (a chain's length is sum of processing times of jobs in it) and the average load. As discussed in the previous section, a naive generalization of the first lower bound implies that the maximum chain length divided by the fastest speed is a lower bound. However it is easy to generate examples where the maximum of this bound and the average load is $O(1/m)$ times the optimal. We describe the general nature of such examples to motivate our new bound. Suppose we have two speeds with $\overline{s}_1 = D$ and $\overline{s}_2 = 1$. The precedence constraints between the jobs are induced by a collection of $\ell > 1$ chains, each of the same length D. Suppose $m_1 = 1$, and $m_2 = \ell \cdot D$. The average load for the instance is upper bounded by 1. In addition the time to process any chain on the fastest machine is 1. However if $D \gg \ell$ it is easy to see that the optimal is $\Omega(\ell)$ since *only* ℓ machines can be busy at any time instant. The key insight we obtain from the above example is that the amount of parallelism in an instance restricts the number of machines that can be used. We capture this insight in our lower bound in a simple way. We need a few definitions to formalize the intuition. We view the precedence relations between the jobs as a weighted poset where each element of the poset has a weight associated with it that is the same as the processing time of the associated job. We will further assume that we have the transitive closure of the poset.

Definition 3.3.1 *A* chain *P is a set of jobs $j_1, \ldots, j_r$ such that for all $1 \leq i < r$, $j_i \prec j_{i+1}$. The* length *of a chain P, denoted by $|P|$, is the sum of the processing times of the jobs in P.*

Definition 3.3.2 *A* chain decomposition *$\mathcal{P}$ of a set of precedence constrained jobs is a partition of the poset into a collection of chains $\{P_1, P_2, \ldots, P_r\}$. A* maximal chain decomposition *is one in which P_1 is a longest chain and $\{P_2, \ldots, P_r\}$ is a maximal chain decomposition of the poset with elements of P_1 removed.*

Though we define a maximal chain decomposition as a set of chains, we will implicitly assume that it is an *ordered* set, that is $|P_1| \geq |P_2| \geq \ldots |P_r|$.

Definition 3.3.3 *Let $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ be a maximal chain decomposition of the given set of jobs. We define a quantity called $L_{\mathcal{P}}$ associated with $\mathcal{P}$ and the machine speeds as follows.*

$$L_{\mathcal{P}} = \max_{1 \leq j \leq \min\{r,m\}} \frac{\sum_{i=1}^{j} |P_i|}{\sum_{i=1}^{j} s_i}.$$

Note that in Definition 3.3.3 the index of summation is over the machines and not the speed classes. With the above definitions in place we are ready to state and prove the new lower bound.

Theorem 3.3.4 *Let $\mathcal{P} = \{P_1, \ldots, P_r\}$ be any chain decomposition (in particular any* maximal *chain decomposition) of the precedence graph of the jobs. Let $AL = (\sum_{j=1}^{n} p_i)/(\sum_{i=1}^{m} s_i)$ denote the average load. Then*

$$C^*_{\max} \geq \max\{AL, L_{\mathcal{P}}\}.$$

Moreover the lower bound is valid for the preemptive case as well.

Proof. It is easy to observe that $C^*_{\max} \geq AL$. We will show the following for $1 \leq j \leq m$

$$C^*_{\max} \geq \frac{\sum_{i=1}^{j} |P_i|}{\sum_{i=1}^{j} s_i}$$

which will prove the theorem. Consider the first j chains. Suppose our input instance was modified to have only the jobs in the first j chains. It is easy to see that a lower bound for this modified instance is a lower bound for the original instance. Since it is possible to execute only one job from each chain at any time instant, only the fastest j machines are relevant for this modified instance. The expression $(\sum_{i=1}^{j} |P_i|)/(\sum_{i=1}^{j} s_i)$ is simply the average load for the modified instance, which as we observed before, is a lower bound. Since the average load is also a lower bound for the preemptive case, the claimed lower bound applies even if preemptions are allowed. □

Horvath, Lam, and Sethi [61] proved that the above lower bound gives the optimal schedule length for preemptive scheduling of chains on uniformly related machines. The idea of extending their lower bound to general precedence graphs using maximal chain decompositions is natural but does not appear to have been effectively used before.

Theorem 3.3.5 *A maximal chain decomposition can be computed in $O(n^3)$ time. If all p_j are the same, the running time can be improved to $O(n^2\sqrt{n})$.*

Proof. It is necessary to find the transitive closure of the given graph of precedence constraints. This can be done in $O(n^3)$ time using a BFS from each vertex. From a theoretical point of view this can be improved to $O(n^\omega)$ where $\omega \leq 2.376$ using fast matrix multiplication [18]. A longest chain in a weighted DAG can be found in $O(n^2)$ time using standard algorithms. Using this at most n times, a maximal chain decomposition can be obtained in $O(n^3)$ time. If all p_j are the same (without loss of generality we can assume they are all 1), the length of a chain is the same as the number of vertices in the chain. We can use this additional structure to obtain an improved time bound as follows. We remove longest chains using the $O(n^2)$ algorithm as long as the longest chain is at least $\sqrt{n}$. The total running time for this phase of the algorithm is clearly bounded by $O(n^2\sqrt{n})$. Once the length of the longest chain falls below $\sqrt{n}$ we run a different algorithm that is outlined in the proof of Lemma 3.3.6 below. That algorithm yields computes a maximal chain decomposition for unit weight jobs in $O(n^2 \cdot d)$ time where d is the maximum chain length. Thus if $d \leq \sqrt{n}$ we obtain an $O(n^2\sqrt{n})$ time for the second phase. Combining the two phases gives an algorithm with the desired bound. □

Lemma 3.3.6 *Given a DAG with all $p_j = 1$, and the longest chain length bounded by d, there is an algorithm to compute a maximal chain decomposition in time $O(n^2 \cdot d)$.*

Proof. We assume we have the transitive closure of the DAG. We partition the vertices into $(d+1)$ layers $L_0, L_1, \ldots, L_d$. Layer L_i is the set of all vertices v such that the longest chain ending at v is of length i. Let $\ell(v)$ denote the layer of a vertex v. For each vertex v we maintain its predecessors in $(d+1)$ classes corresponding to the layer to which they belong. Given the transitive closure it is easy to construct this partition and the predecessor classes in $O(n^2)$ time. Given a layered representation we can find a longest chain in $O(d)$ time by taking an arbitrary vertex in the highest numbered non-empty layer and walking down the layers looking for predecessors. Once we find a longest chain we remove the vertices in the chain and all the edges incident on them. We update the layered data structure and repeat the process. The update happens as follows. For each edge removed we change the predecessor structure of the vertex incident on it. Let S be the vertices that are incident on the edges removed. We examine vertices in S in increasing order of $\ell(v)$. We first update $\ell(v)$ to its new value. If $\ell(v)$ does not change by the removal of the chain we remove v from S. If $\ell(v)$ is reduced, we examine all the successors of v, update their predecessor data structure and add them to S. This takes time proportional to the out-degree of v. We continue this process as long as S is non-empty. We analyze the running time as follows. The time to find the chain, remove the vertices and the associated edges, and form the initial set S can be amortized to total number of edges removed. Thus this time is bounded over all by $O(n^2)$. The time to update the layer information is amortized as follows. We examine the successors of a vertex v only if $\ell(v)$ is reduced. Since $\ell(v)$ can change at most $(d+1)$ times the total time is bounded by $(d+1)\sum_v \deg(v)$ which is $O(n^2 \cdot d)$. □

3.4 The Approximation Algorithm

The approximation algorithm we develop in this section is based on the maximal chain decompositions defined in the previous section. As mentioned in Section 3.2, our algorithm produces an assignment of jobs to speeds. Then we use the speed based list scheduling of [17] with the job assignment produced by our algorithm.

1. **compute** a maximal chain decomposition of the jobs $\mathcal{P} = \{P_1, \ldots, P_r\}$.
2. **set** $B = \max\{AL, L_{\mathcal{P}}\}$.
3. **set** $\ell = 1$.
4. **foreach** speed $1 \leq i \leq K$ **do**
   (a) **let** $t \leq r$ be the maximum index such that $\sum_{\ell \leq j \leq t} |P_j|/(m_i s_i) \leq 4B$.
   (b) **assign** jobs in chains $P_\ell, \ldots, P_t$ to speed i.
   (c) **set** $\ell = t + 1$. If $\ell > r$ **return**.
5. **return**.

Figure 3.1: Algorithm Chain-Alloc

Algorithm Chain-Alloc is described in Figure 3.1. It first computes a lower bound B on the optimal using Theorem 3.3.4. Then it orders the chains in non-increasing lengths and allocates the chains to speeds such that no speed is loaded by more than four times the lower bound. We now prove several properties of the described allocation. Recall that D_i is the average load on a machine in speed class i.

Lemma 3.4.1 *Let $P_{\ell(u)}, \ldots, P_r$ be the chains remaining when Chain-Alloc considers speed u in step 4 of the algorithm. Then*

1. $|P_{\ell(u)}|/\overline{s}_u \leq 2B$ *and*
2. *Either $P_{\ell(u)}, \ldots, P_r$ are allocated to speed u* or $D_u > 2B$.

Proof.[2] We prove the above assertions by induction on u. Consider the base case when $u = 1$ and $\ell(1) = 1$. From the definition of $L_{\mathcal{P}}$ it follows that $|P_1|/\overline{s}_1 \leq B$. Since P_1 is the longest chain, it also follows that $|P_j|/\overline{s}_1 \leq B$ for $1 \leq j \leq r$. Let t be the last chain allocated to $\overline{s}_1$. If $t = r$ we are done. If $t < r$, it must be the case that adding $P_{(t+1)}$ increases the average load on $\overline{s}_1$ to more than $4B$. Since $P_{(t+1)}/\overline{s}_1 \leq B$, we conclude that $D_1 = \sum_{j=1}^{t} |P_j|/m_1\overline{s}_1 > 3B > 2B$.

[2]We thank Monika Henzinger for simplifying an earlier version of this proof.

Assume that the conditions of the lemma are satisfied for speeds s_1 to s_{u-1} and consider speed s_u. We will assume that $\ell(u) < r$ for otherwise there is nothing to prove. We observe that the second condition follows from the first using an argument similar to the one used above for the base case. Therefore it is sufficient to prove the first condition. Suppose $|P_{\ell(u)}|/\overline{s}_u > 2B$. We will derive a contradiction later. Let $j = \ell(u)$ and let v be the index such that $M_1^{v-1} < j \leq M_1^v$ (recall that $M_1^v = \sum_{k=1}^{v} m_k$). If $j > m$, no such index exists and we set v to K, the slowest speed. If $j \leq m$, for convenience of notation we assume that $j = M_1^v$ simply by ignoring other machines of speed $\overline{s}_v$. It is easy to see that $v \geq u$ and $j > M_1^{u-1}$. From the definition of $L_{\mathcal{P}}$, AL, and B, we get the following two facts. If $j \leq m$ then $L_{\mathcal{P}} \geq (\sum_{i=1}^{j} |P_i|)/(\sum_{k=1}^{v} m_k\overline{s}_k)$. If $j > m$ then $AL \geq (\sum_{i=1}^{j} |P_i|)/(\sum_{k=1}^{K} m_k\overline{s}_k)$. Combining them we obtain the following,

$$\frac{\sum_{i=1}^{j} |P_i|}{\sum_{k=1}^{v} m_k\overline{s}_k} \leq \max\{L_{\mathcal{P}}, AL\} = B. \tag{3.1}$$

Since $|P_j|/\overline{s}_u > 2B$, it is the case that $|P_i|/\overline{s}_u > 2B$ for all $M_1^{u-1} < i \leq j$. This implies that

$$\begin{aligned} \sum_{M_1^{u-1}<i}^{j} |P_i| &> 2B(j - M_1^{u-1})\overline{s}_u \\ &\geq 2B \sum_{k=u}^{v} m_k\overline{s}_k \\ \Rightarrow \sum_{i=1}^{j} |P_i| &> 2B \sum_{k=u}^{v} m_k\overline{s}_k \end{aligned} \tag{3.2}$$

The last inequality follows since we are summing up more terms on the left hand side. From the induction hypothesis it follows that speeds $\overline{s}_1$ to $\overline{s}_{u-1}$ have an average load greater than $2B$. From this we obtain

$$\sum_{i=1}^{j-1} |P_i| > 2B \sum_{k=1}^{u-1} m_k\overline{s}_k \tag{3.3}$$

$$\Rightarrow \sum_{i=1}^{j} |P_i| > 2B \sum_{k=1}^{u-1} m_k\overline{s}_k \tag{3.4}$$

Combining Equations 3.2 and 3.4 we obtain the following.

$$
\begin{aligned}
2\sum_{i=1}^{j}|P_i| &> 2B\sum_{k=1}^{u-1} m_k\overline{s}_k + 2B\sum_{k=u}^{v} m_k\overline{s}_k \\
&> 2B\sum_{k=1}^{v} m_k\overline{s}_k \\
\Rightarrow \sum_{i=1}^{j}|P_i| &> B\sum_{k=1}^{v} m_k\overline{s}_k \qquad (3.5)
\end{aligned}
$$

Equation 3.5 contradicts Equation 3.1. □

Corollary 3.4.2 *If chain P_j is assigned to speed i, then $\frac{|P_j|}{\overline{s}_i} \le 2B$.*

Corollary 3.4.3 *Algorithm Chain-Alloc allocates all chains.*

Lemma 3.4.4 *For $1 \le k \le K$, $D_k \le 4C^*_{\max}$.*

Proof. Since $B \le C^*_{\max}$ and the algorithm never loads a speed by more than an average load of $4B$, the bound follows. □

Lemma 3.4.5 *For the job assignment produced by Chain-Alloc $C \le 2KC^*_{\max}$.*

Proof. Let P be any chain. We will show that $\sum_{j\in P} p_j/\overline{s}_{k(j)} \le 2KC^*_{\max}$ where $k(j)$ is the speed to which job j is assigned. Let A_i be the set of jobs in P which are assigned to speed i. Let P_ℓ be the longest chain assigned to speed i by the algorithm. We claim that $|P_\ell| \ge \sum_{j\in A_i} p_i$. This is because the jobs in A_i form a chain when we picked P_ℓ to be the longest chain in the maximal chain decomposition. From Corollary 3.4.2 we know that $|P_\ell|/\overline{s}_i \le 2B \le 2C^*_{\max}$. Therefore it follows that

$$\sum_{j\in P}\frac{p_j}{\overline{s}_{k(j)}} = \sum_{i=1}^{K}\frac{|A_i|}{\overline{s}_i} \le 2KC^*_{\max}.$$

□

Theorem 3.4.6 *Using speed based list scheduling on the job assignment produced by Algorithm Chain-Alloc gives a $6K$ approximation where K is the number of distinct speeds. Furthermore the algorithm runs in $O(n^3)$ time. The running time can be improved to $O(n^2\sqrt{n})$ if all p_j are the same.*

Proof. From Lemma 3.4.4 we have $D_k \leq 4C^*_{\max}$ for $1 \leq k \leq K$ and from Lemma 3.4.5 we have $C \leq 2KC^*_{\max}$. Putting these two facts together, for the job assignment produced by the algorithm Chain-Alloc, speed based list scheduling gives the following upper bound by Theorem 3.2.1.

$$C_{\max} \leq C + \sum_{k=1}^{K} D_k \leq 2KC^*_{\max} + 4KC^*_{\max} \leq 6KC^*_{\max}.$$

It is easy to see that the speed based list scheduling can be implemented in $O(n^2)$ time. The running time is dominated by the time to compute a maximal chain decomposition. Theorem 3.3.5 gives the desired bounds. □

Corollary 3.4.7 *There is an algorithm which runs in $O(n^3)$ time and gives an $O(\log m)$ approximation ratio for the problem of scheduling precedence constrained jobs on uniformly related machines to minimize makespan.*

Remark 3.4.8 *The maximal chain decomposition depends only on the jobs of the given instance and is independent of the machine environment. If a maximal chain decomposition is given the schedule can be computed in $O(n \log n)$ time.*

We note here that the leading constant in the LP based algorithm in [17] is better. We also observe that the above bound is based on our lower bound which is valid for preemptive schedules as well. Hence our approximation ratio is also valid for preemptive schedules. In [17] it is shown that the lower bound provided by the LP relaxation is a factor of $\Omega(\log m/\log\log m)$ away from the optimal. Surprisingly it is easy to show using the same example as in [17] that our lower bound from Section 3.3 is also a factor of $\Omega(\log m/\log\log m)$ away from the optimal.

Theorem 3.4.9 *There are instances where the lower bound given in Theorem 3.3.4 is a factor of $\Omega(\log m/\log\log m)$ away from the optimal.*

Proof. The proof of Theorem 3.3 in [17] provides the instance and it is easily verified that *any* maximal chain decomposition of that instance is a factor of $\Omega(\log m/\log\log m)$ away from the optimal. □

3.4.1 Release Dates

Now consider the scenario where each job j has a release date r_j before which it cannot be processed. By a general result of Shmoys, Wein, and Williamson [103] an approximation algorithm for the problem of minimizing makespan without release dates can be transformed to one with release dates losing only a factor of 2 in the process. Therefore we obtain the following.

Theorem 3.4.10 *There is an $O(\log m)$ approximation algorithm for scheduling to minimize makespan of jobs with precedence constraints and release dates on uniformly related machines ($Q|prec, r_j|C_{\max}$) that runs in time $O(n^3)$.*

3.4.2 Scheduling Chains

In this subsection we show that Chain-Alloc followed by speed based list scheduling gives a constant factor approximation if the precedence constraints are induced by a collection of chains. We first observe that any maximal chain decomposition of a collection of chains is simply the collection itself. The crucial observation is that the algorithm Chain-Alloc allocates all jobs of any chain to the same speed class. The two observation together imply that there are no precedence relations between jobs allocated to different speeds. Suppose we run Chain-Alloc with m speed classes where each machine is in its own distinct speed class. Lemma 3.4.4 implies that $\max_{1\leq k\leq K} D_k \leq 4B$. Since each machine is in its own speed class it follows that the load on each machine is at most $4B$. But if chains are allocated whole, each machine can process the jobs assigned to it in a serial fashion and finish them in time equal to its load. It follows that the makespan of the schedule is at most $4B$.

Theorem 3.4.11 *There is a 4 approximation for the problem $Q|chains|C_{\max}$ and a 8 approximation for the problem $Q|chains, r_j|C_{\max}$.*

Computing the maximal chain decomposition of a collection of chains is trivial and the above algorithm can be implemented in $O(n \log n)$ time.

3.5 Concluding Remarks

Chudak and Shmoys [17] provide an $O(\log m)$ approximations for the more general problem of minimizing the sum of weighted completion times ($Q|prec|\sum w_j C_j$) using linear programming relaxations. Obtaining a simpler algorithm for that problem as well is an interesting problem. It is known that the problem of minimizing makespan is hard to approximate to within a factor of $4/3$ even if all machines have the same speed [74]. However, for the single speed case Graham's list scheduling gives a 2 approximation, while we are able to obtain only an $O(\log m)$ ratio for the multiple speed case. We conjecture the following.

Conjecture 3.5.1 *There is a $O(1)$ approximation algorithm for the problem $Q|prec|C_{\max}$ and even for the more general problem $Q|prec|\sum w_j C_j$.*

Improving the hardness of $4/3$ for the multiple speed case is also an interesting open problem.

Chapter 4

Scheduling Problems in Parallel Query Optimization

4.1 Introduction

Large database systems[1] with sophisticated query processing capabilities are of growing importance in todays computing environment. Their importance will continue to grow in the near future fueled by the explosive growth in the data available for processing and concomitant applications such as data mining, full text searching, and many others. Parallel machines assembled from commodity hardware are offering cost-performance benefits that rival or beat those provided by sequential mainframes. All the major commercial database vendors offer parallel database solutions today. Thus exploiting parallel computing for speeding up large database queries is an important problem. The shared-nothing architecture in particular offers flexibility and scalability and and seems to be the favored architecture for parallel database systems [24].

Database queries are almost always written in a high level declarative language such as SQL. It is the job of the underlying database management system (DBMS) to translate the users queries into a query execution plan that can be run on the underlying hardware environment to produce the query results. The declarative nature of the query language leaves several choices for the DBMS to execute the query, and *query optimization* refers to the process of obtaining the *best* query execution plan for a given query. A DBMS is a large

[1]This chapter (except Section 4.6) is joint work with Waqar Hasan and Rajeev Motwani [11].

system with many components and query optimization even for a sequential machine is an involved process. Query optimization for parallel databases offers many more challenges and is still a relatively less understood area.

An aspect of query optimization that is novel in the parallel setting is the cost of communication incurred between processes running on different processors. Ignoring communication in parallel databases could lead to query plans that unnecessarily transfer very large relations between processors resulting in very poor performance. Thus exploiting parallel execution to speed up database queries presents a parallelism-communication tradeoff [24, 111]. While work is divided among processors, the concomitant communication increases total work itself [45, 89]. To reduce the complexity of the parallel query optimization problem a natural two-phase approach [57, 51] is used. The first phase transforms the given query into an equivalent one that minimizes the overall query execution cost, and is similar to query optimization for sequential machines. The output of the first phase is an annotated query tree. The second phase parallelizes the plan produced by the first phase and schedules the execution on the parallel machine. In this work we study only with the scheduling phase. We postpone to Section 4.2 detailed description of the phases. The task to be scheduled is represented as a *weighted operator tree* [57, 95, 50] in which nodes represent atomic units of execution (operators) and directed edges represent the flow of data as well as timing constraints between operators. Weight of a node represents the time to process the operator and weight of an edge represents the communication cost between the operators connected by the edge.

Scheduling a weighted operator tree on a parallel machine poses a class of novel multiprocessor scheduling problems that differ from the classical ones [44] in several ways. First, edges represent two kinds of timing constraints — parallel and precedence. Second, since data is transmitted in long streams, the important aspect of communication is the CPU overhead of sending/receiving messages and not the delay for signal propagation (see [83, 84] for models of communication as delay). Third, the set oriented nature of queries has led to intra-operator parallelism (relations are horizontally partitioned and a clone of the operator applied to each partition) in addition to inter-operator parallelism [24].

We introduce several problems and focus on the specific problem of scheduling a *pipelined operator tree* (POT scheduling). All edges in such a tree represent parallel constraints, i.e., all operators run in parallel. A schedule is simply an assignment of operators to processors. Since edge weights represent the cost of remote communication, this cost is saved if adjacent operators share a processor. Given a schedule, the load on a processor is the sum of the weights of nodes assigned to it and the weights of all edges that connect nodes on the processor to nodes on other processors. The response time (makespan) of a schedule is the maximum processor load and the optimization problem is to find a schedule with minimum response time.

POT scheduling is NP-Hard since the special case in which all communication costs are zero is classical *multi-processor scheduling* problem [34]. We assess algorithms by their *performance ratio* which is the ratio of the response time of the generated schedule to that of an optimal schedule. We give two constant factor algorithms that run in $O(n \log n)$ time. The first algorithm LocalCuts has a ratio of 3.56 while the second algorithm BoundedCuts has ratio of $(1+\epsilon)2.87$. BoundedCuts runs in time $O(\log(1/\epsilon) n \log n)$ and thus has a larger hidden constant. We then show that we can obtain a polynomial time approximation scheme[2] for the problem by generalizing the ideas for multi-processor scheduling [55]. Though the PTAS is not very practical, it settles the approximability of the problem.

In Section 4.2, we provide an overview of parallel query optimization and develop a model for scheduling problems. In Section 4.3, we review past work on the POT problem and describes our two-stage approach to the development of approximation algorithms for POT. In Section 4.4, we develop the LocalCuts algorithm and show it to have a performance ratio of 3.56. In Section 4.5, this algorithm is modified to yield the BoundedCuts algorithms which is shown to have a performance ratio of 2.87. We give our PTAS in Section 4.6.

4.2 A Model for Scheduling Problems

Figure 4.1 shows a two-phase approach [57, 51] for parallel query optimization. The first phase, JOQR (for Join Ordering and Query Rewrite), minimizes total cost and produces an annotated query tree that fixes aspects such as the order of joins and the strategy for computing each join. The second phase, *parallelization*, converts the annotated query tree into a parallel plan. Parallelization itself has two steps. The first converts the annotated query tree to an operator tree [31, 57, 95]. The second schedules the operator tree on a

[2] A PTAS for the problem was first obtained by Schuurman and Woeginger [101]. We obtained ours independently after hearing about their result.

parallel machine.

Several approaches exist for the first phase; Hong and Stonebraker [58] used a conventional query optimizer while Hasan and Motwani [50] develop algorithms that incorporate communication costs. In this work we are only concerned with the second phase.

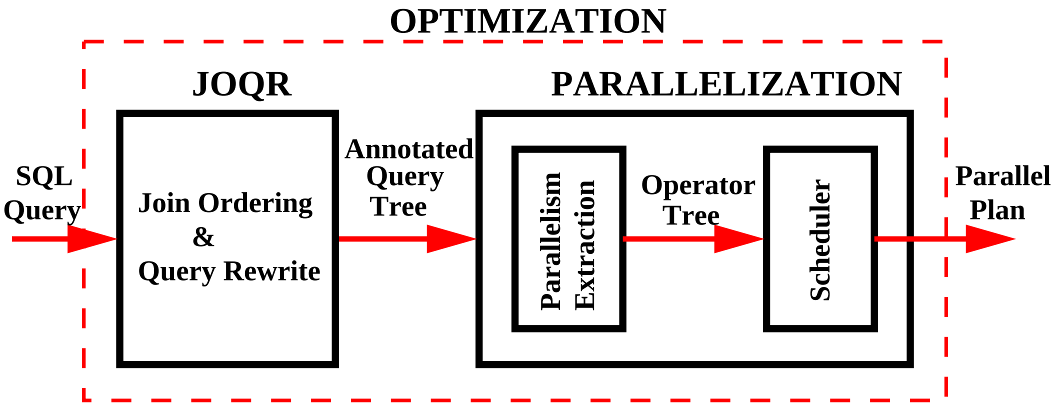

Figure 4.1: Parallel Query Processing:Software Architecture

We will first discuss the forms of available parallelism and how they are captured by the operator tree representation. We then describe how we model communication. Finally, we describe a variety of scheduling problems. The reader is referred to the thesis of Waqar Hasan [51] for more details and related issues.

4.2.1 Forms of Parallelism

Parallel database systems speed-up queries by exploiting *independent* and *pipelined* forms of inter-operator parallelism as well as intra-operator or *partitioned* parallelism. Independent parallelism simultaneously runs two operators with no dependence between them on distinct processors. Pipelined parallelism runs a consumer operator simultaneously with a producer operator on distinct processors. Partitioned parallelism uses several processors to run a single operator. It exploits the set-oriented nature of operators by partitioning the input data and running a copy of the operator on each processor.

4.2.2 Operator Trees

Available parallelism is represented as an operator tree $T = (V, E)$ with $V = \{1, \ldots, n\}$. Nodes represent operators. Functionally, an operator takes zero or more input sets and produces a single output set. Physically, it is a piece of code that is *deemed* to be atomic. Edges between operators represent the flow of data as well as timing constraints. As argued in [50], operators may be *designed* to ensure that any edge represents either a parallel or a precedence constraint.

Example 4.2.1 Figure 4.2 shows a query tree and the corresponding operator tree. Edges with unfilled arrow heads are pipelining edges, edges with filled arrow heads are blocking. A simple hash join is broken into `Build` and `Probe` operators. Since a hash table must be fully built before it can be probed, the edge from `Build` to `Probe` is blocking. A sort-merge join sorts both inputs and then merges the sorted streams. The merging is implemented by the `Merge` operator. In this example, we assume the right input of sort-merge to be pre-sorted. The operator tree shows the sort required for the left input broken into two operators `FormRuns` and `MergeRuns`. Since the merging of runs can start only after run formation, the edge from `FormRuns` to `MergeRuns` is blocking. □

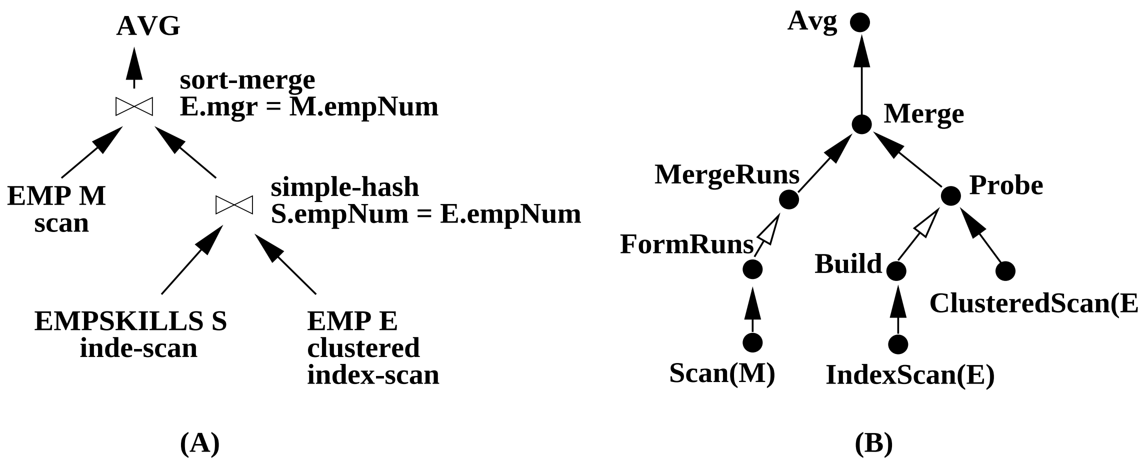

Figure 4.2: Macro-Expansion of Query Tree to Operator Tree (Parallelism Extraction)

The operator tree exposes the available parallelism. Partitioned parallelism may be used for any operator. Pipelined parallelism may be used between two operators connected by a pipelining edge. Two subtrees with no (transitive) timing constraints between them may run independently (eg: subtrees rooted at `FormRuns` and `Build`).

Definition 4.2.2 *A* pipelining *edge from operator i to j represents a parallel constraint that requires i and j to start at the same time and terminate at the same time. A* blocking *edge from i to j represents a* precedence constraint *that requires j to start after i terminates.*

A pipelining constraint is symmetric in i and j. The direction of the edge indicates the direction in which tuples flow but is immaterial for timing constraints. Since all operators in a pipelined subtree start and terminate simultaneously[3], operators with smaller processing times use a smaller *fraction* of the processor on which they run.

4.2.3 Model of Communication

The weight p_i of node i in an operator tree is the time to run the operator in isolation assuming all communication to be local. The weight c_{ij} of an edge from node i to j is the *additional* CPU overhead that both i and i would incur for inter-processor communication if they are scheduled on different processors. A specific schedule incurs communication overheads only for the *fraction* of data that it actually communicates across processors. As discussed in [50], conventional cost models (such as System R [102]) that estimate the sizes of the intermediate results can be adapted to estimate node and edge weights.

Figure 4.3 shows the extreme cases of communication costs of blocking and pipelining edges. In the figure the thick edge represents a blocking edge and the thin edge represents a pipelining edge. Communication is saved when the two operators are on the same processor and totals to twice the edge weight when they are on distinct processors. For a blocking edge, communication occurs after the producer terminates and before the consumer starts. For a pipelined edge, communication is spread over the execution time of the entire operator. Note that since all operators in a pipeline start and terminate simultaneously, heavier operators use a larger fraction of the CPU of the processor they run on.

4.2.4 Scheduling Problems

We assume a parallel machine to consist of m identical processors. A schedule is an assignment of operators to processors. We model partitioned parallelism as permitting processors to execute *fractions* of an operator. Depending on whether partitioned parallelism is allowed or not, the assignment of operators to processors is a fractional or 0-1 assignment. Since the goal of a parallel database system is to speedup queries, we are interested in finding

[3]Pipelining in other contexts (such as instruction pipelining) connotes a sequential form of execution. In the database context, even though individual tuples (or blocks of them) are processed sequentially in a pipeline, the data is assumed to be large enough that all operators in a pipeline start and finish simultaneously.

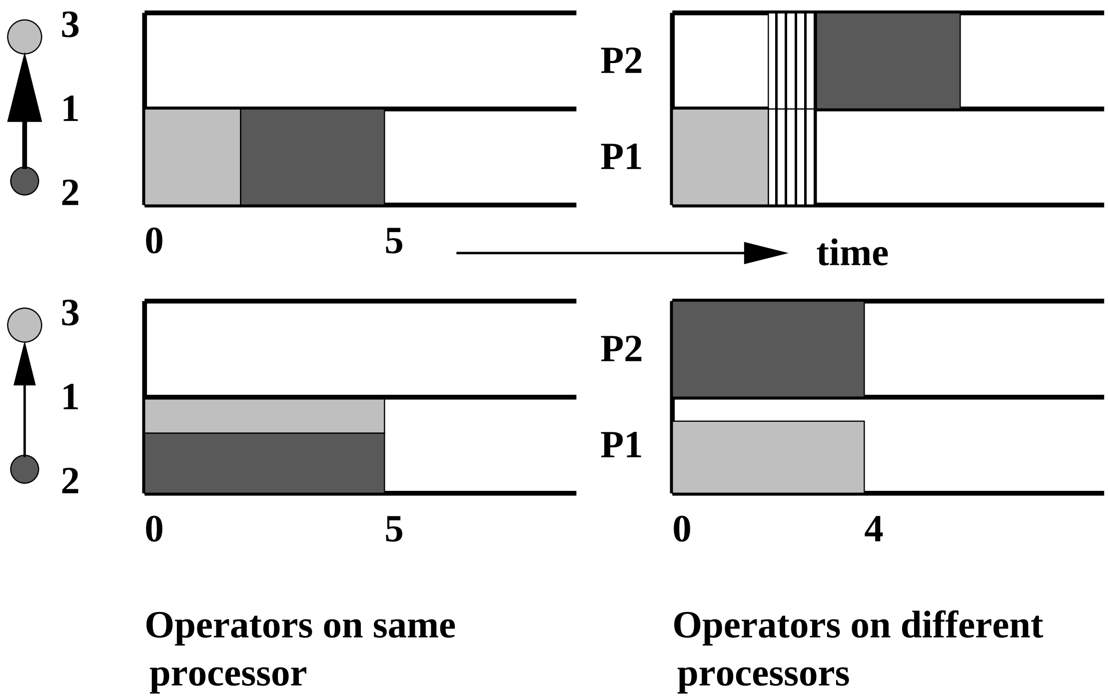

Figure 4.3: Communication Costs for Blocking and Pipelining Edges: Gantt Charts

schedules with minimal response time. The *response time* of a schedule is the elapsed time between starting query execution and fully producing the result.

Figure 4.4 shows the division of the problem of scheduling operator trees into subproblems along two dimensions: the kinds of edges in an operator and whether schedules allow fractional assignment.

	No Partitioned Parallelism	With Partitioning
Pipelined Edges Only	Pipelined Operator Tree (POT)	(POTP)
Blocking Edges Only	Blocking Operator Tree (BOT)	(BOTP)
Pipelining & Blocking	Operator Tree (OT)	(OTP)

Figure 4.4: Classification of Scheduling Problems

In the rest of this chapter we address only one of the above problems namely the POT scheduling problem. Even though POT scheduling is a restricted version of the most general problem we still need non-trivial techniques and analysis to obtain efficient and good approximation algorithms.

4.3 POT Scheduling

We begin by reviewing relevant definitions and results from earlier work on POT scheduling by Hasan and Motwani [50] where algorithms were designed for restricted shapes of trees. We then describe a two-stage approach that we shall use in developing and analyzing our algorithms.

4.3.1 Problem Definition and Prior Results

In the POT scheduling problem the given tree $T = (V, E)$ is restricted to have only pipelining edges. The following definitions make the problem precise.

Definition 4.3.1 *Given m processors and an operator tree $T = (V, E)$, a* schedule *is a partition of V, the set of nodes, into m sets $F_1, \ldots, F_m$ with set F_k allocated to processor k.*

Definition 4.3.2 *The load L_k on processor k is the cost of executing all nodes in F_k plus the overhead for communicating with nodes on other processors, $L_k = \sum_{i \in F_k}(p_i + \sum_{j \notin F_k} c_{ij})$.*

Definition 4.3.3 *The response time of a schedule is the maximum processor load, $L_{\max} = \max_{1 \leq k \leq m} L_k$.*

Example 4.3.4 Figure 4.5 shows a 2-processor schedule. Sets F_1 and F_2 are encircled. F_1 results in a load of $L_1 = 31$ on processor 1 since the processor must pay for the three nodes in F_1 (5+5+10) as well as for the edges that connect to nodes on the other processor (5+6). Similarly $L_2 = 34$. The response time of the schedule is $L_{\max} = \max(L_1, L_2) = 34$. The edges are shown undirected since the stream direction is not relevant for communication costs. □

Prior work [50] showed that any given tree can be transformed into what is called a *monotone* tree, and therefore it suffices to consider algorithms for these restricted trees. It is also shown that the transformation can be accomplished by a simple algorithm called GreedyChase. Further, a schedule for the original tree can be easily obtained from a schedule for the monotone tree. The cost of a set of nodes A denoted by $cost(A)$ is defined to be $\sum_{i \in A}(p_i + \sum_{j \notin A} c_{ij})$. A set of nodes is *connected* if the vertices in the set induce a connected subtree of T.

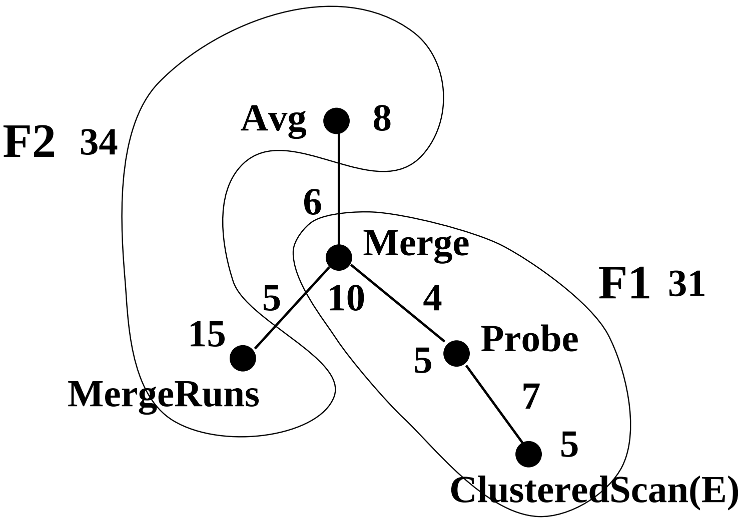

Figure 4.5: 2-processor Schedule

Definition 4.3.5 *An operator tree is* monotone *if and only if, for any two connected sets of nodes,* X *and* Y *such that* $X \subset Y$, $cost(X) < cost(Y)$.

Definition 4.3.6 *An edge* e_{ij} *is* worthless *if and only if* $(c_{ij} \geq p_i + \sum_{k \neq j} c_{ik})$ *or* $(c_{ij} \geq p_j + \sum_{k \neq i} c_{jk})$.

Another interpretation of monotone trees is that they do not contain any *worthless* edges. These are edges whose communication cost is high enough to offset any benefits of using parallel execution for the two end points. The following lemma shows the equivalence.

Lemma 4.3.7 ([50]) *A tree is monotone if and only if it does not have worthless edges.*

The GreedyChase algorithm converts any tree into a monotone tree by repeatedly collapsing worthless edges. The following theorem shows that instead of scheduling a given tree, we may schedule the corresponding monotone tree. We prove it for the sake of completeness.

Theorem 4.3.8 ([50]) *For any operator tree* T *with a worthless edge* (i, j), *there exists an optimal schedule for* T *in which nodes* i *and* j *are assigned to the same processor.*

Proof. Let (i, j) be a worthless edge and let S be an optimal schedule in which i and j are scheduled on different processors, say 1 and 2. Assume without loss of generality that

$(c_{ij} \geq p_j + \sum_{k \neq i} c_{jk})$. Suppose we move j to processor 1 on which i is scheduled, in the process obtaining a new schedule S'. Then it is easy to see that the increase in L_1, the load of i's processor, is bounded by $(p_j + \sum_{k \neq i} c_{jk}) - c_{ij}$. The increase in L_2, the load of j's processor, is bounded by $\sum_{k \neq i} c_{jk} - p_j - c_{ij}$. Since $(c_{ij} \geq p_j + \sum_{k \neq i} c_{jk})$, both increases are negative. Thus S' is at least as good a schedule as S. □

Example 4.3.9 In Figure 4.5, the edge between `Probe` and `ClusteredScan` is worthless since its weight exceeds the weight of `ClusteredScan`. The corresponding monotone tree is created by collapsing `Probe` and `ClusteredScan` into a single node of weight 10. □

Monotone trees have the property that, for any node, the sum of its weight and the weights of edges incident on it, is a lower bound on the optimal response time. This lower bound is useful in proving the performance ratios achieved by our algorithms.

Lemma 4.3.10 ([50]) *The response time of any schedule (independent of number of processors) for a monotone operator tree has a lower bound of* $\max_{i \in V}(p_i + \sum_{j \in V} c_{ij})$.

Proof. Consider any optimal schedule S. In S, let X_i be the connected set of nodes that i is scheduled with. Since the tree is monotone it follows that $cost(X_i) \geq cost(\{i\}) = (p_i + \sum_{j \in V} c_{ij})$. □

In the remainder of the chapter, we will assume operator trees to be monotone.

4.3.2 A Two-stage Approach

We divide the POT scheduling problem into two stages, *fragmentation* followed by *scheduling*. Fragmentation partitions the tree into *connected* fragments by *cutting* some edges. A cut edge is deleted from the tree. This should be interpreted as a decision to allocate the two end-points to distinct processors. The additional communication cost is captured by adding the weight of the deleted edge to the weights of both its end-points. Any edge which is not cut is *collapsed*. This should be interpreted as a decision to schedule the two end-points on the same processor. Collapsing merges the two end-points into a single node which is assigned all the incident edges of the merged nodes and has weight equal to the sum of the weights of the merged nodes. We view the result of fragmentation as a set of fragments that are free to be scheduled independently of each other. The scheduling stage assigns the fragments produced by the first stage to processors.

The two stage approach offers conceptual simplicity and does not restrict the space of schedules. Any schedule defines a natural fragmentation corresponding to cutting exactly the inter-processor edges. For any given schedule, some scheduling algorithm will produce it from its natural fragmentation. Notice that the scheduling stage may assign two fragments that were connected by a cut edge to the same processor thus "undoing" the cutting. Thus, several fragmentations may produce the same schedule. In our analysis, we will ignore the decrease in communication cost caused by this implicit undoing of an edge cutting operation. This can only over-estimate the cost of our solution.

The two-stage approach allows us to use standard multi-processor scheduling algorithms for the second stage. For the constant factor approximation algorithms in Sections 4.4 and 4.5 we use the classical LPT algorithm (largest processing time first) [43]. This is a greedy algorithm that assigns the largest unassigned job to the least loaded processor. For the PTAS in Section 4.6 we use a different approach, the details of which we defer to that section.

We first develop conditions on fragmentation that when combined with LPT for the scheduling stage yield a good approximation ratio. There is an inherent tradeoff between total load and the weight of the heaviest connected fragment. If an edge is cut, communication cost is incurred thus increasing total load. If an edge is collapsed, a new node with a larger net weight is created, potentially increasing the weight of the largest connected fragment. Lemma 4.3.16 captures this trade-off. Before proceeding further, we define a few quantities.

Definition 4.3.11 *Let $R_i = cost(\{i\}) = p_i + \sum_j c_{ij}$ denote the cost of node i and let $R = \max_i R_i$. $W = \sum_i p_i$ is the sum of the weights of all nodes. $W_{\text{avg}} = W/m$ is the average node weight per processor.*

Assuming fragmentation to produces q fragments with costs $M_1, \ldots, M_q$, we make the following definitions.

Definition 4.3.12 *Let $M = \max_i M_i$ denote the weight of heaviest fragment. Let C be the the total communication cost incurred, which is twice the sum of the weights of the* cut *edges. We define $L_{\text{avg}} = (W + C)/m$ to be the* average *load per processor.*

We use the superscript $*$ to denote the same quantities for the natural fragmentation corresponding to some fixed optimal schedule. For example, M^* denotes the weight of the heaviest fragment in the optimal fragmentation.

Example 4.3.13 Figure 4.6 shows the natural fragmentation for the schedule of Example 4.3.4. After the remaining edges are collapsed, we get three nodes with weights $M_1 = 14$, $M_2 = 20$, and $M_3 = 31$. Thus $M = \max\{M_1, M_2, M_3\} = 31$. $C = 22$ since the fragmentation cuts two edges with weights 5 and 6. Since the total node weight in the original tree is $W = 43$, we have $L_{\text{avg}} = (W + C)/m = (43 + 22)/2$. □

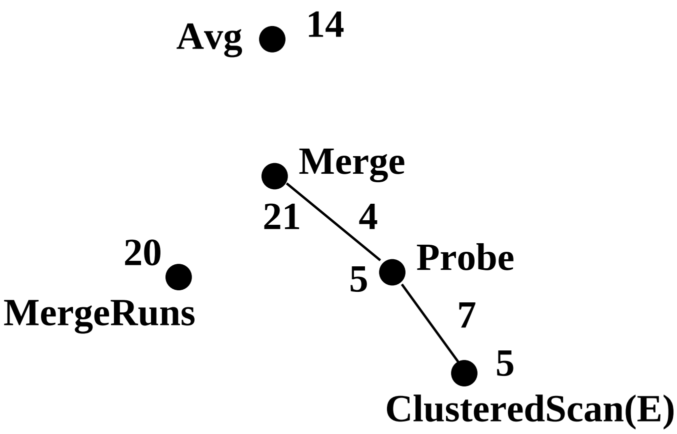

Figure 4.6: Fragments

The following two propositions provide lower bounds on the value of the optimal solution. The proofs are straightforward.

Proposition 4.3.14 *For any schedule* $W_{\text{avg}} \leq L_{\text{avg}} \leq L_{\max}$. *In particular,* $W_{\text{avg}} \leq L^*_{\text{avg}} \leq L^*_{\max}$.

Proposition 4.3.15 *For any schedule* $R \leq M \leq L_{\max}$. *In particular,* $R \leq M^* \leq L^*_{\max}$.

In the following lemma, k_1 captures the effect of size of the largest fragment and k_2 the load increase due to communication.

Lemma 4.3.16 *Given a fragmentation with* $M \leq k_1 L^*_{\max}$ *and* $L_{\text{avg}} \leq k_2 L^*_{\text{avg}}$, *scheduling using LPT yields a schedule with* $L_{\max}/L^*_{\max} \leq \max\{k_1, 2k_2\}$.

Proof. For a fragmentation with the stated properties, let P be a *heaviest* loaded processor in an LPT schedule, with response time $L_{\max}$. Let F_j be the last fragment, of cost M_j, that is assigned to P. We divide the analysis into two cases based on whether F_j is the only fragment on P or not.

If F_j is the only fragment on P, $L_{\max} = M_j$, and by our assumption on the fragmentation,

$$L_{\max} = M_j \leq M \leq k_1 L^*_{\max}.$$

Now consider the case when the number of fragments on P is at least two. Since LPT assigns the current job to the least loaded processor at that point, when F_j is assigned to P, the load on *each* processor is at least $L_{\max} - M_j$. The total load $\sum_k L_k$ may be bounded as

$$\begin{aligned} & \sum_k L_k \geq m \cdot (L_{\max} - M_j) + M_j \\ \Rightarrow \quad & L_{\max} \leq \frac{1}{m} \sum_k L_k + \left(1 - \frac{1}{m}\right) M_j \\ \Rightarrow \quad & L_{\max} \leq L_{\text{avg}} + M_j. \end{aligned}$$

LPT schedules the largest m jobs on distinct processors. Since there was at least one other fragment on P before M_j, there are at least $(m+1)$ fragments, each of them no lighter than M_j. Thus,

$$\begin{aligned} & \sum_k L_k \geq (m+1) M_j \\ \Rightarrow \quad & M_j \leq \frac{1}{m+1} \sum_k L_k < L_{\text{avg}}. \end{aligned}$$

Combining the two observations and using the assumption $L_{\text{avg}} \leq k_2 L^*_{\text{avg}}$, we obtain

$$\begin{aligned} L_{\max} &\leq L_{\text{avg}} + M_j \\ &< 2L_{\text{avg}} \\ &< 2k_2 L^*_{\text{avg}} \\ &< 2k_2 L^*_{\max}. \end{aligned}$$

Combining the two cases, we conclude $L_{\max}/L^*_{\max} \leq \max\{k_1, 2k_2\}$. □

Using the above lemma, the best we can do is to find a fragmentation with $k_1 = k_2 = 1$ which would guarantee a performance ratio of 2. A *star* is a tree with only one non-leaf node. We show that even for this simple case, the problem of finding the best fragmentation is NP-Complete.

Theorem 4.3.17 *Given a star $T = (V, E)$, bounds B and C, the problem of determining whether there is a partition of V such that no fragment is heavier than B and the total communication is no more than C is NP-Complete.*

Proof. We reduce the classical *knapsack* problem [34] to the above problem. Let an instance of the knapsack problem be specified by a bag size S and n pairs (w_i, p_i) where each pair corresponds to an object of weight w_i with profit p_i. By scaling appropriately we can assume without loss of generality that $p_i \leq w_i$ for $1 \leq i \leq n$. We construct a star T with $n+1$ nodes from the knapsack instance. We label the nodes of T from 0 to n with the center as 0. For $1 \leq i \leq n$, we set $c_{i0} = p_i/2$ and $p_i = w_i + c_{i0}$ and $B = S + \sum_i c_{i0}$. We set $p_0 = 0$. We claim that the minimum communication cost for the star instance is C if and only if the maximum profit for the knapsack instance is $\sum_i p_i - C$. □

We remark that the problem is polynomial time solvable when the tree is restricted to be a path. Moreover we can obtain a pseudo-polynomial time algorithm for the above problem even on a tree. We will use this observation in Section 4.6. The next two section focus on algorithms to find good fragmentations that guarantee low values for k_1 and k_2.

4.4 The LocalCuts Algorithm

We now develop a linear time algorithm for fragmentation called LocalCuts. We show bounds on the weight of the heaviest fragment as well as on the load increase due to communication. Application of Lemma 4.3.16 shows the algorithm to have a performance ratio of 3.56.

LocalCuts repeatedly picks a leaf and determines whether to cut or collapse the edge to its parent. It makes the decision based on *local* information, the ratio of the leaf weight to the weight of the edge to its parent. The basic intuition is that if the ratio is low, then collapsing the edge will not substantially increase the *net* weight of the parent. If the ratio is high, the communication cost incurred by cutting will be relatively low and can be amortized to the weight of the node cut off. One complication is that cutting or collapsing an edge changes node weights. Our analysis *amortizes* the cost of cutting an edge, over the total weight of all nodes that were collapsed to produce the leaf.

In the following discussion we assume that the tree T has been *rooted* at some arbitrary node. We will refer to the fragment containing the root as the *residual tree*. A *mother* node in a rooted tree is a node all of whose children are leaves. The algorithm uses a parameter $\alpha > 1$. We will later show (Theorem 4.4.3) how this parameter may be chosen to minimize the performance ratio.

Algorithm 1 The *LocalCuts* Algorithm

Input: Monotone operator tree T, parameter $\alpha > 1$.
Output: Partition of T into fragments $F_1, \ldots, F_k$.
1. **while** there is a mother node s with a child j **do**
2. **if** $p_j > \alpha c_{js}$ **then** cut e_{js}
3. **else** collapse e_{js}
4. **end while**

The running time of the LocalCuts algorithm is $O(n)$. The following lemma shows a bound on the weight of the resulting fragments.

Lemma 4.4.1 *For the fragmentation produced by* LocalCuts $M < \alpha R$.

Proof. Consider an arbitrary fragment produced in the course of the algorithm. Let s be the *highest* level node in the fragment, and assume that its children are numbered $1, \ldots, d$. The node s is picked as a mother node at some stage of the algorithm. Now, $R_s = c' + p_s + c_{s1} + \ldots + c_{sd}$ where c' is the weight of the edge from s to its parent. Collapsing child j into s, corresponds to replacing c_{sj} by p_j. Since the condition for collapsing is $p_j \leq \alpha c_{sj}$, collapsing children can increase R_s to at most αR_s which is no greater than αR. □

We now use an amortization argument to show that the communication cost incurred by the LocalCuts algorithm is bounded by W, the total node weight, times a factor depending on α.

Lemma 4.4.2 *The total communication cost of the partition produced by the* LocalCuts *algorithm is bounded by* $\frac{2}{\alpha-1}W$*, that is* $C \leq \frac{2}{\alpha-1}W$.

Proof. We associate a *credit* q_i with each node i and *credit* q_{jk} with each edge e_{jk}. Initially, edges have zero credit and the credit of a node equals its weight; thus, the total initial credit is W. The total credit will be conserved as the algorithm proceeds. When a node is cut or collapsed, its credit is taken away and either transferred to another node or to an edge that is cut. The proof is based on showing that when the algorithm terminates, every edge that is cut has a credit equal to $(\alpha - 1)$ times its weight. This allows us to conclude that the total weight of the cut edges is bounded by $W/(\alpha - 1)$. This would then imply that $C \leq \frac{2}{\alpha-1}W$. We abuse notation by using p_i for the *current* weight of a node in the residual tree. We now prove the following invariants by induction on the number of iterations in LocalCuts.

1. Each node has a credit greater than or equal to its *current* weight in the residual tree, i.e., $q_i \geq p_i$.

2. Each *cut* edge e_{is} has a credit equal to $(\alpha - 1)$ times its weight, i.e., $q_{is} = (\alpha - 1)c_{is}$.

To take care of the base case we observe that the invariants are trivially true at the beginning of the algorithm. For the inductive step, assume that the invariants are true up to k iterations. Consider a leaf node j with mother s in the $(k + 1)$st iteration. We use the superscript NEW to indicate the values at the next iteration. If j is collapsed, $p_s^{\text{NEW}} = p_s + p_j$. By transferring the credit of j to s, we get $q_s^{\text{NEW}} = q_j + q_s$. By the inductive hypothesis $q_j \geq p_j$ and $q_s \geq p_s$. Therefore $q_s^{\text{NEW}} \geq p_s^{\text{NEW}}$ and both invariants are preserved.

If j is cut, $p_s^{\text{NEW}} = p_s + c_{js}$. We need to transfer a credit of c_{js} to s to maintain the first invariant. The remaining credit $q_j - c_{js}$ may be transferred to the edge e_{js}. By the induction hypothesis, we have $q_j - c_{js} \geq p_j - c_{js}$ and since edge e_{js} was cut, $q_j - c_{js} > (\alpha - 1)c_{js}$. Thus sufficient credit is available for the second invariant as well. □

The previous two lemmas combined with Lemma 4.3.16, allow us to bound the performance ratio guaranteed by LocalCuts. The following theorem states the precise result and provides a value for the parameter α.

Theorem 4.4.3 *Using LPT to schedule the fragments produced by* LocalCuts *with* $\alpha = (3 + \sqrt{17})/2$ *gives a performance ratio of* $(3 + \sqrt{17})/2 \sim 3.56$.

Proof. From Lemma 4.4.2 and Proposition 4.3.14,

$$L_{\text{avg}} = \frac{W + C}{m} \leq \frac{\alpha + 1}{\alpha - 1} W_{\text{avg}} \leq \frac{\alpha + 1}{\alpha - 1} L^*_{\text{avg}}.$$

Combining this with Lemma 4.4.1 and using Lemma 4.3.16 we conclude that

$$\frac{L_{\max}}{L^*_{\max}} \leq \max\left\{\alpha, \frac{2(\alpha+1)}{\alpha-1}\right\}.$$

The ratio is minimized when $\alpha = 2(\alpha+1)/(\alpha-1)$, that is when $\alpha = (3+\sqrt{17})/2$. Thus we obtain $L_{\max}/L^*_{\max} \leq (3+\sqrt{17})/2$. □

The performance ratio of LocalCuts is tight. Consider a star in which the center node with weight δ is connected by edges of weight 1 to $n-1$ leaves, each of weight $\alpha = 3.56$. Suppose the star is scheduled on $m = n$ processors. LocalCuts will collapse all leaves and produce a single fragment of weight $(n-1)\alpha + \delta$. The optimal schedule consists of cutting all edges to produce $n-1$ fragments of weight $1+\alpha$ and one fragment of weight $n-1+\delta$. When $n > 5$, the performance ratio is $((n-1)\alpha+\delta)/(n-1+\delta)$ which approaches α as δ goes to zero.

4.5 The BoundedCuts Algorithm

The LocalCuts algorithm determines whether to collapse a leaf into its mother based on the ratio of the leaf weight to the weight of the edge to its mother. The decision is independent of the current weight of the mother node. From the analysis of LocalCuts, we see that the weight of the largest fragment is bounded by αR_s, where s is the highest level node in the fragment (Lemma 4.4.1). If R_s is small compared to M^*, we may cut expensive edges needlessly. Intuitively, using a uniform bound that is larger than M^* should help in reducing communication cost. The analysis of LocalCuts showed the trade-off between total communication ($C \leq \frac{2}{\alpha-1}W$) and the bound on fragment size ($M < \alpha R$). Reduced communication should allow us to afford a lower value of α, thus reducing the largest fragment size and the performance ratio.

We now discuss a modified algorithm called BoundedCuts that uses a uniform bound B at each mother node. It also cuts off light edges in a manner similar to LocalCuts. Our analysis of communication costs uses lower bounds on C^*, the communication incurred in some fixed optimal schedule. The algorithm below is stated in terms of three parameters α, β and B that are assumed to satisfy $\beta \geq \alpha > 1$, and $M^* \leq B \leq L^*_{\max}$. Our analysis uses these conditions and we shall later show how the values of these parameters may be obtained.

Algorithm 2 The *BoundedCuts* Algorithm

Input: Monotone tree T, parameters α, β, and B where $\beta \geq \alpha > 1$ and $M^* \leq B \leq L^*_{\max}$.
Output: Partition of T into connected fragments $F_1, \ldots, F_k$.

1. **while** there exists a mother node s
2. partition children of s into sets N_1, N_2 such that child $j \in N_1$ if and only if $p_j/c_{sj} \geq \beta$;
3. cut e_{sj} for $j \in N_1$; **(β rule)**
4. **if** $R_s + \sum_{j \in N_2}(p_j - c_{sj}) \leq \alpha B$ **then**
5. collapse e_{sj} for all $j \in N_2$
6. **else** cut e_{sj} for all $j \in N_2$; **(α rule)**
7. **end while**
8. **return** resulting fragments $F_1, \ldots, F_k$.

Lemma 4.5.1 *Any fragment produced by* BOUNDEDCUTS *has weight at most αB. As a consequence, $M \leq \alpha B \leq \alpha L^*_{\max}$.*

Proof. Since the weight of a fragment increases only when some edge is collapsed, the explicit check in line 4 ensures the lemma. □

Let $\mathcal{C}$ denote the set of edges cut by BOUNDEDCUTS. We cut edges using two rules, the β rule in Step 3 and the α rule in Step 6. Let $\mathcal{C}^\beta$ and $\mathcal{C}^\alpha$ denote the edges cut using the respective rules. $\mathcal{C}^\beta$ and $\mathcal{C}^\alpha$ are disjoint and $\mathcal{C}^\beta \cup \mathcal{C}^\alpha = \mathcal{C}$. Let C^β and C^α denote the communication cost incurred due to edges in $\mathcal{C}^\beta$ and $\mathcal{C}^\alpha$ respectively. We bound C^β and C^α in Lemmas 4.5.2 and 4.5.4.

We need a few definitions. Let T_i denote the subtree rooted at node i and let W_i denote the sum of the node weights of T_i. Let $\mathcal{C}_i^\alpha$ denote the set of edges cut in T_i using the α rule (the set is empty if the rule is not applied in T_i) and let C_i^α be twice the sum of the weights of the edges in $\mathcal{C}_i^\alpha$. Similar definitions apply for the β rule. Let $\mathcal{C}_i^*$ denote the set of edges cut in some optimal schedule in T_i and let C_i^* denote twice the sum of the weights of the edges in $\mathcal{C}_i^*$.

Lemma 4.5.2 $C^\alpha \leq \dfrac{\beta - 1}{\alpha - 1} C^*$.

Proof. Let s be the first node where the α rule is applied. This implies that the α rule is not applied at any descendent of s.

Claim 4.5.3 $C_s^\alpha \leq \frac{\beta-1}{\alpha-1} C_s^*$.

We first prove the above claim and then show why the lemma follows from it. Let X be twice the sum of the weights of edges in $\mathcal{C}_s^* - \mathcal{C}_s^\beta$. We prove a stronger claim, that $C_s^\alpha \leq \frac{\beta-1}{\alpha-1}X$. In order to prove this stronger claim, we assume that the optimal is allowed to cut the edges in $\mathcal{C}_s^\beta$ free of cost (this is for purposes of the proof only). In other words we can assume that no edges were cut in T_s using the β rule. This also implies that the only edges cut in T_s are by the α rule at s.

Let the set $\mathcal{C}_s^*$ consist of q edges $e_{s_1z_1}, \ldots, e_{s_qz_q}$. These edges partition T_s into $(q+1)$ fragments. Let F_s be the fragment that contains s and without loss of generality assume that $s_1, \ldots, s_\ell$ (some of these may be the same as s) belong to F_s. Let $F_1, \ldots, F_\ell$ be the fragments that contain nodes $z_1, \ldots, z_\ell$ respectively. From the definitions we have

$$C_s^* = 2 \sum_{1 \leq j \leq q} c_{s_jz_j} \geq 2 \sum_{1 \leq j \leq \ell} c_{s_jz_j}.$$

Since no fragment in the optimal is larger than M^*, the total node weight in fragment F_s is at most $M^* - c_{sk} - \sum_{1 \leq j \leq \ell} c_{s_jz_j}$ where e_{sk} is the edge from s to its parent. Thus, we have

$$M^* - c_{sk} - \sum_{1 \leq j \leq \ell} c_{s_jz_j} + \sum_{1 \leq j \leq \ell} W_j \geq W_s.$$

The α rule was applied at s. Therefore it follows that $W_s + c_{sk} > \alpha B$. Since $B \geq M^*$, we have $W_s + c_{sk} > \alpha M^*$ which reduces the above equation to the following.

$$\sum_{1 \leq j \leq \ell} (W_j - c_{s_jz_j}) > (\alpha - 1)M^*$$

Since no edge in T_s was cut by the β rule, $W_j < \beta c_{s_jz_j}$ for $1 \leq j \leq \ell$. Therefore

$$\sum_{1 \leq j \leq \ell} (\beta - 1)c_{s_jz_j} > (\alpha - 1)M^*$$
$$\Rightarrow \quad M^* < \frac{\beta - 1}{\alpha - 1} \sum_{1 \leq j \leq \ell} c_{s_jz_j} \leq \frac{\beta - 1}{\alpha - 1} C_s^*/2.$$

Since $C_s^\alpha < 2R_s \leq 2M^*$, we have proved our claim:

$$C_s^\alpha \leq \frac{\beta - 1}{\alpha - 1} C_s^*.$$

We can inductively apply the claim to the rest of the tree to obtain the lemma. There is a small subtlety in applying the induction. BOUNDEDCUTS cuts all the edges incident on s when it applies the α rule. The optimal fragmentation might collapse some edges into s. Therefore the weight of s in T_1, the tree obtained after BOUNDEDCUTS processes nodes in the subtree T_s, is no more than in T_2, the tree obtained after the optimal fragmentation processes T_s. However this can only help the analysis in favor of BOUNDEDCUTS in the later stages of the induction. □

Using techniques similar to those in the proof of Lemma 4.4.2, we show the following bound on C^β.

Lemma 4.5.4 $C^\beta \le \dfrac{2}{\beta-1}W - \dfrac{\alpha-1}{\beta-1}C^\alpha$.

Proof. We use a credit based argument similar to that of Lemma 4.4.2. For each edge in $\mathcal{C}^\beta$ we associate a credit of $(\beta-1)$ times its weight and for each $\mathcal{C}^\alpha$ edge we maintain a credit of $(\alpha-1)$ times its weight. The proof for $\mathcal{C}^\beta$ edges is similar to that in Lemma 4.4.2. For $\mathcal{C}^\alpha$ edges, we cannot use a similar argument since the weight of the leaf being cut off, is not necessarily α times the weight of the edge to its parent. But consider all the edges cut off at a mother node s. From the algorithm we have $R_s + \sum_{j\in N_2}(p_j - c_{sj}) > \alpha B$. From this we see that even though each leaf is not heavy enough, the combined weight of all the leaves being cut off at a mother node is sufficient for a credit of $(\alpha-1)$ times the weight of the edges cut. Since we start with an initial credit of W, the result follows. □

Combining Lemmas 4.5.2 and 4.5.4, we obtain the following.

Lemma 4.5.5 $C = C^\beta + C^\alpha \le \dfrac{2}{\beta-1}W + \dfrac{\beta-\alpha}{\alpha-1}C^*$.

We need the following technical lemma before we prove the main theorem.

Lemma 4.5.6 *For $\beta \ge \alpha > 1$, the function*

$$\ell(\alpha,\beta) = \max\left\{\alpha, \frac{2(\beta+1)}{\beta-1}, \frac{2(\beta-\alpha)}{\alpha-1}\right\}$$

is minimized when

$$\alpha = \frac{2(\beta+1)}{\beta-1} = \frac{2(\beta-\alpha)}{\alpha-1}$$

The minimum value is 2.87 *when* $\alpha \sim 2.87$ *and* $\beta \sim 5.57$.

Proof. We observe that $f(\alpha,\beta) = \alpha$ is strictly increasing in α, $h(\alpha,\beta) = 2(\beta-\alpha)/(\alpha-1)$ is strictly decreasing in α, $g(\alpha,\beta) = 2(\beta+1)/(\beta-1)$ is strictly decreasing in β, and h

is strictly increasing in β. From this it is easy to verify that at the optimum point, both f and g must be equal to the optimum value. If either them is not the max-value of the max, then appropriately change α/β to make this happen, and note that this can only reduce the value of h. From this it follows that all three terms are equal at the optimum. Eliminating β from the above two equations gives us

$$\alpha^3 - \alpha^2 - 4\alpha - 4 = 0$$

which on solving yields the claimed values for α, β and the minimum. □

Theorem 4.5.7 *Using LPT to schedule the fragments produced by* BoundedCuts *with* $\alpha = 2.87$, *and* $\beta = 5.57$ *gives a performance ratio of* 2.87.

Proof. Using Lemma 4.5.5, we have

$$\begin{aligned} L_{\text{avg}} = \frac{W+C}{m} &\leq \left(W + \frac{2}{\beta-1}W + \frac{\beta-\alpha}{\alpha-1}C^*\right)/m \\ &\leq \max\left\{\frac{\beta+1}{\beta-1}, \frac{\beta-\alpha}{\alpha-1}\right\} \cdot (W+C^*)/m \\ &\leq \max\left\{\frac{\beta+1}{\beta-1}, \frac{\beta-\alpha}{\alpha-1}\right\} \cdot L^*_{\text{avg}}. \end{aligned}$$

The above equation upper bounds L_{avg} and Lemma 4.5.1 upper bounds M ($M \leq \alpha L^*_{\max}$). Plugging these bounds in Lemma 4.3.16 we obtain

$$\begin{aligned} \frac{L_{\max}}{L^*_{\max}} &\leq \max\left\{\alpha, 2\left(\max\left\{\frac{\beta+1}{\beta-1}, \frac{\beta-\alpha}{\alpha-1}\right\}\right)\right\} \\ &\leq \max\left\{\alpha, \frac{2(\beta+1)}{\beta-1}, \frac{2(\beta-\alpha)}{\alpha-1}\right\}. \end{aligned}$$

From Lemma 4.5.6, the right hand side of the above inequality is minimized at the values stated in the theorem, and this shows that $L_{\max}/L^*_{\max} \leq 2.87$. □

The performance ratio of BoundedCuts is tight. The example is similar to that for LocalCuts i.e. a star in which the center node with weight δ is connected by edges of weight 1 to $n-1$ leaves each of weight $\alpha = 2.87$. Suppose the star is scheduled on $m = n$ processors. The optimal schedule consists of cutting all edges to produce $n-1$ fragments of weight $1+\alpha$ and one fragment of weight $n-1+\delta$. Taking $n > 4$, $M^* = L^*_{\max} = n-1+\delta$. BoundedCuts will collapse all leaves and produce a single fragment of weight $(n-1)\alpha+\delta$

(since $B = L^*_{\max}$, this does not exceed αB). The performance performance ratio is therefore $((n-1)\alpha + \delta)/(n-1+\delta)$ which approaches α as δ goes to zero.

The results in this section rely on the fact that the bound B used in BOUNDEDCUTS satisfies $M^* \leq B \leq L^*_{\max}$. Since we do not know the optimal partition, we do not know M^* or $L^*_{\max}$. However, we can ensure that we try a value of B that is as close as we want to $L^*_{\max}$. The following theorem makes the idea more precise.

Theorem 4.5.8 *For any $\epsilon > 0$, we can ensure that we run* BOUNDEDCUTS *with a bound B satisfying $L^*_{\max} \leq B \leq (1+\epsilon)L^*_{\max}$. This yields a performance ratio of $(1+\epsilon)2.87$ with a running time of $O(\frac{1}{\epsilon} n \log n)$.*

Proof. From Propositions 4.3.14 and 4.3.15, $\max\{W/m, R\}$ is a lower bound on $L^*_{\max}$. Analysis of LOCALCUTS establishes that $L^*_{\max} \leq 3.56 \max\{W/m, R\}$. Thus, $A \leq L^*_{\max} \leq 3.56A$ where $A = \max\{W/m, R\}$. The quantity A can easily be computed in linear time. For each integer i such that $1 \leq i \leq \lceil(\ln 3.56)/\epsilon\rceil$ we run BOUNDEDCUTS with $B = (1+\epsilon)^i$ followed by LPT. Among all the schedules found we choose the best schedule. This guarantees that in some iteration $L^*_{\max} \leq B \leq (1+\epsilon)L^*_{\max}$. From the previous analysis, if we use such a bound, we get a performance ratio of $(1+\epsilon)2.87$. Each run of BOUNDEDCUTS followed by LPT takes $O(n \log n)$ time. $O(1/\epsilon)$ runs take $O(\frac{1}{\epsilon} n \log n)$ time. □

4.6 An Approximation Scheme for POT Scheduling

In this section we describe a polynomial time approximation scheme (PTAS) for the POT scheduling problem. The basic idea is to obtain a fragmentation that closely approximates the fragmentation of the optimal schedule. We can then use the PTAS for multi-processor scheduling of Hochbaum and Shmoys [55] on the approximate fragmentation.

For every fixed $\epsilon > 0$ we will describe an algorithm A_ϵ that achieves a $(1+\epsilon)$ approximation. For the rest of the section we will assume that we know $L^*_{\max}$. As discussed in the proof of Theorem 4.5.8 we can guess $L^*_{\max}$ to within an ϵ precision. The algorithm A_ϵ breaks the interval $(\epsilon, 1]$ into $s = \lceil(\log 1/\epsilon)/\epsilon\rceil$ intervals $(\epsilon = l_1, l_2], (l_2, l_3], \ldots, (l_s, l_{s+1} = 1]$ where $l_i = (1+\epsilon) \cdot l_{i-1}$. For a fragmentation F, let n_i^F denote the number of fragments with weight in the interval $(l_i \cdot L^*_{\max}, l_{i+1} \cdot L^*_{\max}]$. Algorithm A_ϵ "guesses" n_i^{OPT} for each $1 \leq i \leq s$. In effect it enumerates all tuples $(k_1, \ldots, k_s)$ where $1 \leq k_i \leq n$ and $\sum_i k_i \leq n$.

There are at most n^s such tuples to enumerate over and for each fixed ϵ this is polynomial in n. The following lemma establishes the requirements of an approximate fragmentation.

Definition 4.6.1 *Two fragmentations F and G are said to be* ϵ-equivalent *if $n_i^F = n_i^G$ for $1 \leq i \leq s$.*

Lemma 4.6.2 *Let F be a fragmentation that is ϵ-equivalent to some optimal fragmentation and satisfies $C \leq C^*$. Then the maximum load of an optimal schedule for the fragments of F is at most $(1+\epsilon)L^*_{\max}$.*

Proof. Let S be an optimal schedule for the tree T, and let G be its natural fragmentation. We assume that F is ϵ-equivalent to G. We create a schedule S' for F as follows. Replace each fragment in S, of weight greater than ϵ, with a fragment from F from the same interval. Since F is ϵ-equivalent to G this step succeeds. We schedule the rest of the fragments of F on top of this schedule in a greedy fashion using standard list scheduling [42]. We claim that the maximum load of S' is at most $(1+\epsilon)L^*_{\max}$. Our first observation is that after scheduling the large fragments the maximum load of S' is at most $(1+\epsilon)L^*_{\max}$. This is because we replace each fragment of weight x by a fragment of weight at most $(1+\epsilon)x$. Suppose scheduling the small fragments violates the required condition. Consider the first fragment, the scheduling of which, increases the maximum load to more than $(1+\epsilon)L^*_{\max}$. Since this fragment is of weight at most $\epsilon L^*_{\max}$, the processor on which it is scheduled must have already had a load of more than $L^*_{\max}$. Since list scheduling schedules the current job on the least loaded processor, the above fact implies that every processor has a load of more than $L^*_{\max}$. Therefore $L_{\text{avg}} > L^*_{\text{avg}}$ which contradicts the assumption of the lemma. Note that $L_{\text{avg}} = (W + C)/m \leq (W + C^*)/m = L^*_{\text{avg}}$. □

We can use the PTAS for multi-processor scheduling [55] to schedule the fragments produces by F. We can obtain a $(1+\delta)$ approximation for any fixed δ in polynomial time and by choosing δ to be ϵ, we obtain an overall $(1+\epsilon)^2$ approximation. Now we describe a *pseudo-polynomial* time algorithm that, given an ϵ, a guess for $L^*_{\max}$, and a guess $(n_1, \ldots, n_s)$ for an optimal fragmentation, finds a minimum communication cost fragmentation that is ϵ-equivalent, or decides that there is no such fragmentation. Using standard techniques we will show how we can modify our original instance to have polynomially bounded weights. We assume without loss of generality that the tree is binary and that all weights in the input are integers. As before let $W = \sum_i p_i$.

We first root the tree at some arbitrary node. The algorithm examines the nodes of the tree bottom up and computes a table at each node. The table at node v denoted by X_v is $(s+1)$ dimensional and has a total of $(n^s \cdot W)$ entries (note that W is assumed to be an integer). The entry $X_v(k_1, \ldots, k_s, B)$ stores the minimum communication cost necessary to fragment T_v (the subtree root rooted at v) such that

- the weight of the fragment containing v is at most B, and
- the number of fragments in each interval $(l_i \cdot L^*_{\max}, l_{i+1} \cdot L^*_{\max}]$ is at most k_i.

We store a value ∞ if such a fragmentation is not feasible. Let u and v be the children of a node r. Given tables X_v and X_u we claim that computing the table X_r is simple. There are 4 possibilities at r with regard to cutting or collapsing the edges e_{ru} and e_{rv}. For each of these possibilities we can compute the table X_r by examining all pairs of entries in X_u and X_v. We describe the computation when e_{ru} is collapsed and e_{rv} is cut. We initialize all entries of X_r to be ∞. Let $C_1 = X_u(g_1, \ldots, g_s, B_u)$ and $C_2 = X_v(h_1, \ldots, h_s, B_v)$. Let $(l_i, l_{i+1}]L^*_{\max}$ be the interval in which $(B_v + c_{rv})$ lies. Let t be the tuple $(g_1 + h_1, \ldots, g_i + h_i + 1, \ldots, g_s + h_s, B_u + p_r + c_{rv})$. We set

$$X_r(t) = \min(X_r(t), C_1 + C_2 + 2c_{rv}).$$

The justification for the above should be clear from our earlier definitions. The entries for the other possibilities are similarly computed. Given the table at the root we look up each entry and compute the fragmentation implied by the entry. This involves adding a 1 to the number of fragments in the interval that contains the size of the root's fragment. Thus we obtain the following lemma.

Lemma 4.6.3 *A minimum communication cost ϵ-equivalent fragmentation can be computed in $O(n^{2s+1}W^2)$ time.*

Proof. From the above discussion it follows that the table at each node can be computed in time proportional to the product of the table sizes of its children which is $O(n^{2s}W^2)$. Thus the table at the root can be computed in $O(n^{2s+1}W^2)$ time. □

Corollary 4.6.4 *A minimum communication cost ϵ-equivalent fragmentation for each possible valid tuple $(k_1, \ldots, k_s)$ can be computed in $O(n^{2s+1}W^2)$ time.*

Proof. The table at the root computes the cost for each tuple so we can obtain all the values in the same time. □

We now describe the global steps of the algorithm A_ϵ and prove its correctness.

1. Round each p_i to $p'_i = \lfloor p_i \cdot \frac{2n}{\epsilon W_{\text{avg}}} \rfloor$. Round each c_{ij} similarly to $c'_{ij} = \lfloor c_{ij} \cdot \frac{2n}{\epsilon W_{\text{avg}}} \rfloor$. Let T' be the modified tree.
2. Guess $L^{*'}_{\max}$.
3. Guess $(n_1, \ldots, n_s)$, the fragmentation used by the optimal for T'. Compute the minimum communication cost ϵ-equivalent fragmentation.
4. Using the algorithm in [55] compute a $(1+\epsilon)$ approximate schedule for the fragments produced.
5. Use the schedule computed for T' for T.

Lemma 4.6.5 *Algorithm A_ϵ can be implemented to run in $n^{O(\log(1/\epsilon)/\epsilon)}$ time.*

Proof. We observe that $W' = \sum_i p'_i$ is $O(nm/\epsilon)$. Thus the running time for finding the minimum communication cost fragmentation for all possible fragmentations, by Lemma 4.6.3, is $O(n^{2s+1}(nm/\epsilon)^2))$ which is $n^{O(\log(1/\epsilon)/\epsilon)}$. Since $L^{*'}_{\max} \leq W'$, to guess $L^{*'}_{\max}$ we enumerate over each of the $W' = O(nm/\epsilon)$ values. The running time of the multi-processor scheduling PTAS to obtain a $(1+\epsilon)$ approximation is also $n^{O(\log(1/\epsilon)/\epsilon)}$ and we use it $n^{O(\log(1/\epsilon)/\epsilon)}$ times. Thus the overall running time for A_ϵ is $n^{O(\log(1/\epsilon)/\epsilon)}$. □

Lemma 4.6.6 *Algorithm A_ϵ yields a $(1+4\epsilon)$ approximation ratio.*

Proof. Let T' be the tree obtained after rounding the weights as in Step 2. Let $L'_{\max}$ be the value of a schedule S for T'. Then for the same schedule (that is the same assignment of nodes to processors) we claim that $L_{\max}$, the value of the schedule for T, is at most $\frac{\epsilon W_{\text{avg}}}{2n} L'_{\max} + \epsilon L^*_{\max}$. To prove this, let S_k be the set of nodes assigned to processor k. Then

$$L'_k = \sum_{i \in S_k} p'_i + \sum_{(i,j) \in (S_k, V-S_k)} c'_{ij}$$

But then

$$L_k = \sum_{i \in S_k} p_i + \sum_{(i,j) \in (S_k, V-S_k)} c_{ij}$$

$$
\begin{aligned}
&\leq \sum_{i \in S_k} (p'_i + 1)\frac{\epsilon W_{\text{avg}}}{2n} + \sum_{(i,j)\in(S_k, V-S_k)} (c'_{ij} + 1)\frac{\epsilon W_{\text{avg}}}{2n} && (4.1)\\
&\leq \frac{\epsilon W_{\text{avg}}}{2n} L'_k + \frac{\epsilon W_{\text{avg}}}{2n}\Big(\sum_{i\in S_k} 1 + \sum_{(i,j)\in(S_k,V-S_k)} 1\Big) \\
&\leq \frac{\epsilon W_{\text{avg}}}{2n} L'_k + \frac{\epsilon W_{\text{avg}}}{2n} 2n && (4.2)\\
&\leq \frac{\epsilon W_{\text{avg}}}{2n} L'_k + \epsilon W_{\text{avg}} \\
&\leq \frac{\epsilon W_{\text{avg}}}{2n} L'_k + \epsilon L^*_{\max}
\end{aligned}
$$

We use the fact that $\lfloor x \rfloor \geq x - 1$ in 4.1, and in 4.2 we use the fact the total number of nodes and edges in a tree is bounded by $2n$. The last inequality uses the fact that $L^*_{\max} \geq W_{\text{avg}}$. For any schedule of T with value $L_{\max}$, the equivalent schedule for T' satisfies $L'_{\max} \leq \frac{2n}{\epsilon W_{\text{avg}}} L_{\max}$. This follows from our rounding procedure. In particular this implies

$$L^{*'}_{\max} \leq \frac{2n}{\epsilon W_{\text{avg}}} L^*_{\max} \tag{4.3}$$

Now we put things together to obtain our result. Let $L'_{\max}$ be value of the schedule obtained for T'. From Lemma 4.6.2 the optimal schedule value for the fragmentation in Step 3 is at most $(1+\epsilon)L^{*'}_{\max}$. We obtain a $(1+\epsilon)$ approximation in Step 4. Therefore $L'_{\max} \leq (1+\epsilon)^2 L^{*'}_{\max} \leq (1+3\epsilon)L^{*'}_{\max}$. From our observations above, the schedule for T obtained using the schedule for T' satisfies

$$
\begin{aligned}
L_{\max} &\leq \frac{\epsilon W_{\text{avg}}}{2n} L'_{\max} + \epsilon L^*_{\max} \\
&\leq \frac{\epsilon W_{\text{avg}}}{2n} (1+3\epsilon) L^{*'}_{\max} + \epsilon L^*_{\max} \\
&\leq (1+3\epsilon) L^*_{\max} + \epsilon L^*_{\max} \\
&\leq (1+4\epsilon) L^*_{\max}.
\end{aligned}
$$

□

Theorem 4.6.7 *There is a polynomial time approximation scheme for POT scheduling.*

4.7 Concluding Remarks

Parallel query optimization poses a variety of novel scheduling problems. We have developed a clean model that takes communication costs into account. In addition we have obtained approximation algorithms for a class of problems that have pipelined parallelism but no partitioned parallelism. The practical importance of the problems presented in this paper rests on the premise that communication is a significant component of the cost of processing a query in parallel. The reader is referred to Pirahesh et al. [89], Gray [45] and Hasan et al. [50] for such evidence.

It is possible to extend the ideas in this chapter to obtain a constant approximation ratio even if the tree has blocking edges, provided there are no communication costs on the blocking edges. Designing algorithms for the general case with communication costs on blocking edges presents the following difficulty. If an operator i precedes an operator j, i will need to communicate to j when j is scheduled even though the computation of i is over. For example i could compute a temporary table that j needs later. Existing techniques are not sophisticated enough to obtain provable good algorithms for such problems. Partitioned parallelism is widely used in parallel database systems and offers substantial speedups. Further it is necessary since the data is usually partitioned over several disks. Due to its importance it is worth while for future work to develop and analyze algorithms that allow this form of parallelism. See [51] for some preliminary observations on the complexity of the problem.

The parallelism-communication trade-off is not the only concern in parallel query optimization. We have assumed that a parallel machine consists of a set of processors that communicate over an inter-connect. Enhancing the machine model to incorporate disks and memories presents challenging problems. Hong [57] develops a method for balancing CPU and disk while ignoring communication costs. Garofalakis and Ionnadis [36, 37] use multi-dimensional work vectors to take into account the variety of resources required by database tasks. We address a basic scheduling problem that arises in their approach in Chapter 5.

Chapter 5

Approximability of Vector Packing Problems

5.1 Introduction

Multi-processor scheduling, bin packing, and the knapsack problem are three very well studied problems in combinatorial optimization. Their study has had a large impact on the field of approximation algorithms. In addition to their theoretical importance they have several applications such as load balancing, cutting stock, resource allocation to name a few. All of these problems involve packing items of different sizes into capacitated bins. In this chapter[1] we study *multi-dimensional* generalizations of these problems where the items to be packed are d-dimensional vectors and bins are d-dimensional as well. We obtain several approximability and inapproximability results that improve on earlier results. Though our primary motivation is multi-dimensional resource scheduling, an underlying problem that arises is the problem of maximizing the numbers of vectors that can be packed into a bin of fixed size. This is a special case of the the multi-dimensional knapsack problem that is equivalent to packing integer programs (PIPs) [93, 105]. PIPs are an important class of integer programs that capture several NP-Hard combinatorial optimization problems in graphs and hypergraphs including maximum independent set, hypergraph matchings, disjoint paths and so on.

A starting motivation for studying these problems came from recent interest [36, 37, 38]

[1]This chapter is joint work with Sanjeev Khanna [13].

in multi-dimensional resource scheduling problems in parallel query optimization. A favored architecture for parallel databases is the so called shared-nothing environment [24] where the parallel system consists of a set of independent processing units each of which has a set of time-shareable resources such as CPU, one or more disks, network controllers and others. No global resources are assumed and tasks communicate via messages on the interconnecting network. A task executing on one of these units has requirements from *each* of these resources and is best described as a multi-dimensional load vector. The resources on a machine are assumed to be preemptive and are time-shared by the set of tasks assigned to the machine. In most scheduling models, both in theory and practice, it is assumed that the load of a task is described by a single aggregate work measure. This simplification is done both to reduce the complexity of the scheduling problem and to overcome the difficulty of specifying the load vector accurately. However for large task systems that are typically encountered in database applications, ignoring the multi-dimensionality could lead to bad performance. This is especially so because tasks in these systems are typically large and could have skewed resource requirements. The work in [35, 36, 37, 38] demonstrates the practical effectiveness of the multi-dimensional approach for parallel query processing and scheduling continuous media databases.

A basic resource allocation problem that is considered in the above papers is the problem of scheduling d-dimensional vectors (tasks) on d-dimensional bins (machines) to minimize the maximum load on any dimension (the load on the most loaded resource). The load vector of a task describes its normalized resource requirements for each of the available resources. For a given assignment of tasks to machines, the load on a particular resource of a machine is defined to be the sum of the requirements for that resource, of all the tasks assigned to that machine. The objective function is based on the *uniformity assumption* that views tasks as requiring resources at a uniform rate described by the load. This assumption is applicable for large tasks when the resources are preemptable and was suggested by the work of Ganguly et al. [31] and Garofalakis et al. [36] where they show its empirical usefulness. Surprisingly, despite the large body of work on approximation algorithms for multi-processor scheduling and several variants [42, 73], the authors in [35] had to settle for a naive $(d+1)$ approximation for the above problem. For this problem we obtain a PTAS for every fixed d and a simpler $O(\log d)$ approximation that performs better than $(d+1)$ for all $d \geq 2$. The database application requires a solution to a more general problem where there are other constraints such as precedence constraints between the tasks [36]. Heuristics for the general problem are all based on using the basic problem that we solve as a subroutine, and our work addresses it as a first step.

Earlier work on resource constrained multi-processor scheduling [32] is based on a different resource usage model. In that model, each job j requires s resources and takes p_j units of processing time on a machine. It is assumed that there is a fixed amount of each resource, and that schedules are constrained in the following way. At all times, for any particular resource, the sum of the requirements for that particular resource of the jobs executing at that time is at most the resource bound. The goal is to schedule the jobs on a set of parallel machines to minimize makespan subject to the resource constraints mentioned above. In this model resources are *global* and are common to all machines. This model is appropriate for non-preemptable resources in a shared-everything environment, and is not applicable to shared-nothing systems with preemptable resources. Non-preemptable resources such as memory requirements of a job can be modeled as above and some theoretical and empirical work as applicable to databases on combining both types of resources is presented in [37].

The close relationship between multi-processor scheduling and bin packing extends to the multi-dimensional case. Thus we also consider the *generalized bin packing* problem or *vector bin packing* problem [33, 67] in which each item is a vector in $[0,1]^d$ and the objective is to pack the items into the minimum number of bins of capacity 1^d. In what follows, we formally define the problems that we study and provide a detailed description of our results.

5.1.1 Problem Definitions

We start by defining the vector scheduling problem. For a vector x, the quantity $\|x\|_\infty$ denotes the standard ℓ_∞ norm.

Definition 5.1.1 (Vector Scheduling (VS)) *We are given a set J of n d-dimensional vectors $p_1, \ldots, p_n$ from $[0,\infty)^d$ and a number m. A valid solution is a partition of J into m sets $A_1, \ldots, A_m$. The objective is to minimize $\max_{1\le i\le m} \|\bar{A}_i\|_\infty$ where $\bar{A}_i = \sum_{j\in A_i} p_j$ is the sum of the vectors in A_i.*

Definition 5.1.2 (Vector Bin Packing (VBP)) *Given a set of n vectors $p_1, \ldots, p_n$ in $[0,1]^d$, find a partition of the set into sets $A_1, \ldots, A_m$ such that $\|\bar{A}_i\|_\infty \le 1$ for $1 \le j \le m$. The objective is to minimize m, the size of the partition.*

The following definition of PIPs is from [105]. In the literature this problem is also referred to as the d-dimensional 0-1 knapsack problem [29].

Definition 5.1.3 (PIP) *Given* $A \in [0,1]^{d\times n}$*,* $b \in [1,\infty)^d$*, and* $c \in [0,1]^n$ *with* $\max_j c_j = 1$*, a packing integer program (PIP) seeks to maximize* $c^T \cdot x$ *subject to* $x \in \{0,1\}^n$ *and* $Ax \leq b$*. Furthermore if* $A \in \{0,1\}^{d\times n}$*, b is assumed to be integral. Finally B is defined to be* $\min_i b_i$*.*

The restrictions on A,b, and c in the above definition are without loss of generality: an arbitrary packing problem can be reduced to the above form (see [105]). We are interested in PIPs where $b_i = B$ for $1 \leq i \leq d$. When $A \in \{0,1\}^{d\times n}$ this problem is known as the *simple B-matching in hypergraphs* [78]: given a hypergraph with non-negative edge weights, find a maximum weight collection of edges such that no vertex occurs in more than B of them. When $B = 1$ this is the usual hypergraph matching problem. We note that the maximum *independent set* problem is a special case of the hypergraph matching problem with $B = 1$.

5.1.2 Related Work and Our Results

All the problems we consider are NP-Complete for $d = 1$ (multi-processor scheduling, bin packing, and the knapsack problem). The dimension of the vectors, d, plays an important role in determining the complexity. We concentrate on two cases, when d is fixed constant and when d is part of the input and can be arbitrary. Below is an outline of the various positive and negative results that we obtain for these problems.

Vector Scheduling: For the vector scheduling problem the best approximation algorithm [37] prior to our work had a ratio of $(d+1)$. When d is a fixed constant (a case of practical interest) we obtain a polynomial time approximation scheme (PTAS), thus generalizing the result of Hochbaum and Shmoys [55] for multi-processor scheduling. In addition we obtain a simpler $O(\log d)$ approximation algorithm that is better than $(d+1)$ for all $d \geq 2$. When d is large we give an $O(\log^2 d)$ approximation that uses as a subroutine, known approximation algorithms for PIPs. We also give a very simple $O(\log dm)$-approximation. Finally, we show that it is hard to approximate the VS problem to within *any* constant factor when d is arbitrary.

Vector Bin Packing: The previous best known approximation algorithms for this problem gave a ratio of $(d+\epsilon)$ for any fixed $\epsilon > 0$ [23] and $(d + 7/10)$ [33]; the latter result holds even in an on-line setting. All the ratios mentioned are asymptotic, that is there is an additive term of d. Karp et al. [67] do a probabilistic analysis and show bounds on the average wastage in the bins. We design an approximation algorithm that for any fixed $\epsilon > 0$, achieves a $(1 + \epsilon \cdot d + O(\ln \epsilon^{-1}))$-approximation in polynomial time, thus significantly improving upon the previous guarantees. One useful corollary of this result is that for a fixed d, we can approximate the problem to within a ratio of $O(\log d)$. Moreover, we show that even for $d = 2$ the problem is APX-Hard; an interesting departure from classical bin packing problem ($d = 1$) which exhibits an asymptotic FPTAS. Our hardness reduction also gives us an APX-Hardness result for the so-called *vector covering problem* (also known as the "dual vector packing" problem). Only NP-Hardness was known for these problems prior to our work.

Packing Integer Programs: For fixed d there is a PTAS for PIPs [29]. For large d randomized rounding technique of Raghavan and Thompson [93] yields integral solutions of value $t_1 = \Omega(\mathrm{OPT}/d^{1/B})$ and $t_2 = \Omega(\mathrm{OPT}/d^{1/(B+1)})$ respectively, if $A \in [0,1]^{d \times n}$ and $A \in \{0,1\}^{d \times n}$. Srinivasan [105] improved these results to obtains solutions of value $\Omega(t_1^{B/(B-1)})$ and $\Omega(t_2^{(B+1)/B})$ respectively (see discussion at the end of Section 5.4.1 concerning when these values are better). Thus the parameter B plays an important role in the approximation ratio achieved, with better ratios obtained as B gets larger (recall that entries in A are upper bounded by 1). It is natural to question if the dependence of the approximation ratio on B could be any better. We show that PIPs are hard to approximate to within a factor of $\Omega(d^{\frac{1}{B+1}-\epsilon})$ for every fixed B, thus establishing that randomized rounding essentially gives the best possible approximation guarantees. Hardness was known only for the case $B = 1$ earlier. An interesting aspect of our reduction is that the hardness result holds even when the optimal is restricted to choosing a solution that satisfies $Ax \leq 1^d$ while the approximation algorithm is only required to satisfy the relaxed constraint of $Ax \leq B^d$. We use Haståd's recent result [52] on approximating the independent set.

5.1.3 Organization

The rest of this chapter is organized as follows. Sections 5.2 and 5.3 present our approximation algorithms for the vector scheduling problem and the vector bin packing problem respectively. In Section 5.4 we present our hardness of approximation results for packing integer programs, vector scheduling, and vector bin packing and covering.

5.2 Approximation Algorithms for Vector Scheduling

In this section we describe our approximation algorithms for the vector scheduling problem. For any set of jobs A, we define $\bar{A}$ to be the vector sum $\sum_{j \in A} p_j$. The quantity $\bar{A}^i$ denotes component i of the vector $\bar{A}$. The sum of the components of a vector plays a role in our algorithms. We define the *volume* of a vector to be sum of its coordinates, and the volume of a set of vectors A, denoted by $\mathcal{V}(A)$, to be the sum of the volumes of the vectors in A.

5.2.1 Preliminaries

We first describe some simple *lower bounds* on the schedule length. For a given instance let OPT denote the optimal schedule length. We observe that the infinity norm of each of the job vectors is clearly a lower bound.

$$\text{OPT} \geq \max_{j \in J} \|p_j\|_\infty \tag{5.1}$$

The second lower bound is obtained by using the average volume per dimension.

$$\text{OPT} \geq \frac{\mathcal{V}(J)}{m \cdot d} \tag{5.2}$$

We can strengthen the above bound by splitting the sum dimension wise.

$$\text{OPT} \geq \max_{i=1}^{d} \frac{\bar{J}^i}{m}. \tag{5.3}$$

A very naive and simple algorithm for our problem is to ignore the multi-dimensional aspect of the jobs and treat them as a one dimensional vectors of size equal to the sum of their components. The dimensionality of the bins is also ignored. Then one can apply the standard list scheduling algorithm of Graham [43] for multi-processor scheduling to obtain the following theorem that uses the simple lower bounds developed above.

Theorem 5.2.1 *Applying list scheduling on the volumes of the vectors results in a schedule of height $L_{\max}$ where*

$$L_{\max} \leq \frac{\mathcal{V}(J)}{m} + \max_{j \in J} \|p_j\|_\infty$$

This yields a $(d+1)$ approximation.

Proof. The upper bound follows from standard analysis of list scheduling. The approximation ratio follows from the lower bounds in Equations 5.1 and 5.2. □

It is easy to construct examples that show that the above algorithm's analysis is tight. We describe two heuristics below that also have a worst case ratio of $(d+1)$ but are more intelligent than naive list scheduling based on volumes. They can be augmented with different tie breaking rules.

- Order jobs in some way. Schedule the next job in the order on the machine that results in the least increase in the schedule height.

- Among the unscheduled jobs schedule the job that results in the least increase in schedule height.

The above algorithms should perform reasonably well in practice especially for small d since they take the dimensionality of the jobs into account. This is in contrast to list scheduling based only on volumes.

Our goal is to obtain algorithms with improved theoretical guarantees. In the rest of the chapter we assume without loss of generality that the optimal schedule value is 1. We can guess the optimal value to an arbitrary precision via binary search since we have an upper and lower bound on the optimal schedule length that are within a multiplicative factor of d (using Theorem 5.2.1).

5.2.2 A PTAS for fixed d

Hochbaum and Shmoys [55] gave a PTAS for the multi-processor scheduling problem (VS problem with $d = 1$) using dual approximation schemes. We now show that a non-trivial generalization of their ideas yields a PTAS for arbitrary but fixed d.

The basic idea used in [55] is a primal-dual approach whereby the scheduling problem is viewed as a bin packing problem. If optimal solution can pack all jobs with load not exceeding some height h, assume $h = 1$ from here on, then the scheduling problem is to pack all the jobs into m bins (machines) of height 1. The authors then give an algorithm to solve this bin packing problem with bin height relaxed to $(1+\epsilon)$ for some fixed $\epsilon > 0$. In order to do so, they classify jobs into large or small depending on whether their size is greater than ϵ or not. Only a fixed number of large jobs can be packed into any bin. The sizes of the large jobs are discretized into $O(\log 1/\epsilon)$ classes and dynamic programming is used to pack all the large jobs into the m bins such that no bin exceeds a height of $(1+\epsilon)$. The small jobs are then greedily packed on top of the large jobs.

We take a similar view of the problem, our dual problem is vector bin packing. The primary difficulty in generalizing the above ideas to the case of vectors of $d \geq 2$ dimensions is the lack of a total order on the "size" of the jobs. It is still possible to classify vectors into large or small depending on their ℓ_∞ norm but the scheme of [55] does not apply. We need to take into account the interaction between the packing of large and small vectors. In addition the packing of small vectors is non-trivial. In fact we use a linear programming relaxation and a careful rounding to pack the small jobs. We describe our ideas in detail below. Following the above discussion we will think of machines as d-dimensional bins and the schedule length as bin capacity (height). Given an $\epsilon > 0$ and a guess for the optimal value (that we assume is normalized to 1), we describe an ϵ-relaxed decision procedure A_ϵ that either returns a schedule of height $(1+5\epsilon)$ or proves that the guess is incorrect. We can use A_ϵ to do a binary search for the optimal value. Let us define δ to be ϵ/d.

Preprocessing Step: Our first idea is to reduce to zero all coordinates of the vectors that are too small relative to the largest coordinate. This allows us to bound the ratio of the largest coordinate to the smallest non-zero coordinate.

Lemma 5.2.2 *Let I be an instance of the VS problem. Let I' be a modified instance where we replace each p_j in I with a vector q_j as follows. For each $1 \leq i \leq d$, $q_j^i = p_j^i$ if $p_j^i \geq \delta \|p_j\|_\infty$ and $q_j^i = 0$ otherwise. Then replacing the vector q_j by the vector p_j in any valid solution to I' results in a valid solution to I of height at most a factor of $(1+\epsilon)$ that of I'.*

Proof. Let S be the index set of a subset of vectors of I'. For a dimension i, let $A^i = \sum_{j\in S} q_j^i$. Without loss of generality assume that $A^1 \geq A^2 \ldots \geq A^d$. Then

$$\begin{array}{rcll}
\sum_{j\in S} p_j^i & \leq & \sum_{j\in S} q_j^i + \sum_{j\in S} \delta \cdot \|p_j\|_\infty & \text{(from definition of } q_j\text{)} \\
& = & A^i + \sum_{j\in S} \delta \cdot \|q_j\|_\infty & (\|q_j\|_\infty = \|p_j\|_\infty) \\
& \leq & A^i + \sum_{j\in S} \delta \sum_{1\leq k\leq d} q_j^k & \\
& = & A^i + \delta \sum_{1\leq k \leq d} A^k & \text{(by change of order of summation)} \\
& \leq & A^1 + \delta \cdot d \cdot A^1 & (A^k \leq A^1 \text{ for } 1 \leq k \leq d) \\
& \leq & (1+\epsilon) A^1 & (\delta = \epsilon/d).
\end{array}$$

If S is the subset of vectors in some bin, from the above equations it follows that replacing the q_j by p_j increases the height by only a multiplicative factor of $(1+\epsilon)$. Since this is true for all bins we obtain the desired result. $\square$

Assume that we have transformed our instance as described in the above lemma.

Large versus Small Vectors: The second step in the algorithm is to partition the vectors into two sets L and S corresponding to *large* and *small*. L consists of all vectors whose ℓ_∞ norm is greater than δ and S is the rest of the vectors. The algorithm A_ϵ will have two stages; the first stage packs all the large jobs, and the second stage packs the small jobs. Unlike the case of $d = 1$, the interaction between the two stages has to be taken in to account for $d \geq 2$. We show that the interaction can be captured in a compact way as follows. Let $(a_1, a_2, \ldots, a_d)$ be a d-tuple of integers such that $0 \leq a_i \leq \lceil 1/\epsilon \rceil$ for $1 \leq i \leq d$. We will call each such distinct tuple a *capacity configuration*. There are at most $t = (1 + \lceil 1/\epsilon \rceil)^d$ such configurations. Assume that the t capacity configurations are ordered in some way and let a_i^k be the value of coordinate i in tuple k. A space configuration approximately describes how a bin is filled. However we have m bins. A t-tuple $(m_1, \ldots, m_t)$ where $0 \leq m_i \leq m$ and $\sum_i m_i = m$ is called a *bin configuration* that describes the number of bins of each capacity configuration. The number of possible bin configurations is clearly $O(m^t)$. Since there are only a polynomial number of such configurations for fixed d and ϵ we can "guess" the configuration used by a feasible packing. A packing of vectors in a bin is said to *respect* a bin configuration $(a_1, \ldots, a_d)$ if the the height of the packing is each dimension i is less than ϵa_i. Given a capacity configuration we can define the corresponding *empty capacity configuration* as the tuple obtained by subtracting each entry from $(\lceil 1/\epsilon \rceil + 1)$. For a bin configuration M we denote by $\bar{M}$ the corresponding bin configuration as the one obtained by taking the empty capacity configurations for each of the bins in M.

Overview of the Algorithm: Algorithm A_ϵ performs the following steps for each bin configuration M:

- decide if vectors in L can be packed respecting M.
- decide if vectors in S can be packed respecting $\bar{M}$.

If both Steps 2 and 3 succeed for some M we have a packing of height at most $(1+\epsilon)$. If the decision procedure fails for all M we will prove that our guess for the optimal is false.

Packing the large vectors: The first stage consists of packing the vectors in L. Observe that the smallest non-zero coordinate of the vectors in L is at least δ^2. We partition the interval $[\delta^2, 1]$ into $q = \lceil \frac{2}{\epsilon} \log \delta^{-1} \rceil$ intervals of the form $(x_0, (1+\epsilon)x_0], (x_1, (1+\epsilon)x_1], \ldots, (x_{q-1}, 1]$ where $x_0 = \delta^2$ and $x_{i+1} = (1+\epsilon)x_i$. We discretize every non-zero coordinate of the vectors in L by rounding the coordinate down to the left end point of the interval in which it falls. Let L' be the resulting set of vectors.

Lemma 5.2.3 *Let I' be an instance obtained from the original instance I by rounding vectors in L as described above. Then replacing each vector in L' by the corresponding vector in L in any solution for I' results in a solution for I of height at most $(1+\epsilon)$ times that of I'.*

Proof. Each coordinate of a vector in L' is at least $(1+\epsilon)^{-1}$ times the coordinate of the corresponding vector in L. The lemma follows trivially. □

Vectors in L' can be classified into one of $s = (1 + \lceil \frac{2}{\epsilon} \log \delta^{-1} \rceil)^d$ distinct classes. Any packing of the vectors into one bin can be described as a tuple $(k_1, k_2, \ldots, k_s)$ where k_i indicates the number of vectors of the ith class. Note that at most d/δ vectors from L' can be packed in any bin. Therefore $\sum k_i \leq d/\delta$. Thus there are at most $(d/\delta)^s$ *configurations*. A configuration is *feasible* for a capacity configuration if the vectors described by the configuration can be packed without violating the height constraints described by the capacity configuration. Let C_k denote the set of all configurations of the discretized jobs in L that are feasible for the kth capacity configuration. From our discussion earlier $|C_k| \leq (d/\delta)^s$.

Lemma 5.2.4 *Let $M = (m_1, m_2, \ldots, m_t)$ be a bin configuration. There exists an algorithm with running time $O((d/\delta)^s mn^s)$ to decide if there is a packing of the jobs in L' that respects M.*

Proof. We use a simple dynamic programming based algorithm. Observe that number of vector classes in L' is at most s. Thus any subset of vectors from L' can be specified by a tuple of size s and there are $O(n^s)$ distinct tuples. The algorithm orders bins in some arbitrary way and with each bin assigns a capacity configuration from M. For $1 \leq i \leq m$, the algorithm computes all possible subsets of vectors from L' (tuples) that can be packed in the first i bins. For each i this information can be maintained in $O(n^s)$ space. Given the tuples for bin i, the tuples for bin $(i+1)$ can be computed in $O(d/\delta)^s$ time per tuple since that is an upper bound on the number of feasible configurations for any capacity configuration. Thus for each bin i, in $O(d/\delta)^s n^s)$ time, we can compute the tuples that can packed into the first i bins given the information for bin $(i-1)$. The number of bins is m so we get the required time bound. □

Packing the small vectors: We now describe the second stage, that of packing the vectors in S. For the second stage we write an integer programming formulation and round the resulting LP relaxation to find an approximate feasible solution. Without loss of generality assume that the vectors in S are numbered 1 to $|S|$. The IP formulation has 0-1 variables x_{ij} for $1 \le i \le |S|$ and $1 \le j \le m$. Variable x_{ij} is 1 if p_i is assigned to machine j. Every vector has to be assigned to some machine. This results in the following equation.

$$\sum_j x_{ij} = 1 \qquad\qquad 1 \le i \le |S| \tag{5.4}$$

Given a bin configuration M we can define for each machine j and dimension k a height bound b_j^k that an assignment should satisfy such that no bin exceeds a height of 1. Thus we obtain

$$\sum_i p_i^k \cdot x_{ij} \le b_j^k \qquad\qquad 1 \le j \le m, 1 \le k \le d. \tag{5.5}$$

In addition we have the integrality constraints, namely, $x_{ij} \in \{0,1\}$. We obtain a linear program by replacing these constraints by the following.

$$x_{ij} \ge 0 \tag{5.6}$$

Proposition 5.2.5 *Any* basic feasible solution *to the LP defined by Equations 5.4, 5.5, and 5.6 has at most $d \cdot m$ vectors that are assigned fractionally to more than one machine.*

Proof. The number of variables in our LP is $n \cdot m$. The number of non-trivial constraints (those that are other than $x_{ij} \ge 0$) is $(n + d \cdot m)$. From standard polyhedral theory [96] any basic (vertex) solution to our LP has $n \cdot m$ tight constraints. Therefore by a simple counting argument at most $(n + d \cdot m)$ variables can be strictly positive. Since each vector is assigned to at least one machine, the number of vectors that are fractionally assigned to more than one machine is at most $d \cdot m$. □

We can solve the above linear program in polynomial time and obtain a basic feasible solution. Let S' be the set of vectors that are not assigned integrally to any machine. By the above lemma $|S'| \leq d \cdot m$. We assign the vectors in S' in a round robin fashion to the machines that ensures that no machine gets more than d vectors each from S'. However since $\|p_j\|_\infty \leq \delta = \epsilon/d$ for every $p_j \in S'$, the above step does not violate the height by more than ϵ in any dimension.

Putting it Together: We are now ready to prove our main theorem.

Lemma 5.2.6 *If a bin height of* 1 *is feasible,* A_ϵ *returns a packing with height at most* $(1 + 5\epsilon)$.

Proof. Consider the instance modified according to Lemma 5.2.2 and Lemma 5.2.3. A feasible schedule of height 1 is still feasible for the modified instance. Let M be the bin configuration induced by the packing of the large jobs L in a feasible packing. Since we use discrete sizes, a bin configuration approximates the packing requirements of L to within an additive error of ϵ. Further the packing of the small vectors S is feasible for $\bar{M}$. Therefore for the choice of M we can pack both L and S. We have an additive error of ϵ in packing S. Thus A_ϵ packs the modified instance with a bin height of $(1+2\epsilon)$. By Lemma 5.2.2 and Lemma 5.2.3, a packing of the modified instance with height $(1+2\epsilon)$, implies a packing of the original instance in height $(1+2\epsilon) \cdot (1+\epsilon)^2$ which is less than $(1+5\epsilon)$ for sufficiently small ϵ. □

Lemma 5.2.7 *The running time of* A_ϵ *is* $(nd/\epsilon)^{O(s)}$ *where* $s = O((\frac{\log(d/\epsilon)}{\epsilon})^d)$.

Proof. The running time is clearly dominated by the time to pack the large vectors L. There are at most $m^t = O(n^{O(\epsilon^{-d})})$ bin configurations and for each of those configurations, using Lemma 5.2.4, feasibility can be checked in $(nd/\epsilon)^{O(s)}$ time. The claim follows. □

Theorem 5.2.8 *For every fixed d there is a PTAS for the vector scheduling problem.*

Proof. We can use A_ϵ do a binary search for the optimal value. The naive list scheduling algorithm gives an estimate of the optimal to within a factor of $(d+1)$. Thus in $O(\log(d/\epsilon))$ calls to A_ϵ we obtain a $(1+\epsilon)$ approximation. The running time is polynomial for every fixed d and ϵ and therefore we have the required PTAS. □

We have not attempted to obtain the best possible running time for the PTAS for two reasons. First, we do not believe that an approximation scheme is practically useful directly, especially for the database application that motivated it. Second, from a theoretical point of view our main goal was to understand the approximability of the problem and the best possible polynomial running time is not necessary for that goal.

5.2.3 The General Case

We now consider the case when d is arbitrary and present two approximation algorithms for this case. The first algorithm is deterministic and has an approximation ratio that is only a function of d ($O(\log^2 d)$) while the second algorithm is randomized and achieves an approximation ratio that is a function of both d and m ($O(\log dm)$). We once again assume that the optimal schedule height is 1.

An $O(\log^2 d)$ Approximation

The basic ideas underlying our approximation algorithm are the following. Recall that the volume of a vector is defined to be the sum of its coordinates and similarly the volume of a set of vectors A denoted by $\mathcal{V}(A)$ is simply sum of the volumes of the vectors in A. Consider an optimal schedule for a given instance. By simple averaging arguments, some machine k in that schedule satisfies the condition

$$\mathcal{V}(J(k)) \geq \mathcal{V}(J)/m,$$

where $J(k)$ is the set of vectors assigned to machine k. We use approximation algorithms for PIPs to find a set of jobs of total volume $\mathcal{V}(J)/m$ that can be packed on a machine in height close to 1. We then remove these jobs and find another set of jobs to pack in the second machine, and so on (that is if jobs are left). A stage ends when we have used all the m machines. Standard arguments allow us to argue that the volume of the jobs left after a stage is at most a constant factor of the initial volume. We repeat the stages $\Omega(\log d)$ stages to reduce the volume of the jobs not scheduled to less than $\mathcal{V}(J)/d$. Then we use the naive list scheduling algorithm on the remaining jobs. Before we state the algorithm formally we need a couple of definitions. The following problem is a special case of a general PIP.

Definition 5.2.9 *Given a set J of n vectors in $[0,1]^d$, the* largest volume packing problem *is the problem of finding a subset S such that $\|\bar{S}\|_\infty \leq 1$ and $\mathcal{V}(S)$ is maximized. Let $\mathcal{V}_{\max}$ denote the value of the optimal solution.*

Definition 5.2.10 *An (α, β) approximation to the largest volume packing problem is a subset S that satisfies the conditions $\|\bar{S}\|_\infty \leq \alpha$ and $\mathcal{V}(S) \geq \beta \mathcal{V}_{\max}$.*

We give below a pseudo-code description of our algorithm.

Algorithm Greedy-Pack

1. **repeat** for t stages
   (a) **foreach** machine $1 \leq k \leq m$ **do**
      i. Find an (α, β) approximation to the largest volume packing problem with the current set of job vectors.
      ii. Allocate jobs in packing to machine i and remove them.
2. Find an independent schedule for the remaining jobs using naive volume based list scheduling and allocate them to the machines on which they are scheduled.

We now prove several simple lemmas to analyze the performance of Greedy-Pack. Observe that $\beta \leq 1$ in the above algorithm.

Lemma 5.2.11 *Let J^i be the set of jobs remaining at the beginning of the ith stage with $J^1 = J$. Let J^i_{k+1} be the set of jobs remaining after machine $(k+1)$ has been packed in stage i. Then*

$$\mathcal{V}(J^i_{k+1}) \leq \mathcal{V}(J^i) \cdot (1 - \beta/m)^{k+1}$$

Proof. We prove the lemma by induction on k. The claim is trivially true for $k = 0$. Suppose the claim is true up to machine k. We will show that the claim is true for machine $(k+1)$. Since all jobs in J can be scheduled on m machines with height 1 it follows that all jobs in J^i_k can be likewise scheduled. By a simple averaging argument we can infer that there exists a set of jobs in J^i_k with volume at least $\mathcal{V}(J^i_k)/m$ that can be packed in a machine with height at most 1. Since we obtain a β approximation to largest volume packing, we pack jobs a volume of at least $\beta \cdot \mathcal{V}(J^i_k)/m$. Therefore $\mathcal{V}(J^i_{(k+1)}) \leq \mathcal{V}(J^i_k) \cdot (1 - \beta/m)$. By our induction hypothesis $\mathcal{V}(J^i_k) \leq \mathcal{V}(J^i) \cdot (1 - \beta/m)^k$. The lemma follows. □

Corollary 5.2.12 *If J^i be the set of jobs remaining at the beginning of stage i then*

$$\mathcal{V}(J^i) \leq \mathcal{V}(J)/e^{(i-1)\beta}.$$

Proof. From our definitions $J^{(i+1)} = N^i_m$. From Lemma 5.2.11, $\mathcal{V}(J^i_m) \leq \mathcal{V}(J^i) \cdot (1 - \beta/m)^m$. Since $(1 + t/n)^n \leq e^t$ for any t we get the required bound. □

Theorem 5.2.13 *Greedy-Pack yields a schedule of height at most* $(t \cdot \alpha + \frac{d}{e^{t\beta}} + 1)$.

Proof. Let $J_1(k)$ and $J_2(k)$ be the set of jobs allocated to machine k in the packing stage and the list scheduling stage respectively. From the packing property it is easy to see that the height of machine k due to jobs in $J_1(k)$ is at most $t\alpha$. Let J' be the set of jobs remaining after the t stages of packing that are scheduled using list scheduling. From Corollary 5.2.12 we have that

$$\mathcal{V}(J') \leq \mathcal{V}(J)/e^{t\beta}.$$

From Theorem 5.2.1, the height increase of machine k due to jobs in $J_2(k)$ is at most

$$\mathcal{V}(J')/m + \max_j \|p_j\|_\infty \leq \frac{d}{e^{t\beta}} \cdot \mathcal{V}(J)/(dm) + 1 \leq \frac{d}{e^{t\beta}} + 1.$$

In the above inequality we are using the fact that the two lower bounds are less than 1, the optimal value. Combining the two equations gives us the desired bound. □

The parameter t in the algorithm can be chosen as a function of α and β to obtain the best ratio. Note that the largest volume packing problem is a special case of a PIP where c_i is simply the volume of vector i. PIPs have a $(1/2, O(\log d))$ approximation via randomized rounding [93, 105] that can be derandomized by techniques from [92]. When d is fixed there is a $(1, 1-\epsilon)$ approximation [29] that runs in time polynomial in $n^{d/\epsilon}$. These observations lead to the following two corollaries.

Corollary 5.2.14 *There is an* $O(\log^2 d)$ *approximation algorithm for the VS problem.*

Corollary 5.2.15 *There is an* $O(\log d)$ *approximation algorithm for the VS problem that runs in time polynomial in* n^d.

An $O(\log dm)$ approximation

The approximations in Corollary 5.2.14 are good when d is small compared to m, that is when $\log d = o(\sqrt{\log m})$. However when d is large we can obtain a $O(\log dm)$ approximation by a simple randomized algorithm that assigns each vector independently to a machine chosen uniformly at random from the set of m machines. We call this algorithm Random.

Theorem 5.2.16 *Random gives an $O(\log dm)$ approximation with high probability.*

Proof. Consider the first machine. Let X_j be the indicator random variable that is 1 if vector j is assigned to the first machine. The X_j are independent. By uniformity $\mathbf{Pr}[X_j = 1] = 1/m$. Let $P = \sum_j p_j X_j$. Note that P is a vector since each p_j is a vector: let P^i denote the ith coordinate of P. By linearity of expectations $\mathbf{E}[P^i] = \sum_j p_j^i/m \leq$ OPT (using Equation 5.2). Also observe that $\max_j p_j^i \leq$ OPT (using Equation 5.1). Now we estimate the probability that P^i deviates significantly from its expected value. By Chernoff bounds $\mathbf{Pr}\left[P^i > (\mathbf{E}[P^i] + \max_j p_j^i)(1+c)\log dm\right] \leq (dm)^{-c}$. Thus with high probability P^i is $O(\log dm) \cdot$ OPT. If A_k^i is the event that the ith dimension of machine k is greater than $2(1+c)\log dm \cdot$ OPT, then from above we know that $\mathbf{Pr}[A_k^i] \leq (dm)^{-c}$. Thus $\mathbf{Pr}\left[(A = \cup_{i=1}^d \cup_{k=1}^m A_k^i) \leq dm(dm)^{-c}\right]$. By choosing c sufficiently large we can ensure that $\mathbf{Pr}[A]$ is less than an inverse polynomial factor. But the complement of A is the event that the schedule length is $O(\log dm) \cdot$ OPT. Thus with high probability we get a $O(\log dm)$ approximation. □

5.3 Vector Bin Packing

We now examine the problem of packing a given set of vectors into smallest possible number of bins. Our main result here is as follows:

Theorem 5.3.1 *For any fixed $\epsilon > 0$, we can obtain in polynomial time a $(1 + \epsilon \cdot d + O(\ln(\epsilon^{-1}))$-approximate solution for vector bin packing.*

This improves upon the long standing $(d+\epsilon)$-approximation algorithm of [23]. Our approach is based on solving a linear programming relaxation for this problem. As in Section 5.2.2, we use a variable x_{ij} to indicate if vector p_i is assigned to bin j. We guess the least number of bins m (easily located via binary search) for which the following LP relaxation is feasible; clearly $m \leq$ OPT.

$$\begin{aligned}
\sum_j x_{ij} &= 1 && 1 \leq i \leq n \\
\sum_i p_i^k \cdot x_{ij} &\leq 1 && 1 \leq j \leq m\ ,\ 1 \leq k \leq d \\
x_{ij} &\geq 0 && 1 \leq i \leq n\ ,\ 1 \leq j \leq m
\end{aligned}$$

Once again, we use the fact that a basic feasible solution would make fractional bin assignments for at most $d \cdot m$ vectors. Thus at this point, all but a set S of at most $d \cdot m$ vectors have integral assignments in m bins. To find a bin assignment for S, we repeatedly find a set $S' \subseteq S$ of up to $k = \lceil 1/\epsilon \rceil$ vectors that can all be packed together and assign them to a new bin. This step is performed greedily, i.e. we seek to find a largest possible such set in each iteration. We can perform this step by trying out all possible sets of vectors of cardinality less than $(k+1)$. We now claim that this procedure must terminate in $\epsilon \cdot d \cdot m + O(\ln \epsilon^{-1}) \cdot \text{OPT}$ steps. To see this, consider the first time that we pack less than k vectors in a bin. The number of bins used thus far is bounded by $(d \cdot m)/k$. Moreover, the total number of vectors that remain at this point is at most $(k-1)\text{OPT}$; let S' denote this remaining set of vectors. Since the optimal algorithm can not pack more than $(k-1)$ vectors of S' in one bin, our greedy bin assignment procedure is identical to a greedy set cover algorithm where each set has size at most $(k-1)$. Thus the total number of bins used in packing vectors in S' is bounded by $H_{k-1} \cdot \text{OPT}$ [49] (H_i is the ith harmonic number). Putting things together, we obtain that the number of bins used by our algorithm, A, is bounded as follows:

$$A \leq m + (d \cdot m)/k + H_{k-1} \cdot \text{OPT} \leq (1 + \epsilon \cdot d + O(\ln \epsilon^{-1})) \cdot \text{OPT}.$$

This completes the proof of Theorem 5.3.1. Substituting $\epsilon = 1/d$, we obtain the following simple corollary:

Corollary 5.3.2 *For any arbitrary but fixed d, vector bin packing can be approximated to within $O(\ln d)$ in polynomial time.*

Using a simple argument, the result of the previous theorem can be strengthened to an $O(\sqrt{d})$-approximation when the vectors are drawn from the space $\{0,1\}^n$. We study this special case since it essentially captures hard instances of VBP for large values of d.

Theorem 5.3.3 *If each vector $p_i \in \{0,1\}^n$, then we can obtain a $(2\sqrt{d})$-approximation for vector bin packing.*

Proof. Partition the vectors into two sets S and L where the set S contains vectors which have a 1 in at most $\sqrt{d}$ dimensions while L contains the rest. Denote by OPT_S (OPT_L) the number of bins needed for an optimal packing of jobs in S (L); clearly $\text{OPT} \geq$

$(\text{OPT}_S + \text{OPT}_L)/2$, and $\text{OPT}_L \geq |L|/\sqrt{d}$. Now consider a greedy packing of jobs in S, followed by a greedy picking of jobs in L. Since each job in S has a 1 in at most $\sqrt{d}$ dimensions, it rules out up to $\sqrt{d}$ other jobs from being packed. Thus S can be greedily packed in at most $\text{OPT}_S\sqrt{d}$ bins. On the other hand, the greedy procedure uses at most $|L|$ bins to pack jobs in L. Combining the two observations, we obtain that the total number of bins used is at most $2\sqrt{d} \cdot \text{OPT}$. □

The above result is essentially tight for arbitrary d, since via a reduction from graph coloring, we can show a $d^{1/2-\epsilon}$-hardness for this problem, for any fixed $\epsilon > 0$.

5.4 Inapproximability Results

In this section we show hardness of approximation results the three problems we consider, vector scheduling, vector bin packing, and packing integer programs.

5.4.1 Packing Integer Programs

Randomized rounding techniques of Raghavan and Thompson [93] yield integral solutions of value $t_1 = \Omega(\text{OPT}/d^{1/B})$ and $t_2 = \Omega(\text{OPT}/d^{1/(B+1)})$ respectively, if $A \in [0,1]^{d\times n}$ and $A \in \{0,1\}^{d\times n}$. Srinivasan [105] improved these results to obtains solutions of value $\Omega(t_1^{B/(B-1)})$ and $\Omega(t_2^{(B+1)/B})$ respectively. We show that PIPs are hard to approximate to within a factor of $\Omega(d^{\frac{1}{B+1}-\epsilon})$ for every fixed integer B. We start by focusing on the case $A \in \{0,1\}^{d\times n}$ and then indicate how our result extends to $A \in [0,1]^{d\times n}$. Our reduction uses the recent result of Håstad [52] that shows that independent set is hard to approximate within a factor of $n^{1-\epsilon}$ for any fixed $\epsilon > 0$, unless $NP = ZPP$. Since the upper bounds are in terms of d, from here on, we will express the inapproximability factor only as a function of d.

To motivate our reduction, we start by sketching an elementary reduction that shows hardness of the problem when $B = 1$. This case is equivalent to stating the well-known fact that PIPs capture the maximum independent set problem. Given a graph $G = (V, E)$ with n vertices $v_1, \ldots, v_n$ we create an instance of a PIP I_G as follows. We create a $d \times n$ matrix A with zero-one entries that has $d = n^2$ rows, one for each pair of vertices of G, and n columns, one for each vertex of G. Let $r_i = (v_{i_1}, v_{i_2})$ be the pair associated with row i. Let a_{ij} denote the entry in the ith row and jth column of A. We set a_{ij} to 1 if (v_{i_1}, v_{i_2}) is an edge of G and v_j is incident on that edge. Otherwise we set a_{ij} to 0. Thus A is essentially the incidence matrix of G. We set the vectors c and b to be the all ones vector. It is clear that given G, I_G can be constructed in polynomial time. If we interpret the PIP in our vector terminology, we have vectors of n^2 dimensions one for each of the n vertices. The goal is to pack as many vectors as possible (since c is the all ones vector) in a d-dimensional bin of height 1.

Proposition 5.4.1 *There is a* 1-1 *correspondence between independent sets of G and feasible integer solutions for I_G where the value of a feasible solution is equal to the size of its corresponding independent set size.*

Corollary 5.4.2 *Unless $NP = ZPP$, PIPs with $B = 1$ are hard to approximate to within a factor of $n^{1-\epsilon}$ for every fixed $\epsilon > 0$. Alternatively, PIPs with $B = 1$ are hard to approximate to within a factor of $d^{1/2-\epsilon}$.*

Proof. Follows from Proposition 5.4.1 and the hardness of independent set [52]. □

Since the upper bounds are in terms of d, from here on, we will express the inapproximability factor only as a function of d. Our goal now is to extend the above result for larger values of B. Notice that in the above construction if we used $B = 2$ instead of 1, it is possible to pack all the vertices. Thus we need a different construction, one that enforces stronger constraints on vertices connected together in G. Our starting point is once again the maximum independent set problem.

Given a graph G and a positive integer B, we construct an an instance of a PIP I_G as follows. Create a $d \times n$ zero-one matrix A with $d = n^{(B+1)}$ such that each row corresponds to an element from $V^{(B+1)}$. Let $r_i = (v_{i_1}, \ldots, v_{i_{(B+1)}})$ denote the tuple associated with the ith row of A. We set a_{ij} to 1 if and only if the following conditions hold, otherwise we set it to 0:

- the vertex v_j occurs in r_i, and
- the vertices in r_i induce a *clique* in G.

We set c to $\{1\}^n$ and b to $\{B\}^d$. For any fixed integer B, the reduction can be done in polynomial time. Note that a feasible solution to I_G can be described as a set of indices $S \subseteq \{1, \ldots, n\}$.

Lemma 5.4.3 *Let S be any feasible solution to I_G and let G_S be the subgraph of G induced by the set of vertices v_i such that $i \in S$. Then $\omega(G_S) \leq B$ where $\omega(G_S)$ is the clique number of G_S.*

Proof. Suppose there is a clique of size $(B+1)$ in G_S; w.l.o.g. assume that $v_1, \ldots, v_{(B+1)}$ are the vertices of that clique. Consider the tuple $(v_1, v_2, \ldots, v_{(B+1)})$ and let i be the row of A corresponding to the above tuple. Then by our construction, $a_{ij} = 1$ for $1 \leq j \leq (B+1)$. There are $(B+1)$ vectors in S with a 1 in the same dimension i, violating the ith row constraint. This contradicts the feasibility of S. $\square$

Lemma 5.4.4 *Let $X \subseteq V$ be an independent set of G. Then $S = \{i \mid v_i \in X\}$ is a feasible solution to I_G of value $|S| = |X|$. Furthermore S can be packed with a height bound of 1.*

Proof. Suppose that in some dimension the height induced by S is greater than 1. Let r be the tuple associated with this dimension. Then there exist $i, j \in S$ such that $v_i, v_j \in r$ and $(v_i, v_j) \in E$. This contradicts the assumptions that X is an independent set. $\square$

The following is a simple Ramsey type result; we prove it here for the sake of completeness.

Lemma 5.4.5 *Let G be a graph on n vertices with $\omega(G) \leq k$. Then $\alpha(G) \geq n^{1/k}$ where $\alpha(G)$ is the size of a maximum independent set in G.*

Proof. By induction on k. Base case with $k = 1$ is trivial. Assume hypothesis is true for integers up to $k-1$. Consider a graph with $\omega(G) = k$. If the degree of every vertex in G is less than $n^{(k-1)/k}$, then any *maximal* independent set has size at least $n^{1/k}$. Otherwise, consider a vertex v in G that has degree at least $n^{(k-1)/k}$. Let G' be the subgraph of G induced by the neighbors of v. Since $\omega(G') \leq k-1$, by the induction hypothesis, $\alpha(G') \geq (n^{(k-1)/k})^{1/(k-1)} \geq n^{1/k}$. $\square$

Corollary 5.4.6 *Let S be any valid solution to I_G of value $t = |S|$. Then $\alpha(G) \geq t^{1/B}$.*

Proof. Follows from Lemmas 5.4.4 and 5.4.5. $\square$

Theorem 5.4.7 *Unless $NP = ZPP$, for every fixed integer B and fixed $\epsilon_0 > 0$, PIPs with bound $b = \{B\}^d$ and $A \in \{0,1\}^{d \times n}$ are hard to approximate to within a factor of $d^{\frac{1}{B+1} - \epsilon_0}$. PIPs with $A \in [0,1]^{d \times n}$ and B rational are hard to approximate to within a factor of $d^{\frac{1}{\lfloor B \rfloor + 1} - \epsilon_0}$.*

Proof. We first look at the case of PIPs with $A \in \{0,1\}^{d\times n}$. Notice that our reduction produces only such instances. Suppose there is a polynomial time approximation algorithm $\mathcal{A}$ for PIPs with bound B that has an approximation ratio $d^{\frac{1}{B+1}-\epsilon_0}$ for some fixed $\epsilon_0 > 0$. This can be reinterpreted as a $d^{\frac{1-\epsilon}{B+1}}$-approximation where $\epsilon = \epsilon_0(B+1)$ is another constant. We will obtain an approximation algorithm $\mathcal{G}$ for the maximum independent set problem with a ratio $n^{1-\delta}$ for $\delta = \epsilon/B$. The hardness of maximum independent [52] will then imply the desired result. Given a graph G, the algorithm $\mathcal{G}$ constructs an instance I_G of a PIP as described above and gives it as input to $\mathcal{A}$. $\mathcal{G}$ returns $\max(1, t^{1/B})$ as the independent set size of G where t is the value returned by $\mathcal{A}$ on I_G. Note that by Corollary 5.4.6, $\alpha(G) \geq t^{1/B}$ which proves the correctness of the algorithm. Now we prove the approximation guarantee. We are interested only in the case when $\alpha(G) \geq n^{1-\delta}$ for otherwise a trivial independent set of size 1 gives the required approximation ratio. From Lemma 5.4.4 it follows that the optimal value for I_G is at least $\alpha(G)$. Since $\mathcal{A}$ provides a $d^{\frac{1-\epsilon}{B+1}}$ approximation $t \geq \alpha(G)/d^{\frac{1-\epsilon}{(B+1)}}$. In the construction of I_G $d = n^{(B+1)}$. Therefore $t \geq \alpha(G)/n^{(1-\epsilon)}$. Simple algebra verifies that $t^{1/B} \geq \alpha(G)/n^{1-\delta}$ when $\alpha(G) \geq n^{1-\delta}$.

Now we consider the case of PIPs with $A \in [0,1]^{d\times n}$. Let B be some real number. For a given B we can create an instance of a PIP as before with $B' = \lfloor B \rfloor$. The only difference is that we set $b = B^d$. Since all entries of A are integral, effectively the bound is B'. Therefore it is hard to approximate to within a factor of $d^{(1-\epsilon)/(B'+1)} = d^{(1-\epsilon)/(\lfloor B \rfloor+1)}$. If $(\lfloor B + 1/d \rfloor + 1) = B+1$ then $d^{(1-\epsilon)/(\lfloor B \rfloor+1)} = \Theta(d^{(1-\epsilon)/B})$. □

Discussion: An interesting aspect of our reduction above is that the hardness results holds even when the optimal algorithm is restricted to a height bound of 1 while allowing a height bound of B for the approximation algorithm. Let an (α,β)-bicriteria approximation be one that satisfies the relaxed constraint matrix $Ax \leq \alpha b$ and gets a solution of value at least $\textsc{opt}/\beta$, here $\textsc{opt}$ satisfies $Ax \leq b$. Then we have the following corollary:

Corollary 5.4.8 *Unless $NP = ZPP$, for every fixed integer B and fixed $\epsilon > 0$, it is hard to obtain a $(B, d^{\frac{1}{B+1}-\epsilon})$ bicriteria approximation for PIPs.*

For a given B, we use $d = n^{B+1}$, and a hardness of $d^{\frac{1}{B+1}-\epsilon}$ is essentially the hardness of $n^{1-\epsilon}$ for independent set. This raises two related questions. First, should d be larger than n to obtain the inapproximability results? Second, should the approximability (and inapproximability) results be parameterized in terms of n instead of d? These questions are important to understand the complexity of PIPs as d varies from $O(1)$ to poly(n). We observe that the hardness result holds as long as d is $\Omega(n^\epsilon)$ for some fixed $\epsilon > 0$. To see this, observe that in our reduction, we can always add poly(n) dummy columns (vectors) that are either useless (their c_j value is 0) or cannot be packed (add a dummy dimension where only B of the dummy vectors can be packed). Thus we can ensure that $n \geq \text{poly}(d)$ without changing the essence of the reduction. We have a PTAS when $d = O(1)$ and a hardness result of $d^{1/(B+1)}$ when $d = \text{poly}(n)$. An interesting question is to resolve the complexity of the problem when $d = \text{polylog(n)}$.

As remarked earlier, Srinivasan [105] improves the results obtained using randomized rounding to obtain solutions of value $\Omega(t_2^{(B+1)/B})$ where $t_2 = \Omega(y^*/d^{1/(B+1)})$ for $A \in \{0,1\}^{d \times n}$. In the above y^* is the optimal fractional solution to the PIP. It might appear that this contradicts our hardness result but a careful examination will reveal that for the instances we create in our reduction $y^*/d^{1/(B+1)} \leq 1$. For such instances Srinivasan's bounds do not yield an improvement over randomized rounding.

5.4.2 Vector Scheduling

We now extend the ideas used in the hardness result for PIPs to show hardness results for the vector scheduling problem. Our result here uses the hardness of graph coloring; Feige and Kilian [28], building on the work of Haståd [52], show that graph coloring is $n^{1-\epsilon}$-hard unless $NP = ZPP$. Our reduction is motivated by the observation that graph coloring is basically partitioning the graph into independent sets. We need the following elementary lemma.

Lemma 5.4.9 *Let G be a graph on n vertices with $\omega(G) \leq k$. Then $\chi(G) \leq O(n^{1-1/k} \log n)$ where $\chi(G)$ is the chromatic number of G.*

Proof. From Lemma 5.4.5 $\alpha(G) \geq n^{1/k}$. Let G' be the graph obtained by removing a largest independent set from G. It is easy to see that $\omega(G') \leq k$. Thus we can apply Lemma 5.4.5 again to G' to remove another large independent set. We can repeat this process until we are left with a single vertex and standard arguments show that the process terminates after $O(n^{1-1/k} \log n)$ steps. Thus we can partition $V(G)$ into $O(n^{1-1/k} \log n)$ independent sets and the lemma follows. □

Theorem 5.4.10 *Unless $NP = ZPP$, for every constant $\gamma > 1$, there is no polynomial time algorithm that approximates the schedule height in the vector scheduling problem to within a factor of γ.*

Let $B = \lceil \gamma \rceil$; we will show that it is hard to obtain a B-approximation using a reduction from chromatic number. Given graph G we construct an instance I of the VS problem as follows. We construct n vectors of n^{B+1} dimensions as in the proof of Theorem 5.4.7. We set m, the number of machines, to be $n^{\frac{1}{2B}}$.

Lemma 5.4.11 *If $\chi(G) \leq m$ then the optimal schedule height for I is 1.*

Proof. Let $V_1, \ldots, V_{\chi(G)}$ be the color classes. Each color class is an independent set and by Lemma 5.4.4 the corresponding vectors can be packed on one machine with height at most 1. Since $\chi(G) \leq m$ the vectors corresponding to each color class can be packed in a separate machine. □

Lemma 5.4.12 *If the schedule height for I is bounded by B then $\chi(G) \leq \beta n^{1-1/2B} \log n$ for some fixed constant β.*

Proof. Let $V_1, V_2, \ldots, V_m$ be the partition of vertices of G induced by the assignment of the vectors to the machines. Let G_i be the subgraph of G induced by the vertex set V_i. From Lemma 5.4.3 we have $\omega(G_i) \leq B$. Using Lemma 5.4.9 we obtain that $\chi(G_i) \leq \beta n^{1-1/B} \log n$ for $1 \leq i \leq m$. Therefore it follows that $\chi(G) \leq \sum_i \chi(G_i) \leq m \cdot \beta n^{1-1/B} \log n \leq \beta n^{1-1/2B} \log n$. □

Proof of Theorem 5.4.10. Feige and Kilian [28] showed that unless $ZPP = NP$, for every $\epsilon > 0$ there is no polynomial time algorithm to approximate the chromatic number to within a factor of $n^{1-\epsilon}$. Suppose there is a B approximation for the VS problem. Lemmas 5.4.11 and 5.4.12 establish that if $\chi(G) \leq n^{1/2B}$ then we can infer by running the B-approximation algorithm for the VS problem that $\chi(G) \leq \beta n^{1-1/2B} \log n$. This implies a $\beta n^{1-1/2B} \log n$ approximation to the chromatic number. From the result of [28] it follows that this is not possible unless $NP = ZPP$. □

5.4.3 Vector Bin Packing

In this section we prove that the vector bin packing problem is APX hard even for $d = 2$. In contrast the standard bin packing problem with $d = 1$ has an asymptotic PTAS[2] [23, 66]. Thus our result shows that bin packing with $d > 1$ has no PTAS and in particular no asymptotic PTAS. Using a similar reduction we obtain APX hardness for the vector covering problem [2] as well. We use a reduction from the optimization version of bounded 3-Dimensional matching (3DM) problem [34]. We define the problem formally below.

Definition 5.4.13 (3-bounded 3DM (3DM-3)) *Given a set $T \subseteq X \times Y \times Z$. A matching in T is a subset $M \subseteq T$ such that no elements in M agree in any coordinate. The goal is to find a matching in T of largest cardinality. A 3-bounded instance is one in which the number of occurrences of any element of $X \cup Y \cup Z$ in T is at most 3.*

Kann [64] showed this problem is Max-SNP Complete (hence also APX-Complete). Our reduction is based on the ideas used in the NP-Completeness reduction for showing that the 3DM problem reduces to the 4-Partition problem [34]. We now describe the reduction.

Let $q = \max\{|X|, |Y|, |Z|\}$, $u = |X \cup Y \cup Z|$ and $t = |T|$. We create a total of $(u + t)$ 2-dimensional vectors. We have one vector each for the elements of T and one for each of the elements of $X \cup Y \cup Z$. Let x_i, y_j, z_k, w_l denote the corresponding vectors for X, Y, Z, and T respectively. The first coordinates are assigned as as follows.

$$\begin{aligned} x_i^1 &= q^4 + i && 1 \leq i \leq |X| \\ y_j^1 &= q^4 + jq && 1 \leq j \leq |Y| \\ z_k^1 &= q^4 + kq^2 && 1 \leq k \leq |Z| \end{aligned}$$

The first dimension of the an element $l = (i, j, k)$ in T is assigned as follows

$$w_l^1 = q^4 - kq^2 - jq - i.$$

The second dimension of each of the vectors is obtained simply by subtracting the first dimension from $2q^4$. We will assume without loss of generality that $q \geq 4$ in which case

[2] An asymptotic PTAS for a minimization problem implies that for every fixed $\epsilon > 0$ there is an approximation algorithm that guarantees a solution of value $(1 + \epsilon)\text{OPT} + c$ where c is an absolute constant that does not depend on the instance. This differs from the standard definition of a PTAS in that a fixed *additive* term is allowed in addition to the multiplicative factor. See [81] for more details and discussion.

all coordinates are positive. Our goal is to pack these vectors in two dimensional bins with capacity $4q^4$.

Proposition 5.4.14 *Any* three *vectors in the instance can be packed in a bin.*

Proof. It is clear from the construction that the largest vector coordinate is bounded by $(q^4 + q^3 + q^2 + q)$. For $q \geq 4$ it is easily verified that $3(q^4 + q^3 + q^2 + q) \leq 4q^4$. $\square$

An element of T is referred to as a *hyper-edge*. The elements of a hyper-edge e are the elements of $X \cup Y \cup Z$ that form the tuple e.

Proposition 5.4.15 Four *vectors can be packed in a bin if and only if they correspond to a hyper-edge and its elements.*

Proof. It is easy to verify that the vectors corresponding to a hyper-edge and its elements can be packed in a bin. To prove the only if direction, assume that the coordinates of four vectors that are packed are $(a_1, 2q^4 - a_1), \ldots, (a_4, 2q^4 - a_4)$. The four vectors satisfy the capacity constraints of the bin. Therefore we obtain that $\sum_i a_i \leq 4q^4$ and $\sum_i (2q^4 - a_i) \leq 4q^4$. These two inequalities imply that $\sum_i a_i = 4q^4$. It is a simple observation from our construction that such an equality hold only for the vectors corresponding to a hyper-edge and its elements. $\square$

Proposition 5.4.15 establishes that bins with four vectors each correspond to a matching. The proof of the following theorem provides a quantitative relationship between the size of an optimal matching and an optimal packing of the vectors.

Theorem 5.4.16 *Vector bin packing in* 2 *dimensions is APX-Complete.*

Proof. VBP in 2 dimensions is in APX by Theorem 5.3.1. We will show APX hardness via an L-reduction [85] (recall Definition 1.1.2) from 3-bounded 3DM to VBP with $d = 2$. Let $\mathcal{I}$ be an instances of 3DM-3. We map $\mathcal{I}$ to an instance $\mathcal{I}'$ of VBP as described in our reduction above. Let m^* be the size of the largest matching in $\mathcal{I}$ and let b^* be the smallest number of bins needed for $\mathcal{I}'$. By Propositions 5.4.15 and 5.4.14 it is easily verified that $b^* = \lceil (t + u - m^*)/3 \rceil$. We claim that there exists a constant $\alpha > 0$ such that $b^* \leq \alpha m^*$ for any instance of 3DM-3. This follows from the fact that $u = O(t)$ (due to 3-boundedness) and $t = O(m^*)$ (due to membership in Max-SNP). Now to map a solution for $\mathcal{I}'$ to a solution for $\mathcal{I}$, we define the matching to be the hyper-edges corresponding to the vectors in T that are packed in bins with 4 vectors each. Proposition 5.4.15 guarantees the correctness of the solution. We assume without loss of generality that all bins except one have at least three vectors each. Thus if m is the number of bins with 4 vectors packed, then b, the total number of bins, is $\lceil (t+u-m)/3 \rceil$. Thus we obtain that

$$\begin{aligned} b - b^* &= \lceil (t+u-m)/3 \rceil - \lceil (t+u-m^*)/3 \rceil \\ &\geq (t+u-m)/3 - (t+u-m^*)/3 + 1 \\ &\geq (m^* - m)/3 + 1 \end{aligned}$$

In other words, there exists a constant β such that for all instances $|m^* - m| \leq \beta |b^* - b|$. It follows that 3DM-3 L-reduces to VBP. □

Vector Covering

A similar reduction as above shows hardness of the *vector covering problem* defined below.

Definition 5.4.17 (Vector Covering (VC)) *Given a set of n rational vectors $p_1, \ldots, p_n$ from $[0,1]^d$, find a partition of the set into sets $A_1, \ldots, A_m$ such that $\bar{A}^i_j \geq 1$ for $1 \leq i \leq d$ and $1 \leq j \leq m$. The objective is to maximize m, the size of the partition.*

The vector covering problem is also referred to as the dual bin packing problem and the one-dimensional version was first investigated in the thesis of Assman [6]. Both on-line and off-line versions have been studied in various papers [7, 22, 21, 2, 113] mostly for the one-dimensional case. We concentrate on the off-line approximability of the problem. For $d = 1$, a PTAS was obtained by Woeginger [113] that improved upon the constant factor algorithms of Assman et al. [7]. Alon et al. [2] gave a $\min\{d, 2\ln d/(1+o(1))\}$ approximation algorithm for arbitrary d. However no hardness of approximation results were known prior to our work. We will use the bin packing terminology: a partition corresponds to bins that are *covered* by the vectors. The basic idea of the reduction is the same as the one above for bin packing. The following proposition is analogous to Proposition 5.4.14.

Proposition 5.4.18 *Any* five *vectors can cover a bin.*

Proof. The smallest coordinate in our construction is at least $(q^4 - q^3)$. For $q \geq 5$, $5(q^4 - q^3) \geq 4q^4$. □

The proof of the following proposition is very similar to that of Proposition 5.4.15, hence we omit it.

Proposition 5.4.19 Four *vectors can cover a bin if and only if they correspond to a hyper-edge and its elements.*

Theorem 5.4.20 *The Vector covering problem is APX-Complete for* $d = 2$.

Proof. Membership in APX follows from the result of Alon et al. [2]. To show hardness we use the same reduction as the one for vector bin packing. If b is the solution value for the VC instance $\mathcal{I}'$ that corresponds to an instance $\mathcal{I}$ of 3DM-3, we claim that there is a matching of size m in $\mathcal{I}$ where m is the largest value that satisfies $b = \lfloor (t + u + m)/5 \rfloor$. Using Proposition 5.4.18 we assume without loss of generality that the solution for $\mathcal{I}'$ is maximal in that every bin that is covered has at most five vectors and that at most four vectors are left out of the solution. We simply take the hyper-edges corresponding to those bins that are covered by exactly four vectors. Proposition 5.4.19 guarantees that we have a valid matching. Simple algebra verifies that if m is the number of bins covered by four vectors then $b = m + \lfloor (t + u - 4m)/5 \rfloor = \lfloor (t + u + m)/5 \rfloor$.

Let m^* and b^* be the optimal solution values to $\mathcal{I}$ and $\mathcal{I}'$ respectively. The arguments above show that $b^* = \lfloor (t + u + m^*)/5 \rfloor$. As in the proof of Theorem 5.4.16 we claim that there is a constant $\alpha > 0$ such that $b^* \leq \alpha m^*$ using the 3-boundedness and Max-SNP membership of 3DM-3. We can relate the gap in b and b^* to that in m and m^* as

$$\begin{aligned} b^* - b &= \lfloor (t + u + m^*)/5 \rfloor - \lfloor (t + u + m)/3 \rfloor \\ &\geq (t + u + m^*)/5 - (t + u + m)/5 - 1 \\ &\geq (m^* - m)/5 - 1 \end{aligned}$$

In other words, there exists a constant $\beta > 0$ such that for all instances $|m^* - m| \leq \beta |b^* - b|$. It follows that 3DM-3 L-reduces to the VC problem with $d = 2$. □

5.5 Concluding Remarks

We obtained several results on the approximability of natural vector packing problems. However many interesting questions remain. Though we obtained a PTAS for the vector scheduling problem for every fixed d, the running time is impractical. Although desirable, we do not believe that a running time of the form $f(d) \cdot \text{poly}(n^{1/\epsilon})$, for a $(1+\epsilon)$ approximation, is achievable. Some investigation into the fixed parameter tractability [25] of the problem may shed light on this issue. For small values of d we believe that a constant factor approximation with much improved running times is possible using the ideas in the PTAS. As mentioned earlier the database application requires a solution to the problem where tasks have tree like precedence constraints. There are further constraints on the assignment of tasks to machines that depend on the assignment of a task's predecessors (see [36] for more details). If we have only precedence constraints a $(d+1)$ approximation follows from standard list scheduling analysis but we do not know if a constant factor approximation independent of d can be obtained for this problem. For the vector bin packing problem an algorithm with a running time polynomial in n^d that gives a constant factor approximation ratio, independent of d, is of interest. Currently we have only an $O(\log d)$ approximation. Our hardness results for PIPs apply when B is a fixed constant. The main difficulty in extending the result to larger values of B is in extending the Ramsey type result we use to show the existence of a large independent set in a graph that excludes small cliques. We believe that, by using stronger graph products, it is possible to extend the result to values of B up to $\Omega(\log n / \log \log n)$. Finally, for the problems we considered, our results shed light on the extreme cases of d fixed and d large (that is polynomial in n). However we do not yet know the complexity of the problems for an intermediate case when d is logarithmic in n.

Chapter 6

Conclusions

We presented approximation algorithms for several NP-Hard scheduling problems. The problems considered in Chapter 4 and Chapter 5 are new problems suggested by practical applications. Our work on the problem of minimizing average completion time in Chapter 2 is also motivated by the practical problem of profile driven instruction scheduling in compiler optimization. We were able to use intuition from theoretically sound approximation algorithms to develop heuristics that performed well in the real application [12]. Our quest for obtaining simpler and more efficient algorithms that did not use linear programming relaxations was also partly due to the motivating application. We believe that this has led to a better understanding of the approximability of the problem.

Scheduling issues are fundamental in many diverse applications and novel problems, variants, and models will continue to come up. A good understanding of the complexity of basic scheduling problems is both necessary and useful for future applications. For example our approximation schemes for the problems in Chapters 4 and 5 are based on earlier ideas in the PTAS for multi-processor scheduling [55]. Scheduling theory has been an active area of research for the last four decades and impressive progress has been made on several fundamental problems. Despite that many open problems and challenges remain (see the survey articles [73, 46, 65]). At the end of each chapter we point out specific open problems related to the topics addressed in that chapter. Here we point out some broader directions for future research.

A broad class of challenging scheduling problems whose complexity is not well understood is that of scheduling jobs in the presence of precedence constraints. Many scheduling variants are well solved (have small constant factor approximations or a PTAS) when there are no precedence constraints. These include scheduling to minimize makespan or average completion time on a variety of machine environments (single machine, parallel identical machines, uniformly related machines, and unrelated machines). An exception is the problem of scheduling to minimize flow time [69, 76]. When jobs have precedence constraints, the same problems become harder to approximate and harder to design approximation algorithms for (the two notions are not necessarily the same). For example, there is a PTAS for multiprocessor scheduling while there is a lower bound of $4/3$ on the approximability for the same problem when jobs have precedence constraints. A 2 approximation for precedence constraints is simple using Graham's list scheduling while the PTAS involves sophisticated ideas. On the other hand, for minimizing makespan on unrelated machines, a 2 approximation and a $3/2$ hardness result for independent jobs are known, while nothing but a trivial n approximation is known if jobs have precedence constraints.

To understand the main bottleneck in the design of approximation algorithms for these problems, we look at the analysis of the well known Graham's list scheduling algorithm [43] for minimizing makespan on parallel identical machines. His analysis shows that, in any schedule produced by list scheduling, we can identify a chain of jobs $j_1 \prec j_2 \ldots \prec j_r$ such that, a machine is idle only when one of the jobs in the above chain is being processed. The time spent processing the chain is a lower bound for the optimal makespan. In addition, the sum total of time intervals during which all machines are busy is also a lower bound via arguments about the average load. These two bounds provide an upper bound of 2 on the approximation ratio of list scheduling. Three decades after Graham's simple analysis appeared we still do not have an algorithm with a better ratio. It is easy to show that the two lower bounds used in the above analysis cannot provide a ratio better than 2. The lower bounds cannot distinguish between the following two types of instances: those for which the optimal schedule keeps all machines busy most of the time and those for which the optimal schedule has a lot of idle time because of precedence constraints. Thus we need a stronger lower bound, potentially via some linear programming relaxation, that is more sensitive to the given instance than the gross lower bounds above. In a more direct way the same bottleneck hinders better algorithms for more complicated parallel environments such as uniformly related machines and unrelated machines. In these environments, since the machines are not identical, in addition to ordering jobs according to their priority, the algorithm also has to decide for each job the machine on which to execute it. Until the algorithm of Chudak and Shmoys [17] and subsequently ours in Chapter 3, algorithms for scheduling on uniformly related machines were all list scheduling based, and the best of them achieved a ratio of $O(\sqrt{m})$ [62]. Though the algorithm in [17] and ours have an improved $O(\log m)$ ratio, the lower bounds provide only partial information for scheduling. We believe that there exists a constant factor approximation for this problem. The situation for unrelated machines is worse and only a trivial n approximation is known. Similar difficulties exist for job shop and flow shop scheduling problems.

Thus an important direction for future work is to find stronger lower bounds for scheduling problems with precedence constraints. In this direction the work of Schulz [98, 97] and subsequently others [47, 17, 82] on completion time variables seems promising. Completion time formulations have a variable C_j for each job j that denotes the completion time of job j. Jobs are ordered according to their completion times and this ordering is used in the scheduling algorithm. This approach has been successful in obtaining good approximation algorithms for minimizing sum of weighted completion times. However, as remarked earlier we need more than on ordering for complex machine models. For the related machines case Chudak and Shmoys [17] use additional variables in their formulation that provide partial information on the assignment of jobs to machines. Future work in this direction using stronger linear programming or semi-definite relaxations might yield improved algorithms. As an intermediate step towards solving the unrelated machine scheduling problem, the following special case is interesting. We are given n jobs to be scheduled on m machines. The machines are partitioned into K sets $S_1, \ldots, S_k$. The instance also specifies, for each job j, an assignment $k(j)$ and the job j is constrained to run on a machine from set $S_{k(j)}$. All the machines in each partition are identical. Assuming all jobs are unit length (arbitrary length jobs can also be considered), the objective is to find a schedule of minimum makespan. This problem appears as a sub-problem in [17] although for a different purpose. In addition to its appealing simplicity it has applications to instruction scheduling in compilers. Modern CPUs have different functional units such as integer, floating point, and load/store and each of them executes the appropriate set of instructions in parallel. Dependencies between instructions force precedence constraints and the correspondence between the abstract model and the application should be clear. A simple generalization of Graham's list scheduling yields a $(K+1)$ approximation for this problem. Improving the ratio for this problem will yield insights for other problems as well.

Designing approximation algorithms with improved ratios is only one side of the coin in understanding the approximability of a problem. Proving hardness of approximation bounds is the other side. For most problems proving hardness of approximation results is harder than proving approximation bounds. Breakthrough work in the recent past by several authors, based on new characterizations of NP via interactive proof systems, has led to a better understanding of the hardness of some basic optimization problems. These results also established canonical hard problems that enable new results via approximation preserving reductions. Surprisingly, many of the known hardness of approximation results for scheduling problems are based on direct reductions to NP-Hard problems. However, for several fundamental problems such as precedence constrained scheduling to minimize makespan on related and unrelated machines, job shop and flow shop scheduling, and others that we mentioned in the concluding remarks of earlier chapters, the known results are weak. It might be possible that the lack of progress in obtaining better approximation algorithms can be explained by improved hardness results. It is worthwhile to explore if the sophisticated techniques based on PCP and others yield improved hardness results for scheduling problems.

Finally, we have an observation regarding average completion time scheduling. Minimizing makespan is a special case of the problem of minimizing average weighted completion time, and thus the latter problem is harder to approximate. Several single machine variants are NP-Hard for average completion time but are trivially solvable for makespan. However, for many scheduling models, the approximation ratio for minimizing average weighted completion time is no more than a constant multiplicative factor away from the ratio for minimizing makespan. Further Stein and Wein [106] show that, for a very general class of scheduling models, there exists a schedule that is simultaneously within a factor of 2 of the optimal schedule values for both average weighted completion time and makespan. Their proof is based on transforming an optimal schedule for average completion time to a schedule that is approximately good for both objective functions. Is there a converse to their transformation? That is, is there a polynomial time algorithm that uses as a subroutine a procedure for minimizing makespan and outputs an approximate schedule for minimizing average weighted completion time?

Bibliography

[1] D. Adolphson. Single machine job sequencing with precedence constraints. *SIAM Journal on Computing*, 6:40–54, 1977.

[2] N. Alon, J. Csirik, S. V. Sevastianov, A. P. A. Vestjens, and G. J. Woeginger. On-line and off-line approximation algorithms for vector covering problems. *Algorithmica*, 21(1):104–18, 1998.

[3] S. Arora. *Probabilistic checking of proofs and the hardness of approximation problems.* PhD thesis, U.C. Berkeley, 1994.

[4] S. Arora and C. Lund. Hardness of approximations. In D. S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 10, pages 399–446. PWS Publishing Company, 1995.

[5] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and intractability of approximation problems. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 14–23, 1992.

[6] S. F. Assman. *Problems in Discrete Applied Mathematics.* PhD thesis, Mathematics Department, MIT, 1983.

[7] S. F. Assman, D. S. Johnson, D. J. Kleitman, and J. Y. T. Leung. On a dual version of the one-dimensional bin packing problem. *Journal of Algorithms*, 5:502–525, 1984.

[8] Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis.* Cambridge University Press, 1998.

[9] S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. In *Proceedings of the 23rd ICALP*, LNCS, pages 646–57. Springer Verlag, 1996.

[10] C. Chekuri and M. Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. In R. E. Bixby, E. A. Boyd, and R. Z. Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, volume 1412 of *Lecture Notes in Computer Science*, pages 383–393. Springer, 1998.

[11] C. Chekuri, W. Hasan, and R. Motwani. Scheduling problems in parallel query optimization. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 255–65, May 1995.

[12] C. Chekuri, R. Johnson, R. Motwani, B. K. Natarajan, B. R. Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with applications to super blocks. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, pages 58–67, 1996.

[13] C. Chekuri and S. Khanna. On multi-dimensional packing problems. Submitted for publication, July 1998.

[14] C. Chekuri and R. Motwani. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine. Submitted for publication, August 1997.

[15] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 609–618, 1997.

[16] F. Chudak and D. Hochbaum. A half-integral linear programming relaxation for scheduling precedence-constrained jobs on a single machine. Submitted for publication, August 1997.

[17] F. Chudak and D. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 581–590, 1997.

[18] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–80, 1990.

[19] P. Crescenzi, R. Silvestri, and L. Trevisan. On the query complexity of complete problems in approximation classes. Unpublished manuscript, 1994.

[20] P. Crescenzi and L. Trevisan. On approximation scheme preserving reducibility and its applications. In *Proceedings of the 14th Ann. Conf. on Foundations of Software Tech. and Theoret. Comp. Sci.*, volume 880 of *Lecture Notes in Computer Science*, pages 330–341. Springer Verlag, 1994.

[21] J. Csirik, J. B. G. Frenk, G. Galambos, and A. H. G. Rinnooy Kan. Probabilistic analysis of algorithms for dual bin packing problems. *Journal of Algorithms*, 12(2):189–203, 1991.

[22] J. Csirik and V. Totik. On-line algorithms for a dual version of bin packing. *Discrete Applied Mathematics*, 21:163–67, 1988.

[23] W. Fernandez de la Vega and G. S. Lueker. Bin packing can be solved within $1+\epsilon$ in linear time. *Combinatorica*, 1:349–355, 1981.

[24] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[25] R. G. Downey and M. R. Fellows. Fixed-parameter tractability and completeness. I. Basic results. *SIAM Journal on Computing*, 24(4):873–921, Aug 1995.

[26] J. Du, J. Y. T. Leung, and G. H. Young. Scheduling chain structured tasks to minimize makespan and mean flow time. *Information and Computation*, 92:219–236, 1991.

[27] Philippe Chretienne et al. (editors). *Scheduling theory and its applications.* John Wiley & Sons, 1995.

[28] U. Feige and J. Kilian. Zero knowledge and the chromatic number. In *Proceedings of the Eleventh Annual IEEE Conference on Computational Complexity*, pages 278–287, 1996.

[29] A. M. Frieze and M. R. B. Clarke. Approximation algorithms for the m-dimensional 0-1 knapsack problem: worst-case and probabilistic analyses. *European Journal of Operational Research*, 15(1):100–9, 1984.

[30] G. Gallo, M. D. Grigoriadis, and R. Tarjan. A fast parametric maximum flow algorithm and applications. *SIAM Journal on Computing*, 18:30–55, 1989.

[31] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, June 1992.

[32] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, June 1975.

[33] M. R. Garey, R. L. Graham, D. S. Johnson, and A. C. Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory Ser. A*, 21:257–298, 1976.

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[35] Minos N. Garofalakis and Yannis E. Ioannidis. Scheduling issues in multimedia query optimization. *ACM Computing Surveys*, 27(4):590–92, December 1995.

[36] Minos N. Garofalakis and Yannis E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 365–76, June 1996.

[37] Minos N. Garofalakis and Yannis E. Ioannidis. Parallel query scheduling and optimization with time-and space-shared resources. In *Proceedings of the 23rd VLDB Conference*, pages 296–305, 1997.

[38] Minos N. Garofalakis, Banu Ozden, and Avi Silberschatz. Resource scheduling in enhanced pay-per-view continuous media databases. In *Proceedings of the 23rd VLDB Conference*, pages 516–525, 1997.

[39] M. X. Goemans. Improved approximation algorithms for scheduling with release dates. In *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 591–598, 1997.

[40] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 2–11, 1997.

[41] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[42] R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Tech. J.*, 45:1563–81, 1966.

[43] R. L. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.

[44] 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. *Ann. Discrete Math.*, 5:287–326, 1979.

[45] J. Gray. The cost of messages. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 1–7, August 1988.

[46] L. Hall. Approximation algorithms for scheduling. In D. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*, chapter 1, pages 1–45. PWS Publishing, 1995.

[47] L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Offline and online algorithms. *Math. of Operations Research*, 22:513–544, 1997.

[48] L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Offline and online algorithms. In *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 142–151, 1996.

[49] M. M. Halldórsson. Approximating k-set cover and complementary graph coloring. In *Proceedings of Fifth IPCO Conference on Integer Programming and Combinatorial Optimization*, volume 1084 of *LNCS*, pages 118–131. Springer Verlag, 1996.

[50] W. Hasan and R. Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 36–47, September 1994.

[51] Waqar Hasan. *Optimizing Response Time of Relational Queries by Exploiting Parallel Execution*. PhD thesis, Stanford University, 1995.

[52] J. Håstad. Clique is hard to approximate to within $n^{1-\epsilon}$. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, pages 627–636, 1996.

[53] J. L. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5:422–448, 1983.

[54] D. S. Hochbaum. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, 1995.

[55] D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[56] D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, 17:539–551, 1988.

[57] W. Hong. *Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays.* PhD thesis, University of California, Berkeley, August 1992.

[58] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 218–225, December 1991.

[59] J. A. Hoogeveen, P. Schuurman, and G. J. Woeginger. Non-approximability results for scheduling problems with minsum criteria. In R. E. Bixby, E. A. Boyd, and R. Z. Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, volume 1412 of *Lecture Notes in Computer Science*, pages 353–366. Springer, 1998.

[60] J. A. Hoogeveen and A. P. A. Vestjens. Optimal on-line algorithms for single-machine scheduling. In *Proceedings of the Fifth Conference On Integer Programming and Combinatorial Optimization (IPCO)*, pages 404–414, 1996.

[61] E. Horvath, S. Lam, and R. Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24(1):32–43, 1977.

[62] J. Jaffe. Efficient scheduling of tasks without full use of processor resources. *Theoretical Computer Science*, 26:1–17, 1980.

[63] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[64] V. Kann. Maximum bounded 3-dimensional matching is max snp-complete. *Information Processing Letters*, 37:27–35, 1991.

[65] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In *CRC handbook on theoretical computer science*. CRC Press, 1998 (to appear).

[66] N. Karmarkar and R. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pages 312–320, 1982.

[67] R. M. Karp, M. Luby, and A. Marchetti-Spaccamela. A probabilistic analysis of multi-dimensional bin packing problems. In *Proceedings of the Annual ACM Symposium on the Theory of Computing*, pages 289–298, 1984.

[68] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations (Proceedings of the Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*, pages 85–103. Plenum Press, New York, 1972.

[69] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 418–426, May 1996.

[70] S. Khanna, R. Motwani M. Sudan, and U. Vazirani. On syntactic versus computational views of approximability. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 819–830, 1994.

[71] E. L. Lawler. *Combinatorial Optimization*. Holt, Rinehart, and Winston, 1976.

[72] E. L. Lawler. Sequencing jobs to minimize total weighted completion time. *Annals of Discrete Mathematics*, 2:75–90, 1978.

[73] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: algorithms and complexity. In S. C. Graves et al. , editor, *Handbooks in OR & MS*, volume 4, pages 445–522. Elsevier Science Publishers, 1993.

[74] J. K. Lenstra and A. H. G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26:22–35, 1978.

[75] J. K. Lenstra and D. B. Shmoys. Computing near optimal schedules. In Philippe Chretienne et al. , editor, *Scheduling Theory and its Applications*, chapter 1. John Wiley & Sons, 1995.

[76] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 110–119, 1997.

[77] J. W. S. Liu and C. L. Liu. Bounds on scheduling algorithms for heterogeneous computing systems. In J. L Rosenfeld, editor, *Information Processing 74*, pages 349–353. North-Holland, 1974.

[78] L. Lovász. On the ratio of the optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975.

[79] F. Margot, M. Queyranne, and Y. Wang, August 1997. Personal communication.

[80] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[81] Rajeev Motwani. Approximation algorithms. Technical Report STAN-CS-92-1435, Computer Science Department, Stanford University, 1992.

[82] A. Munier, M. Queyranne, and A. S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. In R. E. Bixby, E. A. Boyd, and R. Z. Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, volume 1412 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 1998.

[83] C. H. Papadimitriou and J. D. Ullman. A communication-time tradeoff. *SIAM Journal on Computing*, 16(4):639–646, 1987.

[84] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–8, 1990.

[85] C. H. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43(3):425–40, 1991.

[86] C. H. Papadimitrou. *Computational Complexity.* Addison-Wesley, 1994.

[87] C. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. In *Proceedings of the Fourth International Workshop on Algorithms and Data Structures (WADS)*, pages 86–97, 1995. To appear in *Mathematical Programming B.*

[88] Michael Pinedo. *Scheduling : theory, algorithms, and systems.* Prentice Hall, 1995.

[89] H. Pirahesh, C. Mohan, J. Cheung, T.S. Liu, and P. Selinger. Parallelism in relational data base systems: Architectural issues and design approaches. In *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*, pages 4–29, July 1990.

[90] C. N. Potts. An algorithm for the single machine sequencing problem with precedence constraints. *Mathematical Programming Studies*, 13:78–87, 1980.

[91] M. Queyranne and Y. Wang. Single-machine scheduling polyhedra with precedence constraints. *Mathematics of Operations Research*, 16(1):1–20, February 1991.

[92] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

[93] P. Raghavan and C. D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7:365–374, 1987.

[94] R. Ravi, A. Agrawal, and P. Klein. Ordering problems approximated: single-processor scheduling and interval graph completion. In *Proceedings of the 18th ICALP*, pages 751–62. Springer Verlag, 1991.

[95] D. A. Schneider. *Complex Query Processing in Multiprocessor Database Machines.* PhD thesis, University of Wisconsin, Madison, September 1990.

[96] Alexander Schrijver. *Theory of Linear and Integer Programming.* Wiley-Interscience series in discrete mathematics. Wiley, 1986.

[97] A. Schulz. *Polytopes and Scheduling.* PhD thesis, Fachbereich Mathematik, Technische Universitt Berlin, Berlin, Germany, 1996.

[98] A. S. Schulz. Scheduling to minimize total weighted completion time: performance guarantees of lp based heuristics and lower bounds. In *Proceedings of the Fifth Conference On Integer Programming and Combinatorial Optimization (IPCO)*, pages 301–315, 1996.

[99] A. S. Schulz and M. Skutella. Random-based scheduling: New approximations and lp lower bounds. In J. Rolim, editor, *Randomization and Approximation Techniques in Computer Science (RANDOM)*, volume 1269 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 1997.

[100] A. S. Schulz and M. Skutella. Scheduling-LPs bear probabilities: Randomized approximations for min-sum criteria. In *Proceedings of the Fifth Annual European Symposium on Algorithms (ESA)*, pages 416–29, 1997.

[101] P. Schuurman and G. Woeginger, June 1998. Personal communication.

[102] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1979.

[103] D. Shmoys, J. Wein, and D. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 24:1313–31, 1995.

[104] W. E. Smith. Various optimizers for single-stage production. *Naval Res. Logist. Quart.*, 3:59–66, 1956.

[105] Aravind Srinivasan. Improved approximations of packing and covering problems. In *Proceedings of the 27th ACM Symposium on the Theory of Computing*, pages 268–276, 1995.

[106] C. Stein and J. Wein. On the existence of schedules that are near-optimal for both makespan and total weighted completion time. *Operations Research Letters*, 21(3):115–22, 1997.

[107] L. Stougie. Personal communication, 1995.

[108] L. Stougie and A. Vestjens. Personal communication, 1996.

[109] J. Sydney. Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Research*, 23(2):283–298, 1975.

[110] Eric Torng and Patchrawat Uthaisombut. Lower bounds for srpt-subsequence algorithms for nonpreemptive scheduling. Manuscript, July 1998.

[111] P. Valduriez. Parallel database systems: Open problems and new issues. *Distributed and Parallel Databases: An International Journal*, 1(2):137–165, April 1993.

[112] S. Weiss and J. E. Smith. A study of scalar compilation techniques for pipelined supercomputers. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–9, 1987.

[113] G. Woeginger. A polynomial time approximation scheme for maximizing the minimum completion time. *Operations Research Letters*, 20:149–154, 1997.