Designing PTASs for MIN-SUM scheduling problems

F. Afrati

NTU Athens, Department of Electrical and Computer Engineering, Heroon Polytechniou 9, 15773, Athens, Greece

I. Milis

Athens University of Economics and Business, Department of Informatics, Patission 76, 10434 Athens, Greece

Abstract

We review approximability and inapproximability results for MIN-SUM scheduling problems and we focus on techniques for designing polynomial time approximation schemes for this class of problems. We present examples which illustrate the efficient use of the *ratio partitioning* and *time partitioning* techniques.

Key words: scheduling, MIN-SUM criteria, approximation algorithms, PTASs

1 Introduction

In a *scheduling problem*, a set of n jobs and a set of m machines are given. We want to schedule these jobs on the machines. A *schedule* is an assignment of time slots in a machine to each job so that certain constraints are satisfied. An *optimal* schedule is the one that minimizes a given objective function. Depending on the constraints and the objective function, many variants of the problem can be obtained. In simple variants, the data associated with each job j is its *processing time*, $p_j^{(i)}$, on each machine i. Objective functions are defined in terms of the completion time of each job. The *completion time* C_j of job j is the time at which the job completes its execution. The most

⋆ This work has been partially supported by European Commission, Project APPOL-IST-1999-14084.

common objective functions are the *maximum completion time* over all jobs, called also *makespan* of the schedule, and the *sum of completion times.*

Example: We are given a set of jobs $\{J_1, J_2, J_3, J_4, J_5\}$ and processing times 3, 1, 2, 3, 2 respectively. Also we have two identical (hence the processing times of jobs are independent of the machine on which they are executed) machines available, $\{M_1, M_2\}$. Suppose we decide on scheduling S_1 which assigns jobs J_1, J_2, J_3 on machine M_1 and in this order, i.e., job J_1 completes at time $C_1 = 3$, job J_2 completes at time $C_2 = 4$, and job J_3 completes at time $C_3 = 6$. The rest of the jobs are assigned to machine M_2 so that job J_4 completes at time $C_4 = 3$ and job J_5 completes at time $C_5 = 5$. For a visualization of this schedule see the Gantt chart in Figure 1. Gantt charts are convenient means of describing a schedule.

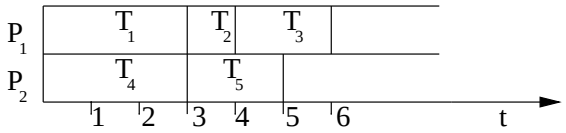

Fig. 1. Gantt chart for schedule S_1

Then the makespan, i.e. the maximum completion time, is equal to 6. The same makespan can be achieved by the following schedule S_2 which assigns jobs J_2, J_3, J_5 to machine M_1 and in this order, and the rest to machine M_2. It is clear that both schedules are optimal for the objective function $\max_j C_j$ which defines the makespan.

However, if we sum the completion times on schedule S_1, we get 3+4+6+3+ 5=21 which is not optimal because we can swap jobs J_1 and J_2 on machine M_1 and reduce this sum to 1+4+6+3+ 5=19. Schedule S_2 has sum of completion times 18. Finally it is easy to see that, in the case we want to minimize the sum of completion times, $\sum_j C_j$, then the best schedule is S_3, defined by the following assignment: machine M_1 is assigned jobs J_2, J_3, J_1 and machine M_2 is assigned jobs J_5, J_4; the sum of completion times in this case is 17. Schedule S_3 may be produced by rearranging jobs within machines on scheduling S_1. Observe, however, that there is no way to get an optimal schedule for the objective function sum-of-completion-times by rearrangement *within* machines starting from schedule S_2. ∎

Besides the processing times of jobs a variety of other constraints may be imposed on the set of jobs such as *release dates*, *due dates*, *weights* and *precedence constraints.* A release date, r_j, represents the time the job j arrives in the system, that is, the earliest time at which the job can start its execution. A due date, d_j, represents a committed completion time of the job j. The completion of the job after this time is allowed, but usually a penalty is incurred. When the due date absolutely must be met, it is referred to as a *deadline.*

Finally, a weight, w_j, is a positive integer and represents the importance of job j relative to the other jobs in the system. Thus, when jobs have not the same importance, then an objective function that is more realistic to minimize is the *weighted* sum of completion times $\sum_j \ w_j C_j$ instead of the the sum of completion times $\sum_j \ C_j$.

Precedence constraints express the inter-dependence among jobs that do not allow certain jobs to start being executed before some other jobs have completed their execution. Precedence constraints are described by a partial order, $\prec$, on the set of jobs. If $i \prec j$, then job j can start its execution after the completion of job i. We say that jobs are *independent* if $\prec$ is empty.

Also two different scheduling models are usually considered depending on whether we allow jobs to be interrupted once started (to be resumed later) or not. These are the *preemptive* schedule, where the execution of a job can be interrupted and continued later on the same or on another machine, and the *non-preemptive* schedule, where once a job starts executed it should be completed.

According to optimization criteria, scheduling problems can be classified into two broad classes: MIN-MAX criteria and MIN-SUM criteria. The most common MIN-MAX criterion is the maximum completion time (makespan) of the schedule, $\max_j \ C_j$, and another is the maximum lateness, $\max_j \ (C_j - d_j)$. The most common MIN-SUM criterion is the total (weighted) completion time, $\sum_j \ w_j C_j$, and another is the total (weighted) flow time, $\sum_j \ w_j F_j$, where $F_j = C_j - r_j$.

It is well known that only some of the simplest variants of scheduling problems can be solved in polynomial time. Most of the recent research is directed towards designing approximation algorithms for the NP-hard variants of scheduling problems. A *ρ-approximation algorithm* computes polynomially a solution within a factor ρ of the optimum one. In this setting we are interested in designing ρ-approximation algorithms with ρ as small as possible as well as on finding lower bounds for ρ. When ρ is a constant, then we say that the approximation algorithm is a *constant factor* one. We say, however, that we have a *Polynomial Time Approximation Scheme* (PTAS) if we give an algorithm which, for any fixed value of ϵ, can construct an $(1+\epsilon)$-approximation solution. If the time complexity of a PTAS is also polynomial in $1/\epsilon$, then it is called a *Fully Polynomial Time Approximation Scheme* (FPTAS). For negative approximation results, the notion of NP-completeness is used to disprove the existence of good approximation algorithms, unless P=NP. It is well known that no *strongly* NP-hard optimization problem, for which the optimum cost is polynomially bounded with respect to the length of its instance, can have a FPTAS. Also, there is no PTAS for MAX SNP-hard problems, unless P=NP (see for example [30]).

Historically, MIN-MAX scheduling problems were the first investigated in the approximation algorithm framework and a lot of results have been proposed after Graham's seminal paper [19]. Most of the early constant approximation algorithms for MIN-SUM scheduling problems appeared during the 90's [31,22,21]. However, during the next years, there was much progress on the approximability of MIN-SUM problems, that led to PTASs for many of them.

In this paper we present a survey of this research on approximation algorithms for MIN-SUM scheduling problems, focusing on PTASs design techniques. In Section 2, we give formal definitions and examples for the reader to be introduced to some of the most common known techniques for deriving approximation algorithms for scheduling problems. In Section 3, we review briefly the known constant approximation algorithms by focusing on ideas that were used later for designing PTASs. In Section 4, we present recent results on PTASs and we classify the techniques into two main categories: the *ratio partitioning* and the *time partitioning*. These techniques are presented in Sections 5 and 6, respectively, where we give examples which illustrate how they may be used to derive PTASs for various problems. Examples are taken from [3] and [1,2]. We conclude with some open questions in Section 7.

2 Background and Preliminaries

In this section we first give some detailed definitions to explain the variants of the scheduling problem that we discuss in this paper and some notation. Then we give an introduction to some of the most well known simple algorithms for scheduling problems. In that respect, we present two polynomial algorithms, one constant ratio approximation algorithm and a PTAS.

2.1 Basic Definitions

In the Introduction we have discussed the general setting for the scheduling problem. However, this applies to many variants and in order to distinguish among them we use the standard three-field, $\alpha|\beta|\gamma$, notation scheme of Graham et al. [20].

The first field describes the machine environment, the second field describes the constraints on jobs and the third field describes the objective function. Thus, the notation $1 \mid\mid \sum w_j C_j$ represents the variant where a number of (independent) jobs are to be executed in one machine and we want to minimize the weighted sum of completion times. The notation $P \mid r_j \mid \sum C_j$ represents the variant where a number of jobs are to be executed on any given number of

identical machines, jobs have release dates (hence are not all available at any time instance) and we want to minimize the sum of completion times. We already discussed and defined in the Introduction the most common constraints on the jobs such us processing times, release dates, deadlines, precedence relation and preemption, which appear in the second field and are denoted by $p_j, r_j, d_j, prec$ and $pmtn$, respectively. We have also discussed the most common objective function which appear in the third field.

The machine environment described by the first field could be, e.g., one of a) any given number of identical machines, or b) a fixed number of identical machines or c) any given number of unrelated (non-identical) machines, and so on. To denote the set of the available machines, we use a standard notation. A single machine environment is denoted by 1. Identical parallel machines are denoted by P. In both single machine and identical parallel machines environments the processing time of each job does not depend on the machine and for each job j we are given a single processing time p_j. Uniformly related parallel machines are denoted by Q. In this case for each job j we are given a processing time p_j and for each machine i a different speed s_i. Hence, the time $p_j^{(i)}$ job j spends on machine i, assuming it is processed only on machine i, equals to p_j/s_i. Unrelated parallel machines are denoted by R and in this case we are given a processing time $p_j^{(i)}$ for processing each job j on each machine i. Thus α takes values from $\{1, P(m), Q(m), R(m)\}$, etc. The optional m denotes a constant number of machines; otherwise, its absence means that the number of machines is part of the problem's instance.

Moreover, in the case of parallel machine environments we might have *multiprocessor jobs*, where a job can (or need to) be executed simultaneously by several machines. In these case the second field describes also how such jobs can be executed. Thus, the notation fix_j is used for the variant where each job requires the simultaneous use of a prespecified set of machines and the notation $size_j$ for the variant where each job requires any set of machines of a given cardinality. In the general variant denoted by set_j, each job has a number of processing modes, each one defined by a set of machines and a processing time on this set of machines.

Although, intuitively, might be already understood what is a feasible schedule (e.g., one that does not assign two different jobs on the same machine at the same time), the following is a formal definition. A schedule is, for each job, an allocation of one or more time intervals to one or more machines. A schedule is *feasible* if (i) no two time intervals overlap in the same machine, (ii) no two time intervals allocated to the same job overlap (unless it is a multiprocessor job), and (iii) it satisfies the given processing restrictions and constraints. A feasible schedule is *optimal* if it minimizes the given optimization criterion. The objective value of an optimum solution is denoted by OPT.

2.2 Algorithms for Simple Cases

In this subsection we give simple algorithms (polynomial, constant approximation factor and PTAS) to familiarize the reader with known techniques.

2.2.1 Polynomial Algorithms

Some simple variants of the scheduling problem have polynomial algorithms. For a thorough listing of complexity results on scheduling problems the reader is referred to the book and a web page of Brucker [5]. We discuss here such a simple polynomial algorithm.

Example: $1||\sum w_j C_j$

The polynomial algorithm that finds an optimal schedule is known as **Smith**'s rule: It schedules according to the rule, "Shortest ratio $ratio_j = \frac{p_j}{w_j}$" first [50]. That is, the algorithm sorts the jobs in non-decreasing order of their ratios and then schedules the jobs on the machine according to this order.

The proof that this algorithm finds an optimal schedule is simple. Let S be an optimal schedule which violates this rule. Suppose the first time the rule is violated is when job i was executed before job j whereas $ratio_i > ratio_j$. By moving job j to be executed before job i the cost is changed from $A + w_i p_i + w_j p_i + w_j p_j$ (before the moving) to $A + w_i p_i + w_i p_j + w_j p_j$ (after the moving), where A is a quantity which remains the same in the two schedules. Thus, the cost is **decreased** by $w_j p_i - w_i p_j$, hence by swapping jobs i and j, we get a better schedule; contradiction.

2.2.2 Constant Ratio Approximation Algorithms

A standard technique to obtain a constant time approximation algorithm is to formulate first a (polynomial-time solvable) relaxation of the problem. The relaxations used include preemptive schedules and various linear and convex programs.

Example: $1|r_j|\sum C_j$

In the following we present how a preemptive relaxation can be used to approximate the problem $1|r_j|\sum C_j$. This result is due to Philips et. al. [31]. The problem is first relaxed to its preemptive variant $1|r_j, pmtn|\sum C_j$, which is known to be solvable in polynomial time by the Shortest Remaining Processing Time (SRPT) rule, i.e. at any point in time, schedule an available job with the least amount of remaining processing time. Let $\tilde{C}_j$ be the completion

time of job j in this preemptive schedule. A non-preemptive schedule can be obtained by the following heuristic: schedule the jobs in order in which they complete in an optimal preemptive schedule, i.e. in non-decreasing order of $\tilde{C}_j$'s. Consider now the derived non-preemptive schedule and let C_j be the completion time of job j. For simplicity assume also that the jobs have been scheduled in the order $1, 2, \ldots, n$. For each job j, let I_j be the total amount of idle time before the completion of job j and P_j be the total amount of processing time before the completion of job j. It clearly holds: $C_j = I_j + P_j$. Now, this quantity can be bounded as follows:

$C_j = I_j + P_j \leq max_{k \leq j} r_k + \sum_{k \leq j} p_k \leq \tilde{C}_j + \tilde{C}_j = 2\tilde{C}_j$.

Therefore, $\sum_j C_j \leq 2 \sum_j \tilde{C}_j \leq 2\text{OPT}$.

2.2.3 PTAS

Example: $1|p_j = p, r_j \leq plogn| \sum C_j$

This is a motivating example which is somewhat contrived for the sake of simplicity. We consider the problem $1|p_j = p, r_j \leq plogn| \sum C_j$, i.e., (i) all n jobs are identical with processing time p and (ii) all release dates are less than $plogn$. In fact, the more general problem $1|prec, p_j = p, r_j| \sum C_j$ is polynomialy solvable [5]. However, in order to make several useful points on the techniques used for PTASs in section 6, we give the following simple PTAS for $1|p_j = p, r_j \leq plogn| \sum C_j$.

Algorithm:
1. If $n \leq (1/\epsilon)^2$, then by exhaustive search find the best schedule.
2. If $n > (1/\epsilon)^2$, then schedule all jobs after time $plogn$ at any order.

Analysis of the Algorithm: The cost of any optimal schedule is greater than the cost of the schedule S_0 where all release dates are equal to 0. In this case, as jobs are identical, we schedule them in any order without idle time and the cost is: $Cost_0 = \sum_{j=1}^{n} C_j = \sum_{j=1}^{n} jp = \frac{n(n+1)}{2} p$

The schedule produced by the algorithm is no worse than adding $\log n$ additional jobs in the end of schedule S_0. In this case the additional cost is: $\Delta Cost = \log n \, max \, C_j + \frac{\log n(\log n+1)}{2} p$. However, $max \, C_j = np$, hence the algorithm produces a schedule which is worse at most by a ratio:

$$\frac{\Delta Cost}{Cost_0} \leq 2 \log n \frac{2}{n+1} < 5\epsilon$$

(Here is a detailed but obvious analysis of the very last inequality. Towards

contradiction, suppose that $5\epsilon < \frac{\log n}{n+1}$. This implies $\epsilon < \frac{\log n}{n}$ which implies $\frac{n}{1/\epsilon} < \log n$ which implies $\log n - \log(1/\epsilon) < \log\log n$, hence $1/2 \log n < \log n - \log\log n < \log(1/\epsilon)$, hence $\log n < 2\log(1/\epsilon)$ or $n < (1/\epsilon)^2$, contradiction.)

Motivated by this example, we can gain some intuition on how we could move jobs around in a schedule without degrading the cost very much. E.g., in the schedule S_0 of the jobs in the example, we can (a) Take any job and move it to the end, thus degrading the cost only by an added factor $O(C_j)$, hence by a ratio $\frac{2}{n+1}$, or (b) Take $\log n$ jobs and move them to the end, thus degrading the cost only by an added factor $O(C_j \log n)$, hence by a ratio $\log n \frac{2}{n+1}$.

So, in the general case, we have reason to expect that the following strategy would work: we can shuffle jobs in an "orderly way" without increasing the cost very much, e.g., move a few jobs within the time horizon where jobs of comparable size are executed. We will build on this intuition to solve more complicated problems in section 6.

3 Constant Approximation Algorithms

Although a FPTAS for $Pm|\ |\sum w_j C_j$ (due to Sahni [37]) was known since 1976, only a few approximation algorithms, for a MIN-SUM scheduling problem, has been proposed for some fifteen years. Gonzalez and Sahni in 1978 have proposed approximation results for MIN-SUM shop scheduling problems [18], while Kawaguchi and Kyan in 1986 have shown an 1.21-approximation algorithm for $P|\ |\sum w_j C_j$ [24], which is the best known even today.

The next non-trivial approximation results were an $O(\log n \log \sum_j w_j)$-approximation algorithm for $1|prec|\sum w_j C_j$, obtained by Ravi et al. [36], in 1991, which improved to an $O(\log n \log\log \sum_j w_j)$ one by Even et al. [14], in 1995. In fact, these results were implied by a general approach proposed in [36] and [14] to build approximation algorithms for *ordering problems*. Note also that some of these algorithms are still the best known for some ordering problems.

However, little was known about the approximability of MIN-SUM scheduling problems even until 1995, despite the fact that there was an extensive literature on their polyhedral structure (especially of single machine problems). In fact, several linear programming formulations have been considered but they were only exploited either to characterize polynomial solvable special cases or to obtain strong lower bounds allowing the optimal solution of moderate sized problems using enumerative methods (see Queyranne and Schulz [34] for a thorough survey of this research area).

Philips et al. [31], in 1995, gave the first constant approximation algorithms

for a number of problems including $(8+\epsilon)$-approximation algorithms for both $1|r_j, pmtn|\sum w_jC_j$ and $P|r_j, pmtn|\sum w_jC_j$), a $(16+\epsilon)$-approximation algorithm for $1|r_j|\sum w_jC_j$ and a $(24+\epsilon)$-approximation algorithm for $P|r_j|\sum w_jC_j$. Subsequently, Hall et al. in [22] and its journal version [21], improved dramatically the known approximation factors and gave algorithms for many new variants. Their results, some of which are the best known even today (see Table 1 below), were motivated by the successful work done on polyhedral structure of scheduling problems and build on earlier research on computing near-optimum solutions for MIN-MAX scheduling problems, by rounding optimum solutions to linear relaxations. In view of future results, the most important point of their work was the introduction of an *time-interval indexed* LP formulation, since it was proven later very fruitful for designing PTASs for many MIN-SUM problems.

The general idea of *time indexed* LP formulation was used in the past in several forms. Potts [32] used decision variables δ_{ij}, where δ_{ij} implies that job i precedes job j in the schedule. Dyer and Wolsey [13] used variables x_{jt}, where $x_{jt} = 1$ means that job j completes its execution at time t. Wolsey [56] and Queyranne [33] used as decision variables the completion times C_j of jobs themselves. Hall et al. [22] adopted this last formulation to obtain their results for single machine problems and proposed a different, more compact linear *time-interval indexed* LP formulation: subdivide the time horizon into the intervals $[1,1], (1,1+\epsilon], (1+\epsilon,(1+\epsilon)^2], \ldots$, where ϵ is an arbitrary small positive constant. Then, the linear program only specifies the interval in which the job is completed and since all completion times within an interval are within a $(1+\epsilon)$ factor of each other, the relative schedule within an interval will be of little consequence. This was the first use of *time partitioning*.

After the results of Hall et al. [22,21] there has been an explosion of research in this area with new algorithms of successively smaller approximation factors as well as with algorithms for new variants of MIN-SUM scheduling problems. Most of these algorithms follow the general and successful idea of problem relaxation: first an optimum solution to a relaxation of the original problem is polynomially obtained and then this solution is used to obtain an approximate solution to this problem. Types of relaxations can be distinguished into two broad classes: preemptive scheduling relaxations and several linear and convex programming relaxations. The preemptive scheduling relaxations consist of solving polynomially the preemptive variant of a non-preemptive problem, while the linear and convex programming relaxations consist of relaxing the integer constraints of an integer programming formulation of the problem. The solution of the relaxed problem is then used either to order the jobs in time (in one machine problems) or to assign the jobs to machines (in many machines problems). A series of ideas have been used for such a construction of a solution to the original problem including scheduling by completion times, scheduling by α-points as well as deterministic and randomized rounding methods. As

this paper focuses on PTASs design, we summarize, in Table 1, only the best known constant approximation algorithms. Clearly we omit a large body of important work on constant approximation algorithms, but the reader can acquire an accurate view of the results in this area by considering the references given in Table 1. The reader is also referred to [44] for a elegant survey on several linear programming relaxation algorithms mainly for $1|prec|\sum w_jC_j$.

Although there is practical evidence [38,53] that linear programming relaxations can provide very good approximate solutions, all algorithms based on such relaxations suffer from a major drawback: the solution to a problem is compared to the solution of the relaxed one and, unfortunately, there is an integrality gap between these two solutions. Thus, comparing any solution of such an algorithm to a fractional one, inherits this gap. Hall et al. [21], Schulz and Skutella [39] and Chekuri and Motwani [10] argue on the fact that several relaxation methods for $1|prec|\sum w_jC_j$ can not improve on the best known 2-approximation algorithm. Moreover, Torng and Uthaisombut [51] have shown that any preemptive relaxation for $1|r_j|\sum C_j$, that uses the shortest remaining processing time (SRPT) rule to solve the relaxed problem, can not led to an approximation factor better than $e/(e-1)$, that matches the best known factor of 1.58 [11]. It seems therefore, that the relaxation approaches can not yield a PTAS for MIN-SUM scheduling problems.

4 Polynomial Time Approximation Schemes

Sahni [37] has proposed in 1976, the first FPTAS, based on dynamic programming, for the *weakly* NP-hard problem $Pm\,||\sum w_jC_j$. However, no other PTAS or FPTAS was known for MIN-SUM scheduling problem before late 90's, when, Woeginger [54] proposed a FPTAS for $Qm|\,|\sum w_jC_j$, based also on dynamic programming, and Alon et al. [4] obtained a PTAS for $P\,||\sum D_i^2$ (D_i denotes the time by which machine i completes the execution of all jobs assigned to it).

Subsequently, Skutella and Woeginger [49] realized that this last result implies also a PTAS for $P\,|\frac{w_j}{p_j}=R|\sum w_jC_j$ and then in a first step they generalized this to a PTAS for $\frac{w_j}{p_j}$ ratios within a constant range. In a second step they obtained the first PTAS for a *strongly* NP-hard MIN-SUM problem ($P\,||\sum w_jC_j$). Their main idea was *ratio partitioning*, i.e. the partitioning of the jobs into subsets such that the ratios $\frac{w_j}{p_j}$ of all the jobs in each subset are within a constant range. Then, using the first step, near optimum solutions are computed for all subsets. Finally, these schedules are concatenated in order of nonincreasing ratios in each machine.

Subsequently, Afrati et al. [3] used also ratio partitioning to develop a PTAS

for $Rm \mid\mid \sum w_j C_j$. However, the concatenation of schedules that is used in ratio partitioning technique can not be applied in the presence of release dates. Thus the *time partitioning* technique was recalled by several researchers and it was proven powerful enough to yield PTASs for a variety of MIN-SUM scheduling problems in the presence of release dates. Several groups obtained, independently, PTASs for several variants of the average weighted completion time problem. All these results merged in to single article by eleven authors [1], where PTASs were obtained for scheduling jobs with release dates on a single machine, on identical parallel machines and on a constant number of unrelated machines in both preemptive and non-preemptive models. These results were a substantial progress in resolving the approximability question of MIN-SUM problems and it seems that the ideas proposed will influence the design of PTASs for other variants.

Indeed, based also on time partitioning technique and building on the ideas proposed in [1], Afrati et al. presented PTASs for $P2|fix_j|\sum C_j$ and $Pm|fix_j, pmtn|\sum C_j$ in [2], and very recently Fishkin et al. presented PTASs for $Pm|fix_j, r_j|\sum w_j C_j$ and $Pm|size_j, r_j|\sum w_j C_j$ in [15] and $Pm|set_j|\sum w_j C_j$ in [16]. Moreover, Chekuri and Khanna using some non-trivial extensions to the ideas introduced in [1], obtained PTASs for $Q|r_j|\sum w_j C_j$ and $Q|r_j, pmtn|\sum w_j C_j$ [8].

However, it seems that it is difficult to extend the techniques for approximating the average weighted completion time in the case of average weighted flow time (recall that the flow time F_j for each job is defined as $F_j = C_j - r_j$). When computing optimal solutions, minimizing the average weighted completion time is equivalent to minimizing the average weighted flow time, since $\sum r_j$ is determined by the input. However, these two objective functions behave quite differently when we look for approximate solutions. Recently, Chekuri et al. [9] proposed initially a quasi-polynomial $2+\epsilon$-approximation algorithm for $1|r_j, pmtn|\sum w_j F_j$, in the special case where the ratio of the largest job processing time over the smallest one, $P = \frac{\max_j p_j}{\min_j p_j}$, is polynomially bounded in n. After this result, Chekuri and Khanna [7] gave a quasi-PTAS for the same problem, in the case where either P is polynomially bounded or the ratio of the largest job weight over the smallest one, $W = \frac{\max_j w_j}{\min_j w_j}$, is polynomially bounded. They also proved that this last quasi-PTAS becomes a PTAS in the cases where (i) both P and W are bounded, (ii) P is bounded and W is unrestricted and (iii) for each job its weight is the inverse of its processing time, i.e. $p_j = 1/w_j$ (also known as *stretch metric*).

In the third column of Table 1 we give a summary of existing (F)PTASs. We present there either the existence of a (F)PTAS or known inapproximability results in terms of either MAX SNP-hardness or existing lower bounds on the variant's approximation factor.

Problem	Best known ρ	(F)PTAS or Lower bound
$1\|r_j\|\sum C_j$	1.58 [11]	PTAS [1]
$1\|r_j\|\sum w_jC_j$	1.69 [17]	PTAS [1]
$1\|r_j,pmtn\|\sum w_jC_j$	4/3 [41]	PTAS [1]
$1\|prec\|\sum w_jC_j$	2 [10,12,21,28]	
$1\|interval\ order\|\sum w_jC_j$	$\frac{1+\sqrt{5}}{2}+\epsilon$ [55]	
$1\|convex\ bipartite\ order\|\sum w_jC_j$	$\frac{1+\sqrt{5}}{2}+\epsilon$ [55]	
$1\|r_j,prec\|\sum w_jC_j$	$e+\epsilon$ [39]	
$1\|r_j,prec,pmtn\|\sum w_jC_j$	2 [21]	
$1\|r_j\|\sum F_j$	$O(n^{1/2})$ [25]	$\Omega(n^{1/2-\epsilon})$ [25]
$1\|r_j,pmtn\|\sum w_jF_j$	$2+\epsilon$ [9] *	quasi-PTAS [7] **
$Pm\|\ \|\sum w_jC_j$		FPTAS [37]
$P\|\ \|\sum D_i^2$	1.04 [27]	PTAS [4]
$P\|prec,p_j=1\|\sum C_j$		8/7 [23]
$P\|prec\|\sum C_j$		4/3 [23]
$P\|\ \|\sum w_jC_j$	1.21 [24]	PTAS [49]
$P\|r_j\|\sum w_jC_j$	2 [40]	PTAS [1]
$P\|r_j,pmtn\|\sum w_jC_j$	2 [40]	PTAS [1]
$P\|r_j,prec\|\sum w_jC_j$	4 [29]	
$P\|r_j,prec,pmtn\|\sum w_jC_j$	3 [21]	
$P\|r_j\|\sum F_j$	$O(\sqrt{\frac{n}{m}}\log\frac{n}{m})$ [26]	$\Omega(n^{1/3-\epsilon})$ [26]
$P\|r_j,pmtn\|\sum F_j$	$O(\log\min\{\frac{n}{m},\log P\})$ [26]	
$P2\|fix_j\|\sum C_j$	2 [6]	PTAS [2]
$Pm\|fix_j,pmtn\|\sum C_j$		PTAS [2]
$Pm\|fix_j,r_j\|\sum w_jC_j$		PTAS [15]
$P\|fix_j,p_j=1\|\sum C_j$		$\Omega(m^{1/2-\epsilon})$ [15]
$Pm\|size_j,r_j\|\sum w_jC_j$		PTAS [15]
$P\|size_j\|\sum C_j$	32 [52]	
$Pm\|set_j\|\sum w_jC_j$		PTAS [16]
$Qm\|\ \|\sum w_jC_j$		FPTAS [54]
$Q\|r_j\|\sum w_jC_j$	$2+\epsilon$ [40]	PTAS [8]
$Q\|r_j,pmtn\|\sum w_jC_j$		PTAS [8]
$Rm\|\ \|\sum w_jC_j$	3/2 [46,48]	PTAS [3,1]
$Rm\|r_j\|\sum w_jC_j$	2 [47,48]	PTAS [1]
$Rm\|r_j,pmtn\|\sum w_jC_j$	3 [47,48]	PTAS [1]
$R\|\ \|\sum w_jC_j$	3/2 [43,46,48]	MAX SNP-hard [23]
$R\|r_j\|\sum w_jC_j$, $R\|r_j\|\sum C_j$	2 [47,48]	MAX SNP-hard [23]
$R\|pmtn\|\sum w_jC_j$, $R\|pmtn\|\sum C_j$	2 [47,48]	
$R\|r_j,pmtn\|\sum w_jC_j$	$2+\epsilon$ [35]	

* Quasi-polynomial algorithm for poly-bounded P

** For poly-bounded P or W

Table 1. Overview of results.

PTASs that are based on ratio partitioning and time partitioning analyze the problem by using a combination of elementary and well understood combinatorial ingredients: partitioning, grouping, geometric rounding, enumeration and dynamic programming. The approach taken, in these PTASs, is to sequence several transformations of the input problem. Some transformations are actual changes to simplify the input, while others are applied as thought experiments to the optimum solution to prove there is a near-optimum solution with nice structure. Each transformation potentially increases the objective value by only $1+O(\epsilon)$, that is, it produces $1+O(\epsilon)$ *loss*. Besides problem-variant-specific transformations, some general transformations are used in these techniques. Such transformations are *geometric rounding* and *time stretching*.

Geometric rounding is a simplification on the given data for each job, e.g., the processing times. It creates a well-structured set of a limited number of distinct processing times (and other data such as release dates): It rounds each processing time (and release date) to integer powers of $1+\epsilon$ [1]. By this simplification, we degrade the cost by at most a factor of $1+\epsilon$. To prove, we change the values in two steps. First multiply every release date and processing time by $1+\epsilon$; this increases the objective value by $1+\epsilon$. Then *decrease* each date and time to the next *lower* integer power of $1+\epsilon$ (which is still greater than the original value). This can only improve things for the problems considered in this paper.

The second simplification, *time stretching*, creates idle time *gaps* evenly spread along the schedule. These gaps will be used conveniently in later transformations to accommodate jobs that we are moving around in order to prove that there is a near-optimum schedule with nice structure [2]. Often, we will use time stretching to add an idle amount of ϵp_j time units before every job j. This will increase each completion time (and hence their sum) by a factor of $1+\epsilon$. According to this last transformation, we may also assume that the starting time of each job j is $S_j \geq \epsilon p_j$: If job j completed at time $t > p_j$ then it now completes at time $(1+\epsilon)t$ and therefore does not start until time $\epsilon t \geq \epsilon p_j$.

[1] In the rest of the paper, as we are only discussing PTASs, we use the given approximation factor ϵ as one of the given parameters of the problem.
[2] Recall also the discussion in Section 2.2.3

5 Ratio Partitioning

In this section, we want to find a PTAS for the problem $R2||\sum w_jC_j$. First, we observe that we only need to decide on which machine each job will be executed. This is so, because, if we know that, then the optimum schedule on each machine is given by Smith's rule (see Section 2.2.1). The structure of this section is as follows. Before we present the algorithm, we show that certain simplifying assumptions can be done to the problem without affecting its cost very much. In fact, it is easy to see that these assumptions apply to a more general variant, i.e., in the case we have any constant number of machines.

We make the following simplifying assumptions:

(a) For a job with its processing times far apart, we can immediately decide on which machine the job will be executed with negligible loss.

(b) It is easy to see that we can view the calculation of the objective function as the sum of products, each product being essentially the contribution to the cost of the interaction between pairs of jobs. Then, we can observe that pairs of jobs with very different ratios do not contribute much to the cost function, hence such contributions can be equated to zero with negligible loss.

(c) Summing up assumptions (a) and (b) and adding geometric rounding and grouping, we observe that only a constant number of pairs of jobs affect each other, namely the ones with comparable ratios. This leads to a dynamic programming algorithm.

Before we move on with discussing the above assumptions in quantitative terms and in order to facilitate this discussion we define a partitioning of the range of ratios into disjoint windows such that the i^{th} window consists of ratios in the range $F_i = (a^iR_{max}, a^{i-1}R_{max}]$, where $R_{max} = \max_{i,j}\{w_j/p_j^{(i)}\}$ and $a = \epsilon^4$. For the same reason, we define a **job-type** $T_{R,R'}$ as the set of jobs having the same pair of ratios, i.e. $T_{R,R'} = \{j \mid R_j^{(1)} = R \text{ and } R_j^{(2)} = R'\}$, where $R_j^{(i)} = w_j/p_j^{(i)}$.

Now, we can repeat our three assumptions in a bit more quantitative (yet not formal) terms:

(a) We can think of each job as having its two ratios either in the same window or in adjacent windows.
(b) "Very different" job types do not affect each other.
(c) We can bound the number of jobs within each job-type by a constant number.

In the following three subsections we state and prove formally these three

simplifying assumptions.

5.1 Ratios in adjacent windows

We prove that for each job, either we can tell in advance in which machine it will be scheduled (we say that it is decided) or else its two ratios do not differ much. If the processing times of a job are very different on the two machines, then we can directly decide to schedule it on the machine on which it has the shortest processing time. In quantitative terms: We can assume that all jobs such that $p_j^{(1)} < \epsilon p_j^{(2)}$ are scheduled on M_1 (we say that this job is M_1-*decided*), and that all jobs such that $p_j^{(2)} < \epsilon p_j^{(1)}$ are scheduled on M_2 (we say that this job is M_2-*decided*). To prove, we show that by moving all such jobs to the machine with the small processing time, the cost does not increase much: we move each such job j in the other machine so that it starts just before the earliest job which completes at time $> C_j$ (the completion time of the moved job in the original schedule). This affects the total cost by increasing the completion time of job j and by delaying the jobs after j in the machine M_1 (suppose we moved the job to M_1). However, the new completion time of job j is at most $C_j + p_j^{(1)} < C_j + \epsilon p_j^{(2)} < C_j(1+\epsilon)$. The other jobs affected are the jobs in machine M_1 which are delayed. Each such job k is delayed by at most ϵ times the sum of M_2-processing times of moved jobs which completed before time C_k in M_2; but this is at most ϵ times C_k. Thus, in this case, we can put $p_j^{(2)} = \infty$ or $p_j^{(1)} = \infty$ accordingly. Moreover, for each undecided job we now know that $p_j^{(1)} > \epsilon p_j^{(2)}$ and $p_j^{(2)} > \epsilon p_j^{(1)}$, that is $\epsilon < R_j^{(1)}/R_j^{(2)} < 1/\epsilon$. Hence, each job is either decided or it has its two ratios either in the same window or in adjacent windows.

5.2 Job types not affecting each other

We want to prove that "very different" job types do not affect each other. To state formally, we rewrite the cost function as:

$$\sum_{j,k \text{ on } M_1} \mathcal{M}_1[j,k] + \sum_{j,k \text{ on } M_2} \mathcal{M}_2[j,k],$$

where

$$\forall j,k \in J, \qquad \mathcal{M}_i[j,k] = \begin{cases} p_k^{(i)} w_j & \text{if } j = k \\ \frac{1}{2} p_k^{(i)} w_j & \text{if } j \neq k \text{ and } \frac{w_k}{p_k^{(i)}} = \frac{w_j}{p_j^{(i)}} \\ p_k^{(i)} w_j & \text{if } \frac{w_k}{p_k^{(i)}} > \frac{w_j}{p_j^{(i)}} \\ 0 & \text{otherwise} \end{cases}$$

Notice that an entry $\mathcal{M}_i[j,k]$ represents the contribution to the cost function of job j, due to job k.

Using this formulation of the cost function, we can prove that if the ratios of two jobs are very different, then we can neglect their interaction:

(i) For every pair of jobs $\{j,k\}$ such that $R_j^{(i)} < \epsilon^2 R_k^{(i)}$, we can replace $\mathcal{M}_i[j,k]$ by 0.

(ii) For $i = 1,2$, and for every pair of jobs $\{j,k\}$ such that j and k are either M_i-decided or undecided, and such that $R_j^{(i)} < \epsilon^4 R_k^{(i)}$, we can replace both $\mathcal{M}_1[j,k]$ and $\mathcal{M}_2[j,k]$ by 0.

Proof of (i). We prove this in two steps: First for each k on M_1, we look at the jobs j on M_1 such that $R_j^{(1)} < \epsilon^2 R_k^{(1)}$, and whose completion times satisfy $C_k < \epsilon C_j$. The presence of k before such a j has only a marginal effect on j's completion time (easy to prove). Second, we look at the pairs $\{j,k\}$ which are scheduled on M_1, such that $R_j^{(1)} < \epsilon^2 R_k^{(1)}$, but whose completion times satisfy $C_j \leq C_k/\epsilon$. The contribution to the cost function of those pairs is very little as compared to $w_k C_k$:

$$\sum_{\substack{j \text{ on } M_1 \\ C_j \leq C_k/\epsilon}} w_j p_k^{(1)} = \sum_{\substack{j \text{ on } M_1 \\ C_j \leq C_k/\epsilon}} r_j^{(1)} p_j^{(1)} p_k^{(1)} < \epsilon^2 \sum_{\substack{j \text{ on } M_1 \\ C_j \leq C_k/\epsilon}} w_k p_j^{(1)}$$

$$= \epsilon^2 w_k \sum_{\substack{j \text{ on } M_1 \\ C_j \leq C_k/\epsilon}} p_j^{(1)} \leq \epsilon^2 w_k \max_{\substack{j \text{ on } M_1 \\ C_j \leq C_k/\epsilon}} C_j \leq \epsilon^2 w_k C_k/\epsilon = \epsilon w_k C_k,$$

Proof of (ii) Consider the case $i = 1$. Because $R_j^{(1)} < \epsilon^2 R_k^{(1)}$, we can replace $\mathcal{M}_1[j,k]$ by 0–according to (i) above which we just proved. If either job is M_1-decided, then it will never be placed on M_2 and so we can replace $\mathcal{M}_2[j,k]$ by 0. If both are undecided, then we have: $p_j^{(2)} > \epsilon p_j^{(1)}$ and $p_k^{(2)} < p_k^{(1)}/\epsilon$. Thus: $\frac{w_j}{p_j^{(2)}} < \frac{1}{\epsilon}\frac{w_j}{p_j^{(1)}} < \frac{\epsilon^4}{\epsilon}\frac{w_k}{p_k^{(1)}} < \frac{\epsilon^4}{\epsilon}\frac{1}{\epsilon}\frac{w_k}{p_k^{(2)}} = \epsilon^2 \frac{w_k}{p_k^{(2)}}$. Thus the condition in (i) is satisfied, hence, we can replace $\mathcal{M}_2[j,k]$ by 0.

5.3 Constant number of jobs

We can bound the number of jobs within each job-type by a constant number. Consider a job-type $T_{R,R'}$ and the sum of the weights, $W_{R,R'}$, of jobs in $T_{R,R'}$. We group small jobs together to build larger compound jobs. Thus, we can assume that every job of a job-type has weight at least $\epsilon^2 W_{R,R'}$, and hence each job-type has at most $\frac{1}{\epsilon^2} = O(1)$ elements. More specifically, we group greedily so that the sum of weights of the tiny jobs in each group (which now represents the weight of the compound job) is between $\epsilon^2 W_{R,R'}$ and $2\epsilon^2 W_{R,R'}$. There are two points to discuss:

(i) The cost does not change much by this grouping. For each group k (in machine i with ratio R say), the contribution to the cost function before the grouping was $\sum_{m,n \ jobs \ in \ group \ k} R p_m^{(i)} p_n^{(i)}$. This quantity, though, is bounded on both sides by $R(\sum_{n \ job \ in \ group \ k} p_n^{(i)})^2$ and $R\frac{1}{2}(\sum_{n \ job \ in \ group \ k} p_n^{(i)})^2$ (upper and lower bound respectively). Hence it is at most $\frac{4}{R}\epsilon^4 W_{R,R'}^2$. Similarly the jobs in this job type contribute to the total cost by a quantity which is at least $\frac{1}{2R} W_{R,R'}^2$. There are at most $\frac{1}{\epsilon^2}$ compound jobs, hence the total shift from the original cost is at most $\frac{4}{\epsilon^2 R}\epsilon^4 W_{R,R'}^2$ which is less than $\epsilon \frac{1}{2R} W_{R,R'}^2$.

(ii) Clearly a schedule that uses the compound jobs results in a cruder fractioning of the jobs in the job type among the machines. To take care of that we stretch time by $1+\epsilon$ and this stretching is enough to accommodate the compound jobs.

5.4 Algorithm for $R2||\sum w_j C_j$

We first simplify the input of the problem by applying geometric rounding on the parameters defining the jobs. Then, we identify the jobs which are M_1-decided or M_2-decided. We replace all irrelevant processing times (i.e. $p_j^{(2)}$ if j is M_1-decided and $p_j^{(1)}$ if j is M_2-decided) by ∞, which means that the corresponding ratio becomes equal to 0. This leaves us with undecided jobs that have their two ratios either in the same window or in adjacent windows.

Then we do all the simplifications of the previous subsection, namely calculating first the $\mathcal{M}_i[j,k]$ and then putting accordingly some of them equal to 0.

Note that, the set of undecided jobs having at least one ratio in a specific window has size $O(\log^2(1/\epsilon)/\epsilon^2)$: Roughly, because of geometric rounding, we have only a constant number of different ratios within a window and, because undecided jobs have their ratios either in the same window or in adjacent windows, there is only a constant number of job types that may contain a

job of a particular ratio in one of the machines. Moreover, there is a constant number of jobs within each job type.

Now, in deciding where to put optimally the undecided jobs, we have to calculate the cost of the schedule. In this calculation, as we proved in Section 5.2, we sum up the contribution to the cost of all pairs of jobs and, moreover, we can neglect with negligible loss the contribution of pairs with very different ratios. This means, in quantitative terms that for such a pair of jobs j, k we set their contribution $\mathcal{M}_i[j,k]$ equal to zero.

Thus, summing up the above analysis, we have reduced the problem to a problem of scheduling jobs in two machines under the following (convenient) constraints: a) Each job type has the two ratios either in the same window or in adjacent windows and b) Job types affect each other only in the stretch of two windows and c) there is a constant number of undecided jobs with at least one of their ratios in a specific window. An immediate consequence of that is dynamic programming with respect to the windows of ratios. In each stage i of the dynamic programming, we have fixed the schedule for jobs with their both ratios in windows $F_1, \ldots F_i$ and keep all possible schedules of jobs with at least one of their ratios in window F_i. Thus, each stage runs in $O(1)$ time.

6 Time Partitioning

In this section we want to find PTASs for the problems, $1|r_j|\sum C_j$ and $P2|fix_j|\sum C_j$.

The complication in the first problem is that we have to deal with release dates too. In the absence of release dates, the problem has a simple polynomial algorithm (the algorithm that schedules according to the SPT rule, see Section 2.2.1). Also, the variant with release dates and preemption has a polynomial algorithm which solves it optimally. It is again a simple algorithm which, now, schedules according to the Shortest Processing Remaining Time first (SPRT rule). The PTAS that we will explain here is using this algorithm. It finds first a preemptive schedule, and then argues that we can move jobs around so that most of them can be completed in a non preemptive manner with negligible added cost.

The complication in the second problem is that we have multiprocessor jobs. In particular, the variant that we consider here deals with *dedicated* jobs, i.e., each job j requires for its execution the simultaneous availability of a *prespecified* subset of machines $\tau_j = \{M_1, \ldots, M_m\}$ for p_j time units. The set τ_j is called the *type* of job j. We restrict attention here to the case with two machines. The

methods and techniques though extend easily to the case of any fixed number of machines. For this problem, we have two kinds of monoprocessor jobs, the 1-jobs and the 2-jobs dedicated to run on machine M_1 and M_2 respectively. Biprocessor 12-jobs are dedicated to both machines. Again the variant of the problem where preemption is allowed has a polynomial algorithm that solves it optimally [6].

Now we explain time partitioning and in the next subsection, we show how we may assume some structure in the near-optimal schedule. Then, we discuss the algorithms for each problem separately.

6.1 Simple properties of time partitioning

We partition the time axis in intervals as follows. For an arbitrary integer x, we define $R_x = (1+\epsilon)^x$. We partition the time interval $(0, \infty)$ into disjoint intervals $I_x = [R_x, R_{x+1})$. We will use I_x to refer to both the interval and the size $(R_{x+1} - R_x)$ of the interval. We will often use the fact that $I_x = \epsilon R_x$, i.e., the length of an interval is ϵ times its start time. We consider an optimum schedule and focus on the schedule of one machine. The reason this time partitioning is useful is that each interval can be used to accommodate jobs of relatively small size and thus we can assume that, in a near-optimal schedule, finishing up jobs on a preemptive schedule comes at negligible cost. To argue that intervals accommodate small jobs, we show that moving around jobs (from smaller intervals to intervals comparable to their size), does not degrade the cost very much. In a way, this generalizes the intuition discussed in Section 2.2.3.

Now, we make it formal. The following are easy to prove useful facts:

(a) By time stretching we can create, in each interval an idle gap of size ϵI_x.
(b) Moving a job j inside the interval in which it runs produces a $1 + O(\epsilon)$ loss because it will only increase its completion time by at most I_x which is less than ϵR_{x+1}, consequently less than ϵC_j.
(c) A job can not be "arbitrarily large", i.e., it crosses a constant number of intervals. In quantitative terms, we argue as follows. In the worst case, a job starts at time ϵp_j and completes at time $\epsilon p_j + p_j$, hence, it crosses at most $s = \lceil \log_{1+\epsilon} \frac{\epsilon p_j + p_j}{\epsilon p_j} \rceil = \lceil \log_{1+\epsilon}(1 + \frac{1}{\epsilon}) \rceil$ intervals.
(d) If a job j crosses several intervals, we know that it crosses at most s intervals, and then we can accumulate idle time for all s intervals in the beginning of job j, and we know that the relative size of the 1st interval to the size of the s-th interval is $\geq \frac{I_x}{I_{x+s}} = \frac{1}{(1+\epsilon)^{s-1}} = \epsilon$.

In many cases, jobs that are much smaller than the interval in which they run are essentially negligible and easy to deal with. The difficulty comes from jobs that are large—taking up a substantial portion of the interval [3].

We assume, here, that *small* jobs are defined to be those with $p_j < \epsilon I_x$, where I_x is the interval where job j runs. Jobs that do not have this property are called *large*. The following easy to prove facts:

(a) We may restrict attention to schedules in which no small job crosses an interval: By time stretching, add in each interval a gap of ϵI_x and finish the single small job that may be crossing an interval.
(b) There is a constant number of large jobs in an interval, namely at most $\frac{1}{\epsilon}$.

Interestingly, we can almost get rid altogether of large jobs in all intervals, except a constant number of them. I.e., we can put structure in a near-optimal schedule so that all jobs are small in the interval they run except a constant number of them in the entire schedule, which we call *huge* jobs. Thus, the huge jobs are large jobs that we can not move in larger intervals without a considerable degradation of the cost. We can enforce such a structure by moving most large jobs in larger intervals without much degrading the quality of the schedule. Thus, we end up with an overall constant number of huge jobs. As expected the huge jobs are the ones whose starting time S_j is very close to the optimum cost OPT:

Lemma 1 *With a $1 + O(\epsilon)$ loss, we assume:*

(i) For each job j, $S_j \geq \min\{\frac{p_j}{\epsilon^2}, \epsilon^7 OPT\}$.

(ii) All jobs are small in the intervals they run except at most $1/\epsilon^7$.

Proof: The proof of (ii) follows from (i): Note that $S_j \geq p_j/\epsilon^2$ means that job j is small in the interval it runs. Also, all jobs starting after time $\epsilon^7 OPT$ are at most $1/\epsilon^7$. The small jobs trivially satisfy this condition in (1). We move around large jobs in larger intervals so that they are now small in the interval that they run. We do that only for large jobs that complete before $\epsilon^7 OPT$.

We move a job that completes before time $(1 + \frac{1}{\epsilon^2})p_j$ in the first gap after time $t = C_j/\epsilon^4$. We have to worry about three considerations:

(a) There is space large enough to accommodate all jobs that are landed on a particular gap. This is so because the gap at time t is equal to $\epsilon^2 t$, that is

[3] Small jobs are a lot like fractional jobs in linear relaxation methods, and fractional solutions are usually easier to find.

at least equal to C_j and hence it accommodate all jobs which complete before time C_j.

(b) There is not some hugely long crossing job in this interval. This is guaranteed by the fact that any crossing job may cross at most s intervals. Thus, we put the moved job in the beginning of the crossing job and still it is in an interval where it is considered small. This is so because the intervals that a job crosses differ in size by a factor at most $1/\epsilon$.

(c) The added cost is not too much: We need to bound the added cost of the large jobs we moved forward. The last interval from which we advance jobs ends at time $t = \epsilon^7 OPT$. Thus the total completion time of the advanced jobs is $\sum_{R_x<t} \frac{1}{\epsilon}\frac{(1+\epsilon)R_x}{\epsilon^4} \leq \frac{t}{\epsilon^5}\sum_{i\geq 0} 1/(1+\epsilon)^i = \frac{t}{\epsilon^5}\frac{(1+\epsilon)^2}{\epsilon} = \epsilon \cdot \text{OPT}(1+\epsilon)^2 \leq 2\epsilon\text{OPT}$ as desired. ∎

6.3 Algorithm for $1|r_j|\sum C_j$

We first simplify the input of the problem by applying geometric rounding on the parameters. Thus, jobs are released only in the end of intervals. Now, we use the simplifications that, as we proved in the previous subsections of this section, do not degrade much the quality of the schedule. Before we present the algorithm that solves the problem in the general case we treat an easy subcase. Suppose that, in the near optimal schedule of the preemptive problem, all jobs are small in the interval they run. Then the following simple algorithm finds a near optimal solution: Step 1: We set release dates equal to $\max\{r_j, \frac{p_j}{\epsilon^2}\}$. According to Lemma 1, this does not change the feasibility of a nearly optimum schedule. Step 2: We use SPRT rule to obtain an optimum preemptive schedule and then stretch each interval by ϵI_x and use the extra space to accommodate the tails of the preempted jobs. Step 2, however, is equivalent to running SPT rule.

Now in the general case, we also have the huge jobs to accommodate. I.e., the optimum schedule may require some jobs to run in intervals such that they are large. By Lemma 1, however, there is still a nearly optimum solution with only a constant number of large jobs. Hence we find an optimum schedule of them by exhaustive enumeration and schedule the small jobs around them as we explained above.

6.4 Algorithm for $P2|fix_j|\sum C_j$

For this case, we need some more discussion to explain how we deal with multiprocessor jobs. We use again a relaxation to the preemptive case. The analysis goes through for any fixed number of machines M, only that, for $M > 2$ the preemptive case is not polynomial and is solved by a PTAS too.

6.4.1 Parallel running jobs have comparable processing times

Besides geometric rounding, we need some more transformations. We begin with the following lemma which states that jobs scheduled to run in parallel have comparable processing times.

We first state a few useful facts:

(a) If p_{max} is the maximum processing time of a set of jobs and ℓ and C is the length and the cost respectively of a schedule of this set of jobs, then, $\ell < 2\sqrt{p_{max}C}$. At least $\frac{\ell}{2p_{max}}$ jobs will be executed after time $\frac{\ell}{2}$, hence the cost is at least: $C \geq \frac{\ell}{2p_{max}}\frac{\ell}{2}$, hence the inequality.

(b) Suppose each processing time is $< \epsilon^{10}$, then the makespan is $\ell < 2\sqrt{p_{max}C} < 2\sqrt{\epsilon^{10}C^2}$, hence $\ell < 2\epsilon^5 C$.

Lemma 2 *With a $1 + O(\epsilon)$ loss, we never have two jobs scheduled in parallel whose processing times differ by a factor greater than $\frac{1}{\epsilon^5}$.*

Proof: Let j be a job executed w.l.o.g. on machine M_1, and let A be the set of jobs scheduled in parallel with j on machine M_2 with processing times $< \frac{1}{\epsilon^5}p_j$. Recall that, since the single-machine-jobs on each machine are scheduled in the order of increasing processing times, all jobs in A run in a single consecutively.

The intuition of this proof is the following: We have two cases. Case (i): All jobs in A occupy when they run time less than ϵp_j. Then, by time stretching we move the job j by at most ϵp_j and the additional cost is small compared to the processing time p_j of job j. Case (ii): All jobs in A occupy when they run more than ϵp_j. Then, the number of jobs in A is comparitively large, hence we move the job j in a larger interval and, by time stretching, the additional cost is small compared to the sum of completion times of jobs in A. We argue now quantitatively on the two cases:

Case (i): $\sum_{i\in A} p_i < \epsilon p_j$. We use time stretching to move starting time of j to $S_j + \epsilon p_j$, thus jobs in A and job j do not run in parallel any more.

Case (ii): $\sum_{i\in A} p_i \geq \epsilon p_j$. Put the job j so that it completes at time $\frac{C_j}{\epsilon^2}$. Let S_j be the starting time of job j in the initial schedule. Since for every $i \in A$,

$p_i < \epsilon^5 p_j$, we get that at least $\frac{1}{2\epsilon^4}$ jobs in A complete in the time-interval $[S_j + \frac{\epsilon p_j}{2}, S_j + \epsilon p_j]$. Then $\sum_{i \in A} C_i > (S_j + \frac{\epsilon p_j}{2}) \frac{1}{2\epsilon^4} > \frac{\epsilon}{2} \frac{1}{2\epsilon^4} (S_j + p_j) = \frac{1}{4\epsilon^3} C_j$. Thus, $\frac{C_j}{\epsilon^2} < 4\epsilon \sum_{i \in A} C_i$. Summing over all jobs j, we get an increase of at most $4\epsilon OPT$. ∎

6.4.2 Gap in processing times

Suppose that there is a *gap* in the values of the processing times of the jobs in the sense that we can partition the set of jobs into "long" and "short" jobs so that the processing times of all long jobs are much larger than any processing time of a small job. Then the following lemma states that all short jobs are scheduled before any long job is scheduled.

Lemma 3 *Suppose that the set of jobs $\mathcal{T}$ can be partitioned into subsets L and S such that $\max_{j \in S} p_j < \epsilon^5 \min_{j \in L} p_j$. Then, all jobs in S are scheduled before any job in L is scheduled with $1 + \epsilon$ loss.*

Proof: Observe that, because of Lemma 2 above, we can view a near-optimal schedule as a series of time intervals, each time interval accommodating either only jobs in L or only jobs in S. Suppose there are two consecutive time intervals so that a subset L' of L jobs is scheduled before a subset S' of S. If we swap the schedules in those time intervals so that S' is scheduled before L', then we get a better schedule. We compute the change in cost. Let $\mathcal{S}$ and $\mathcal{L}$ are the length of the time interval in which the jobs in S' and L', respectively, are scheduled. The jobs of L' are delayed by $\mathcal{S}$ while the jobs of S' are moved up by $\mathcal{L}$. Thus the change of the cost is

$$\begin{aligned} \mathcal{S}|L'| - \mathcal{L}|S'| &\leq \left(\frac{2}{\min_{j \in L} p_j} - \frac{1}{\max_{j \in S} p_j}\right) \mathcal{L}\mathcal{S} \\ &\leq \left(2 - \frac{\min_{j \in L} p_j}{\max_{j \in S} p_j}\right) \frac{\mathcal{L}\mathcal{S}}{\min_{j \in L} p_j}. \end{aligned}$$

By the hypothesis we have that $\frac{\min_{j \in L} p_j}{\max_{j \in S} p_j} > \frac{1}{\epsilon^5}$, hence the upper bound on the change of cost is a negative number for $\epsilon < 1/2$. ∎

However this lemma cannot be used unless there is such a gap. It turns out that there is an "almost gap". Namely, a separation result allows for partitioning the jobs into "long" jobs and "short" jobs with a few (negligible) "medium" jobs in between. U denotes the upper bound to the optimal cost obtained by processing all jobs sequentially. Note that this upper bound is within a factor of 2 of the optimal.

Definition: *Let L_i denote the set of jobs with processing time $p_j \geq \epsilon^{5i+5}U$, M_i denote the set of jobs j with processing time $\epsilon^{5i+10}UB \leq p_j < \epsilon^{5i+5}U$, and S_i denote the remaining jobs.*

Lemma 4 *If $L_1 \neq \emptyset$, then there is some $i < \log_{1+\epsilon} \frac{1}{\epsilon^{10}}$ such that $OPT(M_i \cup L_i) \leq (1+\epsilon^2)OPT(L_i)$. We call this i the* seperation index.

Proof: The proof is a simple algebraic manipulation. Observe that $L_i = L_1 \cup M_1 \cup M_2 \cup \ldots \cup M_{i-1}$. Taking the inequality $OPT(L_1) > \epsilon^{10}OPT$, and i inequalities, one for each i: $OPT(M_i \cup L_i) > (1+\epsilon^2)OPT(L_i)$ or equivalently $OPT(L_1 \cup M_i \cup M_1 \cup M_2 \cup \ldots \cup M_{i-1}) > (1+\epsilon^2)OPT(L_1 \cup M_1 \cup M_2 \cup \ldots \cup M_{i-1})$ and multiplying the RHS parts and the LHS parts yields the inequality: $OPT > OPT(L_1 \cup M_1 \cup \ldots \cup M_i) > (1+\epsilon^2)^i\epsilon^{10}OPT$, i.e. $(1+\epsilon^2)^i\epsilon^{10} < 1$. Consequently, $i < \log_{1+\epsilon^2} \frac{1}{\epsilon^{10}}$. ∎

Combining the results in Lemmas 3 and 4, we get the following:

Lemma 5 *Let i be the separation index. Then, with a $1+\epsilon$ loss, we schedule all jobs in S_i before all jobs in $L_i \cup M_i$.*

Proof: We will show that the cost does not increase much. By Lemma 3, we know that there is a schedule with a $1+\epsilon$ loss from the optimal, which schedules first the jobs in S_i and then the jobs in L_i. The cost of this schedule is $C(S_i \cup L_i) = OPT(S_i) + OPT(L_i) + M^{S_i}|L_i|$ where M^{S_i} is the maximum makespan of the optimum schedule in S_i.

The cost of a schedule which schedules first S_i and then $M \cup L_i$ is $OPT(S_i) + OPT(M_i \cup L_i) + M^{S_i}|M_i \cup L_i|$. Since, we already know that $OPT(M_i \cup L_i) < (1+\epsilon)OPT(L_i)$ (by Lemma 4), we need to bound $M^{S_i}|M_i|$.

Let $p_{\min}^{L_i}$ be the shortest processing time of a job in L_i, then:

$$\frac{|L_i|^2}{4}p_{\max}^{S_i} < \frac{|L_i|^2}{4}p_{\min}^{L_i} < OPT(L_i) < \epsilon^2 OPT(L_i).$$

Hence $|L_i| < 2\epsilon\sqrt{\frac{OPT(L_i)}{p_{\max}^{S_i}}}$. From the observation in the beginning of this subsection $M^{S_i} < 2\sqrt{p_{\max}^{S_i}OPT(S_i)}$, and thus $M^{S_i}|L_i|$ is at most $2\epsilon OPT$. ∎

The above transformations together with a polynomial algorithm for the preemptive variant [2] yield a PTAS for this problem. We develop the algorithm in two stages. First suppose there are no jobs with processing time $> \epsilon^{10}OPT$. Then we get the algorithm:

(1) Solve the preemptive problem with at most $1+\epsilon$ loss.
(2) Reorder jobs in each type by non-decreasing processing time.
(3) Reorder to obtain a schedule with at most one preemption per interval.
(4) Move large uniprocessor jobs in long intervals so that starting time for uniprocessor jobs is greater than $\frac{1}{\epsilon^2}p_j$, according to Lemma 1.
(5) Consider each preempted job, j, (it is always a single-machine job) and finish it at the time of preemption.

The analysis of this algorithm is straightforward from the above lemmas.

Now, the algorithm for the general case is:

(1) For every $i < \log_{1+\epsilon} \frac{1}{\epsilon^{10}}$, partition the jobs into L_i, M_i, S_i.
(2) Construct a non-preemptive schedule of S_i as above.
(3) Construct an optimal non-preemptive schedule of $L_i \cup M_i$ by exhaustive search.
(4) Concatenate the schedule of S_i and the schedule of $L_i \cup M_i$.
(5) Choose the best schedule over all choices of i.

7 Open Questions

The recent activity on MIN-SUM scheduling problems has led to resolving many of the approximability questions arising in this area as it is shown in Table 1. Several open questions can be picked up by looking at this table. Schuurman and Woeginger in [42] provide also an excellent listing of ten open problems in scheduling theory, including MIN-SUM scheduling problems. In [42], they also discuss related MIN-SUM scheduling problems that we do not address here such as shop scheduling problems and scheduling problems with communication delays.

However, we believe that some of the most interesting open questions can be classified into three directions.

(i) The variants with precedence constraints are still far from being resolved. Any ρ-approximation algorithm, with $\rho < 2$ or any negative result for $1|prec|\sum w_jC_j$ will be of great interest. Note that at least four different 2-approximation algorithms are known (see Table 1) for this variant. Moreover, Woeginger [55], gave a proof that eight special cases of $1|prec|\sum w_jC_j$ (including $1|prec|\sum C_j$) all have the same approximation threshold. Thus, what we can, possibly, expect is a negative result like the following.

Conjecture 1 *There is no* $(2-\epsilon)$*-approximation algorithm for* $1|prec|\sum C_j$.

(ii) The unrelated machines case, with the number of machines as part of the

problem's instance, is also of interest. We know that $R|\ |\sum w_jC_j$ as well as $R|r_j|\sum C_j$ (and hence $R|r_j|\sum w_jC_j$) are MAXSNP-hard [23] and that they can be approximated within a factor of 3/2 [43,46,48] and 2 [47,48], respectively. We also know that their preemptive variants, i.e. $R|pmtn|\sum w_jC_j$ and $R|r_j, pmtn|\sum w_jC_j$, can be approximated within a factor of 2 [47,48] and $2+\epsilon$ [35] respectively. However, the existence of an approximation threshold (for the non-preemptive variants) and a PTAS (for the preemptive variants) are two major open questions. Last but not least note that the complexity of $R|pmnt|\sum C_j$ was an longstanding open question and very recently it was shown to be strongly NP-hard [45]. However, the existence of an approximation result better than than the one to its weighted variant is unknown. It follows, therefore, that there is enough room for improvements here and that any conjecture will be very risky.

(iii) Neither PTASs nor constant approximation algorithms are known for the case with objective function the total (weighted) flow time. Such a result is not however expected for the non preemptive variants due to known non-approximability results [25,26]. Recent work in [9] and [7] (see the discussion in Section 3) for $1|r_j, pmtn|\sum w_jF_j$ provides a strong evidence that the following conjecture holds.

Conjecture 2 *There is a PTAS for* $1|r_j, pmtn|\sum w_jF_j$.

For the parallel machines preemptive unweighted variant $P|r_j, pmtn|\sum F_j$ an $O(\log\min\{\frac{n}{m}, \log P\})$-approximation algorithm is known [26], whereas no negative result has been shown (recall that P refers to the ratio of the largest job processing time over the smallest one,i.e. $P = \frac{\max_j p_j}{\min_j p_j}$). It would be interesting to have either an better approximation factor or a negative result for this variant or even for $P2|r_j, pmtn|\sum F_j$.

8 Conclusion

In this paper, we have discussed approximation algorithms for scheduling problems. We have presented a brief survey on successful methods and techniques used for deriving constant ratio approximation algorithms. However, for a more detailed exposition on this subject and especially on methods that use relaxation techniques on integer programming formulations of the problem, the reader is asked to refer to surveys in the literature. We have focused in Polynomial Time Approximation Schemes and we have presented an extended formal exposition of algorithms in the literature. We presented PTAS algorithms for three problems: $R2||\sum w_jC_j$, $1|\,r_j\,|\sum C_j$, and $P2|fix_j|\sum C_j$. Finally note that it was not our goal here to discuss variants of shop scheduling problem.

References

[1] F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, M. Sviridenko: Approximation schemes for minimizing average weighted completion time with release dates. In Proceedings of 40th FOCS (1999) 32–43.

[2] F. Afrati, E. Bampis, A. V. Fishkin, K. Jansen, C. Kenyon: Scheduling to minimize the average completion time of dedicated tasks. In Proceedings of 20th FSTTCS (2000) 454-464.

[3] F. Afrati, E. Bampis, C. Kenyon, I. Milis: Scheduling to minimize the weighted sum of completion times. Journal of Scheduling **3** (2000) 323–332.

[4] N. Alon, Y. Azar, G. J. Woeginger, T. Yadid: Approximation schemes for scheduling on parallel machines. In Proceedings of 8th SODA (1997) 493–500; Journal of Scheduling **1** (1998) 55–66.

[5] P. Brucker: Scheduling Algorithms. Springer, 1998. See also http://www.mathematik.uni-osnabrueck.de/research/OR/class/

[6] X. Cai, C. Y. Lee, C. L. Li: Minimizing total completion time in two-processor task systems with prespecified processor allocations. Naval Research Logistics, **45** (1998) 231–242.

[7] C. Chekuri, S. Khanna: Approximation schemes for preemptive weighted flow time. Electronic Colloquium on Computational Complexity (ECCC) **8** (2001) 65; In Proceedings of the 34th STOC (2002) 297–305

[8] C. Chekuri, S. Khanna: A PTAS for minimizing weighted completion time on uniformly related machines. In Proceedings of 28th ICALP (2001) 848-861.

[9] C. Chekuri, S. Khanna, An Zhu: Algorithms for minimizing weighted flow time. In Proceedings of 33rd STOC (2001) 84-93.

[10] C. Chekuri, R. Motwani: Minimizing weigthed completion time on a single machine. In Proceedings of 10th SODA (1999) 873–874; Discrete Applied Mathematics **98** (1999) 29–38.

[11] C. Chekuri, R. Motwani, B. Natarajan, C. Stein: Approximation techniques for average completion time scheduling. In Proceedings of 8th SODA (1997) 609–618; SIAM Journal on Computing **31** (2001) 146–166.

[12] F. A. Chudak, D. S. Hochbaum: A half-integer linear programming relaxation for scheduling precedence-constrained jobs on a single machine. Operations Research Letters **25** (1999) 199–204.

[13] M. E. Dyer, L. A. Wolsey: Formulating the single machine sequencing problem with release dates as a mixed integer program. Discrete Applied Mathematics **26** (1990) 255–270.

[14] G. Even, J. Naor, S. Rao, B. Schieber: Divide-and-conquer approximation algorithms via spreading metrics. In Proceeding of 36th FOCS (1995) 62–71; Journal of ACM **47** (2000) 585–616.

[15] A. V. Fishkin, K. Jansen, L. Porkolab: On minimizing average weigthed time completion time of multiprocessor tasks with release dates. In Proceedings of 28th ICALP (2001) 875–886 .

[16] A. V. Fishkin, K. Jansen, L. Porkolab: On minimizing average weigthed time completion time: A PTAS for scheduling general multiprocessor tasks. In Proceedings of 13th FCT-Workshop on Efficient Algorithms (WEA), LNCS-2138 (2001) 495-507.

[17] M. X. Goemans, M. Queyranne, A. S. Shulz, M. Skutella, Y. Wang: Single machine scheduling with release dates. SIAM Journal on Discrete Mathemetics **15** (2002) 165–192.

[18] T. F. Gonzalez, S. Sahni: Flowshop and jobshop schedules:complexity and approximation, Operations Research **26** (1978) 36–52.

[19] R.L. Graham: Bounds on certain multiprocessor anomalies. Bell Systems Technical Journal **45** (1996) 1563–1581.

[20] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan: Optimization and approximation in deterministic sequencing and scheduling. Annals of Discrete Mathematics **5** (1979) 287–326.

[21] L. A. Hall, A. S. Schulz, D. B. Shmoys, J. Wein: Scheduling to minimize average completion time: on-line and off-line approximation algorithms. Mathematics of Operations Research **22** (1997) 513–544.

[22] L.A. Hall, D.B. Shmoys, J. Wein: Scheduling to minimize average completion time: on-line and off-line algorithms. In Proceedings of 7th SODA (1996) 142–151.

[23] H. Hoogeveen, P. Schuurman, G.J. Woeginger: Non-approximability results for scheduling problems with minsum criteria. In Proceedings of 6th IPCO, LNCS 1412 (1998) 353–366; INFORMS Journal of Computing **13** (2001) 157-168.

[24] T. Kawaguchi, S. Kyan: Worst case bound of an LRF schedule for the mean weighted flow-time problem. SIAM Journal on Computing **15** (1986) 1119–1129.

[25] H. Kellerer, T. Tautenhahn, G. J. Woeginger: Approximability and nonappproximability results for minimizing total flow time on a single machine. In Proceedings of 28th STOC (1996) 418–426; SIAM Journal on Computing **28** (1999) 1155–1166.

[26] S. Leonardi, D. Raz: Approximating total flow time on parallel machines. In Proceedings of 29th STOC (1997) 110–119.

[27] J. Y. T. Leung, W. D. Wei: Tighter bounds on a heuristic for a partition problem. Information Processing Letters **56** (1995) 51–57.

[28] F. Margot, M. Queyranne, Y. Wang: Decompositions, network flows, and a precedence constrained single machine scheduling problem. Report No. 2000-29, Department of Mathematics, University of Kentucky, Lexington.

[29] A. Munier, M. Queyranne, A. S. Schulz: Approximation bounds for a general class of precedence-constrained parallel machine scheduling problems. In Proceedings of 6th IPCO, LNCS 1412 (1998) 367–382.

[30] C. H. Papadimitriou: Computational Complexity. Addison Wesley, 1994.

[31] C. Philips, C. Stein, J. Wein: Scheduling jobs that arrive over time. In Proceedings of 4th WADS, LNCS 955 (1995) 290–301; Mathematical Programming B **82** (1998) 199-224.

[32] C. N. Potts: An algorithm for the single machine sequencing problem with precedence constraints. Math. Programming Stud. **13** (1980) 78–87.

[33] M. Queyranne: Structure of a simple scheduling polyhedron. Mathematical Programming **58** (1993) 263–285.

[34] M. Queyranne, A. S. Shulz: Polyhedral approaches to machine scheduling. Preprint No. 408/1994, Department of Mathematics, Technical University Berlin, 1994.

[35] M. Queyranne, M. Sviridenko: A $(2 + \epsilon)$-approximation algorithm for the generalized preemptive open shop problem with minsum objective. In Proceedings of 8th IPCO, LNCS-2081 (2001) 361–369; Journal of Algorithms **45** (2002) 202–212.

[36] R. Ravi, A. Agrawal, P. Klein: Ordering problems approximated: single processor scheduling and interval graph completion. In Proceedings of 18th ICALP, LNCS 510 (1991) 751–762.

[37] S. Sahni: Algorithms for scheduling independent tasks. Journal of ACM **23** (1976) 116–127.

[38] M. W. P. Savelsbergh, R. N. Uma, J. M. Wein: An experimental study of LP-based approximation algorithms for scheduling problems. In Proceedings of 9th SODA (1998) 453–462.

[39] A. S. Schulz, M. Skutella: Random-based scheduling: New approximations and LP lower bounds. In Proceedings of RANDOM'97, LNCS 1269 (1997) 119-133.

[40] A. S. Schulz, M. Skutella: Scheduling-LPs bear probabilities: Randomized approximations for min-sum criteria. In Proceedings of 5th ESA, LNCS 1284 (1997) 416-429; SIAM Journal of Discrete Mathematics **15** (2002) 450-469.

[41] A. S. Schulz, M. Skutella: The power of alpha-points in preemptive single machine scheduling. Journal of Scheduling **5** (2002) 121–133.

[42] P. Schuurman, G.J. Woeginger: Polynomial time approximation algorithms for machine scheduling: Ten open problems. Journal of Scheduling **2** (1999) 203-213.

[43] J. Sethuraman, M. S. Squillante: Optimal scheduling of multiclass parallel machines. In Proceedings of 10th SODA (1999) 963–964.

[44] D. B. Shmoys: Using linear programming in the design and analysis of approximation algorithms: Two illustrative problems. In Proceedings of APPROX'98, LNCS 1444 (1998) 15-32.

[45] R. Sitters: Two NP-hardness results for preemptive minsum scheduling of unrelated parallel machines. In Proceedings of 8th IPCO, LNCS-2081 (2001) 396–405.

[46] M. Skutella: Semidefinite relaxations for parallel machine scheduling. In Proceedings of 39th FOCS (1998) 472–481;

[47] M. Skutella: Convex quadratic programming relaxations for network scheduling problems. In Proceedings of 7th ESA, LNCS 1643 (1999) 127–138.

[48] M. Skutella: Convex quadratic and semidefinite programming relaxations in scheduling Journal ACM 48(2001) 206–242.

[49] M. Skutella, G.J. Woeginger: A PTAS for minimizing the weigthed sum of job completion times on parallel machines. In Proceedings of 31th STOC (1999) 400–407; Mathematics of Operations Research **25** (2000) 63–75.

[50] W. Smith: Various optimizers for single-stage production. Naval Research Logistics Quarterly **3** (1956) 59-66.

[51] E. Torng, P. Uthaisombut: Lower bounds for SRPT-subsequence algorithms for nonpreemptive scheduling. In Proceedings of 10th SODA (1999) 973–974.

[52] J. Turek, U. Schwiegelshohn, J.L. Wolf, P.S. Yu: Scheduling parallel tasks to minimize verage response time. In Proceedings of 5th SODA (1994) 112-121.

[53] R. N. Uma, J. Wein: On the relationship between combinatorial and LP-based approaches to NP-hard scheduling problems. In Proceedings of 6th IPCO (1998) 394–408.

[54] G. J. Woeginger: When does a dynamic programming formulation guarantee the existence of an FPTAS? In Proceedings of 10th SODA (1999) 820-829; Electronic Colloquium on Computational Complexity (ECCC) **8** (2001) 84.

[55] G.J. Woeginger: On the approximability of average completion time scheduling under precedence constraints. In Proceedings of 28th ICALP (2001) 887-897.

[56] L. A. Wolsey: Mixed integer programming formulations for production planning and scheduling problems. Invited talk at 12th International Symposium on Mathematical Programming, MIT, Cambriddge, 1985.