# Approximation Schemes for Minimizing Average Weighted Completion Time with Release Dates

Foto Afrati[1] Evripidis Bampis[2] Chandra Chekuri[3] David Karger[4] Claire Kenyon[5]
Sanjeev Khanna[6] Ioannis Milis[7] Maurice Queyranne[8] Martin Skutella[9] Cliff Stein[10]
Maxim Sviridenko[11]

## Abstract

*We consider the problem of scheduling $n$ jobs with release dates on $m$ machines so as to minimize their average weighted completion time. We present the first known polynomial time approximation schemes for several variants of this problem. Our results include PTASs for the case of identical parallel machines and a constant number of unrelated machines with and without preemption allowed. Our schemes are efficient: for all variants the running time for a $(1+\epsilon)$ approximation is of the form $f(1/\epsilon, m)poly(n)$.*

[1]NTUA, Division of Computer Science, Heroon Polytechniou 9, 15773, Athens, Greece.

[2]LaMI, Université d'Evry, Boulevard Francois Mitterand, 91025 Evry Cedex, France.

[3]Bell Labs, 600-700 Mountain Ave, Murray Hill, NJ 07974. chekuri@research.bell-labs.com.

[4]MIT Laboratory for Computer Science, Cambridge, MA 02139. karger@lcs.mit.edu. Research supported by NSF contract CCR-9624239, an Alfred P. Sloane Foundation Fellowship, and a David and Lucille Packard Foundation Fellowship.

[5]LRI, Bât 490, Université Paris-Sud, 91405 Orsay Cedex, France.

[6]Department of Fundamental Mathematics Research, Bell Labs, 700 Mountain Avenue, Murray Hill, NJ 07974. sanjeev@research.bell-labs.com.

[7]Athens University of Economics, Dept. of Informatics, Patisson 76, 10434 Athens, Greece.

[8]Faculty of Commerce and Business Administration, University of British Columbia, Vancouver, B.C., Canada. E-mail: Maurice.Queyranne@commerce.ubc.ca.. Supported by a research grant from NSERC.

[9]FB Mathematik, Technische Universität Berlin, Germany. E-mail: skutella@math.tu-berlin.de. Research supported by DONET within the frame of the TMR Programme (contract number ERB FMRX-CT98-0202) while staying at C.O.R.E., Louvain-la-Neuve, Belgium, for the academic year 1998/99.

[10]Dartmouth College. cliff@cs.dartmouth.edu. Research supported by NSF Career Award CCR-9624828 and an Alfred P. Sloane Foundation Fellowship.

[11]Sobolev Institute of Mathematics, Novosibirsk, Russia. E-mail:svir@math.nsc.ru.

## 1. Introduction

Recently, there has been a great deal of study of scheduling problems in which the objective is to minimize the average (weighted) completion time. Until about five years ago, almost nothing was known about the approximability of these problems. But since that time constant factor approximation algorithms were found for many variants involving one, identical parallel or unrelated parallel machines, with and without release dates, precedence constraints, and preemption. The techniques introduced in these algorithms are powerful and elegant, but do not seem to lead to the design of *polynomial-time approximation schemes* (PTASs): algorithms that, for any fixed $\epsilon > 0$, find a solution within a $(1+\epsilon)$ factor of the optimum in polynomial time. In this paper, we present the first PTASs for scheduling to minimize average weighted completion time in the presence of release dates in various machine models.

Formally, we are given a set of $n$ jobs where job $j$ has a processing time $p_j$, a positive weight $w_j$ and a release date $r_j$ before which it cannot be scheduled. The objective is to schedule the jobs on a set of $m$ machines so as to minimize $\sum_j w_j C_j$, where $C_j$ denotes the completion time of job $j$ in the schedule. This objective function is referred to as the *average weighted completion time* or *sum of weighted completion times;* when all $w_j = 1$ it is referred to as the *average* or *total* completion time. We will consider three different machine environments: one machine, identical parallel machines, and a fixed number of unrelated parallel machines (on unrelated machines the processing time of a job depends on which machine processes it). We consider variants with and without preemption. In the scheduling notation introduced by Graham et al. [10], a scheduling problem is denoted by a 3-tuple $\alpha \,|\, \beta \,|\, \gamma$, where $\alpha$ denotes the machine environment, $\beta$ denotes the additional constraints on the jobs, and $\gamma$ denotes the objective function. In this paper $\alpha$ will take on the values 1, $P$, $R$, and $Rm$ denoting one machine, parallel (identical) machines, unrelated machines, and a fixed number $m$ (not part of the input) of unrelated machines, respectively. The field $\beta$ will contain value $r_j$ (meaning the problem has release dates). Preemption is permitted if and only if $pmtn$ is part of $\beta$. The final field $\gamma$ will take on the values $\sum w_jC_j$ and $\sum C_j$ to denote the objective functions average weighted and total completion time, respectively.

### 1.1 New Results

In this paper we introduce the first PTASs for scheduling to minimize average weighted completion time with release dates. In particular, we give PTASs for each of the following problems:

- $1|\,r_j\,|\sum C_j$
- $P\,|\,r_j\,|\sum w_jC_j$ and $P\,|\,r_j,\,pmtn\,|\sum w_jC_j$.
- $Rm\,|\,r_j\,|\sum w_jC_j$ and $Rm\,|\,r_j,\,pmtn\,|\sum w_jC_j$.

The second and third results include the first result as a special case, however the PTAS presented for the first problem is much simpler and hence of independent interest. The running time of our algorithm for identical machines is polynomial in the number of machines $m$, while that of our algorithm for unrelated machines is exponential in $m$. We summarize our results in Table 1.

We note, with some surprise, that our algorithms for these long-open problems are not based on poweful new techniques that have recently advanced approximation algorithms but instead use (carefully) rounding and enumeration ideas which are well understood by now.

### 1.2 Previous Results

Several of the simplest average completion time scheduling problems have polynomial time algorithms. The problem $P\,|\,|\sum C_j$ can be solved by the greedy *shortest processing time first* (SPT) rule: schedule the jobs in order of increasing processing time. Generalizing, the problem $1|\,|\sum w_jC_j$ can be solved *Smith's rule*: schedule the jobs in order of non-decreasing $p_j/w_j$ [27]. The problem $R\,|\,|\sum C_j$ can be solved by matching techniques [3, 14].

With the addition of release dates, these rules no longer yield optimal schedules, and almost all such problems in the set we consider are strongly NP-hard. The two exceptions are $1|\,r_j,\,pmtn\,|\sum C_j$, which can be solved via the *shortest remaining processing time* rule (SRPT) [2], and $R\,|\,pmtn\,|\sum C_j$, whose polynomial time solvability is open.

It is therefore natural to consider approximation algorithms for these problems. We define a $\rho$-approximation algorithm to be one that, in polynomial time, returns a solution whose objective value is at most $\rho$ times the optimal objective value.

Many researchers have given $O(1)$-approximation algorithms for average completion time scheduling problems [19, 11, 4, 5, 7, 22, 8, 9, 18, 24, 6, 21, 25]. All these algorithms first formulate a (polynomial-time solvable) relaxation of the problem. They solve this relaxation and use information from the relaxation to obtain an ordering on the jobs and/or an assignment of the jobs to machines. The jobs are then scheduled accordingly on the machines. The relaxations used include preemptive schedules and various linear and convex programs, and the ordering and assignment rules include both deterministic and randomized rules. A series of ideas along these lines have led to many nice algorithms with successively smaller constant factor approximation bounds (see Table 1 for the best bounds), and there is evidence in practice of the power of these techniques [20, 29].

Unfortunately, there seem to be fundamental barriers to turning these algorithms into approximation schemes. First, there are gaps between the objective value of the relaxation and the average completion time of the optimal schedule. Any algorithm that compares itself to the fractional optimum inherits this gap and so cannot be a PTAS. Also, a recent result of Torng and Uthaisombut [28] shows that any algorithm for $1|\,r_j\,|\sum C_j$ that starts with the preemptive relaxation created by the shortest remaining processing time algorithm cannot find an approximation ratio better than $e/(e-1)$, matching the best known upper bound [5].

The technique of time-partitioning was used by Hall et al. [11] and refined by Chakrabati et al [4] to approximate average completion time. The idea is to divide time into geometrically increasing intervals, and in each interval schedule all jobs that, by a certain time, had been released but not yet been processed. Thus the overall problem can be solved by solving a series of subproblems, each of which involves scheduling in an interval a set of jobs without release dates. This idea is a basic building block of our algorithms.

The only PTAS for a strongly NP-hard problem involving average completion time is the recent result of Skutella and Woeginger [26] who give a PTAS for the problem $P\,|\,|\sum w_jC_j$. Their algorithm, generalizing a result of Alon et al. [1], is based on ratio-partitioning. Jobs are grouped according to their $p_j/w_j$ ratios, which are then geometrically rounded. A near optimal schedule is computed for each group, and the schedules are concatenated according to Smith's rule. Since this technique relies on the fact that sequencing is easy on each machine in the absence of release dates, it cannot be easily generalized to scheduling problems involving release dates. Time-partitioning is better suited to our goals.

Recently, Hoogeveen, Schuurman, and Woeginger [12]

| Problem | Previous Best | PTAS Running Time |
|---|---|---|
| $1 \mid r_j \mid \sum C_j$ | 1.58 [5] | $O(n \log n + 2^{\mathrm{poly}(1/\epsilon)})$ |
| $1 \mid r_j \mid \sum w_j C_j$ | 1.69 [8] | $O(2^{\mathrm{poly}(1/\epsilon)} n + n \log n)$ |
| $1 \mid r_j,\ pmtn \mid \sum w_j C_j$ | 4/3 [21] | $O(2^{\mathrm{poly}(1/\epsilon)} n + n \log n)$ |
| $P \mid r_j \mid \sum w_j C_j$ | 2 [22] | $O((m+1)^{\mathrm{poly}(1/\epsilon)} n + n \log n)$ |
| $P \mid r_j,\ pmtn \mid \sum w_j C_j$ | 2 [22] | $O(2^{\mathrm{poly}(1/\epsilon)} n + n \log n)$ |
| $Rm \mid r_j \mid \sum w_j C_j$ | 2 [25] | $O(f(m, 1/\epsilon)\mathrm{poly}(n))$ |
| $Rm \mid r_j,\ pmtn \mid \sum w_j C_j$ | 3 [25] | $O(f(m, 1/\epsilon) n + n \log n)$ |
| $Rm \mid \mid \sum w_j C_j$ | 3/2 [24] | $O(f(m, 1/\epsilon) n + n \log n)$ |

**Table 1. Summary of results.**

showed a number of non-approximability results for average completion time scheduling problems. In particular, they showed that the problems $R \mid r_j \mid \sum C_j$ and $R \mid \mid \sum w_j C_j$ do not have a PTAS unless P=NP. They also conjectured that $1 \mid r_j \mid \sum C_j$ and $P \mid \mid \sum w_j C_j$ do have PTASs but $P \mid r_j \mid \sum C_j$ does not.

### 1.3 Our Approach

Our approach to approximation is to perform several transformations that simplify the input problem without dramatically increasing the objective value, such that the final result is amenable to a fast dynamic programming solution. Many of our transformations are thought experiments applied to the optimal solution to argue that some solution nearly as good has very simple structure. Others are actual simplifying transformations of the input that do not significantly increase the objective value.

The first transformation is *geometric rounding:* we round processing times and release dates to integral powers of $(1+\epsilon)$. This changes no quantity by more than $1+\epsilon$, ensuring a small change in the objective function. However, rounding guarantees that there are only a small number of distinct processing times and release dates to worry about. Since (as we will see) release dates are the only places where scheduling decisions really need to be made, having few of them simplifies the problem. Rounding lets us break time into geometrically increasing intervals, where intervals start and end at release dates, which is useful for dynamic programming.

Our second transformation is *time stretching.* We add small amounts of idle time spread throughout the schedule. These additions change completion times only slightly, but can be used to "clean up" the schedule. In particular, if a job is "large" compared to an interval in which it executes, we can advance it into the idle time in a later interval where it is small. This lets us assume that most jobs are small. Small jobs are a lot like fractional jobs, and fractional problem solutions are often easier to find.

A careful application of these first two techniques enables us to prove that a surprisingly simple algorithm is in fact a PTAS for the one machine, unweighted case: this algorithm is in essence SPT except for exhaustive search on the last few jobs scheduled. The ideas of enumerating the schedules of the jobs at the end of the schedule (as opposed to enumerating "large" jobs throughout the schedule), and enumerating after scheduling the other jobs are, to the best of our knowledge, new. This approach can be extended to the multi-machine setting but not to weighted completion times.

For the problems with job weights, the algorithms are more sophisticated. A simple observation is that whenever two small jobs are available we can execute them in the order of increasing $p_j/w_j$ (Smith's rule). However large and small jobs interact in complex ways when weights are present and we use structured enumeration to explore these interactions.

Our third transformation, *weight-shifting,* compacts the set of jobs released at any point in time and helps in the above mentioned enumeration. If many jobs are released at once, we know that some of them will have to wait to be processed. Shifting refers to the process of moving the excess jobs to the next interval. For small jobs this can be done by prioritizing them by $p_j/w_j$ and retaining only those that can be executed in the current interval. For large jobs of each particular size, we order them in decreasing weight and simply retain the maximum number that could be potentially scheduled in the given interval. After the transformation, the processing time of the jobs released in any interval is a small multiple of the length of the interval. Though simple, this transformation is a key idea. Coupled with time stretching it shows that every job can be scheduled within $O(1/\epsilon^3)$ intervals after its release date. Thus, our dynamic program only needs to remember limited history, which reduces the state space.

When multiple machines are present the above transformations can still be applied. However several other aspects become complex and we defer the details till Section 4.

## 1.4 History of this Work and Overview of Paper

As mentioned above, the design of algorithms to minimize average completion time has been a very active area of research over the last five years. However, prior to this work, no PTAS existed for any problem with release dates; the design of a PTAS for the simplest such problem, $1|\,r_j\,|\sum C_j$, was a natural next question. No fewer than five groups of authors (Afrati, Bampis, Kenyon and Milis; Chekuri and Khanna; Karger and Stein; Queyranne and Sviridenko; Skutella) independently discovered a PTAS for the problem $1|\,r_j\,|\sum C_j$. All five groups used the natural idea of geometric rounding, but with variations that are more or less amenable to generalizations such as handling weights, preemption, and multiple machines. In Section 3 we present the algorithm due to Karger and Stein, which is by far the simplest but unfortunately cannot be extended to handle job weights.

In Section 4 we present the most general algorithm, due to Chekuri and Khanna. It handles an arbitrary number of identical machines, with release dates, job weights, and with or without preemption, i.e., $P|r_j|\sum w_j C_j$ and $P|r_j, pmtn|\sum w_j C_j$.

In Section 5 we present PTASs for a fixed number of unrelated machines, $Rm|r_j|\sum w_j C_j$ and $Rm|r_j, pmtn|\sum w_j C_j$. Skutella obtained the first PTAS for the problem $Rm|r_j|\sum C_j$ (no weights), and Afrati *et al.* obtained the first PTAS for the problem $Rm||\sum w_j C_j$ (no release dates), but here we present the general algorithm due to Chekuri and Khanna, which builds upon the ideas used in Section 4.

# 2. Preliminaries

In this section we discuss some general techniques and lemmas that apply throughout our paper. We aim to transform any input into one with simple structure. This will help for efficient enumeration and dynamic programming techniques.

Our approach 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 function value by $1 + O(\epsilon)$, so we can perform a constant number of them while still staying within $1 + O(\epsilon)$ of the original optimum. When we describe such a transformation, we shall say it produces $1 + O(\epsilon)$ *loss*.

To simplify notation we will assume throughout the paper that $1/\epsilon$ is integral (and in particular that $\epsilon \leq 1/4$). We use $C_j$ and $S_j$ to denote the completion and start time respectively of job $j$, OPT to denote the objective value of the optimal schedule.

The properties we prove in this section pertain to the case of single or identical parallel machines, but the ideas will be used in modified or generalized form for unrelated machines as well.

## 2.1 Geometric Rounding

Our first simplification creates a well-structured set of possible processing times and release dates.

**Lemma 2.1** *With $1 + \epsilon$ loss, we can assume that all processing times and release dates are integer powers of $1 + \epsilon$.*

**Proof Sketch.** We round up in two steps. First multiply every release date and processing time by $1 + \epsilon$; this increases the objective by the same amount (we are simply changing time units). 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 an arbitrary integer $x$, we define $R_x := (1 + \epsilon)^x$. As a result of Lemma 2.1 we can assume that all release dates are of the form $R_x$ for some integer $x$. We partition the time interval $(0, \infty)$ into disjoint intervals of the form $I_x := [R_x, R_{x+1})$ (Lemma 2.2 below ensures that no jobs are released at time 0). 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 $\epsilon$ times its start time.

## 2.2 Large and Small Jobs

In all of our algorithms, 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. Our notion of small versus large changes from algorithm to algorithm. We say that a job is *small* with respect to an interval if its size is less than $\epsilon$ (in the single-machine case), $\epsilon^2$ (in the parallel case), or $\epsilon^3$ (in the unrelated case) times the size of the interval where it runs. It is useful to show that jobs are not arbitrarily large:

**Lemma 2.2** *With $1 + \epsilon$ loss, we can enforce $r_j \geq \epsilon p_j$ for all jobs $j$.*

**Proof Sketch.** Multiply every *completion* time by $1 + \epsilon$ and increase start times to match (without changing job sizes). It is easy to verify that this gives a feasible schedule. 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 $\epsilon t \geq \epsilon p_j$.

It follows that we can increase release dates to enforce $r_j \geq \epsilon p_j$, and still have a $(1 + \epsilon)$-optimal schedule. □

### 2.3 Crossing Jobs

While most jobs run completely inside one interval, some jobs *cross* over multiple intervals, creating complexity we would like to avoid. The next two lemmas simplify this problem: we can assume that no job crosses too many intervals, and we can assume there are no small crossing jobs at all.

**Lemma 2.3** *Each job crosses at most* $s := \lceil \log_{1+\epsilon}(1+\frac{1}{\epsilon}) \rceil$ *intervals.*

**Proof Sketch.** Suppose job $j$ starts in interval $I_x = [R_x, R_{x+1})$. Since $R_x \geq r_j \geq \epsilon p_j$ (Lemma 2.2), we have $I_x = \epsilon R_x \geq \epsilon^2 p_j$. The $s$ intervals following $x$ sum in size to $I_x/\epsilon^2 \geq p_j$. □

To prove the second lemma, we make first use of *time-stretching*, a technique mentioned in the Introduction that is used often in subsequent sections. We describe the technique in some detail in this first use of it; later, similar, uses will be abbreviated due to space limitations.

**Lemma 2.4** *With* $1+\epsilon$ *loss we restrict attention to schedules in which no small job crosses an interval.*

**Proof.** Suppose we increase the size of each of our geometrically increasing time intervals by $1+\epsilon$. We can move jobs with the increase so that they continue to execute in or cross the same intervals. This stretching of intervals increases the completion time of each job by at most a $1+\epsilon$ factor, so the increase in objective value is bounded by the same factor.

At most one job $j$ can cross out of any given interval $I_x$; suppose it is small (size at most $\epsilon I_x$). Since at most $I_x$ units of work are processed in interval $I_x$, the expansion of the interval creates $\epsilon I_x$ units of empty space in the interval. The newly created empty space can be used to completely process job $j$, so it need no longer cross the interval. Thus we have given a $1+\epsilon$ times optimal schedule with no small crossing jobs. □

## 3. Scheduling on a single machine with unit weights

In this section we present a very simple and easy to analyze approximation scheme for the problem $1|\, r_j \,| \sum C_j$. In the end of the section, we sketch how a somewhat more involved analysis can lead to a better dependence on $\epsilon$.

Recall that SPT as the algorithm that repeatedly chooses, among all jobs that have been released but not processed, the one with the smallest processing time and runs this job to completion. Our PTAS is as follows:

1. Run SPT until at most $\frac{3}{\epsilon^7}$ jobs are left; during the run assume that each job $j$ is released at time $\max\{r_j, \frac{p_j}{\epsilon^2}\}$, and that time has been stretched by a $(1+3\epsilon)$-factor.
2. Enumerate all orderings of the remaining jobs to find the best one.

The running time of this algorithm is $O(n \log n)$ (to sort the jobs for SPT) plus $(3/\epsilon^7)!$. A tighter analysis of the first part (and corresponding change in the threshold for ending SPT) can improve the enumeration time to $1/\epsilon^5!$. A more careful enumeration technique improves the enumeration time to $2^{o(1/\epsilon^3)}$.

In the remainder of this section, we prove that the algorithm yields a $1+\epsilon$ approximation to the optimum. For this section we say that a job $j$ running in interval $I_x$ is *small* if $p_j \leq \epsilon I_x$, and large otherwise. We motivate our algorithm with the following simple lemma.

**Lemma 3.1** *If in the optimum schedule all jobs are small, then the above algorithm gives a* $(1+\epsilon)$ *times optimum solution.*

**Proof.** Consider the optimum schedule. The fact that all jobs are small means that job $j$ running in interval $I_x$ satisfies $S_j \geq R_x = I_x/\epsilon \geq p_j/\epsilon^2$. Thus increasing release dates as in the algorithm does not change the feasibility (and optimality) of the optimum schedule. With these new release dates, consider the *preemptive* version of the problem, which can be solved optimally using the shortest *remaining* processing time first (SRPT) rule (this algorithm runs like SPT, but may preempt a running job when a shorter job than it is released). This solution's objective value is clearly no more than the non-preemptive optimum. To convert this solution into a non-preemptive schedule, note that preemptions only happen at release dates, which occur at the ends of intervals. For an interval $I_x$, whichever job (if any) is preempted at the end of $I_x$ is small in $I_x$ (since it is small when released). Thus, by stretching each interval by a $1+\epsilon$ factor, we add an extra $\epsilon I_x$ space, which is enough to let that job complete without being preempted.

Stretching the intervals only increased completion times, and thus the objective, by at most a $1+\epsilon$ factor, giving us a non-preemptive schedule that is within $1+\epsilon$ of the optimum preemptive schedule, and thus within $1+\epsilon$ of the optimum non-preemptive schedule. □

Our only problem, then, is that the optimum schedule may require some jobs to run when they are large. We use time stretching to modify the optimum schedule to make most of these large jobs small, and apply enumeration to the remaining few large jobs. This essentially lets us reduce to the case covered by the previous lemma. For a given instance of this problem, let OPT be the value of the optimal schedule. Clearly, at most $\frac{1}{\epsilon^7}$ jobs can complete after a *threshold* time $t := \epsilon^7\text{OPT}$. We now show that all large jobs that run before time $t$ in the optimum solution can be delayed until they are small with only $(1+\epsilon)$ loss.

Our proof uses time-stretching heavily. We will expand each interval by a $1+O(\epsilon)$ factor, adding idle time into each interval. This idle time will provide room for jobs that were large in their optimally scheduled interval to advance to an interval where they are small.

**Lemma 3.2** *There exists a $(1+3\epsilon)$–optimal schedule in which, for each job $j$, $S_j \geq \min\{\frac{p_j}{\epsilon^2}, t\}$.*

Note that $S_j \geq p_j/\epsilon^2$ means that job $j$ is small when it runs.

**Proof.** Consider an optimal schedule and let $x(j)$ be the index of the interval in which job $j$ starts. Since small jobs have $p_j \leq \epsilon I_{x(j)} = \epsilon^2 R_{x(j)} \leq \epsilon^2 S_j$, they satisfy the lemma; the large jobs, however, may not. We will show that we can move the large jobs later, so that the resulting schedule is both feasible and $(1+3\epsilon)$ optimal.

To deal with the large jobs, move each large job that starts before $t$ forward for $k := \lceil \log_{1+\epsilon} \frac{1}{\epsilon^4} \rceil$ intervals. For any job $j$, let $x'(j) = x(j) + k$ and $S_j^{'}$ be the new starting time of job $j$. Then

$$p_j \leq \frac{r_j}{\epsilon} \leq \frac{R_{x(j)}}{\epsilon} \leq \epsilon^3 R_{x'(j)} \leq \epsilon^3 S'_j, \qquad (1)$$

and the condition of the lemma is fulfilled—job $j$ is small.

Of course we need to make room for the jobs moved forward to run in their new locations. To do so, increase the size of every interval by a $1+\epsilon$ factor. There are at most $1/\epsilon$ large jobs that landed in interval $I_{x'}$, since each one originated in interval $I_{x'-k}$ and had $p_j > \epsilon I_{x'-k}$. Each, by (1), has $p_j \leq \epsilon^3 R_{x'} = \epsilon^2 I_{x'}$. Therefore, the total processing time of these jobs is at most $\epsilon I_{x'}$, and all these jobs can fit into the extra $\epsilon I_{x'}$ space created by stretching the interval.

One thing can go wrong: interval $I_{x'}$ might be entirely covered by a crossing job $c$, preventing us from inserting the extra $\epsilon I_{x'}$ units of space. However, we can instead place this extra space (and all the jobs that want to land in it) immediately *before* $c$. This does not increase the completion time of any job. By Lemma 2.3, we know that $c$ crosses at most $s = \log_{1+\epsilon}(1/\epsilon)$ intervals, so our new space "backs up" by at most $s$ intervals to an interval $I_{y'} \geq \epsilon I_{x'}$. Since the jobs that wanted to land in $I_{x'}$ had size at most $\epsilon^2 I_{x'}$, they will have size at most $\epsilon I_{y'}$ and will therefore be small in the interval where they run.

It remains to bound the cost of the new solution. Expanding the schedule by $1+\epsilon$ increased all costs by $1+\epsilon$. Now we need only bound the added cost of the large jobs we moved forward. Jobs advancing from $I_x$ end up in interval $I_{x+k}$ with completion time at most $R_{x+k+1} = (1+\epsilon)R_x/\epsilon^4$, and there are at most $1/\epsilon$ of them. The last interval from which we advance jobs ends at time $t$. Thus the total completion time of the advanced jobs is

$$\begin{aligned}\sum_{R_x<t} \frac{1}{\epsilon}\frac{(1+\epsilon)R_x}{\epsilon^4} &\leq \frac{t}{\epsilon^5}\sum_{i\geq 0} 1/(1+\epsilon)^i \\ &= \frac{t}{\epsilon^5}\frac{(1+\epsilon)^2}{\epsilon} \\ &= \epsilon \cdot \text{OPT}(1+\epsilon)^2 \leq 2\epsilon\text{OPT}\end{aligned}$$

as desired. Since we stretched by a $1+\epsilon$-factor and then added an additional $2\epsilon \cdot \text{OPT}$ cost, the resulting schedule is $(1+3\epsilon)$-optimal. □

We combine the previous two lemmas to bound the performance of our algorithm. Consider the input modified as in Lemma 3.2. Its optimum schedule has large jobs only after time $t$. We now argue as in Lemma 3.1 that all the small jobs can be rescheduled to run in SPT order with $1+\epsilon$ loss. To see this, fix the large jobs in place and reorder all the small jobs around them using SRPT. This schedule is preemptively optimal (SRPT is still optimal in the presence of the fixed large jobs) and can be made non-preemptive with $1+\epsilon$ loss.

In sum, by manipulating the release dates we lose at most a factor $(1+3\epsilon)$ before time $t$; by the argument above, we might loose another $(1+\epsilon)$ factor due to the SPT rule. As a result, there can be at most $(1+\epsilon)(1+3\epsilon)\frac{1}{\epsilon^7} \leq \frac{3}{\epsilon^7}$ jobs left at time $t = \epsilon^7\text{OPT}$; thus the first step ends before time $t$ as required to apply Lemma 3.2. In the second step we have at most $\frac{3}{\epsilon^7}$ jobs remaining; hence we can simply enumerate all possible orderings and take the smallest one. Thus we have shown the following

**Theorem 3.3** *We can find a $(1+\epsilon)$–optimal solution to $1|\,r_j\,|\sum C_j$ in $O(n\log n + \frac{3}{\epsilon^7}!)$ time.*

By a more involved analysis, a better dependence on $\epsilon$ is possible. First, we notice that while there can be at most $1/\epsilon$ large jobs in each interval, at most one of these is a crossing job. We can bound the sum of the sizes of all the non-crossing large jobs by the size of the interval; this allows us to move them forward only $\frac{1}{\epsilon^2}$ intervals. Second, for the large crossing jobs, we can try to move them forward by the minimum amount necessary to make them small. This may result in too many jobs arriving in any one interval, however, by scheduling them in SPT order, we can achieve better bounds (based on a potential function). Using both these ideas lets us continue running SPT until only $1/\epsilon^5$ jobs remain. This improves the running time to $\epsilon^{-5}!$.

We can also improve the time cost of the enumeration step. After time $t$, there are only $O(\log_{1+\epsilon} 1/\epsilon)$ time intervals where jobs can run and only $O(\log_{1+\epsilon} 1/\epsilon)$ distinct job sizes that are large enough to be hard to schedule. Instead of enumerating all possible orderings we can simply enumerate over how many jobs of each size are executed in each interval. This improves the time of the enumeration step to $2^{o(1/\epsilon^3)}$.

## 4. Scheduling on identical parallel machines

In this section we sketch an approximation scheme for the scheduling problems $P\,|\,r_j\,|\sum w_jC_j$ and $P\,|\,r_j,\,pmtn\,|\sum w_jC_j$. The approximation schemes presented here contain our central ideas for the parallel and weighted case; in the next section, we build on the ideas and techniques presented here to develop approximation schemes for models with a constant number of unrelated parallel machines. Our approach is based on dividing the time horizon into a sequence of blocks, each containing a constant number of intervals dates, and then using dynamic programming over the blocks. There are three main ideas needed to make this approach work. First, we show that there exists a $(1+\epsilon)$-approximate schedule such that any two consecutive blocks interact with each other in only $m^{O(1)}$ different ways. Second, we show that there exists a $(1+\epsilon)$-approximate schedule such that one can represent *compactly* at each block the information about jobs that were released earlier and have not been yet completed. Finally, we show that there exists a $(1+\epsilon)$-approximate procedure for scheduling jobs within a block, subject to constraints specifying interactions between the block and its neighboring blocks. Put together, these elements give us our approximation scheme. We start with the non-preemptive case and then sketch in Subsection 4.5 the modifications needed for the preemptive case.

### 4.1. The structure of parallel schedules

**Lemma 4.1** *Consider an instance of $P\,|\,r_j\,|\sum w_jC_j$ or $P\,|\,r_j,\,pmtn\,|\sum w_jC_j$ with two small jobs $j$ and $k$ such that $r_j \le r_k$ and $\frac{p_j}{w_j} \le \frac{p_k}{w_k}$. There exists a $(1+\epsilon)$–approximate schedule in which $S_j \le S_k$ for all such pairs of jobs and no small job is ever preempted.*

**Proof Sketch.** As in Lemma 3.1, we can consider expanding time by $1+\epsilon$ and running the small jobs (without preempting) using Smith's rule. □

As a result of Lemma 4.1 we can order all small jobs released at $R_x$ according to their ratio $\frac{p_j}{w_j}$ and consider them for scheduling only in that order. Let $T_x$ and $H_x$ denote the small and large jobs released at $R_x$ ($T$ for tiny and $H$ for huge). Note that in this section small means an $\epsilon^2$ fraction of the interval. Let $p(S)$ denote the sum of the processing times of the jobs in set $S$. The next lemma says that any input instance $I$ can be modified with $1+\epsilon$ loss to an instance $I'$ so that the total size of the small and large jobs released at any release date $R_x$ is $O(mI_x)$. This lemma plays an important role in our implementation of the dynamic programming framework since it allows us to represent compactly information about unfinished jobs as we move from one block to the next.

**Lemma 4.2** *An instance of $P\,|\,r_j\,|\sum w_jC_j$ can be modified with $1+O(\epsilon)$ loss to an instance $I'$ such that the following conditions hold.*

- *$p(T'_x) \le 2mI_x$ for all $x$.*
- *The number of distinct job sizes in $H'_x$ is at most $\lfloor 1 + 4\log_{1+\epsilon}\frac{1}{\epsilon}\rfloor$.*
- *The number of jobs of each distinct size in $H'_x$ is at most $\frac{m}{\epsilon^2}$.*

**Proof.** Consider the input instance $I$. The total processing time available in interval $I_x$ is $mI_x$. Order the small jobs in $T_x$ by non-decreasing ratios $\frac{p_j}{w_j}$ and pick jobs according to this order until the processing time of jobs picked just exceeds $mI_x$. Picking jobs according to this order is justified by Lemma 4.1. The remaining jobs, which are released at $R_x$ but cannot be processed in $I_x$, can safely be moved to the next release date $R_{x+1}$.

For each job $j$ in $H_x$, Lemma 2.2 yields $R_x \ge \epsilon p_j$. On the other hand, since $j$ is large we get $p_j \ge \epsilon^2 I_x = \epsilon^3 R_x$. Since all job sizes are powers of $1+\epsilon$, the number of distinct job sizes in $H_x$ is as claimed. Within a particular size we can order jobs by non-increasing weights. The number of jobs of each size class that can be executed in the current interval is limited to $\frac{mI_x}{\epsilon^3 R_x} = \frac{m}{\epsilon^2}$. □

### 4.2. The dynamic programming framework

We now present an overview of our dynamic programming framework. The implementation of this framework for the parallel machine case requires additional ideas, presented in Subsections 4.3 and 4.4. However, as we sketch at the end of this subsection, an approximation scheme for $1\,|\,r_j\,|\sum w_jC_j$ immediately follows from our framework.

The basic idea is to decompose the time horizon into a sequence of *blocks*. A block is a set of $s = \lceil \log_{1+\epsilon}(1+\frac{1}{\epsilon})\rceil$ consecutive intervals. Let $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_\ell$ be the partition of the time interval $[\min_j r_j, D)$ into blocks where $D$ is an upper bound on the schedule makespan (we can bound $D$ crudely by $(\sum_j p_j + \max_j r_j)$). Our goal is to do dynamic programming with blocks as units. There is interaction between blocks since jobs from an earlier block can cross into the current block. However by the choice of the block size and Lemma 2.3, no job crosses an entire block. In other words jobs that start in $\mathcal{B}_i$ finish either in $\mathcal{B}_i$ or $\mathcal{B}_{i+1}$. A *frontier* describes the potential ways that jobs in one block finish in the next. An incoming frontier for a block $\mathcal{B}_i$ specifies for each machine the time at which the crossing job from $\mathcal{B}_{i-1}$ finishes on that machine.

**Lemma 4.3** *There exists a $(1+\epsilon)$-approximate schedule which considers only $(m+1)^{s/\epsilon}$ feasible frontiers between any two blocks.*

**Proof.** By Lemma 2.4 we can restrict attention to schedules in which small jobs never cross an interval. Each block consists of a fixed number $s$ of intervals. Fix an optimal schedule and consider any machine in a block $\mathcal{B}_i$. A large job $j$ continuing from the preceding block finishes in one of the $s$ intervals of block $\mathcal{B}_i$ which we denote by $I_{x(j)}$. We can round up $C_j$ to $C'_j$ where $C'_j = R_{x(j)} + i \cdot \epsilon I_{x(j)}$ for some integer $0 \leq i \leq \frac{1}{\epsilon} - 1$. This will increase the schedule value by only a $1+\epsilon$ factor. Thus we can restrict the completion times of crossing jobs to $\frac{s}{\epsilon}$ discrete time instants. Each machine realizes one of these possibilities. A frontier can thus be described as a tuple $(m_1, \ldots, m_{s/\epsilon})$ where $m_i$ is the number of machines with crossing jobs finishing at the $i^{\text{th}}$ discrete time instant. Therefore there are at most $(m+1)^{s/\epsilon}$ frontiers to consider. □

Let $\mathcal{F}$ denote the possible set of frontiers between blocks. The high level idea behind the dynamic programming is now easy to describe. The dynamic programming table entry $O(i, F, U)$ stores the minimum weighted completion time achievable by starting the set $U$ of jobs before the end of block $\mathcal{B}_i$ while leaving a frontier of $F \in \mathcal{F}$ for block $\mathcal{B}_{i+1}$. Given all the table entries for some $i$, the values for $i+1$ can be computed as follows. Let $W(i, F_1, F_2, V)$ be the minimum weighted completion time achievable by scheduling the set of jobs $V$ in block $\mathcal{B}_i$, with $F_1$ as the incoming frontier from block $\mathcal{B}_{i-1}$ and $F_2$ the outgoing frontier to block $\mathcal{B}_{i+1}$. We obtain the following equation.

$$O(i{+}1, F, U) = \min_{F' \in \mathcal{F}, V \subset U} \big(O(i, F', V) + W(i{+}1, F', F, U{-}V)\big)$$

There are two difficulties in implementing the dynamic programming. First, we cannot maintain the table entries for each possible subset of jobs in polynomial time. Therefore we need to show the existence of approximate schedules that have compact representations for the set of subsets of jobs remaining after each block. Second, we need a procedure that computes the quantity $W(i, F_1, F_2, V)$. The next two subsections describe how to achieve these two objectives. However, at this point, we can already sketch an approximation scheme for $1 \,|\, r_j \,|\, \sum w_j C_j$.

By Lemma 4.2 we know that the processing time of all the jobs released at any release date $R_x$ is $O(I_x)$. If we stretch our intervals by $1+\epsilon$, we create enough idle space in interval $I_{x+O(s)}$ to execute all this work. Thus we can assume that all work finishes within $O(s)$ intervals of its release. This means that we can explicitly maintain a list of large jobs that remain to be scheduled as we move from one block to the next.

To maintain information about small jobs, we use the fact that the small jobs arriving at any given release date are executed in the order specified by Smith's ratio rule. We partition this *ordered* list into $O(1/\epsilon^2)$ pieces of roughly equal size and show that time stretching lets us schedule an integral number of these pieces in each block. Thus information can be compactly maintained for small jobs as well. Finally, the procedure for computing $W(i, F_1, F_2, V)$ is trivial for a single machine; simply try all possible ways of scheduling the large jobs in $V$ (there are only $O(1/\epsilon^4)$ large jobs to be considered), and place the small jobs in $V$ in accordance with Smith's ratio rule. However, as we see in the next two subsections, both these steps require significant additional ideas for the parallel machine case.

## 4.3. Compact representation of job subsets

The difficult part in the dynamic programming is to show that it is sufficient to maintain information in the table for only a few (polynomial) subsets of jobs. Recall that $H_x$ and $T_x$ denote the large and small jobs released at $R_x$. Let $X_{xi}$ and $Y_{xi}$ denote the set of small and large jobs released at $R_x$ that are scheduled in block $\mathcal{B}_i$. Let $U_{xi}$ and $V_{xi}$ denote the set of small and big jobs among jobs released at $R_x$ that remain *after* block $\mathcal{B}_i$. Our goal is to show that there exist $(1+\epsilon)$–approximate schedules with compact representations for these sets. Let $b(x)$ denote the block containing the interval $I_x$.

We start with small jobs. Recall that we ordered the set $T_x$ using Smith's ratio rule. The lemma below shows that each block $\mathcal{B}_i$ has enough space to execute a constant fraction of small jobs released at each of the release dates in the preceding blocks $\mathcal{B}_1, \ldots, \mathcal{B}_{i-1}$.

**Lemma 4.4** *There is a $(1+2\epsilon)$–approximate schedule such that for each release date $R_x$ and each $i > b(x)$, either*

- *$p(X_{xi}) \geq \epsilon^2 m I_x$, or*
- *$p(X_{xi}) < \epsilon^2 m I_x$ and $p(X_{xk}) = 0$ for all $k > i$.*

**Proof.** Consider an optimal schedule that does not satisfy the properties of the lemma. Fix a release date $R_x$ for which the conditions of the lemma are violated. Let $k$ be the smallest index such that $p(X_{xk}) < \epsilon^2 m I_x$ and let $l$ be the smallest index greater than $k$ with $p(X_{xl}) > 0$. We simply move jobs from $X_{xl}$ to block $\mathcal{B}_k$ until either $\epsilon^2 m I_x \leq p(X'_{xk}) \leq \epsilon^2 m I_x + \epsilon^2 I_x$, or $p(X'_{xl}) = 0$ . This is possible since jobs in $X_{xl}$ are small. We repeat the procedure until the conditions of the lemma are satisfied for $R_x$.

It is clear that the procedure terminates. The processing time of the new jobs assigned to a block from $T_x$ cannot be more than $\epsilon^2 m I_x$. We apply a similar transformation for each $R_x$. A simple volume summation argument shows that

$$\sum_{x:b(x)<i} \epsilon^2 m I_x \leq \epsilon m I_y \qquad (2)$$

where $I_y$ is the first interval in block $\mathcal{B}_i$. Now we have for each block a set of new jobs that are assigned to it from later blocks but have not been scheduled. We schedule these as follows. In block $\mathcal{B}_i$ on machine $M_j$ let $t_j$ be the first time at which a job can be started. Note that $t_j$ exists since no job spans a block. We create a space of $2\epsilon I_y$ at the point $t_j$ by pushing forward the previously scheduled job. We use this space to greedily fill the new jobs assigned to each block. This is possible by (2). The new jobs have their completion time reduced and the old jobs have their completion time increased by at most a $1 + 2\epsilon$ factor. $\square$

Using the same idea as in the proof of Lemma 4.4, we can partition the ordered set $T_x$ into $O(\frac{1}{\epsilon^2})$ sets such that in every block, an integral number of these sets is scheduled. Thus we can capture $U_{xi}$ for $i > b(x)$ by specifying the number of these sets that have been scheduled. Observe that this is only a constant amount of information. We however did not deal with the case of $i = b(x)$. We again use the idea in the proof of Lemma 4.4 but now we can only show that $p(X_{xi}) \geq \min\{p(T_x), \epsilon^2 I_x\}$ for $i = b(x)$. This involves scheduling small jobs at the end of the frontier of block $\mathcal{B}_i$, in particular right after the crossing job with the smallest finish time among all crossing jobs. This violates our earlier property that jobs in $X_{xi}$ start in $\mathcal{B}_i$. We treat these jobs as a special case and for simplicity of presentation we ignore the full details in this extended abstract. To summarize, we can specify $U_{xi}$ by an integer in $[0, \frac{1}{\epsilon^2}]$ for $i > b(x)$ and by an integer in $[0, \frac{m}{\epsilon^2}]$ for $i = b(x)$.

We now turn our attention to big jobs. By Lemma 4.2, there are $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$ distinct size classes in $H_x$. We order jobs of the same size by decreasing weights. It is easy to see that jobs in $H_x$ can be treated as small jobs from block $\mathcal{B}_{b(x)+3}$ on. For $i = b(x), b(x) + 1, b(x) + 2$, we specify the set $Y_{xi}$ by explicitly listing the number of jobs from each size class of $H_x$. From Lemma 4.2 it is easy to see that there are only $m^{O(1)}$ distinct possibilities. For $i > b(x)+2$, we treat jobs in $H_x$ as small. Using ideas similar to those in Lemma 4.4, it suffices to use only a coarse precision of $O(1)$. However, for ease of exposition, we maintain separate information for each different size class of $H_x$.

In summary, for a block $\mathcal{B}_i$ the sets $U_{xi}$ and $V_{xi}$ such that $b(x) \geq i - \frac{2}{\epsilon^2}$ are specified by:

- $U_{xi}$ is specified by an integer in $[0, \frac{1}{\epsilon^2}]$ for $i > b(x)$ and by an integer in $[0, \frac{m}{\epsilon^2}]$ for $i = b(x)$.
- For $i = b(x), \ldots, b(x) + 2$, $V_{xi}$ is specified by $\lfloor 1 + 4 \log_{1+\epsilon} \frac{1}{\epsilon} \rfloor$ integers each in the range $[0, \frac{m}{\epsilon^2}]$.
- For $i > b(x) + 2$, $V_{xi}$ is specified by $\lfloor 1 + 4 \log_{1+\epsilon} \frac{1}{\epsilon} \rfloor$ integers each in the range $[0, \frac{1}{\epsilon^2}]$.

We summarize our considerations and results of this subsection in the following lemma.

**Lemma 4.5** *There is a $(1 + \epsilon)$–approximate schedule $\mathcal{S}$ such that for each block $\mathcal{B}_i$ the following is true:*

- *There are $k = (\frac{m}{\epsilon^2})^{O(1/\epsilon^4)}$ sets $G_i^1, \ldots, G_i^k$ that can be constructed in polynomial time, and*
- *$G_i$, the set of jobs remaining in $\mathcal{S}$ after block $\mathcal{B}_i$, is one of $\{G_i^1, \ldots, G_i^k\}$.*

### 4.4. Scheduling jobs within a block

We now describe how to compute $W(i, F_1, F_2, V)$. Since this is itself an NP-hard problem we settle for a relaxation. A $1 + \epsilon$ decision procedure for computing $W(i, F_1, F_2, V)$ outputs a schedule that is within $1 + \epsilon$ of $W(i, F_1, F_2, V)$ and shifts the frontier $F_2$ by at most a $1+\epsilon$ factor. Clearly such a procedure suffices in order to compute a $(1+O(\epsilon))$–optimal solution to the dynamic program given above. We now describe a $1 + \epsilon$ decision procedure that runs in polynomial time for each fixed $\epsilon$.

We partition the job set $V$ into small and large as before. Our objective is to enumerate over all potential schedules of large jobs. In particular, we restrict ourselves to schedules where, in each interval $I_x$, a large job starts only at one of the $\frac{1}{\epsilon^3}$ times specified by $R_x + i\epsilon^3 I_x$, for $i = 0, \ldots, \frac{1}{\epsilon^3} - 1$. Furthermore, in our enumeration of large job schedules we will only specify the sizes and the start times of the large jobs scheduled. This is sufficient information to reconstruct their schedule: whenever we have two jobs of same size available, we always schedule the one with the larger weight first. With these restrictions, the schedule of large jobs on a machine within a block is completely determined by three things: its incoming frontier, its outgoing frontier, and the sizes of jobs started at each of the discrete time units in each of the $s$ intervals. By arguments similar to those in the previous section, the number of different possibilities is $k = 2^{O(1/\epsilon^5)}$. Thus the configurations of all machines is from one of $(m + 1)^k$ possibilities. Out of these we consider only those that are compatible with the incoming and outgoing frontiers $F_1$ and $F_2$ and have a feasible schedule for the large jobs in $V$. Both conditions can be checked in a straightforward way. We schedule the small jobs in a greedy fashion in the spaces left by the large jobs. We move all the large jobs that start and finish in an interval to the end of the interval. We enlarge each of the spaces by a $1 + \epsilon$ factor to accommodate all the small jobs. Thus we have the following lemma.

**Lemma 4.6** *There is a $1+\epsilon$ decision procedure to compute $W(i, F_1, F_2, V)$ that runs in time $(m+k)^k$ where $k = 2^{O(1/\epsilon^5)}$.*

We remark that the running time of the procedure can be improved by doing dynamic programming between intervals of the block instead of brute force enumeration of all large job schedules. The improved running time will be $m^{\mathrm{poly}(1/\epsilon)}$. However in interests of space we omit the details and give our main result.

**Theorem 4.7** *There is a PTAS for $P\,|\,r_j\,|\sum w_jC_j$ that constructs a $(1+\epsilon)$–approximation in time $O((m+1)^{poly(1/\epsilon)} \cdot n + n\log n)$.*

The number of potential blocks for the dynamic programming is $O(\log D)$ where $D$ is an upper bound on the schedule makespan. However there are only $O(n/\epsilon^3)$ interesting blocks since each job $j$ finishes by $r_j/\epsilon^4$.

### 4.5. Scheduling with preemption

In the preemptive case, several computational aspects of the preceding algorithm can be simplified, leading to an approximation scheme with a better running time. Specifically, since large jobs can be executed fractionally, we do not need to keep track of the frontier formed by the crossing jobs. Moreover, we can do dynamic programming directly with intervals instead of blocks and an approximate schedule can be specified by the fractions of jobs that are processed in any interval. This significantly reduces the amount of enumeration needed in the dynamic programming. For instance, since there are no release dates within an interval, we can use McNaughton's wrap around rule [17] to compute a preemptive schedule with optimal makespan in $O(n)$ time. Thus if we knew the job fragments that execute within an interval, they can be efficiently scheduled. We omit here the various technical details involved and summarize below the running time of our approximation scheme.

**Theorem 4.8** *There is a PTAS for $P\,|\,r_j,\, pmtn\,|\sum w_jC_j$ that constructs a $(1+\epsilon)$–approximation in time $O(2^{poly(1/\epsilon)} \cdot n + n\log n)$.*

## 5. Scheduling on a constant number of unrelated machines

We now treat the unrelated parallel machine case where job $j$ has processing time $p_{ij}$ on machine $i$. It is easy to see that Lemma 2.1 still applies and that Lemma 2.2 applies if we now define $p_j := \min_i p_{ij}$. We refer to $p_j$ as the *size* of job $j$. Moreover, we use the notation $a(j)$ to denote the machine to which job $j$ is assigned.

The following observation is crucial to dealing with unrelated machines. One would like to claim that, in an optimal schedule, each job $j$ will be processed on a machine on which it does not take much more processing time than $p_j$, i.e. that $p_{a(j)j} = O_{\epsilon,m}(p_j)$. Although this is true in the preemptive case or when there are no release dates, it is not true for instances of $Rm\,|\,r_j\,|\sum w_jC_j$. For example, consider the 2-machine case. Take one job with $r_1 = 0, w_1 = 1, p_{11} = 4$ and $p_{21} = \infty$, and another job with $r_2 = 2, w_2 = 1, p_{12} = 0$ and $p_{22} = 1$. Here machine 2 is extremely slow for job 2, and yet it is appropriate to schedule job 2 on machine 2 instead of waiting for machine 1 to become available. The typical such situation is when, at the release date of a job, all its fast machines are busy processing different large jobs; however then as soon as a fast machine becomes available after the release date of a job, there is no more need to process it on a slow machine. We capture this in the following lemma.

We change the notion of small and large jobs slightly to adapt to this setting: here a job $j$ is said to be *small* if $p_j \le \frac{\epsilon^3}{m} r_j$, otherwise it is *large*.

**Lemma 5.1** *For instances of $Rm\,|\,r_j\,|\sum w_jC_j$, there exists a $(1+\epsilon)$–approximate schedule such that, for each job $j$, either $p_{a(j)j} \le \frac{m}{\epsilon} p_j$ or $C_j \le r_j/\epsilon$.*

**Proof Sketch.** Given a schedule $S$, let $j$ be a job whose fastest machine is machine $i$, but which is scheduled on machine $l$; if $j$ violates the conditions of the lemma, then we remove $j$ from machine $l$ and place it on machine $i$, right before the first job on $i$ whose completion time is greater than $C_j$; this creates a delay on machine $i$ of up to $r_j + p_j$, which is negligible. One can check that when this is done for every job of $S$ which violates the conditions of the lemma, the delays incurred by any job $k$ do not exceed $2\epsilon C_k$. □

We note that the special case $p_{a(j)j} > \frac{m}{\epsilon} p_j$ only occurs when, at time $r_j = R_x$, job $j$'s fastest machine is busy processing a very large job, whose processing time spans all of the interval $I_x$ and beyond. Thus, if we know the schedule of the large jobs, we can, for each small job, replace $p_j$ by $\min\{p_{ij} | \text{machine } i \text{ is not busy at time } r_j \text{ with a large job spanning all of } I_x\}$. Although we do not know ahead of time the schedule of the large jobs, the dynamic program will perform these updates dynamically.

For simplicity, in the remainder of this section we only discuss the problems $Rm\,|\;|\sum w_jC_j$ and $Rm\,|\,r_j,\, pmtn\,|\sum w_jC_j$, for which $p_{a(j)j} \le m/\epsilon p_j$. Thus we can set $p_{ij} = \infty$ whenever $p_{ij} > \frac{m}{\epsilon} p_j$. Now, define the *execution profile* of a job $j$ to be an $m$-tuple $< i_1, \ldots, i_m >$ such that $p_{ij} = p_j \cdot (1+\epsilon)^{i_k}$. We adopt the convention that $i_k = \infty$ if $p_{ij} = \infty$.

**Corollary 5.2** *The number of distinct profiles is bounded by $\ell := \lfloor 2 + \log_{1+\epsilon} \frac{m}{\epsilon^2} \rfloor^m$.*

Let $T_x(t)$ and $H_x(t)$ denote the set of small and large jobs released at $R_x$ with profile $t$. The next lemma is an adaptation of Lemma 4.2 to the unrelated machine case.

**Lemma 5.3** *The input instance $I$ can be modified to an instance $I'$ with $\text{OPT}(I') \leq (1+\epsilon)\text{OPT}(I)$ such that the following conditions hold.*

- *For every profile t, we have $p(T'_x(t)) \leq 2mI_x$.*
- *For every profile t, the number of distinct job sizes in $H'_x(t)$ is at most $\lfloor 1 + \log_{1+\epsilon} \frac{m}{\epsilon^4} \rfloor$. The number of jobs of each distinct size is at most $(\frac{m}{\epsilon})^2$.*

We can now pursue an approach similar to that in Section 4. Thus, we do dynamic programming over the blocks, keeping track of incoming and outgoing frontiers, enumerate the schedules of large jobs in each block, and fill in the block with small jobs for each profile. However another difficulty of unrelated machines is that we cannot schedule small jobs greedily by Smith's rule, since Lemma 4.1 no longer applies. Instead, we use LP techniques, more precisely use the result of Shmoys and Tardos [23] for the generalized assignment problem.

Notice that the amount of information that has to be maintained is multiplied by a factor of roughly $\ell$ since we have to treat jobs with distinct profiles separately. We omit the details and claim the following:

**Theorem 5.4** *There is a PTAS for the problem $Rm \,|\, r_j \,|\, \sum w_j C_j$. For instances of $Rm \,|\; |\, \sum w_j C_j$ and $Rm \,|\, r_j,\, pmtn \,|\, \sum w_j C_j$ there is a PTAS that constructs a $(1+\epsilon)$–approximation in time $O(n \log n)$ for each fixed $\epsilon$ and $m$.*

**Remark**. When there are no release dates, i.e. in the case $Rm \,|\; |\, \sum w_j C_j$, it is possible to take a slightly different approach to obtain a PTAS, based on ratios $\frac{p_{ij}}{w_j}$ instead of time. One can then use ratio-stretching and ratio-rounding to prove a variant of the preliminary Lemma 2.1 and simplify the input jobs. From this viewpoint, the profile of a job is just the $m$-tuple of its ratios on the $m$ machines. The main simplification in the absence of release dates is that on each machine, Smith's ratio rule applies. To bound the number of jobs of each profile, it is possible to merge jobs with identical profiles when their processing time is very short, which further simplifies the input (and altogether gets rid of the "small jobs" issue). Moreover, a variant of Lemma 5.1 also implies a bound on the number of profiles with the same minimum ratio, so that, if one defines intervals on the ratio scale, there are only a constant number of jobs in each interval. To complete the dynamic program, all that remains is to show that each interval only needs to interact with a neighboring block of intervals. This is done by showing, via an elementary calculation, that if two jobs have very different ratios, then, even if they are scheduled on the same machine, the earlier job will only have a marginal effect on the later job's completion time (an observation which is perhaps of independent interest). It is then easy to perform dynamic programming on the ratio scale.

Because the above approach does not have to deal with the placement of many small jobs and appeal to LP techniques, it leads to a slightly simpler algorithm for the problem $Rm \,|\; |\, \sum w_j C_j$; unfortunately this approach breaks down completely when jobs have release dates.

## References


[1] N. Alon, Y. Azar, G. J. Woeginger, and T. Yadid. Approximation schemes for scheduling on parallel machines. *Journal of Scheduling*, 1:55–66, 1998.

[2] K. R. Baker. *Introduction to Sequencing and Scheduling*. Wiley, 1974.

[3] J.L. Bruno, E.G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 17:382–387, 1974.

[4] S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. In F. Meyer auf der Heide and B. Monien, editors, *Automata, Languages and Programming*, number 1099 in Lecture Notes in Computer Science. Springer, Berlin, 1996. Proceedings of the 23rd International Colloquium (ICALP'96).

[5] 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.

[6] C. Chekuri and R. Motwani. Precedence constrained scheduling to minimize weighted completion time on a single machine. Short abstract in *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 873–74, 1999. To appear in *Discrete Applied Mathematics*.

[7] 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.

[8] M. X. Goemans, M. Queyranne, A. S. Schulz, M. Skutella, and Y. Wang. Single machine scheduling with release dates. Manuscript, 1999.

[9] M. X. Goemans, J. Wein, and D. P. Williamson. A 1.47–approximation algorithm for a preemptive single-machine scheduling problem. Manuscript, 1997.

[10] 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.

[11] 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.

[12] J. A. Hoogeveen, P. Schuurman, and G. J. Woeginger. Non-approximability results for scheduling problems with min-sum 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.

[13] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the Association for Computing Machinery*, 23:317–327, 1976.

[14] W. Horn. Minimizing average flow time with parallel machines. *Operations Research*, 21:846–847, 1973.

[15] 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.

[16] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[17] R. McNaughton. Scheduling with deadlines and loss functions. In *Management Science*, 6:1–12, 1959.

[18] 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.

[19] C. Phillips, C. Stein, and J. Wein. Minimizing average completion time in the presence of release dates. *Mathematical Programming B*, 82:199–223, 1998.

[20] M. W. P. Savelsbergh, R. N. Uma, and J. Wein. An experimental study of LP-based scheduling heuristics. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, pages 453–461, 1998.

[21] A. S. Schulz and M. Skutella. The Power of alpha-Points in Preemptive Single Machine Scheduling. Manuscript 1999, submitted.

[22] A. S. Schulz and M. Skutella. Scheduling–LPs bear probabilities: Randomized approximations for min–sum criteria. In R. Burkard and G. J. Woeginger, editors, *Algorithms – ESA '97*, volume 1284 of *Lecture Notes in Computer Science*, pages 416 – 429. Springer, Berlin, 1997.

[23] D. B. Shmoys and E. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62:461–474, 1993.

[24] M. Skutella. Semidefinite relaxations for parallel machine scheduling. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS'98)*, pages 472–481, 1998.

[25] M. Skutella. Convex quadratic programming relaxations for network scheduling problems. In J. Nešetřil, editor, *Algorithms — ESA '99*, volume 1643 of *Lecture Notes in Computer Science*, pages 127–138. Springer, 1999.

[26] M. Skutella and G. J. Woeginger. A PTAS for minimizing the weighted sum of job completion times on parallel machines. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 400–407, 1999.

[27] W. E. Smith. Various optimizers for single-stage production. *Naval Res. Logist. Quart.*, 3:59–66, 1956.

[28] E. Torng and P. Uthaisombut. Lower bounds for srpt-subsequence algorithms for nonpreemptive scheduling. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 973–974, 1999.

[29] R. N. Uma and J. Wein. On the relationship between combinatorial and lp-based approaches to NP-hard scheduling problems. In *Proceedings of the 7th Conference on Integer Programming and Combinatorial Optimization*, pages 394–408, 1999.