A PTAS for Minimizing Weighted Completion Time on Uniformly Related Machines (Extended Abstract)

Chandra Chekuri[1] and Sanjeev Khanna[2]

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

[2] Dept. of CIS, University of Pennsylvania, Philadelphia, PA 19104.
sanjeev@cis.upenn.edu. Supported in part by an
Alfred P. Sloan Research Fellowship.

Abstract. We consider the well known problem of scheduling jobs with release dates to minimize their average weighted completion time. When multiple machines are available, the machine environment may range from identical machines (the processing time required by a job is invariant across the machines) at one end of the spectrum to unrelated machines (the processing time required by a job on each machine is specified by an arbitrary vector) at the other end. While the problem is strongly NP-hard even in the case of a single machine, constant factor approximation algorithms are known for even the most general machine environment of unrelated machines. Recently a PTAS was discovered for the case of identical parallel machines [1]. In contrast, the problem is MAX SNP-hard for unrelated machines [11]. An important open problem was to determine the approximability of the intermediate case of uniformly related machines where each machine has a speed and it takes p/s time to process a job of size p on a machine with speed s. We resolve the complexity of this problem by obtaining a PTAS. This improves the earlier known approximation ratio of $(2+\epsilon)$.

Keywords: Polynomial time approximation scheme; average completion time; scheduling; uniformly related machines; weighted completion time.

1 Introduction

Scheduling to minimize average weighted completion time is one of the most well studied class of problems in scheduling theory. In this paper we concentrate on the following variant. We are given a set of n jobs where each job j has a processing time p_j, a weight w_j, and a release date r_j before which it cannot be scheduled. The goal is to schedule the jobs on a set of m machines *non-preemptively* with the objective of minimizing $\sum_j w_j C_j$ where C_j is the completion time of j in the schedule. The specific machine environment we consider in this paper is the *uniformly related* case. Machine i has a *speed* s_i and it takes

p_j/s_i time for machine i to process job j. In the $\alpha \mid \beta \mid \gamma$ scheduling notation introduced by Graham et al. [7] this problem is denoted by $Q|r_j|\sum_j w_j C_j$. Using some non-trivial extensions to the ideas introduced in [1] we obtain a polynomial time approximation scheme (PTAS) for this problem. Our ideas also extend to the preemptive case $Q|r_j, pmtn|\sum_j w_j C_j$ but we omit the details of that result in this extended abstract.

Most variants of scheduling to minimize average completion time are strongly NP-hard including preemptive problems [12]. Polynomial time solvable cases include $P||\sum_j C_j$, $1||\sum_j w_j C_j$, and $R||\sum_j C_j$. In the last few years considerable progress has been made in understanding the approximability of many of these NP-hard problems. Constant and logarithmic ratio approximations were found for several variants: diverse machine environments (one, parallel, unrelated) and with a variety of constraints on the jobs (release dates, precedence constraints, delays) [14, 8, 3, 5, 6, 13]. See [8, 1] for more details on the history of these developments. Hoogeven et al. [11] obtained MAX SNP-hardness for some problems, especially those that had precedence constraints on jobs and/or involved unrelated machines. These results led to some conjectures regarding the approximability of variants with only release dates on the jobs. In particular, the problem $1|r_j|\sum_j w_j C_j$ was conjectured to have a PTAS, and $P|r_j|\sum_j w_j C_j$ was conjectured not to have a PTAS (the problem $R|r_j|\sum_j W_j C_j$ was shown to be MAX SNP-hard in [11]). Most of the ideas that led to constant factor approximation algorithms did not seem to lead to the design of approximation schemes. They were based on either preemptive relaxations or linear programming relaxations that had integrality gaps. Skutella and Woeginger [16] obtained a PTAS for the problem $P||\sum_j w_j C_j$ using some ideas from Alon et al. [2]. The basic idea in [16] was to group jobs based on similar values of w_j/p_j and then finding good schedules for each group separately. The schedules for the different groups could be combined together on the same machine using Smith's rule since there are no release dates. These ideas do not have a straight forward extension when jobs have release dates, especially in a multiple machine environment. More recently substantial progress was made in [1] where PTASs were obtained for scheduling jobs with release dates on single, identical parallel machines, and a constant number of unrelated machines both with and without preemptions allowed.

The above mentioned results improved our understanding of the approximability of scheduling with release dates by showing that problems admitting a PTAS (identical parallel and constant number of unrelated) were sufficiently close to the case that is MAX SNP-hard (unrelated machines). An open problem that remained was to determine the approximability of the related machine case, a strong generalization of the identical machine problem, and an important special case of the unrelated machine problem. In this paper we obtain a PTAS for this case, improving the earlier known ratio of $(2 + \epsilon)$ [15].

Techniques and Relation to Previous Work: Scheduling on related machines generalizes the case of identical parallel machines in a natural way. Not surprisingly, we use as our starting point a dynamic programming framework presented in [1]. Informally speaking, the framework requires three key problems to be solved: (i) how to maintain polynomial size description of jobs that remain to be scheduled at any given time, (ii) a polynomial size description of how machines interact as one proceeds from one instant of time to the next, and (iii) a polynomial-time algorithm for $(1+\epsilon)$-approximating the special case where we have only a constant number of distinct release dates. Unfortunately, the ideas used for solving these problems in the identical machines case do not generalize easily to the case of related machines. The main contribution of this paper is the development of the substantial new technical machinery required to obtain a PTAS for our problem. In what follows we give a brief overview of the difficulties involved in moving from the identical machine to the related machine case. In the identical machine case we could separate jobs into large and small based on their size. A critical part of the earlier approximation schemes was that at any point in time there are only $O(1)$ distinct large job sizes to consider. This allowed for explicit maintenance of certain parameters associated with each large job size class (such as how many are left, how many to schedule etc.) in time and space $m^{O(1)}$. The small jobs are easy to handle using a greedy approach. At a high level the main difference in the related machine case is in the possibility of having up to $\Omega(\log m)$ geometrically spaced speeds (we show how to reduce an arbitrary instance to such a restricted instance). The fastest machine could be m times faster than the slowest one. Because of the different speeds, jobs cannot be classified as large and small in an absolute manner. Thus, at any time instant, there could be up to $\Omega(\log m)$ job sizes that could potentially be executed as large. The dynamic programming framework of [1] gives a running time of $m^{\Theta(\log m)}$. Each of the three key steps in the framework has this dependence on the number of speeds.

To get around these difficulties we use two approaches. First, we show the existence of approximate schedules with *weaker* requirements than previously known. In particular, we give a procedure to preprocess the input in such a way that there is an approximate schedule in which every job finishes by $O(1)$ times its release date.

Second, we use more sophisticated enumeration and dynamic programming techniques. We mention two main ideas in this context. We use an approximate guessing tool from the recent work on the multiple knapsack problem [4] that allows, under certain conditions, to reduce an enumeration that would take $\Omega(m^{\log m})$ time if done naively to $O(\text{poly}(m))$ time. One of the components in [1] is an approximation scheme for the special case where the number of release dates is a fixed constant. This component is substantially more complicated in the related machines case. In this regard, the differences are akin to the differences in the approximation schemes for minimizing makespan (without release dates) on identical machines [9] and on related machines [10]. In particular, we use ideas from [10] which requires dynamic programming across machines with different speeds.

In this extended abstract we concentrate on getting the central ideas across and we omit formal proofs and several technical details in the interest of clarity and conciseness. We focus here only on the non-preemptive case. Various dependencies on ϵ have not been optimized.

2 Preliminaries

We first discuss some general techniques and lemmas that apply throughout our paper. We aim to transform any input into one with simple structure. Some of these transformations will be similar to those in [1] but some are new to the related machine case. A key shifting transformation is considerably more involved in its implementation and proof of correctness. We sequence several transformations of the input problem. 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. Using notation consistent with [1] we say that a transformation produces $1+O(\epsilon)$ *loss*. To argue that a transformation does not produce more than a $1+O(\epsilon)$ loss we typically take an optimal schedule and show how a near optimal schedule exists with the properties desired after the transformation. We go over two such ideas since we will be using them repeatedly. The first is ordering certain subsets of jobs by the ratio w_j/p_j (Smith's ratio). This is motivated by Smith's optimal algorithm for scheduling on a single machine when jobs are all released at the same time [17]. When we have many jobs that are released at the same time we will be able to show that there are approximate schedules that use this order in selecting the jobs for execution. The second transformation is *time stretching*. It is best understood by mapping time t to $(1+\epsilon)t$. Consider a single machine schedule where we map the completion time of each job according to the above mapping. This will worsen the schedule value by only a $(1+\epsilon)$ factor. However, since the processing times of the jobs remain the same, this leaves extra "space" in the schedule which we exploit to schedule other jobs. This allows us to obtain schedules with nicer structure while losing only a $1+O(\epsilon)$ factor. After the preprocessing of the input we use a dynamic programming framework to find a schedule with a special structure that is guaranteed to be within a $1+O(\epsilon)$ factor of the optimum.

Notation: For simplicity we will assume throughout the paper that $1/\epsilon$ is integral. We use C_j and S_j to denote the completion time and start time respectively of job j, and OPT to denote the weighted completion time of an optimal schedule. The number of jobs and machines is denoted by n and m respectively. We denote the speed of a machine i by s_i and assume w.l.o.g. that $s_1 \geq s_2 \geq \ldots \geq s_m$.

2.1 Input Transformations

We start with some transformations that are generalizations of those in [1]. We then introduce several new ones that are crucial for the related machines case.

Geometric Rounding: Our first simplification creates a well-structured set of possible processing times, release dates, and machine speeds.

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

For an arbitrary integer x, we define $R_x := (1+\epsilon)^x$. As a result of Lemma 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 $I_x := [R_x, R_{x+1})$. 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. Observe that the notion of time is independent of the machine speeds.

Large and Small Jobs: As in [1] we classify jobs as small and large. Jobs are small if their processing time is sufficiently small relative to the interval in which they run. Large jobs are those that take up a substantial portion of the interval. Note that this definition is both a function of the job size and the interval. A difficulty with related machines is that a job in an interval could be small or large depending on the machine on which it is processed. Therefore we say that a job is large or small by qualifying with the speed we have in mind. To be more precise we say that a job p_j is *small* with respect to an interval I_x for speed s_ℓ if $p_j/s_\ell \leq \epsilon^3 I_x$, otherwise it is *large*. We will often simply say that a job p_j is scheduled as small to indicate that it will be scheduled in some interval I_x on some machine with speed s_ℓ so that $p_j/s_\ell \leq \epsilon^3 I_x$. Similarly for large jobs. The following lemma states that a job is not arbitrarily large relative to its start time.

Lemma 2. *With* $1+\epsilon$ *loss, we can enforce* $S_j \geq \epsilon p_j / s_{k(j)}$ *for all jobs* j *where* $k(j)$ *is the machine on which* j *is processed.*

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 3. *With* $1+\epsilon$ *loss we can ensure that every job crosses at most* $s := \lceil \log_{1+\epsilon}(1+\frac{1}{\epsilon}) \rceil$ *intervals.*

Lemma 4. *With* $1+\epsilon$ *loss we can restrict attention to schedules in which no small job crosses an interval.*

Lemma 5. *With* $1+\epsilon$ *loss we can restrict attention to schedules in which each job that is scheduled as large starts at one of* $1/\epsilon^4$ *equi-spaced instants within any interval.*

$O(\log m)$ Speed Classes It is easy to see that rounding the speeds to powers of $(1+\epsilon)$ results only in an $1+O(\epsilon)$ loss. However, this can still leave m distinct speeds. We reduce the number of distinct speeds to $O(\log m)$ as follows. Intuition suggests that machines much slower than the fastest machine can be ignored, with little loss in the schedule value. We formalize this intuition below. Let $s_1 \geq s_2 \geq \ldots s_m$ be the speeds of the machines. We can assume that $m > 1/\epsilon^3$ for otherwise we can use the algorithm in [1] to obtain a PTAS.

Lemma 6. *With $(1+\epsilon)$ loss we can ignore machines with speed less than $\frac{\epsilon}{m} s_{1/\epsilon^3}$.*

Proof Sketch. Consider an optimal schedule S and let $k(j)$ be the machine on which job j is executed in S. Let $A_i = \sum_{k(j)=i} w_j C_j$ be the contribution of machine i to the schedule value. Let ℓ be such that $A_\ell = \min_{2 \leq i \leq 1/\epsilon^3} A_i$. We obtain a new schedule as follows. We remove the jobs allocated to M_ℓ and execute them on M_1 in a delayed fashion. By time stretching on M_1 it is clear that we can execute the jobs of M_ℓ with no more than a $1/\epsilon^2$ factor delay. Since $A_\ell \leq \epsilon^3 \cdot \text{OPT}$ this does not effect the schedule by more than a $(1+\epsilon)$ factor. We schedule the jobs allocated on all the slow machines (the ones with speed smaller than $s_{1/\epsilon^3} \cdot \frac{\epsilon}{m}$) and assign them to M_ℓ. We do this as follows. All the jobs that start in each of the slow machines in the interval I_x are scheduled in the interval I_x on M_ℓ. It is easy to see that all the jobs will complete on M_ℓ within the same interval I_x and hence their completion times are affected by no more than a $(1+\epsilon)$ factor.

Following Lemma 1 and Lemma 6 we can assume that our instance has $O(\log m/\epsilon)$ distinct speeds. We group machines with the same speed and refer to the group as a speed class. Let K be the exact number of classes we have with the implicit understanding that $K = O(\log m)$. For $1 \leq i \leq K$, let M_i denote the machines in the ith speed class. Let $m_i = |M_i|$ denote the number of machines in M_i and let s_i denote the common speed of machines in M_i. We assume w.l.o.g. that $s_1 > s_2 > \ldots > s_K$.

Generating Extra Machines We describe another technique that allows us to simplify things. The lemma below shows that any schedule can be transformed into a $1+O(\epsilon)$-approximate schedule where we use only a $(1-\epsilon)$-fraction of the machines from any sufficiently large machine class. Note that a similar lemma does not hold for minimizing makespan. We will assume from here on that we are working with this reduced allocation of machines. The remaining extra machines would be useful in a key step for implementing the dynamic programming.

Lemma 7. *Given m machine instance of identical parallel machines where $m > 1/\epsilon^3$ there is a $1+O(\epsilon)$-approximate schedule on $m(1-\epsilon^3)$ machines.*

The proof of the above lemma uses a similar line of reasoning as in the proof of Lemma 6.

Shifting Our next goal is to show that we can preprocess the input instance I in such a way that we can guarantee a schedule in which every job will be completed within a constant number of intervals from its release. We accomplish this by selectively retaining only a fraction of the jobs released in each interval and *shifting* the rest to later intervals. This basic idea plays a crucial role in obtaining the PTAS for the parallel machine case $P|r_j|\sum_j w_j C_J$ [1]. A brief description of the procedure in [1] follows. Jobs released in an interval I_x are either small or fall in to one of $O(1)$ large size classes. Small jobs are ordered in non-increasing order according to the ratio w_j/p_j and large jobs are ordered in each size class in decreasing order of their weights. In each class the number of jobs retained is restricted by the volume that could be processed in the interval I_x. The rest are shifted to the next interval. Since the number of classes is $O(1)$ the total volume of jobs released at R_x in the modified instance is $O(1)$ times the volume of I_x. By time shifting one can show that there exists an approximate schedule in which all the jobs at I_x could be finished within $O(1)$ intervals after R_x. This enables locality in dynamic programming.

There is no simple generalization of the above ideas to the related machine case because the notion of small and large jobs is now relative to the machine speed. The number of distinct job sizes that can be executed as large in an interval could be $\Omega(\log m)$ and we cannot afford to have a volume of jobs released at I_x that is $\Omega(\log m)$ times the processing capability of the machines in I_x. We design a new procedure below that essentially retains the property concerning the volume. The proof that this procedure results in only an $1 + O(\epsilon)$ loss is however more involved. We describe the shifting procedure formally below.

Let J_x be the set of jobs released at R_x. For each speed class i from K down to 1 the following process is done.

- Let T_x^i and H_x^i be the small and large jobs with respect to speed s_i released at R_x that are still to be processed.
- The number of distinct size classes in H_x^i is $O(1/\epsilon^2)$. In each size class we pick jobs in order of non-increasing weights until the sum of processing times of jobs picked just exceeds $m_i s_i I_x/\epsilon^2$ or we run out of jobs.
- We pick jobs in T_x^i in non-increasing w_j/p_j ratio until the processing time of the jobs picked just exceeds $m_i s_i I_x$ or we run out of jobs.
- We remove the jobs picked from T_x^i and H_x^i from J_x.

Jobs that are not picked in any speed class are shifted to the next interval. We repeat this process with each successive interval. Let I' be the modified instance obtained after the shifting process above and for each x let J'_x be the set of jobs released at R_x in I'.

Lemma 8. *For any given rounded job size s let $a_x^s(S)$ and $b_x^s(S)$ denote the number of jobs of size s started in I_x as small and large respectively in an optimal schedule S. There exists a $1 + O(\epsilon)$-approximate schedule S' such that for each s and x either $a_x^s(S') < \frac{1}{\epsilon^2} b_x^s(S')$ or $b_x^s(S') = 0$.*

Proof. Consider an optimal schedule S. Suppose $a_x^s(S) > \frac{1}{\epsilon^2} b_x^s(S)$ for some size s and interval x. We create a modified schedule S' as follows. We take all the

$b_x^s(S)$ jobs executed as large and execute them as small within the same interval I_x by scheduling them on faster machines. Since the number of jobs executed as small, $a_x^s(S)$, is much larger that those executed as large, we claim that this can be accomplished by stretching the interval by only a $(1+\epsilon)$ factor. In the modified schedule $b_x^s(S')=0$. This can be done simultaneously for all s and x which do not satisfy the lemma and no interval stretches by more than a $1+\epsilon$ factor. The schedule S' is a $1+O(\epsilon)$-approximation to S.

Lemma 9. *For the modified instance I' obtained from I by the shifting procedure*

1. $\text{OPT}(I') \le (1+O(\epsilon))\text{OPT}(I)$.
2. *There exists a $(1+O(\epsilon))$-approximate schedule for I' in which all jobs in J_x are finished by $R_{x+O(\log(1/\epsilon)/\epsilon)}$.*

Proof Sketch. We prove (2) first. Let J_x^i be the set of jobs picked by the shifting procedure in speed class i, $1 \le i \le K$ at R_x. We note that all jobs in J_x^i can be executed by machines of speed class i in time $O(I_x)$. This implies that $p(J_x^i)$ will be small relative to interval $I_{x+O(\log(1/\epsilon)/\epsilon)}$ because of the geometrically increasing property of interval sizes. Therefore time stretching any arbitrary but fixed optimal schedule allows us to create the required space to execute all the jobs in J_x^i by then.

Now we prove (1). We observe that the shifting procedure does the following. For each size class s that can be executed as large in I_x the procedure picks the n_x^s/ϵ^2 jobs in non-increasing weight order from J_x where n_x^s is the maximum number of jobs that can executed as large of size class s in I_x. From Lemma 8 there exists a $(1+O(\epsilon))$-approximate schedule in which the jobs executed as large in I_x of size s are contained in the set we pick. The small jobs that are executed in I_x can be treated as fractional jobs and this enables us to pick them in a greedy fashion in non-increasing order of w_j/p_j and we pick enough jobs to fill up the volume of I_x. The proof of the near optimality of greedily picking small jobs is similar to that of the parallel machine case in [1] and we omit the details in this version.

2.2 Overview of Dynamic Programming Framework

We give a brief overview of the dynamic programming framework from [1].

The idea is to divide the time horizon into a sequence of blocks, say $\mathcal{B}_1, \mathcal{B}_2, \ldots$, each containing a constant number (depending on ϵ) of intervals dates, and then do a dynamic programming over these blocks by treating each block as a unit. There is interaction between blocks since jobs from an earlier block can cross into the current block. We choose the number of intervals in each block to be sufficiently large so that no job crosses an entire block. From Lemma 3 we conclude that $O(1/\epsilon^2)$ intervals per block suffice. In other words jobs that start in $\mathcal{B}_i$ finishes no later than $\mathcal{B}_{i+1}$. 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. Let $\mathcal{F}$ denote the possible set of frontiers between blocks. The dynamic programming table maintains entries of the form $O(i, F, U)$: the minimum weighted completion time achievable by starting the set U of jobs in 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 $C(i, F_1, F_2, V)$ 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 as the outgoing frontier to block $\mathcal{B}_{i+1}$. We obtain the following equation.

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

3 Implementing the Dynamic Programming Framework

Broadly speaking, we need to solve three problems for using the dynamic programming framework described in the preceding section. First, we need a mechanism to compactly describing for any block $\mathcal{B}_i$, the set of jobs that were released prior to $\mathcal{B}_i$ and have already been scheduled. Second, we need to ensure that the number of distinct frontiers in $\mathcal{F}$ is polynomial for any block $\mathcal{B}_i$. Finally, given a set of jobs to be scheduled within a block, we should be able to find a $(1+\epsilon)$-approximate schedule. A basic theme underlying our implementation of these steps is to relax the requirements slightly. In the parallel machine case we could enumerate the set of jobs U that are released in $\mathcal{B}_i$ and started in $\mathcal{B}_i$ itself. This was done by separating out the small and large jobs. Since there were only $O(1)$ large job sizes in each $\mathcal{B}_i$ this was relatively easy. Now we have $\Omega(K)$ large job sizes. We would have to enumerate $m^{\Omega(K)}$ possibilities. To get around this difficulty we use a global scheme that is inspired by the recent work on the multiple knapsack problem [4]. We will be able to figure out most of the important jobs using the above scheme in polynomial time and we show this approximate enumeration suffices. A similar situation arises in enumerating the frontiers. Here we use a different idea based on Lemma 7. Finally, another difficult part is the problem of scheduling jobs in a fixed number of intervals. The approach we adopt is some what akin to the approach taken by Hochbaum and Shmoys [10] to obtain a PTAS for the makespan problem on related machines. The basic idea is to do dynamic programming across the speed classes going from the slowest speed class to the fastest. The advantage of this approach is the following. Any fixed size class is large for only $O(1)$ consecutive speed classes because of the geometrically increasing speeds. This implies that while we are doing the dynamic programming the number of size classes for which we have to maintain detailed information (in terms of the exact number of jobs remaining etc) is only $O(1)$ as opposed to $\Omega(K)$ if we tried to solve the problem all at once. The many subtle details that we need to make all these ideas work are explained in the remainder of this section.

In what follows, we assume each block consists of $\alpha = O(1/\epsilon^2)$ intervals, the precise constant is of not much importance.

3.1 Compact Description of Remaining Jobs

We start by observing that by Lemma 9 and our choice of block size, there exists a $(1+\epsilon)$-approximate schedule such that all jobs released in a block $\mathcal{B}_i$ are always scheduled by the end of the block $\mathcal{B}_{i+1}$. In fact we will be able to schedule all jobs released in $\mathcal{B}_i$ by the end of $\mathcal{B}_{i+1}$ irrespective of how many of them have been executed in $\mathcal{B}_i$ itself. We will restrict our attention to only such schedules. Thus, to compactly describe the set of jobs that remain from B_i, we need only describe a mechanism for compact representation of the set of jobs chosen to be scheduled within B_i. However, due to the non-uniform nature of machine speeds, this process turns out to be more involved than the identical machine case. In particular, we rely on some ideas from the recent approximation scheme for the multiple knapsack problem [4]. We show that there exists a $(1+\epsilon)$-approximate schedule that needs to enumerate over only a polynomial number of distinct possibilities for sets of jobs chosen for scheduling within a block $\mathcal{B}_i$. We will use the following elementary fact from [4]:

Proposition 1. *Let $h = O(\log m)$. Then the number of h-tuples $\langle k_1, k_2, ..., k_h \rangle$ such that $k_i \in [0...h]$ and $\sum k_i \leq h$ is $m^{O(1)}$.*

We can now describe our scheme for enumerating the job subsets. For each interval $I_j \in \mathcal{B}_i$ we separately enumerate the jobs that are released at I_j and will be scheduled in $\mathcal{B}_i$. Since the number of intervals in each block is a constant depending only on ϵ, we concentrate on a single interval. Let X_l be the set of all jobs released in I_j that are large for the slowest speed and let X_s be the remaining jobs. We focus here on the enumeration of the set X_l and later sketch the idea for the set X_s. Let w be the total weight of jobs in X_l. As a result of our shifting procedure, total number of jobs in X_l can be bounded by $m^2 f(1/\epsilon)$ for some suitably large function f. We use this fact to ignore from our consideration all jobs in X_l whose weight is less than $\delta(\epsilon) \cdot w/m^2$ where δ is a suitably smaller than $1/f(1/\epsilon)$ — we will schedule these jobs in $\mathcal{B}_{i+1}$ and by our choice of δ, their completion time can be amortized to an ϵ fraction of the weighted completion of other jobs in X_l. Once we eliminate these jobs from consideration, there are only $O(\log m)$ distinct weights for the remaining jobs in X_l.

1. For interval $I_j \in \mathcal{B}_i$ we first specify W_j the total weight of jobs in X_l that will be scheduled in $\mathcal{B}_i$. We specify this weight in multiples of $\delta(\epsilon) \cdot w/m^2$ by an integer ℓ such that $0 \leq \ell \leq m^2/\delta(\epsilon)$. The set of jobs that are lost due to the downward rounding are scheduled in $\mathcal{B}_{i+1}$ and as above the increase in the schedule value can be bounded. The number of choices for ℓ is polynomial in m.
2. For a given W_j (specified by the integer ℓ), we specify a partition of W_j into $h = O(\log m)$ classes one for each of the distinct large size classes. Since an exact partition would require quasi-polynomial possibilities, we need to do this step approximately. We specify an approximation to an exact partition of the form $\langle W_j^1, W_j^2, ..., W_j^h \rangle$ by guessing an integer vector $\langle k_1, k_2, ..., k_h \rangle$ such that $k_l(\delta(\epsilon) \cdot W_j/h) \leq W_j^l < (k_l + 1)(\delta(\epsilon) \cdot W_j/h)$. By Proposition 1, the

number of tuples enumerated above is bounded by $m^{O(1)}$ for any fixed $\epsilon > 0$. The error introduced by this approximate under-guessing can be bounded by $\delta(\epsilon) \cdot W_j$ over all h size classes. Since all jobs released in $\mathcal{B}_i$ are always scheduled by the end of the block $\mathcal{B}_{i+1}$, the cost of the schedule as a result of the under-guessing above increases by at most a factor of $1 + O(\epsilon)$.
3. Finally, for each size class of jobs released in I_j, we greedily pick the smallest number of jobs whose cumulative weight exceeds the weight guessed above.

For jobs in X_s, we order them by w_j/p_j and guess the fraction of them that will be scheduled with a precision of $\delta(\epsilon)/m^2$, and using similar reasoning as above, conclude that we do not incur more than a $1 + \epsilon$ loss.

In summary we showed that by restricting the choice to important jobs based on weights we need to consider only a polynomial number of sets as candidates for jobs scheduled within a block.

3.2 Frontiers

A frontier describes the set of jobs that are crossing over from a block $\mathcal{B}_i$ to the next block $\mathcal{B}_{i+1}$. By Lemma 4, we know that only a job that is scheduled as large can participate in a frontier, and by Lemma 5 we know that there are only $1/\epsilon^4$ distinct time instants in any interval by which a job scheduled as large starts or ends. Further the number of distinct job sizes that can execute as large in a block is $O(1/\epsilon^4)$. Hence a crossing job on a large machine can be specified by the size and the time instant it starts in the interval. Let $q = O(1/\epsilon^8)$ denote the total number of such distinct frontiers for any machine. In order to describe the over all frontier, we need to specify this information for each machine. Consequently, we can describe the frontier by a vector $\langle m_{11}, m_{12}, ..., m_{1q}, m_{21}, ..., m_{Kq}\rangle$ where m_{ij} denotes the number of machines in the speed class $\mathcal{C}_i$ that have a job finishing at the jth distinct instant in block $\mathcal{B}_{i+1}$. Clearly, an exact enumeration would require considering quasi-polynomial number of possibilities. We now argue that in order to obtain a $(1+\epsilon)$-approximation it suffices to work with a polynomial-sized set $\mathcal{F}$ of frontiers. With any vector of the above form, we associate a vector $\langle l_{11}, l_{12}, ..., l_{1q}, l_{21}, ..., l_{Kq}\rangle$ in $\mathcal{F}$ where $l_{ij} = m_{ij}$ if $m_i \leq 1/\epsilon^3$, and otherwise, $(l_{ij} - 1)(\epsilon^{11} m_i) < m_{ij} < l_{ij}(\epsilon^{11} m_i)$. Clearly, there are only $O(1/\epsilon^{11K}) = m^{O(1)}$ such vectors to be considered. However, the above approximation of an exact frontier description *over-allocates* machines for large machine classes, and thus would necessitate extra machines. The total number of extra machines needed by any large speed class is bounded by $\epsilon^{11} \cdot q \cdot m_i$ which is at most $\epsilon^3 \cdot m_i$. We allocate these extra machines by using Lemma 7 which allowed us to keep aside an $\epsilon^3 \cdot m_i$ machines for each speed class of size at least $1/\epsilon^3$.

3.3 Scheduling Jobs Within a Block

We now describe a $(1+\epsilon)$-approximate implementation of the procedure $C(i, F_1, F_2, Z)$. Recall that $C(i, F_1, F_2, Z)$ is the procedure that computes the best schedule for a set of jobs Z that are to be scheduled in block $\mathcal{B}_i$ with incoming and outgoing frontiers specified by F_1 and F_2.

In what follows, it will be useful to assume that F_1^j and F_2^j denote the components of F_1 and F_2 that correspond to the jth speed class M_j. Our scheduling procedure is based on a dynamic programming across the classes; we move from the slowest class to the fastest class and treat each class as a unit. In [1] a procedure was given to schedule on a single speed class. The basic idea was to enumerate large job placements and schedule the small jobs greedily in the space left by the large jobs. Enumerating the large job placements was relatively easy because there were only $O(1)$ sizes that were large in each block. We do not know how to efficiently enumerating all large job placements with K speed classes in polynomial time, hence we resort to doing dynamic programming across classes. When considering a current class we would like to know the jobs that are already scheduled in the preceding classes. The main difficulty in implementing the dynamic program is to maintain a compact description of this information. To achieve this we use the notion of *template schedules.*

Template Schedules: A template schedule at a machine class M_j provides information about the jobs that remain to be scheduled along with some "coarse" information constraining how these jobs may be scheduled. It is this additional scheduling information that implicitly encodes information concerning weights of the remaining jobs. Specifically, a template schedule at a machine class M_j specifies scheduling information for all jobs that are eligible to be scheduled as large on a machine in M_j, as well as global information concerning the volume of jobs that must be scheduled as small on machines in M_{j-1} through M_1. We describe these two aspects next.

Let $L(j)$ denote the set of job sizes that can be scheduled as large at the jth machine class, and let $Z_{L(j)}$ denote the job set Z restricted to the sizes in $L(j)$. At the jth speed class we consider all possible extensions of template schedules for the $(j-1)$th machine class so as to incorporate scheduling information for the jobs in the set $Q = Z_{L(j)} \setminus Z_{L(j-1)}$. A template schedule specifies the following information for each size class in Q.

- The number of jobs that are executed as large and the number that are executed as small.
- For those executed as small, the number that will be executed in each interval of $\mathcal{B}_i$.
- For those executed as large, the number that will be executed for each possible placement in each of the speed classes where that size can be executed as large. We note that this information includes speed classes greater than j, that is classes that have already been processed.

Lemma 10. *The template schedule information is polynomial size for each size class in the set Q.*

Proof Sketch. For any fixed size class the number of speed classes on which it can be scheduled as large is $O(\log(1/\epsilon)/\epsilon)$ since the speeds are increasing geometrically. Further the number of distinct start times of large jobs in each class is also fixed for fixed ϵ, following Lemma 5. Hence specifying the numbers is polynomial.

At each class M_j the number of job sizes in Q is $O(1/\epsilon^2)$, hence the total information for all sizes in Q is still polynomial. Observe that template schedules do not maintain any explicit information about the weight of the jobs that remain. However, this information is implicit and can be recovered as follows. Consider the scheduling at a machine class M_j that receives the template schedules for all job sizes that can be scheduled as large on a machine in M_j. Fix one such size class, say p_k, and let a_t denote the number of jobs of size p_k that are required to start at the tth starting time in block $\mathcal{B}_i$ on a machine in M_j. Since a template schedule completely determines the finishing times of all jobs of size p_k, it is straightforward to determine the weights associated with each one of the a_t jobs (we resolve all ties in some fixed canonical manner).

The idea of template schedule as described so far allows us to identify the jobs that are to be scheduled as large at any machine class. However, we need additional information to determine what jobs are available to be scheduled as small at any machine class. We do this by maintaining a vector of the form $\langle V_1, ..., V_\alpha \rangle$ such that V_l specifies the total volume of the small jobs that must be scheduled in the lth interval of the block $\mathcal{B}_i$ in classes M_{j-1} through M_1.

Lemma 11. *The template schedule information for small jobs is of poly size.*

Proof Sketch. We claim that the precision needed for each V_i is $O(\epsilon^5/m^2)$. Assume without loss of generality that $s_K = 1$ and hence $s_1 = O(m)$. Consider the smallest large job in the block and let s be its size. From our assumption that $s_K = 1$, s is at least ϵ^3 times the smallest interval in $\mathcal{B}_i$. We claim that the volume can be maintained in multiples of ϵ^2 times s. This is because the size of each job in the block can be approximated to within a $(1+\epsilon)$ factor by multiples of the above quantity. Coupled with the fact that the total volume that can be executed in the block is $O(m^2)$ the lemma follows.

Dynamic Programming with Template Schedules We maintain a table $T(j, X, Y)$ where j ranges over machine speed classes and X and Y are template schedules for M_j and M_{j-1} respectively. $T(j, X, Y)$ stores the best weighted completion time that is consistent with X and Y. Note that by knowing X and Y the job set that is to be scheduled in M_j is determined. Given X and Y computing $T(j, X, Y)$ involves the following.

- Checking for consistency between X and Y.
- Checking the feasibility of scheduling the jobs in M_j implied by X and Y.

Note that the template schedules implicitly determine the best weighted completion times. We briefly describe the feasibility computation below.

Scheduling Jobs within a Machine Class: For any machine class M_j, once we know the position of jobs to be scheduled as large, as well as the volume of jobs to be scheduled as small in each one of the α intervals, it is relatively easy to determine whether or not there exists a feasible schedule (with $1+\epsilon$ loss) that is consistent with this specification and the in-coming and out-going frontiers F_1^j and F_2^j.

Theorem 1. *There is a PTAS for the problem* $Q|r_j|\sum_j w_j C_j$.

References

1. F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Sviridenko. Approximation Schemes for Minimizing Average Weighted Completion Time with Release Dates. *FOCS '99*.
2. N. Alon, Y. Azar, G. J. Woeginger, and T. Yadid. Approximation schemes for scheduling on parallel machines. *Journal of Scheduling*, 1:55–66, 1998.
3. S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. *ICALP '96*.
4. C. Chekuri and S. Khanna. A PTAS for the Multiple Knapsack Problem. *SODA '00*.
5. C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. *SODA '97*.
6. M. X. Goemans. Improved approximation algorithms for scheduling with release dates. *SODA '97*.
7. 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.
8. L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Offline and online algorithms. *Math. of OR*, 513–44, '97.
9. D. S. Hochbaum and D. B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *JACM*, 34:144–162, 1987.
10. D. S. Hochbaum and D. B. Shmoys. A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, 17:539–551, 1988.
11. J. A. Hoogeveen, P. Schuurman, and G. J. Woeginger. Non-approximability results for scheduling problems with minsum criteria. *IPCO '98*.
12. J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.
13. A. Munier, M. Queyranne, and A. S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. *IPCO '98*.
14. C. Phillips, C. Stein, and J. Wein. Minimizing average completion time in the presence of release dates. *Mathematical Programming B*, 82:199–223, 1998.
15. A. S. Schulz and M. Skutella. Scheduling–LPs bear probabilities: Randomized approximations for min–sum criteria. *ESA '97*.
16. M. Skutella and G. J. Woeginger. A PTAS for minimizing the weighted sum of job completion times on parallel machines. *STOC '99*.
17. W. E. Smith. Various optimizers for single-stage production. *Naval Res. Logist. Quart.*, 3:59–66, 1956.