RC25320 (WAT1210-078) October 25, 2012
Computer Science

IBM Research Report

Preemptive Scheduling with Job-Dependent Setup Times

Rohit Khandekar[1], Kirsten Hildrum[2], Deepak Rajan[3], Joel Wolf[2]

[1]Knight Capital Group
Jersey City, NJ
USA

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

[3]Lawrence Livermore National Laboratories
Livermore, CA
USA

Preemptive Scheduling with Job-Dependent Setup Times

Rohit Khandekar
Knight Capital Group, Jersey City, NJ

Kirsten Hildrum
IBM T.J. Watson Research, Yorktown Heights, NY

Deepak Rajan
Lawrence Livermore National Laboratories, Livermore, CA

Joel Wolf
IBM T.J. Watson Research, Yorktown Heights, NY

October 23, 2012

Abstract

We consider single processor preemptive scheduling with job-dependent setup times. In this model, a job-dependent setup time is incurred when a job is started for the first time, and each time it is restarted after preemption. This model is a common generalization of preemptive scheduling, and actually of non-preemptive scheduling as well. The objective is to minimize the sum of any general non-negative, non-decreasing cost functions of the completion times of the jobs — this generalizes objectives of minimizing weighted flow time, flow-time squared, tardiness or the number of tardy jobs among many others. Our main result is a randomized polynomial time $O(1)$-speed $O(1)$-approximation algorithm for this problem. Without speedup, no polynomial time finite multiplicative approximation is possible unless $\mathbb{P} = \mathbb{NP}$.

We extend the approach of Bansal et al. (FOCS 2007) of rounding a linear programming relaxation which accounts for costs incurred due to the non-preemptive nature of the schedule. A key *new* idea used in the rounding is that a point in the intersection polytope of two matroids can be decomposed as a convex combination of incidence vectors of sets that are independent in both matroids. In fact, we use this for the intersection of a partition matroid and a laminar matroid, in which case the decomposition can be found efficiently using network flows. Our approach gives a randomized polynomial time offline $O(1)$-speed $O(1)$-approximation algorithm for the broadcast scheduling problem with general cost functions as well.

1 Introduction

In this paper, we consider a very general preemptive scheduling problem with job-dependent setup times. This model captures the necessity of performing a setup whenever a job is started for the first time, or restarted after being preempted. Such a setup time might be needed for a variety of practical reasons, such as loading the job context or acquiring the necessary resources. Furthermore, we set as our goal the minimization of the sum of arbitrarily given non-decreasing cost functions of the completion times of the jobs. (For this paper we will restrict our attention to non-negative cost functions.) This problem is general enough to capture several interesting min-sum cost functions such as weighted flow-time, flow-time squared, tardiness or the number of tardy jobs. One can also encode min-max cost functions such as makespan or maximum stretch by doing binary searches on the optimum cost and setting job deadlines appropriately.

We now define our problem, which can be classified as $1 \mid pmtn, r_j, setup = s_j \mid \sum f_j(C_j)$. We call this problem a *general scheduling problem* or GSP. Let $\mathbb{Z}_+$ denote the set of non-negative integers, and $\mathbb{R}$ the set of all reals. Consider a set $\mathcal{J}$ of jobs, where each job $j \in \mathcal{J}$ is associated with release time $r_j \in \mathbb{Z}_+$, setup time $s_j \in \mathbb{Z}_+$, processing time $p_j \in \mathbb{Z}_+$ and a non-decreasing cost function[1] $f_j : \mathbb{Z}_+ \to \mathbb{Z}_+ \cup \{\infty\}$. A feasible schedule on a single processor works on no job before its release time and works on at most one job at any time. The jobs can be preempted. However, every time a job j is started or restarted, a setup time of s_j must be spent before processing can begin. Thus, without loss of generality, any schedule which starts or restarts job j works on it for at least s_j time before preempting or completing it. A job j requires a total of p_j time for processing.[2] Thus if job j is preempted k times before it completes, the total amount of time (including time for setup and processing) it requires is $(k+1) \cdot s_j + p_j$. Given a feasible schedule, let C_j denote the completion time of job j. The objective is to find a feasible schedule that minimizes the total cost $\sum_{j \in \mathcal{J}} f_j(C_j)$.

The above problem generalizes both preemptive scheduling (when $s_j = 0$ for all j) and non-preemptive scheduling (when $p_j = 0$ for all j). As a result, obtaining small multiplicative approximation factors for GSP in polynomial time can be ruled out. More precisely, it is impossible to obtain an $n^{1/2-\epsilon}$-approximation for any $\epsilon > 0$ in polynomial time even for minimizing non-preemptive unweighted flow-time (i.e., $f_j(t) = \max\{0, t - r_j\}$) on n-job instances unless $\mathbb{P} = \mathbb{NP}$ [11].

A commonly used method for dealing with such problems is *resource augmentation analysis*, first proposed by Kalyanasundaram and Pruhs [10] and named so by Phillips et al. [12]. In this methodology, one compares the candidate algorithm, equipped with a faster processor, to an optimum algorithm with a unit speed processor. Define an s-speed ρ-approximation algorithm to be one which, using a processor of speed s, can achieve an objective value no more than ρ times the optimum value on a processor of speed 1. One also considers an analogous notion of extra processors instead of or in addition to extra speed: an m-processor s-speed ρ-approximation algorithm is one which, using m processors of speed s each, can achieve an objective value no more than ρ times the optimum value on a single processor of speed 1. This method of analysis can help elucidate the problem structure. For example, it can be used to explain why some algorithms work well in practice. It can also be used to explain why hardness proofs fall apart when hard instances are perturbed even slightly, for example a reduction from 3-Partition that shows non-preemptive flow-time is hard. We refer the reader to Bansal et al. [3] for further explanation. In this paper, we use resource augmentation analysis.

1.1 Our Results

We summarize our results now. The main result is given in Theorem 1.1.

Theorem 1.1 *There exists a randomized polynomial time $O(1)$-speed $O(1)$-approximation algorithm for* GSP.

Lemma 1.2 *For any $\epsilon > 0$, there exists a randomized polynomial time $(1+\epsilon)$-speed $(1+\frac{1}{\epsilon})(1+\epsilon)$-approximation for preemptive scheduling ($s_j = 0$ for all j) and a randomized polynomial time 12-speed $2(1+\epsilon)$-approximation for non-preemptive scheduling ($p_j = 0$ for all j).*

Theorem 1.3 *There exists a randomized polynomial time $O(1)$-speed $O(1)$-approximation algorithm for the broadcast version of* GSP.

[1] We assume that f_j is given by a value oracle that given $t \in \mathbb{Z}_+$ returns $f_j(t)$.

[2] In the case with $p_j = 0$ for some job j, we insist that job j must get its setup time s_j contiguously at least once.

To determine if there exists a randomized polynomial time $(1+\epsilon)$-speed $f(\epsilon)$-approximation algorithm for GSP for any $\epsilon > 0$, where $f(\epsilon)$ is any computable function of ϵ alone, is an interesting open question. However it is easy to show that a speedup, greater than 1, is needed to obtain any finite multiplicative approximation, even for the special case of non-preemptive scheduling to minimize the number of tardy jobs, if $\mathbb{P} \neq \mathbb{NP}$, as shown by the lemma below.

Lemma 1.4 *Consider a special case of non-preemptive scheduling (i.e., $p_j = 0$ for all $j \in \mathcal{J}$) to minimize the number of tardy jobs (i.e., $f_j(t) = 0$ if $t \leq d_j$, and 1 otherwise for deadline $d_j \in \mathbb{Z}_+$). It is strongly $\mathbb{NP}$-hard to distinguish between the instances that have zero optimum cost and the instances that have positive optimum cost.*

The definition of broadcast scheduling problem and proofs of Lemma 1.2, Theorem 1.3 and Lemma 1.4 are omitted from this version due to lack of space.

1.2 Our Techniques

Our algorithm is based on rounding a linear programming relaxation. Our LP relaxation and overall approach are motivated by the work of Bansal et al. [3], who consider min-sum non-preemptive scheduling problems like weighted flow-time, weighted tardiness and *unweighted* number of tardy jobs on single processor. The LP has a time-indexed formulation that pays a job-dependent setup time in each fractional processing of a job, and also gets partial credit for completing jobs fractionally. The LP also has the obvious constraints to ensure that each job is scheduled to the full extent and that at most one job is scheduled at any single time. Apart from these, the LP has one more crucial set of constraints used to lower bound the cost of any feasible schedule. These constraints are similar to those used by Bansal et al. [3] and are based on the following non-preemptive nature of the problem. Consider a job k that the LP decides to schedule continuously in a certain time interval $[t, t+\ell)$. Then any job j that is released in the interval $[t, t+\ell)$ must start no earlier than $t+\ell$. Thus it must pay at least $f_j(t+\ell)$ cost in the objective function. These constraints provide a good lower bound that a rounding scheme can charge against when the penalty function f_j of job j increases significantly between r_j and $t+\ell$.

1.2.1 New: rounding using total unimodularity of network flows

The main technical contribution of this paper as compared to Bansal et al. [3], however, is in rounding the fractional LP solution. The rounding scheme of Bansal et al. [3] works only when the fractional solution is so-called *laminar*. Intuitively, being laminar means that if the fractional schedule "preempts" job j in favor of scheduling job k, it starts processing job j again only after finishing job k. Such a property holds, for example, when there exists a total ordering $\prec$ on the jobs, such that $j \prec k$ iff the partial credit that the fractional solution gets by scheduling job k is at least that for job j at any point in time. Such an ordering exists for the case of weighted flow-time: $j \prec k$ iff k has higher weight than j, or if they have same weight and k is released before j.

The fractional solution may not be laminar, however, for arbitrary non-decreasing cost functions. Consider, for example, a cost function that encodes weighted completion time and a strict deadline, so that $f_j(t) = \omega_j t$ if $t \leq d_j$ and ∞ otherwise. Suppose two jobs j and k have a cost function of this form with release dates $r_j < r_k$, weights $\omega_j < \omega_k$ and deadlines $d_j < d_k$. In the absence of any other jobs, the fractional solution schedules job j starting at r_j. Once job k is released, it preempts job j in favor of job k because of the partial credit due to $\omega_k > \omega_j$. At some point later, it preempts job k in favor of job j to finish it by its deadline. It then schedules job k again. Thus the fractional solution is not laminar. There may not be a general way to massage such a solution to make it laminar without increasing its cost by too much.

Our approach works even if the fractional solution is *not* laminar. It first partitions the time into "aligned" intervals of length a power of β, an integer greater than 1. Thus these intervals are of the form $[a \cdot \beta^c, (a+1) \cdot \beta^c)$ where a and c are integers. We refer to c as the *class* of an interval of this type. It is easy to see that aligned intervals from all classes form a laminar family.[3] Intuitively speaking, our algorithm uses a rounding procedure on bipartite graphs where jobs on one side are fractionally assigned to aligned intervals from a certain class on the other side. We would like to convert this assignment into an integral assignment randomly while preserving the expectation and ensuring that the maximum number of jobs in any aligned interval is at most the ceiling of its expectation. We reduce this problem to rounding a fractional max-flow to an integral max-flow in a network with integral arc capacities. This can be done using the total unimodularity of network flow matrices, see details in Section 2.4. Our rounding is reminiscent of the bipartite graph based dependent rounding of Gandhi et al. [8]. Their rounding, however, cannot be used here, because our problem cannot be formulated as a rounding problem on bipartite graphs since we need to satisfy the so-called "degree-preservation constraints" for all aligned intervals which form a laminar family.

From a more general perspective, one can see this rounding as decomposing a point in the intersection polytope of a partition matroid and a laminar matroid as a convex combination of incidence vectors of sets which are independent in both the matroids. It turns out that the intersection polytope of a partition and a laminar matroid can be expressed as the set of feasible source-sink flows in a network with integral arc capacities. Thus the problem of decomposing a point in such a polytope as a convex combination of integral extreme points can be reduced to network flow computations. An algorithm, for rounding a point in the intersection polytope of two matroids, with some concentration properties was recently presented by Chekuri et al. [6]. We do not need their complex rounding scheme since we do not need the concentration properties in our analysis.

1.3 Related Work

The closest works to ours are Bansal et al. [3] and Bansal and Pruhs [4]. As mentioned before, Bansal et al. [3] consider *non-preemptive* single processor scheduling for min-sum objectives like weighted flow-time, weighted tardiness, *unweighted* number of tardy jobs. We generalize their rounding procedure to work with arbitrary cost functions. Bansal and Pruhs [4] consider single processor scheduling with the same general cost functions as we do, plus preemption and release dates. There are, however, no setup times. Reducing this problem to a particular geometric set-cover problem yields a randomized polynomial time algorithm with approximation ratio $O(\log \log(nP))$, where P is the maximum job size. They also give an $O(1)$ approximation in the special case of identical release times. Recently, Im et al. [9] showed that the Highest-Density-First algorithm is $(2+\epsilon)$-speed $O(1)$-competitive for general monotone penalty functions. On the one hand, their algorithm is *online*, which is stronger, but on the other hand, they allow job preemption without any setup time penalty.

Several models for preemption penalties have been considered before. These include sequence-dependent, job-dependent or processor-dependent penalties. See Potts and van Wassenhove [13] and Allahverdi et al. [2] for surveys of the area. Most of the results deal with specific cost functions such as total completion time, total flow-time, or makespan. Schuurman and Woeginger [14], for example, present $(4/3+\epsilon)$-approximation for minimizing makespan in the context of multiple parallel processors, with preemption, job-migration and job-dependent setup times. There has also been some work in online scheduling with preemption penalties as well. For example, Divakaran and Saks [7] consider the single processor online problem with release times and setup times. The goal is to minimize the *maximum* flow time for a job. They present an $O(1)$-competitive algorithm for this problem. They also show that

[3] A laminar family is not to be confused with a non-laminar fractional solution.

the offline problem is $\mathbb{NP}$-hard. Chan et al. [5] study the online flow time scheduling in the presence of preemption overheads and present a simple algorithm that is $(1+\epsilon)$-speed $(1+1/\epsilon)$-competitive.

2 Algorithm for GSP

2.1 Outline of the Algorithm

We give a randomized polynomial time $O(1)$-speed $O(1)$-approximation algorithm. Our algorithm and analysis have the following high-level steps.

1. Incurring a constant speedup factor, we argue that GSP can be reduced to a problem called *multi-piece non-preemptive scheduling* problem (MPSP) and one can restrict the search to so-called "aligned" schedules.
2. We then create and solve an LP relaxation to lower bound the cost of the optimum schedule.
3. We next perform randomized rounding of the fractional solution using network flow techniques to obtain a "pseudo"-schedule.
4. Losing another constant speedup factor, we convert the pseudo-solution into a feasible schedule.
5. Finally, incurring yet another constant speedup factor, we get a feasible solution with cost at most constant times the LP lower bound.

We remark that steps 1, 2 and 4 are very similar to the corresponding steps of the algorithm of Bansal et al. [3]. Our main contribution is in step 3. Step 5 is a final (and simple) wrap-up needed to bound the cost of the computed solution.

2.2 Step 1: Reduction to mpsp and Restricting to Aligned Schedules

We begin by reducing our problem to *multi-piece non-preemptive scheduling* problem (MPSP), defined as follows. The input to MPSP consists of a set $\mathcal{J}$ of jobs, where each job $j \in \mathcal{J}$ is associated with release time $r_j \in \mathbb{Z}_+$, number of pieces $n_j \in \mathbb{Z}_+$, processing time $p_j \in \mathbb{Z}_+$ of each piece and a non-decreasing cost function[4] $f_j : \mathbb{Z}_+ \to \mathbb{Z}_+ \cup \{\infty\}$. A feasible schedule on a single processor works on no job before its release time and works on at most one job at any time. A feasible schedule also schedules each job j in exactly n_j intervals, each of length exactly p_j. Given a feasible schedule, let C_j denote the completion time of job j, i.e., the maximum end time of any interval corresponding to job j. The MPSP is to find a feasible schedule that minimizes the total cost $\sum_{j\in\mathcal{J}} f_j(C_j)$.

Lemma 2.1 *If there is a polynomial-time σ-speed ρ-approximation algorithm for* MPSP*, there is a polynomial-time 2σ-speed ρ-approximation algorithm for* GSP*.*

Proof: Given instance $(\mathcal{J}, \{(r_j, s_j, p_j, f_j) \mid j \in \mathcal{J}\})$ of GSP, define an instance $(\mathcal{J}, \{(r'_j, n'_j, p'_j, f'_j) \mid j \in \mathcal{J}\})$ of MPSP by letting $r'_j = r_j$, $n'_j = \max\{1, \lceil p_j/\max\{1, s_j\}\rceil\}$, $p'_j = \max\{1, 2s_j\}$ and $f'_j = f_j$ for each job $j \in \mathcal{J}$. Let OPT(GSP) denote the optimum value of the GSP instance and let OPT(MPSP) denote the optimum value of the MPSP instance on a processor with twice the speed. We first argue that OPT(MPSP) $\leq$ OPT(GSP). Consider the optimum schedule S of the GSP instance. We construct a feasible solution S' for the MPSP instance on a processor with twice the speed as follows. Consider an interval $[t, t + s_j + t')$ in which S schedules a job j. Let S' schedule $\lceil t'/\max\{1, s_j\}\rceil$ pieces of length $p'_j = \max\{1, 2s_j\}$ each in this interval. Since there is at least one such interval and the sum of processings

[4] We again assume that f_j is given by a value oracle that given $t \in \mathbb{Z}_+$ returns $f_j(t)$.

t' over all such intervals is p_j, the total number of pieces scheduled for job j is at least n'_j. Therefore S' gives a feasible solution for the MPSP instance on a processor with twice the speed. It is easy to see that the cost of S' is at most that of S.

Now it is enough to show that given a solution S' for the MPSP instance, one can construct a solution S for the GSP instance feasible on a processor with the same speed and with cost at most that of S'. Consider an interval $[t, t+p'_j)$ in which S' schedules a piece of job j. Let S schedule job j in this interval so that it spends s_j time in setup and $\max\{1, s_j\}$ time in processing. Thus job j gets $n'_j \cdot \max\{1, s_j\} \geq p_j$ processing overall. It is easy to see that the cost of S is at most that of S'. ■

In light of this lemma, we focus on designing an $O(1)$-speed $O(1)$-approximation algorithm for MPSP. Fix an instance $(\mathcal{J}, \{(r_j, n_j, p_j, f_j) \mid j \in \mathcal{J}\})$ of MPSP. Let $\beta > 1$ be an integer to be determined later. Incurring a speedup factor of β, we assume that all processing times in the instance are integer powers of β by replacing p_j by $\beta^{\lfloor \log_\beta p_j \rfloor}$. We define a notion of aligned schedules.

Definition 2.2 *We say that a schedule for* MPSP *is* aligned *if each piece of each job j is scheduled in an interval of the form $[ap_j, (a+1)p_j)$ where $a \geq 0$ is an integer.*

Lemma 2.3 *On a processor with speedup factor* 2*, there exists an aligned schedule with cost at most that of the original* MPSP *instance.*

Proof: Fix an optimum schedule S for MPSP. Suppose S processes a piece of job j in the interval $[t, t+p_j)$. Let $t' \in [t, t+p_j/2)$ be an integral multiple of $p_j/2$. On a processor with twice the speed, we can process this piece of job j in the interval $[t', t'+p_j/2)$. It is easy to see that the resulting schedule is aligned and has cost at most that of S. ■

To summarize, by incurring an overall speedup factor of 4β, we reduce GSP to MPSP, assume that each p_j is an integer power of β and set our goal to finding the aligned schedule with minimum cost. Let OPT denote the cost of an optimum aligned schedule of the new MPSP instance.

2.3 Step 2: Solving the Linear Programming Relaxation

We now present a linear programming relaxation for lower bounding OPT. Since any schedule, without loss of generality, completes all jobs by time $T = \max_j r_j + \sum_j n_j p_j$, it is enough to work within this time horizon. We introduce a variable $x(j,t)$ for each job j and each multiple t of p_j such that $t \geq r_j$. In the "intended" solution, $x(j,t) = 1$ if a piece of job j is scheduled in the interval $[t, t+p_j)$. We also introduce a variable W_j intended to lower bound the cost of job j.

$$\begin{aligned}
&\text{minimize} \quad \sum_j W_j && (1)\\
\forall j \in \mathcal{J}, \quad & n_j = \sum_t x(j,t) && (2)\\
\forall \tau \in \mathbb{Z}_+, \quad & 1 \geq \sum_j \sum_{t:\tau\in[t,t+p_j)} x(j,t) && (3)\\
\forall j \in \mathcal{J}, \quad & W_j \geq \frac{1}{n_j}\sum_t x(j,t)\cdot f_j(t+p_j) && (4)\\
\forall j \in \mathcal{J}, \quad & W_j \geq \sum_{k\neq j} \sum_{t:r_j\in(t,t+p_k)} x(k,t)\cdot f_j(t+p_k) && (5)\\
\forall j,t, \quad & x(j,t) \geq 0 && (6)
\end{aligned}$$

Lemma 2.4 *The optimum value of the LP (1)-(6) is at most* OPT.

Proof: Consider an optimal aligned schedule S with cost OPT. Using S, we construct a feasible solution $\{x^*, W^*\}$ to the LP with value at most OPT. Let $x^*(j,t) = 1$ if S schedules a piece of j in the interval $[t, t+p_j)$ and 0 otherwise. Let $W_j^* = f_j(C_j)$ denote the cost of job j in S, where C_j denotes the completion time of j in S. It is easy to see that constraints (2) and (3) are satisfied by this solution, since each job j has exactly n_j pieces scheduled and S works on at most one job at a time, respectively. We now argue that the cost of job j in S is at least the right-hand-side of constraint (4) (the "average lower bound") or (5) (the "displacement lower bound") for job j. Fix a job j and let $[t_i, t_i + p_j)$ for $i = 1, 2, \ldots$ be the intervals containing the pieces of job j. The right-hand-side of constraint (4) for job j is the average of $f_j(t_i + p_j)$ for $i = 1, 2, \ldots$ which is clearly at most $W_j^* = \max_i f_j(t_i + p_j)$, the cost of job j in S. Thus the constraint (4) is satisfied. Now note that there is at most one job $k \neq j$ that has $x^*(k,t) > 0$ for values of t with $r_j \in (t, t+p_k)$. This is because S works on at most one job at a time. If such a job does not exist, it is easy to see that the constraint (5) is satisfied. If such a job k exists, the earliest time the first piece of job j can start is at least $t + p_k$. Thus its cost is at least $f_j(t + p_k)$. Since $x^*(k,t) = 1$ holds in such a case, the constraint (5) is satisfied. ■

The number of variables and constraints in this LP is pseudo-polynomial. Using an approach similar to Bansal et al. [3], we can make this LP polynomial-sized at a small loss. We omit detailed from this version due to lack of space. For now, let us assume that we can compute a fractional optimum, denoted by $x^*(j,t)$ and W_j^* for $j \in J$ and $0 \leq t \leq T$, of this LP.

2.4 Step 3: Obtaining a Pseudo-Schedule via Network Flows

We say that a job j belongs to *class* $c \geq 0$ if $p_j = \beta^c$. Let $\mathcal{J}_c$ denote the set of jobs in class c. We obtain a pseudo-schedule for jobs in $\mathcal{J}_c$ for each class c separately. Fix a class c. To obtain a pseudo-schedule for class c, we create an instance of the network flow problem. For an integer $d \geq 0$, we call an interval of the form $[a \cdot \beta^d, (a+1) \cdot \beta^d)$ an *aligned* β^d*-interval* or simply an *aligned interval*, where $a \geq 0$ is an integer.

2.4.1 Creating a flow network

Refer to Figure 1 for an example for the case of $\beta = 2$. The flow network we create is a layered directed acyclic graph with all arcs going between consecutive layers from top to bottom. The first layer consists of the `source` node. The second layer has a node v_j for each job $j \in \mathcal{J}_c$. For $k \geq 0$, the $(k+3)$rd layer has a node v_I for each aligned β^{c+k}-interval I. The last layer has a node for the single aligned interval spanning the entire instance – we call this node the `sink`.

We add an arc from the `source` to v_j with capacity $n_j = \sum_t x(j,t)$, i.e., the total x-value job j receives. We add an arc from v_j to v_I for β^c-aligned interval $I = [t, t+\beta^c)$ such that $x(j,t) > 0$. We assign such an arc a capacity of

$$\lceil x(j,t) \rceil ,$$

i.e., the x-value that job j receives in aligned β^c-interval I rounded up to the nearest integer. For $k \geq 0$, we add an arc from v_I to $v_{I'}$ for an aligned β^{c+k}-interval I and aligned β^{c+k+1}-interval I' such that $I \subset I'$, provided v_I is not the `sink`. We give this arc a capacity of

$$\left\lceil \sum_{j \in \mathcal{J}_c} \sum_{t : [t, t+\beta^c) \subseteq I} x(j,t) \right\rceil ,$$

i.e., the total x-value all jobs in $\mathcal{J}_c$ receive in aligned β^c-intervals contained in I rounded up to the

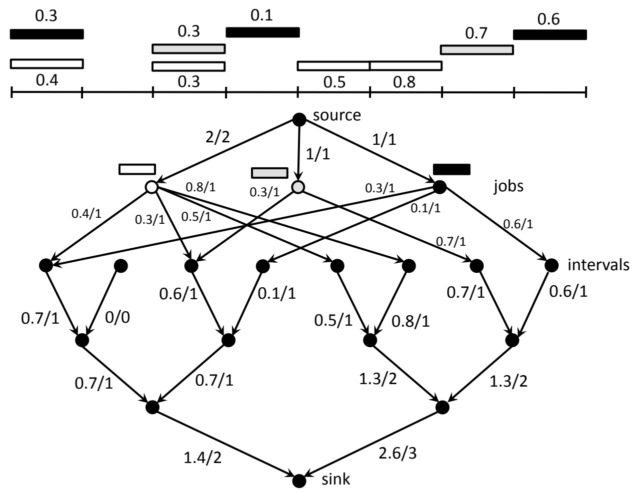

Figure 1: Creating a network flow instance. Consider the adjacent example with 3 jobs and 8 aligned intervals in a certain class. Assume that $\beta = 2$. The total x-values white, grey and black jobs receive are 2, 1 and 1 respectively. The corresponding flow network has a source node in the top layer, 3 nodes corresponding to jobs in the second layer and nodes corresponding to aligned intervals arranged in form of a β-ary tree. The bottom layer has a sink node. The numbers on the arcs denote their flows/integral capacities. The fractional maximum flow given is computed from the fractional solution.

nearest integer. Thus the flow network below layer 3 looks like an inverted β-ary tree. Note also that all arc-capacities in this network are integral.

Observe that the fractional solution $\{x(j,t) \mid j \in \mathcal{J}_c, 0 \le t \le T\}$ gives a fractional feasible maximum flow in this network from `source` to `sink` of total value $\sum_{j\in\mathcal{J}_c} n_j = \sum_{j\in\mathcal{J}_c}\sum_t x(j,t)$ as follows. Send a flow of $n_j = \sum_t x(j,t)$ on $(\texttt{source}, v_j)$ for all $j \in \mathcal{J}_c$, a flow of $x(j,t)$ on $(v_j, v_{[t,t+\beta^c)})$ for all $j \in \mathcal{J}_c$ and t with $x(j,t) > 0$ and a flow of $\sum_{j\in\mathcal{J}_c}\sum_{t:[t,t+\beta^c)\subseteq I} x(j,t)$ on the unique out-going arc $(v_I, v_{I'})$ for all aligned intervals I except the one corresponding to the `sink`.

We now use the fact that a network flow matrix is totally unimodular and hence the polytope of flows in a network with integral arc capacities has integral extreme points. Therefore any point inside such a polytope can be decomposed as a convex combination of integral flows.

Lemma 2.5 *Consider a flow network with a source node, a sink node and integral arc capacities. Given a fractional maximum flow f in the network, one can compute a collection of integral maximum flows $F_1, \ldots, F_q$ and corresponding $\lambda_1, \ldots, \lambda_q \ge 0$ with $\sum_i \lambda_i = 1$ such that $f = \sum_i \lambda_i F_i$. Furthermore, the size q of the convex combination and its computation time is bounded polynomially in the input size.*

Proof: We cast the problem of decomposing the given maximum flow f as a convex combination of integral maximum flows as a linear program. Introduce a variable λ_i for each integral maximum flow F_i (that satisfies the arc capacity constraints). There may be exponentially many such flows. Now consider the following LP and its dual.

$$\text{Primal:} \qquad \max\{\sum_i \lambda_i \mid \sum_i \lambda_i F_i(e) = f(e)\ \forall \text{ arcs } e, \lambda_i \ge 0\ \forall\, i\},$$
$$\text{Dual:} \qquad \min\{\sum_e f(e) l_e \mid \sum_e F_i(e) l_e \ge 1\ \forall\, i, l_e \in \mathbb{R}\ \forall \text{ arcs } e\}.$$

The dual has exponentially many constraints but only polynomially many variables. We can solve the dual using the *ellipsoid* algorithm using the separation oracle that given not-necessarily-positive "arc-lengths" $\{l_e\}$ computes a maximum flow F_i with "minimum cost" $\sum_e F_i(e)l_e$. Several polynomial time algorithms exist for this well-known min-cost max-flow problem [1]. Recall that since all arc capacities are integral, this oracle returns an integral flow. The ellipsoid algorithm finds polynomially many maximum flows while computing the dual optimum solution. We can then restrict our attention to these maximum flows (i.e., set $\lambda_i = 0$ for all maximum flows F_i *not* found in the ellipsoid algorithm) and solve the new primal that now has polynomially many variables and constraints. Since each F_i as well as f are maximum flows, it is easy to see that the optimum primal solution thus computed has value $\sum_i \lambda_i = 1$. ■

Our rounding procedure to obtain a pseudo-schedule for class c works as follows.

> **Procedure round:** Use Lemma 2.5 to compute a convex combination of integral flows. Pick exactly one integral flow F_i with probability λ_i. Schedule $F_i(v_j, v_I)$ pieces of job j in the aligned β^c-interval I for all jobs j and aligned β^c-intervals I.

Lemma 2.6 *The pseudo-schedule for all classes constructed by the above rounding procedure satisfies the following properties.*

1. *The expected number of pieces any job j receives in an aligned interval $[t, t+\beta^c)$ is $x(j,t)$.*
2. *With probability 1, each job j has exactly n_j pieces scheduled overall.*
3. *With probability 1, at most $\lceil \sum_{j\in\mathcal{J}_c}\sum_{t:[t,t+\beta^c)\subseteq I} x(j,t)\rceil$ pieces of jobs in class c, counting multiplicities from the same job, are scheduled in any interval I of a class $d \geq c$.*
4. *Consider any interval in class d. With probability 1, the total size of all pieces of jobs in classes $0,\ldots,d$ scheduled in this interval is at most $\beta^d(2+\frac{1}{\beta-1})$.*

Proof: The properties 1, 2 and 3 hold directly from the rounding. To see property 4, fix an interval I in class d. Summing the volume constraint (3) that the fractional solution satisfies over all $\tau \in I$

$$\sum_{c=0}^{d}\beta^c\left(\sum_{j\in\mathcal{J}_c}\sum_{t:[t,t+\beta^c)\subseteq I} x(j,t)\right) \leq \beta^d.$$

Now from property 3, at most $\lceil \sum_{j\in\mathcal{J}_c}\sum_{t:[t,t+\beta^c)\subseteq I} x(j,t)\rceil$ pieces of jobs in class c are scheduled in I. Therefore the total size of all the pieces of jobs in classes $0,\ldots,d$ scheduled in I is at most

$$\begin{aligned}\sum_{c=0}^{d}\beta^c\left\lceil\sum_{j\in\mathcal{J}_c}\sum_{t:[t,t+\beta^c)\subseteq I} x(j,t)\right\rceil &< \sum_{c=0}^{d}\beta^c\left(1+\sum_{j\in\mathcal{J}_c}\sum_{t:[t,t+\beta^c)\subseteq I} x(j,t)\right)\\ &\leq \sum_{c=0}^{d}\beta^c + \beta^d\\ &= \beta^d\left(\frac{\beta}{\beta-1}+1\right).\end{aligned}$$

■

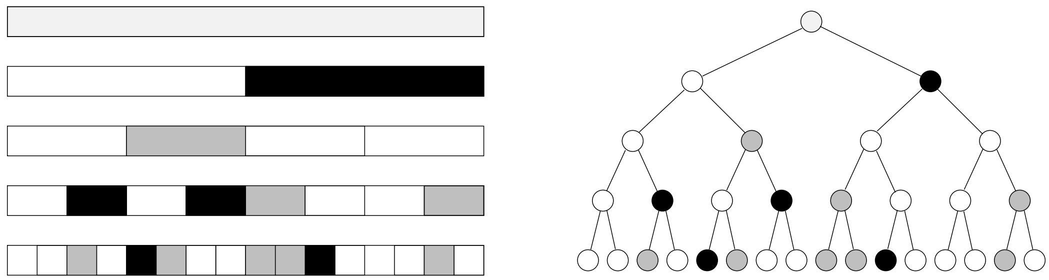

Figure 2: Example of a pseudo-schedule output by the rounding procedure for $\beta = 2$ and its corresponding β-ary tree. Each shaded box (and the respective tree-node) corresponds to one or more pieces of one or more jobs. From Lemma 2.6-4, the total size of pieces in any sub-tree (containing a node and all its descendants) is at most $(2 + \frac{1}{\beta-1})$ times the size of the root of the sub-tree. The black boxes/tree-nodes represent pieces of early jobs while grey boxes/tree-nodes represent pieces of late jobs. A box/tree-node can be both black or grey. (Source: Bansal et al. [3])

2.5 Step 4: Converting the Pseudo-Schedule into a Feasible Schedule

We use a factor $(2+\frac{1}{\beta-1})$ speedup to convert the pseudo-schedule into a feasible schedule that schedules at most one job at any single time. Consider the pseudo-schedule produced in step 3. Call an aligned interval *maximal* if it contains a piece of a job of equal size and does not overlap with any other piece of a job of larger size. There can be multiple pieces corresponding to any maximal aligned interval. Fix a maximal interval I. We associate a natural β-ary tree corresponding to all the pieces overlapping with I. The tree-nodes in level d correspond to the aligned β^d-intervals overlapping with I. See Figure 2 for an example of such a tree for the case $\beta = 2$.

We give a procedure that uses a speedup factor of $(2+\frac{1}{\beta-1})$, and given a tree corresponding to a maximal interval I, feasibly schedules all the pieces in that tree in the aligned β^c-interval corresponding to the root. The schedule is feasible in the sense that pieces of each job are not scheduled before its release time and no two pieces overlap with each other. Since all the maximal intervals are non-overlapping, applying the above procedure to each corresponding tree produces a schedule for the entire instance.

Procedure fit: We first shrink all pieces in J_I by a factor of $(2 + \frac{1}{\beta-1})$. We then compute the POSTORDER[5] traversal of the β-ary tree T_I. We schedule all pieces of early jobs in the order they appear in POSTORDER(T_I), pieces of equal lengths overlapping with each other ordered arbitrarily. We then compute the PREORDER traversal of T_I. We schedule all pieces of late jobs in the order they appear in PREORDER(T_I), pieces of equal lengths overlapping with each other ordered arbitrarily. These pieces of late jobs are then "right-justified", shifted as far right as possible so that the last piece completes at the end-point of interval I.

Consider a maximal interval $I = [\tau, \tau + \beta^c)$. Let J_I denote the set of jobs corresponding to pieces scheduled in the interval I, and let T_I denote the β-ary tree associated with the pseudo-schedule in I. We

[5]The POSTORDER traversal of a single-node tree v is defined as POSTORDER$(v) := v$ and that of a β-ary tree T with root r and left-to-right sub-trees $T_1, \dots, T_\beta$ is recursively defined as POSTORDER$(T) :=$ POSTORDER$(T_1), \dots,$ POSTORDER$(T_\beta), r$. Similarly, the PREORDER traversal of a single-node tree v is defined as PREORDER$(v) := v$ and that of a β-ary tree T with root r and left-to-right sub-trees $T_1, \dots, T_\beta$ is recursively defined as PREORDER$(T) := r,$ PREORDER$(T_1), \dots,$ PREORDER(T_β).

partition the jobs J_I into two sets, denoted *early* and *late*. The *early* jobs are the jobs $\{j \in J_I \mid r_j \leq \tau\}$ that are released not later than time τ. The pieces of these jobs can be scheduled anywhere in I. Note that even though an early job k is scheduled during the interval I, it does not pay the "penalty term" in the constraint (5). The *late* jobs are the jobs $\{j \in J_I \mid \tau < r_j < \tau + \beta^c\}$ that are released in I. A piece of a late job j can be scheduled no earlier than its release time r_j. Note that a late job j pays the "penalty term" in the constraint (5). We now describe our procedure FIT, to convert the pseudo-schedule into a feasible schedule. The following lemma shows that the schedule computed by the FIT procedure is feasible.

Lemma 2.7 *The schedule output by the* FIT *procedure satisfies the following properties.*

1. *The pieces of jobs in J_I are scheduled in the interval I such that no two pieces overlap.*
2. *Each piece of each early job in J_I completes no later than its completion time in the pseudo-schedule.*
3. *Each piece of each late job in J_I starts no earlier than its start time in the pseudo-schedule, and it completes within I.*

Proof: The first property follows from the observation that the total size of all the jobs in J_I is at most $(2 + \frac{1}{\beta-1})$ times the length of the interval I and the fact that we shrink all the pieces by a factor of $(2 + \frac{1}{\beta-1})$. We now prove the second property. Consider a piece π of an early job in J_I. Let its completion time in the pseudo-schedule be $\tau + \tau_\pi$. It is sufficient to argue that the total size of pieces of early jobs (including π) that come no later than π in POSTORDER(T_I) is at most $\tau_\pi(2 + \frac{1}{\beta-1})$ before shrinking. To this end, consider the prefix of POSTORDER(T_I) up to the tree-node corresponding to π. Let $T_1, \ldots, T_q$ be the disjoint subtrees of T_I that are traversed in POSTORDER(T_I) up to node π. Note that the root of T_q is π. Now let $I_1, \ldots, I_q$ be the (disjoint) intervals occupied by the roots of $T_1, \ldots, T_q$. Note that the total size of $I_1, \ldots, I_q$ is precisely τ_π. Furthermore, from Lemma 2.6-4, the total size of pieces of jobs in J_I that are contained in intervals $I_1, \ldots, I_q$ is at most $(2 + \frac{1}{\beta-1})$ times the total size of these intervals. Thus, in particular, the total size of pieces of the early jobs in these intervals is at most $\tau_\pi(2 + \frac{1}{\beta-1})$, and the property follows. The third property can be proved analogously, but with late jobs and PREORDER(T_I). ■

2.6 Step 5: Final Wrap-up

The solution obtained in step 4 is feasible on a processor with speedup factor $4\beta(2 + \frac{1}{\beta-1})$. We now define pieces-wise average costs of a job $j \in \mathcal{J}$ in the pseudo-schedule and a schedule computed by procedure FIT.

Definition 2.8 *Consider a job $j \in \mathcal{J}$. Let $A_j = \frac{1}{n_j} \sum_{i=1}^{n_j} f_j(t_i + p_j)$ be the* average *cost of job j in the pseudo-schedule where $\{[t_i, t_i+p_j) \mid 1 \leq i \leq n_j\}$ denotes the set of intervals in which the pseudo-schedule computed by Procedure* ROUND *schedules job j. Furthermore let $A'_j = \frac{1}{n_j} \sum_{i=1}^{n_j} f_j(t'_i + p'_j)$ be the* average *cost of job j in the feasible schedule computed by Procedure* FIT. *Here $p'_j = p_j/(2 + \frac{1}{\beta-1})$ denotes the processing time of job j after the scaling and $\{[t'_i, t'_i + p'_j) \mid 1 \leq i \leq n_j\}$ denotes the set of intervals in which Procedure* FIT *schedules job j.*

The following lemma bounds the expected average cost of a job in the feasible schedule.

Lemma 2.9 *For any job $j \in \mathcal{J}$, the expected value of A'_j is at most $2W_j$.*

Proof: From Lemma 2.6-1 and the constraint (4), it is clear that the expected value of A_j is at most W_j. We next prove that the expected value of $A'_j - A_j$ is at most W_j.

To this end, fix a job $j \in \mathcal{J}$ and consider a piece $\pi_i = [t_i, t_i + p_j)$, where $1 \le i \le n_j$ in the pseudo-schedule. During Procedure FIT, job j was labelled as early for π_i with a certain probability and late for π_i with a certain probability. From Lemma 2.7, if job j was labelled as early for π_i, the piece π_i completed in the feasible schedule at or before its completion time in the pseudo-schedule and thus its contribution to $A'_j - A_j$ is non-positive. Now suppose that job j was labelled as late for π_i. Let $I_{\max}$ denote the random variable denoting the maximal interval that contains π_i. For any interval $I = [t, t+\ell) \ni r_j$, we have $I_{\max} = I$ with probability at most $\sum_{t,\ell:r_j\in(t,t+\ell)} \sum_{k:k\neq j,p_k=\ell} x(k,t)$. This follows from Lemma 2.6-1. In the event that $I_{\max} = I$, from Lemma 2.7, the piece π_i completes by time $t+\ell$. Thus the expected contribution of π_i to $A'_j - A_j$ is at most

$$\frac{1}{n_j} \sum_{t,\ell:r_j\in(t,t+\ell)} \sum_{k:k\neq j,p_k=\ell} x(k,t)\cdot f_j(t+\ell) = \frac{1}{n_j} \sum_{k\neq j} \sum_{t:r_j\in(t,t+p_k)} x(k,t)\cdot f_j(t+p_k).$$

Summing this over all pieces π_i of job j, we get from constraint (5) that the expected value of $A'_j - A_j$ is at most W_j. Hence the lemma holds. ■

Note that the actual cost of job j is the maximum value of $f_j(t'_i + p'_j)$ over all of its pieces $\{[t'_i, t'_i + p'_j) \mid 1 \le i \le n_j\}$. For any non-decreasing cost function f_j, this cost can be arbitrarily larger than its average cost A'_j. Using a simple trick to bound the actual cost of the solution, we incur another factor α in speedup, where $\alpha > 1$ is an integer to be fixed later. We use the following simple observation. For a random variable x with range $\in \mathbb{Z}_+$ and a non-decreasing function $w : \mathbb{Z}_+ \to \mathbb{Z}_+$, we have $\mathbb{E}[w(x)] \ge \Pr[x \ge x_0] \cdot w(x_0)$ for all $x_0 \in \mathbb{Z}_+$, where $\mathbb{E}$ denotes the expectation operator.

Suppose the pieces $[t'_i, t'_i + p'_j)$ for $1 \le i \le n_j$ are sorted in the increasing order of their completion times $t'_i + p'_j$ in the feasible schedule. For any job j, shrink its processing time by factor α and schedule α pieces in each of the intervals $[t'_i, t'_i + p'_j)$ for $1 \le i < i_0 := \lceil n_j/\alpha \rceil$ and at most α pieces in the interval $[t'_{i_0}, t'_{i_0} + p'_j)$. It is easy to see that the actual cost of job j is at most

$$f_j(t'_{i_0} + p'_j) \le \frac{A'_j}{1 - \frac{\lfloor n_j/\alpha \rfloor}{n_j}} \le \frac{A'_j}{1 - \frac{1}{\alpha}} = \frac{\alpha A'_j}{\alpha - 1}.$$

Thus the expected actual cost of job j is at most $2\alpha W_j/(\alpha-1)$.

In summary, we obtain a randomized algorithm that gives a feasible schedule on a processor with $4\beta(2 + \frac{1}{\beta-1})\alpha$ speedup having expected cost at most $2\alpha/(\alpha-1)$ times the optimum. If $\alpha = \beta = 2$, we get a randomized polynomial time 48-speed 4-approximation algorithm for MPSP, and (from Lemma 2.1) a randomized polynomial time 96-speed 4-approximation algorithm for GSP.

Acknowledgements. We thank Nikhil Bansal for useful discussions.

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows : Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] A. Allahverdi, C. T. Ng, T. C. E. Cheng, and M. Y. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal on Operations Research*, 2008.

[3] N. Bansal, H.-L. Chan, R. Khandekar, K. Pruhs, C. Stein, and B. Schieber. Non-preemptive min-sum scheduling with resource augmentation. In *Proceedings of the 48th Annual Symposium on Foundations of Computer Science*, pages 614–624, 2007.

[4] N. Bansal and K. Pruhs. Geometry of scheduling. In *Proceedings of the 51st Annual Symposium on Foundations of Computer Science*, pages 81–90, 2004.

[5] Ho-Leung Chan, Tak-Wah Lam, and Rongbin Li. Online flow time scheduling in the presence of preemption overhead. In *Proceedings of International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*, pages 85–97, 2012.

[6] C. Chekuri, J. Vondrák, and R. Zenklusen. Multi-budgeted matchings and matroid intersection via dependent rounding. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1080–1097, 2011.

[7] Srikrishnan Divakaran and Michael Saks. An online algorithm for a problem in scheduling with set-ups and release times. *Algorithmica*, 56, 2009.

[8] Rajiv Gandhi, Samir Khuller, Srinivasan Parthasarathy, and Aravind Srinivasan. Dependent rounding and its applications to approximation algorithms. *J. ACM*, 53(3):324–360, 2006.

[9] Sungjin Im, Benjamin Moseley, and Kirk Pruhs. Online scheduling with general cost functions. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 1254–1265, 2012.

[10] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000.

[11] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 418–426, May 1996.

[12] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32:163–200, 2001.

[13] C.N. Potts and L.N. van Wassenhove. Integrating scheduling with batching and lotsizing: a review of algorithms and complexity. *Journal of the Operational Research Society*, 43:395–406, 1992.

[14] Petra Schuurman and Gerhard Woeginger. Preemptive scheduling with job dependent-setup times. In *Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 759–767, 1999.