Minimizing maximum (weighted) flow-time on related and unrelated machines

S. Anand[1] Karl Bringmann[2] Tobias Friedrich[3]
Naveen Garg[1] Amit Kumar[1]

[1] Department of Computer Science and Engineering, IIT Delhi, India
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[3] Friedrich-Schiller-Universität Jena, Germany

Abstract. In this paper we initiate the study of job scheduling on related and unrelated machines so as to minimize the maximum flow time or the maximum weighted flow time (when each job has an associated weight). Previous work for these metrics considered only the setting of parallel machines, while previous work for scheduling on unrelated machines only considered $L_p, p < \infty$ norms. Our main results are:

(i) We give an $O(\varepsilon^{-3})$-competitive algorithm to minimize maximum weighted flow time on related machines where we assume that the machines of the online algorithm can process $1+\varepsilon$ units of a job in 1 time-unit (ε speed augmentation).

(ii) For the objective of minimizing maximum flow time on unrelated machines we give a simple $2/\varepsilon$-competitive algorithm when we augment the speed by ε. For m machines we show a lower bound of $\Omega(m)$ on the competitive ratio if speed augmentation is not permitted. Our algorithm does not assign jobs to machines as soon as they arrive. To justify this "drawback" we show a lower bound of $\Omega(\log m)$ on the competitive ratio of *immediate dispatch* algorithms. In both these lower bound constructions we use jobs whose processing times are in $\{1, \infty\}$, and hence they apply to the more restrictive *subset parallel* setting.

(iii) For the objective of minimizing maximum weighted flow time on unrelated machines we establish a lower bound of $\Omega(\log m)$-on the competitive ratio of any online algorithm which is permitted to use $s = O(1)$ speed machines. In our lower bound construction, job j has a processing time of p_j on a subset of machines and infinity on others and has a weight $1/p_j$. Hence this lower bound applies to the subset parallel setting for the special case of minimizing maximum stretch.

1 Introduction

The problem of scheduling jobs so as to minimize the flow time (or response time) has received much attention. In the online setting of this problem, jobs arrive over time and the flow time of a job is the difference between its release time (or arrival time) and completion time (or finish time). We assume that the jobs can be preempted. The task of the scheduler is to decide which machine to schedule a job on and in what order to schedule the jobs assigned to a machine.

One way of combining the flow times of various jobs is to consider the sum of the flow times. An obvious drawback of this measure is that it is not *fair* since some job might have a very large flow time in the schedule that minimizes sum of their flow times. A natural way to overcome this is to minimize the L_p norm of the flow times of the jobs [3, 5, 10, 11] which, for increasing values of p, would ensure better fairness. Bansal and Pruhs [5], however, showed that even for a single machine, minimizing, the L_p-norm of flow times requires speed augmentation — the online algorithm must have machines that are, say, ε-fraction faster (can do $1+\varepsilon$ unit of work in one time-unit) than those of the offline algorithm. With a $(1+\varepsilon)$-speed augmentation Bansal and Pruhs [5] showed that a simple algorithm which schedules the shortest job first is $\mathrm{O}(\varepsilon^{-1})$-competitive for any L_p-norm on a single machine; we refer to this as an $(1+\varepsilon, O(1/\varepsilon))$-competitive algorithm. Golovin *et.al.* [10] used a majorizing technique to obtain a similar result for parallel machines. While both these results have a competitive ratio that is independent of p, the results of Im and Moseley [11] and Anand *et.al.* [3] for unrelated machines have a competitive ratio that is linear in p and which therefore implies an unbounded competitive ratio for the L_∞-norm.

Our main contribution in this paper is to provide a comprehensive treatment of the problem of minimizing maximum flow time for different machine models. The two models that we consider are the *related machines* (each machine has speed s_i and the time required to process job j on machine i is p_j/s_i) and the *unrelated machines* (job j has processing time p_{ij} on machine i). A special case of the unrelated machine model is the *subset-parallel* setting where job j has a processing time p_j independent of the machines but can be assigned only to a subset of the machines.

Besides maximum flow time, another metric of interest is the maximum weighted flow time where we assume that job j has a weight w_j and the objective is to minimize $\max_j w_j F_j$, where F_j is the flow time of j in the schedule constructed. Besides the obvious use of job weights to model priority, if we choose the weight of a job equal to the inverse of its processing time, then minimizing maximum weighted flow time is the same as minimizing maximum *stretch* where stretch is defined as the ratio of the flow time to the processing time of a job. Chekuri and Moseley [9] considered the problem of minimizing the maximum *delay factor* where a job j has a deadline d_j, a release date r_j and the delay factor of a job is defined as the ratio of its flow time to $(d_j - r_j)$. This problem is in fact equivalent to minimizing maximum weighted flow time and this can be easily seen by defining $w_j = (d_j - r_j)^{-1}$.

The problem of minimizing maximum stretch was first considered by Bender *et.al.* [7] who showed a lower bound of $\Omega(P^{1/3})$ on the competitive ratio for a single machine where P is the ratio of the largest to the smallest processing time. Bender *et.al.* [7] also showed a $\mathrm{O}(P^{1/2})$-competitive algorithm for a single machine, which was improved by [8], while the lower bound was improved to $\Omega(P^{0.4})$ by [9].

For minimizing maximum weighted flow time, Bansal and Pruhs [6] showed that the highest density first algorithm is $(1+\varepsilon, O(\varepsilon^{-2}))$-competitive for single machines. For parallel machines, Chekuri and Moseley [9] obtained a $(1+\varepsilon, O(\varepsilon^{-1}))$-competitive algorithm that is both *non-migratory* (jobs once assigned to a machine are scheduled only on that machine) and *immediate dispatch* (a job is assigned to a machine as soon as the job arrives). Both these qualities are desirable in any scheduling algorithm since they reduce/eliminate communication overheads amongst the central server/machines.

Our main results and the previous work for these three metrics (Max-Flow-time, Max-Stretch and Max-Weighted-Flow-time) on the various machine models (single, parallel, related, subset parallel and unrelated) are expressed in Table 1. Note that the Max-Flow-time metric is not a special case of the Max-Stretch metric, and neither is the model of related machines a special case of the subset-parallel setting. Nevertheless, a lower bound result (respectively an upper bound result) for a machine-model/metric pair extends to all model/metric pairs to the right and below (respectively to the left and above) in the table.

	Max-Flow-time	Max-Stretch	Max-Weighted-Flow-time
Single Machine	polynomial time	$(1, \Omega(P^{2/5}))$ [9] and $(1, O(P^{1/2}))$ [7, 8]	$(1+\varepsilon, O(\varepsilon^{-2}))$ [6]
Parallel Machines	$(1, 2)$ [1]		$(1+\varepsilon, O(\varepsilon^{-1}))$ [9]
Related Machines			$(1+\varepsilon, O(\varepsilon^{-3}))$
Subset Parallel	$(1, \Omega(m))$	$(O(1), \Omega(\log m))$	
Unrelated Machines	$(1+\varepsilon, O(\varepsilon^{-1}))$		

Table 1. Previous results and the results obtained in this paper for the different machine models and metrics considered. The uncited results are from this paper.

Our main results are:

(i) We give an $O(\varepsilon^{-3})$-competitive non-migratory algorithm to minimize maximum weighted flow time on related machines with ε speed augmentation. When compared to a migratory optimum our solution is $O(\varepsilon^{-4})$-competitive.

(ii) For the objective of minimizing maximum flow time on unrelated machines we give a simple $2/\varepsilon$-competitive algorithm when we augment the speed by ε. For m machines we show a lower bound of $\Omega(m)$ on the competitive ratio if speed augmentation is not permitted. Our algorithm does not assign jobs to machines as soon as they arrive. However [4] show a lower bound of $\Omega(\log m)$ on the competitive ratio of any *immediate dispatch* algorithm. Both these lower bound constructions use jobs whose processing times are in $\{1, \infty\}$, and hence they apply to the more restrictive *subset parallel* setting.

(iii) For the objective of minimizing maximum weighted flow time on unrelated machines, we establish a lower bound of $\Omega(\log m)$-on the competitive ratio of any online algorithm which is permitted to use $s = O(1)$ speed machines. In our lower bound construction, job j has a processing time of p_j on a subset of machines and infinity on others and has a weight $1/p_j$. Hence this lower bound applies to the subset parallel setting for the special case of minimizing maximum stretch.

(iv) For minimizing the L_p-norm of stretch on subset parallel machines with a speed augmentation of $1+\varepsilon$, we show a lower bound of $\frac{p}{\varepsilon^{1-O(1/p)}}$ on the competitive ratio. This compares well with the $O(\frac{p}{\varepsilon^{2-O(1/p)}})$-competitive algorithm in [3] for minimizing l_p norm of weighted flow time on unrelated machines.

The problem of minimizing maximum (weighted) flow time also has interesting connections to *deadline scheduling.* In deadline scheduling besides its processing time and release time, job j has an associated deadline d_j and the objective is to find a schedule which meets all deadlines. For single machine it is known that the Earliest Deadline First (EDF) algorithm is optimum, in that it would find a feasible schedule if one exists. This fact implies a polynomial time algorithm for minimizing maximum flow time on a single machine. This is because, job j released at time r_j should complete by time $r_j + \texttt{opt}$, where $\texttt{opt}$ is the optimal value of maximum flow time. Thus $r_j + \texttt{opt}$ can be viewed as the deadline of job j. Hence EDF would schedule jobs in order of their release times and does not need to know $\texttt{opt}$.

For parallel machines it is known that no online algorithm can compute a schedule which meets all deadlines even when such a schedule exists. Phillips *et.al.* [12] showed that EDF can meet all deadlines if the machines of the on-line algorithm have twice the speed of the offline algorithms. This bound was improved to $\frac{e}{e-1}$ by Anand *et.al.* [2] for a schedule derived from the Yardstick bound. Our results imply that for related machines a constant speedup suffices to ensure that all deadlines are met while for the subset parallel setting, no constant (independent of number of machines) speedup can ensure that we meet deadlines.

The paper is organized as follows. In Section 2 and Section 4 we consider the problem of minimizing maximum weighted flow time on related machines and unrelated machines respectively. Section 3 considers the problem of minimizing maximum flow time on unrelated machines.

2 Max-Weighted-Flow-time on Related Machines

In this section we consider the Max-Weighted-Flow-time on related machines where the on-line algorithm is given $(1+\varepsilon)$-speed augmentation for some arbitrary small constant $\varepsilon > 0$. In the related machines setting, each job j has weight w_j, release date r_j and processing requirement p_j. We are given m machines with varying speed. Instead of working with speed, it will be more convenient to work with *slowness* of machines: the slowness of a machine i, denoted by s_i, is the reciprocal of its speed. Assume that $s_1 \leq \ldots \leq s_m$. For an instance $\mathcal{I}$, let $\texttt{opt}(\mathcal{I})$ denote the value of the optimal off-line solution for $\mathcal{I}$. We assume that the on-line algorithm is given $(1 + 4\varepsilon)$-speed augmentation. We say that a job j is *valid* for a machine i, if its processing time on i, i.e., $p_j s_i$, is at most $\frac{T}{w_j}$. Observe that a (non-migratory) off-line optimum algorithm will process a job j on a valid machine only.

We assume that all weights w_j are of the form 2^k, where k is a non-negative integer (this affects the competitive ratio by a factor of 2 only). We say that a job is of *class* k if its weight is 2^k. To begin with, we shall assume that the on-line algorithm knows the value of $\texttt{opt}(\mathcal{I})$ — call it T. In the next section, we describe an algorithm, which requires a small amount of "look-ahead". We describe it as an off-line algorithm. Subsequently, we show that it can be modified to an on-line algorithm with small loss of competitive ratio.

2.1 An off-line algorithm

We now describe an off-line algorithm $\mathcal{A}$ for $\mathcal{I}$. We allow machines speedup of $1 + 2\varepsilon$. First we develop some notation. For a class k and integer l, let $I(l, k)$ denote the interval $\left[\frac{lT}{\varepsilon 2^k}, \frac{(l+1)T}{\varepsilon 2^k}\right)$. We say that a job j is of *type* (k, l) if it is of class k and $r_j \in I(k, l)$. Note that the intervals $I(k, l)$ form a nested set of intervals.

The algorithm $\mathcal{A}$ is described in Figure 1. It schedules jobs in a particular order: it picks jobs in decreasing order of their class, and within each class, it goes by the order of release dates. When considering a job j, it tries machines in order of increasing speed, and schedules j in the first machine on which it can find enough free slots (i.e., slots which are not occupied by the jobs scheduled before j). We will show that it will always find some machine. Note that $\mathcal{A}$ may not respect release dates of jobs.

```
Algorithm $\mathcal{A}(\mathcal{I}, T)$:

For $k = K$ downto 1 ($K$ is the highest class of a job)
  For $l = 1, 2, \ldots$
    For each job $j$ of type $(k, l)$
      For $i = m_j$ downto 1 ($m_j$ is the slowest machine on which $j$ is valid)
        if there are at least $p_j s_i$ free slots on machine $i$ during $I(k, l)$ then
          schedule $j$ on $i$ during the first such free slots (without caring about $r_j$).
```

Fig. 1. The off-line algorithm

Analysis In this section, we prove that the algorithm $\mathcal{A}$ will always find a suitable machine for every job. We prove this by contradiction: let $j^\star$ be the first job for which we are not able to find such a machine. Then we will show that the $\texttt{opt}(\mathcal{I})$ must be more than T, which will contradict our assumption.

In the discussion below, we only look at jobs which were considered before $j^\star$ by $\mathcal{A}$. We build a set S of jobs recursively. Initially S just contains $j^\star$. We add a job j' of type (k', l') to S if there is a job j of type (k, l) in S satisfying the following conditions:

- The class k of j is at most k'.
- The algorithm $\mathcal{A}$ processes j' on a machine i which is valid for j as well.
- The algorithm $\mathcal{A}$ processes j' during $I(k, l)$, i.e., $I(k', l') \subseteq I(k, l)$.

We use this rule to add jobs to S as long as possible. For a machine i and interval $I(k, l)$, define the *machine-interval* $I_i(k, l)$ as the time interval $I(k, l)$ on machine i. We construct a set $\mathcal{N}$ of machine-intervals as follows. For every job

$j \in S$ of type (k,l), we add the intervals $I_i(k,l)$ to $\mathcal{N}$ for all machines i such that j is valid for i. We say that an interval $I_i(k,l) \in \mathcal{N}$ is *maximal* if there is no other interval $I_i(k',l') \in \mathcal{N}$ which contains $I_i(k,l)$ (note that both of the intervals are on the same machine). Observe that every job in S except $j^\star$ gets processed in one of the machine-intervals in $\mathcal{N}$. Let $\mathcal{N}'$ denote the set of maximal intervals in $\mathcal{N}$. We now show that the jobs in S satisfy the following crucial property.

Lemma 1. *For any maximal interval $I_i(k,l) \in \mathcal{N}$, the algorithm $\mathcal{A}$ processes jobs of S on all but $\frac{\varepsilon}{1+2\varepsilon}$-fraction of the slots in it.*

Proof. We prove that this property holds whenever we add a new maximal interval to $\mathcal{N}$. Suppose this property holds at some point in time, and we add a job j' to S. Let j,k,l,k',l',i be as in the description of S. Since $k \le k'$, and j is valid for i, $\mathcal{N}$ already contains the intervals $I_{i'}(k,l)$ for all $i' \le i$. Hence, the intervals $I_{i'}(k',l')$, $i' \le i$, cannot be maximal. Suppose an interval $I_{i'}(k',l')$ is maximal, where $i' > i$, and j' is valid for i' (so this interval gets added to $\mathcal{N}$). Now, our algorithm would have considered scheduling j' on i' before going to i — so it must be the case that all but $p_{j'}s_{i'}$ slots in $I_{i'}(k',l')$ are busy processing jobs of class at least k'. Further, all the jobs being processed on these slots will get added to S (by definition of S, and the fact that $j' \in S$). The lemma now follows because $p_{j'}s_{i'} \le \frac{T}{2^{k'}} \le \varepsilon|I(k',l')|$, and $\mathcal{A}$ can do $(1+2\varepsilon)|I(k,l)|$ amount of processing during $I(k,l)$. □

Corollary 1. *The total volume of jobs in S is greater than $\sum_{I(k,l)\in\mathcal{N}'}(1+\varepsilon)|I(k,l)|$.*

Proof. Lemma 1 shows that given any maximal interval $I_i(k,l)$, $\mathcal{A}$ processes jobs of S for at least $\frac{1+\varepsilon}{1+2\varepsilon}$-fraction of the slots in it. The total volume that it can process in $I(k,l)$ is $(1+2\varepsilon)|I(k,l)|$. The result follows because maximal intervals are disjoint (we have strict inequality because $\mathcal{A}$ could not complete j^*). □

We now show that the total volume of jobs in S cannot be too large, which leads to a contradiction.

Lemma 2. *If $\texttt{opt}(\mathcal{I}) \le T$, then the total volume of jobs in S is at most $\sum_{I(k,l)\in\mathcal{N}'}(1+\varepsilon)|I(k,l)|$.*

Proof. Suppose $\texttt{opt}(\mathcal{I}) \le T$. For an interval $I_i(k,l)$, let $I_i^\varepsilon(k,l)$ be the interval of length $(1+\varepsilon)|I_i(k,l)|$ which starts at the same time as $I(k,l)$. It is easy to check that if $I(k',l') \subseteq I(k,l)$, then $I^\varepsilon(k',l') \subseteq I^\varepsilon(k,l)$.

Let $j \in S$ be a job of type (k,l). The off-line optimal solution must schedule it within $\frac{T}{2^k}$ of its release date. Since $r_j \in I(k,l)$, the optimal solution must process a job j during $I^\varepsilon(k,l)$. So, the total volume of jobs in S can be at most

$$\left|\bigcup_{I(k,l)\in\mathcal{N}} I^\varepsilon(k,l)\right| = \left|\bigcup_{I(k,l)\in\mathcal{N}'} I^\varepsilon(k,l)\right| \le \sum_{I(k,l)\in\mathcal{N}'} |I^\varepsilon(k,l)| = \sum_{I(k,l)\in\mathcal{N}'} (1+\varepsilon)|I(k,l)|.$$

□

Clearly, Corollary 1 contradicts Lemma 2. So, algorithm $\mathcal{A}$ must be able to process all the jobs.

2.2 Off-line to on-line

Now, we give an on-line algorithm for the instance $\mathcal{I}$. Recall that $\mathcal{A}$ is an off-line algorithm for $\mathcal{I}$ and may not even respect release dates. The on-line algorithm $\mathcal{B}$ is a non-migratory algorithm which maintains a queue for each machine i and time t. For each job j, it uses $\mathcal{A}$ to figure out which machine the job j gets dispatched to.

Note that the algorithm $\mathcal{A}$ can be implemented in a manner such that for any job j of type (k, l), the slots assigned by $\mathcal{A}$ to j are known by the end of interval $I(k, l)$ — jobs which get released after $I(k, l)$ do not affect the schedule of j. Also note that the release date of j falls in $I(k, l)$. This is described more formally in Figure 2.

```
Algorithm $\mathcal{A}(\mathcal{I}, T)$:

For $t = 0, 1, 2, \ldots$
  For $k = 1, 2, \ldots$
    If $t$ is the end-point of an interval $I(k, l)$ for some $l$, then
      For each job $j$ of type $(k, l)$
        For $i = m_j$ downto 1 ($m_j$ is the slowest machine on which $j$ is valid)
          If there are at least $p_j s_i$ free slots on machine $i$ during $I(k, l)$ then
            schedule $j$ on $i$ during the first such free slots (without caring about $r_j$).
```

Fig. 2. An alternate implementation of $\mathcal{A}$

We now describe the algorithm $\mathcal{B}$. It maintains a queue of jobs for each machine. For each job j of class k and releasing during $I(k, l)$, if j gets processed on machine i by $\mathcal{A}$, then $\mathcal{B}$ adds j to the queue of i at end of $I(k, l)$. Observe that $\mathcal{B}$ respects release dates of jobs — a job j of type (k, l) has release date in $I(k, l)$, but it gets dispatched to a machine at the end of the interval $I(k, l)$. For each machine i, $\mathcal{B}$ prefers jobs of higher class, and within a particular class, it follows the ordering given by $\mathcal{A}$ (or it could just go by release dates). Further, we give machines in $\mathcal{B}$ $(1 + 3\varepsilon)$-speedup.

Analysis We now analyze $\mathcal{B}$. For a class k, let $J_{\geq k}$ be the jobs of class at least k. For a class k, integer l and machine i, let $Q(i, k, l)$ denote the jobs of $J_{\geq k}$ which are in the queue of machine i at the beginning of $I(k, l)$. First we note some properties of $\mathcal{B}$:

(i) A job j gets scheduled in $\mathcal{B}$ only in later slots than those it was scheduled on by $\mathcal{A}$: A job j of type (k, l) gets scheduled during $I(k, l)$ in $\mathcal{A}$. However, it gets added to the queue of a machine by $\mathcal{B}$ only at the end of $I(k, l)$.

(ii) For a class k, integer l and machine i, the total remaining processing time (on the machine i) of jobs in $Q(i, k, l)$ is at most $\frac{(1+2\varepsilon)T}{\varepsilon 2^k}$: Suppose this is true for some i, k, l. We want to show that this holds for $i, k, l + 1$ as well. The jobs in

the queue $Q(i,k,l+1)$ could consist of either (i) the jobs in $Q(i,k,l)$, or (ii) the jobs of $J_{\geq k}$ which get processed by $\mathcal{A}$ during $I_i(k,l)$. Indeed, jobs of $J_{\geq k}$ which get released before the the interval $I_i(k,l)$ finish before this interval begins (in $\mathcal{A}$). Hence, in $\mathcal{B}$, any such job would either finish before $I(k,l)$ begins, or will be in the queue $Q(i,k,l)$. The jobs of $J_{\geq k}$ which get released during $I(k,l)$ will complete processing in this interval (in $\mathcal{A}$) and hence may get added to the queue $Q(i,k,l+1)$.
Now, the total processing time of the jobs in (ii) above would be at most $(1+2\varepsilon)|I(k,l)|$ (recall that the machines in $\mathcal{A}$ have speedup of $(1+2\varepsilon)$). Suppose in the schedule $\mathcal{B}$, the machine i processes a job of class greater than k during some time in $I_i(k,l)$ — then it must have finished processing all the jobs in $Q(i,k,l)$, and so $Q(i,k,l+1)$ can only contain jobs from (ii) above, and hence, their total processing time is at most $(1+2\varepsilon)|I(k,l)|$ and we are done. If the machine i is busy during $I_i(k,l)$ processing jobs from $J_{\geq k}$ (in $\mathcal{B}$), then it does at least $(1+2\varepsilon)|I(k,l)|$ amount of processing , and so, the property holds at the end of $I(k,l)$ as well.

We are now ready to prove the main theorem.

Theorem 1. *In the schedule $\mathcal{B}$, a job j of class k has flow-time at most $\frac{T(1+3\varepsilon)}{\varepsilon^2 2^k}$. Hence, for any instance, $\mathcal{B}$ is an $\left(\frac{2(1+3\varepsilon)}{\varepsilon^2}\right)$-competitive algorithm with $(1+3\varepsilon)$-speedup.*

Proof. Consider a job j of class type (k,l). Suppose it gets processed on machine i. The algorithm $\mathcal{B}$ adds j to the queue $Q(i,k,l)$. Property (ii) above implies that the total remaining processing time of these jobs (on i) is at most $(1+2\varepsilon)|I(k,l)|$. Consider an interval I which starts at the beginning of $I(k,l)$ and has length $\frac{(1+2\varepsilon)|I(k,l)|}{\varepsilon} = \frac{(1+2\varepsilon)T}{\varepsilon^2 2^k}$. The jobs of $J_{\geq k}$ that $\mathcal{B}$ can process on i during I are either (i) jobs in $Q(i,k,l)$, or (ii) jobs processed by $\mathcal{A}$ on machine i during I (using property (i) above). The total processing time of the jobs in (ii) is at most $(1+2\varepsilon)|I|$, whereas $\mathcal{B}$ can process $(1+3\varepsilon)|I|$ volume during I (on machine i). This still leaves us with $\varepsilon|I| = \frac{(1+2\varepsilon)T}{\varepsilon 2^k}$ — this is enough to process all the jobs in $Q(i,k,l)$. So the flow-time of j is at most $|I| + |I(k,l)| = \frac{T}{2^k}\left(\frac{1}{\varepsilon} + \frac{1+2\varepsilon}{\varepsilon^2}\right)$. Finally, given any instance, we lose an extra factor of 2 in the competitive ratio because we scale all weights to powers of 2. □

Extensions We mention some easy extensions of the result above.

Comparison with migratory off-line optimum: Here, we allow the off-line optimum to migrate jobs across machines. To deal with this, we modify the definition of when a job is valid on a machine. We will say that a job j of class k is valid for a machine i if its processing time on i is at most $\frac{T}{2^k} \cdot \frac{1+\varepsilon}{\varepsilon}$. Note that even a migratory algorithm will process at most $\frac{\varepsilon}{1+\varepsilon}$-fraction of a job on machines which are not valid for it. Further, we modify the definition of $I(l,k)$ to be $\left[\frac{(1+\varepsilon)lT}{\varepsilon^2 2^k}, \frac{(1+\varepsilon)(l+1)T}{\varepsilon^2 2^k}\right)$. The rest of the analysis can be carried out as above. We can show that the on-line algorithm is $O\left(\frac{(1+\varepsilon)^2}{\varepsilon^3}\right)$-competitive with $(1+\varepsilon)$-speedup.

Deadline scheduling on related machines: In this setting, the input instance also comes with deadline d_j for each job j. The assumption is that there is a schedule (off-line) which can schedule all jobs (with migration) such that each job finishes before its deadline. The question is: is there a constant s and an on-line algorithm $\mathcal{S}$ such that with speedup s, it can meet all the deadlines? Using the above result, it is easy to show that our online algorithm has this property provided we give it constant speedup. We give the proof in the appendix.

Corollary 2. *There is a constant s, and a non-migratory scheduling algorithm which, given any instance of the deadline scheduling problem, completes all the jobs within their deadline if we give speed-up of c to all the machines.*

So far our on-line algorithm has assumed that we know the optimal value of an instance. In the appendix B, we show how to get rid of this assumption.

3 Max-Flow-time on Unrelated Machines

We consider the (unweighted) Max-Flow-time on unrelated machines. We first show that a constant competitive algorithm cannot have the property of immediate dispatch and it requires speed augmentation. Since our instances use unit-sized jobs, the lower bound also holds for Max-Stretch. Recall that a scheduling algorithm is called *immediate dispatch* if it decides, at the time of a job's arrival, which machine to schedule the job on.

The lower bound for an immediate dispatch algorithm follows from the lower bound of Azar et al. [4] for minimizing total load in the subset parallel settings. Here, we are given a set of machines, and jobs arrive in a sequence. Each job specifies a subset of machines it can go to, and the on-line algorithm needs to dispatch a job on its arrival to one such machine. The goal is to minimize the maximum number of jobs which get dispatched to a machine. Azar et al. [4] prove that any randomized on-line algorithm for this problem is $\Omega(\log m)$-competitive. From this result, we can easily deduce the following lower bound for Max-Flow-time in the subset parallel setting with unit size jobs (given an instance of the load balancing problem, give each job size of 1 unit, and make them arrive at time 0 in the same sequence as in this given instance).

Theorem 2. *Any immediate dispatch randomized on-line algorithm for* Max-Flow-time *in the subset parallel setting with unit job sizes must have competitive ratio of $\Omega(\log m)$.*

Any randomized on-line algorithm with bounded competitive ratio needs speed augmentation. We give the proof in the appendix.

Theorem 3. *Any online algorithm for minimizing* Max-Flow-time *on subset-parallel machines which allows non-immediate dispatch but does not allow speed augmentation has a competitive ratio of $\Omega(m)$. This holds even for unit-sized jobs.*

3.1 A $(1+\varepsilon, O(1/\varepsilon))$-competitive algorithm

We now describe an $\left(\frac{2}{\varepsilon}\right)$-competitive algorithm for Max-Flow-time on multiple unrelated machines with $(1+\varepsilon)$-speed augmentation. The algorithm proceeds in several phases: denote these by $\Pi_1, \Pi_2, \ldots$, where phase Π_i begins at time t_{i-1} and ends at time t_i. In phase Π_i, we will schedule all jobs released during the phase Π_{i-1}.

In the initial phase, Π_1, we consider the jobs released at time $t_0 = 0$, and find an optimal schedule which minimizes the makespan of jobs released at time t_0. This phase ends at the time we finish processing all these jobs. Now, suppose we have defined $\Pi_1, \ldots, \Pi_l$, and have scheduled jobs released during $\Pi_1, \ldots, \Pi_{l-1}$. We consider the jobs released during Π_l, and starting from time t_l, we find a schedule which minimized their makespan (assuming all of these jobs are released at time t_l). Again, this phase ends at the time we finish processing all these jobs. Note that this algorithm is a non-immediate dispatch algorithm and does not require migration. We now prove that this algorithm has the desired properties.

Theorem 4. *Assuming $\varepsilon \le 1$, The algorithm described above has competitive ratio $\frac{2}{\varepsilon}$ with $(1+\varepsilon)$-speed augmentation.*

Proof. Consider an instance $\mathcal{I}$ and assume that the optimal off-line schedule has maximum flow time of T. We will be done if we show that each of the phases Π_i has length at most $\frac{T}{\varepsilon}$. For Π_1, this is true because all the jobs released at time 0 can be scheduled within T units of time. Suppose this is true for phase Π_i. Now, we know that the jobs released during Π_i can be scheduled in an interval of length $\Pi_i + T$. Using $(1+\varepsilon)$-speed augmentation, the length of the next phase is at most

$$\frac{|\Pi_i| + T}{1+\varepsilon} \le \frac{T/\varepsilon + T}{1+\varepsilon} = \frac{T}{\varepsilon}. \qquad \square$$

4 Max-Weighted-Flow-time on Unrelated Machines

In this section, given any constant speedup, any on-line algorithm for Max-Weighted-Flow-time on unrelated machines is $\Omega(\log m)$-competitive. This bound holds for the special case of subset parallel model, and even extends to the Max-Stretch metric. We give the proof of the following theorem in the appendix.

Theorem 5. *Given any large enough parameter c, integer $s \ge 1$, and an on-line algorithm $\mathcal{A}$ which is allowed speedup of $(s+1)/2$, there exists an instance $\mathcal{I}(s,c)$ of* Max-Weighted-Flow-time *on subset parallel machines such that $\mathcal{A}$ is not c-competitive on $\mathcal{I}(s,c)$. The instance $\mathcal{I}(s,c)$ has jobs with s different weights only, and uses $(O(s))^{O(cs^2)}$ machines.*

5 Lower bound for L_p norm of stretch

We show a lower bound for the competitive ratio for the L_p-norm of the stretches, with speed augmentation by a factor of $1+\varepsilon$. We assume that there is an online algorithm with competitive ratio $c = o(\frac{p}{\varepsilon^{1-3/p}})$ and derive a contradiction.

The construction uses $m = 2^p$ machines. We start with the typical construction to get a large load on one machine. For this we consider 2 machines. At time 0 we release two jobs of size 1 (and weight 1) - each can go on exactly one machine. Then until time 1 we release tiny jobs, i.e., at each δ time step a job of size δ (and weight $1/\delta$) is released that can go on any of the two machines. Note that at time 1 at least one of the machines has load (of size 1 jobs) at least $1/2 - \varepsilon - c\delta$. This is because, the total volume of jobs is 3, the two machines can process at most $2(1+\varepsilon)$ units, and all tiny jobs except the last c have to be processed. It makes sense to set $\delta = \varepsilon/c$ and hence $c\delta \leq \varepsilon$.

Now, we can use this as a gadget, starting with $m/2$ pairs of machines we then take the $m/2$ machines with large load and pair them up arbitrarily and recursively do the same construction. We end up with one machine with load $\Omega(\log m)$ (if ε is sufficiently smaller than 1/2). This concludes the first of two phases.

Now that we have a machine with large load, we release tiny jobs for a time interval of length $\log(m)/\varepsilon$. Since the tiny jobs have to be processed first, the initial load of $\Omega(\log m)$ needs time $\Omega(\log(m)/\varepsilon)$ to be fully processed, as it can be processed only in the time that we have additional due to resource augmentation. Hence, at least one size 1 job has stretch at least $\Omega(\log(m)/\varepsilon)$. This concludes the second phase.

Let us bound the number of jobs k that we release in these 2 phases. In the first phase of the construction we release $m + m/2 + m/4 + ... = O(m)$ jobs of size 1 and $O(m/\delta)$ tiny jobs. In the second phase we release $O(\log(m)/(\varepsilon\delta))$ tiny jobs. Thus, $k = O(m/\delta + \log(m)/(\varepsilon\delta))$. Note that we can bound $1/\delta \leq p/\varepsilon^{2-3/p}$ and hence $k = O(mp/\varepsilon^{3-3/p})$.

We want to repeat these two phases n/k times. After the first 2 phases have been completed (by the optimal offline algorithm) we release again the 2 phases, and we repeat this n/k times. Thus, for the optimal offline algorithm all repetitions will be independent. Then in total we released any desired number n of jobs, where $n \geq k$.

Note that the optimal offline algorithm would have a max-stretch of 2 and, thus, also an L_p norm of the stretches of $\left(\frac{1}{n}\sum_i v_i^p\right)^{1/p} \leq 2$.

We now lower bound the L_p norm of the stretches of the online algorithm. We already have a lower bound on the maximal stretch of any job, $\Omega(\log(m)/\varepsilon)$, and we know that there are at least n/k jobs with such a large stretch, one for each repetition of the 2 phases. Now, let v_i be the stretch of the i-th job. Then the L_p norm of the stretches is

$$c \geq \Omega\left(\frac{1}{n}\sum_i v_i^p\right)^{1/p}$$

Since we know that there are n/k jobs with $v_i = \Omega(\log(m)/\varepsilon)$ this is at least

$$c \geq \Omega\left(\frac{\log(m)}{\varepsilon}\left(\frac{n/k}{n}\right)^{1/p}\right) = \Omega\left(\frac{\log(m)}{\varepsilon k^{1/p}}\right).$$

Plugging in our bound on $k = O(mp/\varepsilon^{3-3/p})$ this yields a bound of

$$c \geq \Omega\left(\frac{\log(m)}{\varepsilon(mp)^{1/p}/\varepsilon^{3/p}}\right).$$

Since $m = 2^p$ and noting that $p^{1/p} = O(1)$ this yields the desired contradiction to c begin too small,

$$c \geq \Omega\left(\frac{p}{\varepsilon^{1-3/p}}\right).$$

The only condition for this was

$$n \geq k = \frac{2^{\Theta(p)}}{\varepsilon^{\Theta(1)}}.$$

which implies that n just has to be sufficiently large.

Bibliography

[1] Christoph Ambühl and Monaldo Mastrolilli. On-line scheduling to minimize max flow time: An optimal preemptive algorithm. *Oper. Res. Lett.*, 33(6):597–602, 2005.

[2] S. Anand, Naveen Garg, and Nicole Megow. Meeting deadlines: How much speed suffices? In *38th Intl. Coll. Automata, Languages and Programming (ICALP)*, pages 232–243, 2011.

[3] S. Anand, Naveen Garg, and Amit Kumar. Resource augmentation for weighted flow-time explained by dual fitting. In *23rd Symp. Discrete Algorithms (SODA)*, pages 1228–1241, 2012.

[4] Yossi Azar, Joseph Naor, and Raphael Rom. The competitiveness of on-line assignments. *J. Algorithms*, 18(2):221–237, 1995.

[5] Nikhil Bansal and Kirk Pruhs. Server scheduling in the ℓ_p norm: A rising tide lifts all boats. In *35th Symp. Theory of Computing (STOC)*, pages 242–250, 2003.

[6] Nikhil Bansal and Kirk Pruhs. Server scheduling in the weighted ℓ_p norm. In *6th Latin American Theoretical Informatics Conference (LATIN)*, pages 434–443, 2004.

[7] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *9th Symp. Discrete Algorithms (SODA)*, pages 270–279, 1998.

[8] Michael A. Bender, S. Muthukrishnan, and Rajmohan Rajaraman. Improved algorithms for stretch scheduling. In *13th Symp. Discrete Algorithms (SODA)*, pages 762–771, 2002.

[9] Chandra Chekuri and Benjamin Moseley. Online scheduling to minimize the maximum delay factor. In *20th Symp. Discrete Algorithms (SODA)*, pages 1116–1125, 2009.

[10] Daniel Golovin, Anupam Gupta, Amit Kumar, and Kanat Tangwongsan. All-norms and all-ℓ_p-norms approximation algorithms. In *28th Conf. Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 199–210, 2008.

[11] Sungjin Im and Benjamin Moseley. An online scalable algorithm for minimizing ℓ_k-norms of weighted flow time on unrelated machines. In *22nd Symp. Discrete Algorithms (SODA)*, pages 95–108, 2011.

[12] Cynthia A. Phillips, Clifford Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002.

A Proof of Corollary 2

Proof. As argued above, there is an algorithm for Max-Weighted-Flow-time with competitive ratio $\frac{c(1+\varepsilon)^2}{\varepsilon^3}$ if we give speedup of $(1+\varepsilon)$ to the machines, where c is a constant. Note that, here ε can be any positive number, and so, if we pick ε to be a large constant, then this ratio becomes less than 1, i.e., the weighted flow-time of each job is even better than the optimal value T. Further, note that there is no assumption of the weights of the jobs – they need not be power of 2. The fact that we rounded them to power of 2 worsens the competitive ratio by a factor of 2, which is getting absorbed in the constant c. We pick s to be $(1+\varepsilon)$, where ε is such that $\frac{c(1+\varepsilon)^2}{\varepsilon^3} < 1$.

Now consider an instance $\mathcal{I}$ of the deadline scheduling problem. We map this to an instance $\mathcal{I}'$ of the Max-Weighted-Flow-time problem where we *know* that the optimal value T is at most 1. The mapping is as follows. When a job j with deadline d_j arrives at time r_j in $\mathcal{I}$, we release j at time r_j in $\mathcal{I}'$ as well (the processing time of j is $\mathcal{I}'$ is same as that in $\mathcal{I}$). Further, we set w_j to be $\frac{1}{d_j - r_j}$ in $\mathcal{I}'$. We claim that the optimal value for $\mathcal{I}'$ is at most 1. Indeed, there is a schedule which finishes each job j by time d_j, and so, its weighted flow-time is at most 1. Now, our on-line algorithm with speedup s will also have objective value of 1, i.e., each job will now finish by its deadline d_j. □

B Removing the assumption about knowledge of T

In this section, we show how to remove the assumption about knowledge of T. Again, we will construct an off-line algorithm $\mathcal{C}$, which will invoke $\mathcal{A}$ for different guesses for T. We begin with some definitions. We fix an instance $\mathcal{I}$. For a parameter T, let $I^{(T)}(k,l)$ be the interval $\left[\frac{lT}{\varepsilon 2^k}, \frac{(l+1)T}{\varepsilon 2^k}\right)$ (this is same as $I(k,l)$ defined in Section 2.1). Similarly, we say that a job of class k is of type $(k,l)_T$ if $r_j \in I^{(T)}(k,l)$.

Our algorithm will work with guess of T which are powers of $C = \frac{1+\varepsilon}{\varepsilon}$. Assume that all release dates and processing times are integers so that the optimum value is at least 1. Let T_u denote C^u. We first slightly generalize the algorithm $\mathcal{A}$ described in Figure 1. The new algorithm $\mathcal{A}'$ will take as parameters an instance $\mathcal{I}'$, guess T, and a starting time t_0 — all release dates in $\mathcal{I}'$ will be at least t_0. It will run $\mathcal{A}(\mathcal{I}', T)$ with the understanding that time starts at t_0. Also it will run the machines at speed $(1+3\varepsilon)$. So the interval $I^{(T)}(k,l)$ will be defined as $\left[t_0 + \frac{lT}{\varepsilon 2^k}, t_0 + \frac{(l+1)T}{\varepsilon 2^k}\right)$. With these definitions, we ready to describe our new off-line algorithm. The algorithm is described in Figure 3.

We first show that the algorithm $\mathcal{C}$ is constant competitive. Suppose during iteration u of Step 2 in the algorithm $\mathcal{C}(\mathcal{I})$, we find a job $j^\star$ as in Step 2(iii), where $j^\star$ is of type $(k^\star, l^\star)_{T_u}$. Recall that t_{u+1} is the end-point of $I^{(T_u)}(k^\star, l^\star)$. For a job $j \in \mathcal{I}_u$, let r_j^u denote its release date in the instance $\mathcal{I}_u$.

Lemma 3. *Any job $j \in \mathcal{I}_{u+1}$ with $r_j^u < t_{u+1}$ must be of class at most $k^\star$. Further, if such a job is of class k, then $t_{u+1} - r_j \leq \frac{T_{u+1}}{2^k}$.*

```
Algorithm $\mathcal{C}(\mathcal{I})$:

1. Initialize $T_0 = 1, t_0 = 0, \mathcal{I}_0 = \mathcal{I}$.
2. For $u = 0, 1, 2, \ldots$
   (i) Run $\mathcal{A}'(\mathcal{I}_u, T_u, t_u)$ as described above.
   (ii) If we are able to finish all jobs, then stop and output the schedule produced.
   (iii) Else let $j$ be the first job which the algorithm $\mathcal{A}'(\mathcal{I}_u, T_u, t_u)$ is not able to schedule.
         Suppose $j$ is of type $(k, l)_{T_u}$. Define $t_{u+1}$ as the end-point of $I^{(T_u)}(k, l)$.
         Define $\mathcal{I}_{u+1}$ as the jobs in $\mathcal{I}_u$ which are not scheduled yet.
         Define release date of a job $j \in \mathcal{I}_{u+1}$ as $\max(t_{u+1}, r_j)$. Set $T_{u+1} = T_u \cdot \frac{1+\varepsilon}{\varepsilon}$.
         Go to the next iteration.
```

Fig. 3. The off-line algorithm which schedules jobs in instance $\mathcal{I}$

Proof. Suppose $j \in \mathcal{I}_u$ and $r_j^u < t_{u+1}$. If j is of type $(k, l)_{T_u}$, where $k > k^\star$, then $I^{(T_u)}(k, l) \subseteq I^{(T_u)}(k^\star, l^\star)$, and so, the interval $I^{(T_u)}(k, l)$ ends on or before t_{u+1}. So $An(\mathcal{I}_u, T_u, t_u)$ would have considered j before $j^\star$. By definition of $j^\star$, the algorithm must have scheduled j in $I^{(T_u)}(k, l)$, and so, before t_{u+1}. This proves the first statement in the lemma. We now prove the second statement in the lemma. We use induction on u. Suppose the statement is true for iteration $u - 1$. We now show that it is true for u. Let j be a job of class $k' \leq k$ such that $j \in \mathcal{I}_{u+1}$ and $r_j^u < t_{u+1}$. Then j of type $(k, l)_{T_u}$, where the interval $I^{(T_u)}(k, l)$ ends on or after t_{u+1}. So, $t_{u+1} - r_j^u \leq |I^{(T_u)}(k, l)| = \frac{T_u}{\varepsilon 2^k}$. If $r_j \geq t_u$, then $r_j^u = r_j$, and we are done. Otherwise, $r_j^u = t_u$. So we get $t_{u+1} - t_u \leq \frac{T_u}{\varepsilon 2^k}$. By induction hypothesis, $t_u - r_j \leq \frac{(1+\varepsilon)T_u}{2^k}$. So,

$$t_{u+1} - r_j \leq \frac{T_u}{\varepsilon 2^k} + \frac{T_u}{2^k} = \frac{T_{u+1}}{\varepsilon 2^k}.$$ □

Now, we show that if $\mathcal{C}$ is not able to process all jobs in iteration u, then the $\texttt{opt}(\mathcal{I})$ must be at least T_u.

Lemma 4. *If during iteration u, $\mathcal{C}$ does not finish all jobs, then* $\texttt{opt}(\mathcal{I}) \geq T_u$.

Proof. The proof is similar to the proof in Section 2.1, so we sketch the main ideas only. The set S is defined as in the section (with respect to the input $\mathcal{I}_u$). Proofs of Lemma 1 and Corollary 1 remain unchanged. However, machines in $\mathcal{A}'$ have $(1 + 3\varepsilon)$-speedup. So, we get that the total volume of jobs in S is more than $\sum_{I^{(T_u)}(k,l) \in \mathcal{N}'} (1 + 2\varepsilon)|I^{(T_u)}(k, l)|$.

We get a contradiction by showing that if $\texttt{opt}(\mathcal{I}_u) \leq T_u$, then the total volume of jobs in S is at most $\sum_{I^{(T_u)}(k,l) \in \mathcal{N}'} (1 + 2\varepsilon)|I^{(T_u)}(k, l)|$. The proof is similar to that of Lemma 2. The only catch is that for a job j of type $(k, l)_{T_u}$, r_j may not even lie in $I^{(T_u)}(k, l)$. So, the optimum algorithm may process j even before this interval. But Lemma 3 shows that r_j may lie at most $\varepsilon|I^{(T_u)}(k, l)|$ to the left of $I^{(T_u)}(k, l)$. So, we define the intervals $I^{\varepsilon,(T_u)}(k, l)$ which attach two segments of length $\varepsilon|I^{(T_u)}(k, l)|$ both before and after $I^{(T_u)}(k, l)$. Rest of the arguments proceed as in the proof of Lemma 2. □

Theorem 6. *Suppose* $\texttt{opt}(\mathcal{I})$ *lies between* T_{u-1} *and* T_u. *Then the algorithm* $\mathcal{C}$ *completes a job of class* k *within* $\frac{(1+\varepsilon)T_u}{\varepsilon 2^k}$ *of its processing time. Further, the schedule for* k *depends only on jobs released till time* $r_j + \frac{(1+\varepsilon)T_u}{\varepsilon 2^k}$.

Proof. Lemma 4 implies that $\mathcal{A}$ must finish in iteration u. So, each job of class k terminates in $I^{(T_{u'})}(k,l)$ for some $u' \leq u$. Lemma 3 now implies that it completes within $\frac{T_{u'}}{2^k} + \frac{T_{u'}}{\varepsilon 2^k}$ of its release date. The second statement in the theorem is also easy to see. □

We now describe the on-line algorithm. The on-line algorithm $\mathcal{D}(\mathcal{I})$ runs $\mathcal{C}(\mathcal{I})$. Let T_u be as in Theorem 6. The theorem implies that for any job j, we will know the machine on which it will get scheduled by time $r_j + \frac{(1+\varepsilon)T_u}{\varepsilon 2^k}$. At this time, we place j on the queue of the machine to which it gets scheduled on by $\mathcal{C}$. We give machines in $\mathcal{D}$ speedup of $(1+4\varepsilon)$. Further, each machine follows the following rule: it prefers jobs of larger class, and within a particular class, it just goes by release date. The following claim shows that the queues do not get big.

Claim. At time $\frac{2lT_u}{\varepsilon 2^k}$, for any integer l, the total remaining processing time of jobs of $J_{\geq}k$ in the queue of machine i is at most $\frac{T_u}{\varepsilon 2^k}$.

Proof. We prove this by induction on l. For ease of notation, let t_l denote $\frac{2lT_u}{\varepsilon 2^k}$. Suppose it is true for some l. Now, the queue on i at time t_l from $J_{\geq k}$ could be (i) jobs which are completely processed by $\mathcal{C}$ during $[t_l, t_{l+1}]$, which have processing time $\frac{(1+3\varepsilon)T_u}{\varepsilon 2^{k-1}}$ on machine i, (ii) jobs in the queue of i at time t_l, which have remaining processing time of $\frac{T_u}{\varepsilon 2^k}$ (by induction hypothesis), and (iii) jobs which were partially processed by $\mathcal{C}$ by time t_l: there will be at most 1 such job from each class, and so their total processing tim will be at most $\frac{T_u}{2^{k-1}}$. The result now follows because $\mathcal{D}$ can do $\frac{(1+4\varepsilon)T_u}{\varepsilon 2^{k-1}}$ amount of processing during $[t_l, t_{l+1}]$. □

The proof of the following theorem is analogous to Theorem 1.

Theorem 7. *The algorithm* $\mathcal{D}$ *completes a job of class* k *within* $\frac{(3+\varepsilon)T_u}{\varepsilon^2 2^k}$ *of its release date. Hence,* $\mathcal{D}$ *is* $\frac{(3+\varepsilon)(1+\varepsilon)}{\varepsilon^3}$*-competitive with* $(1+4\varepsilon)$*-speed augmentation.*

C Proof of Theorem 3

Proof. Let the machines be numbered from 1 to m. Consider an online algorithm $\mathcal{A}$. We will use the decisions made by $\mathcal{A}$ to build an instance $\mathcal{I}$ on which $\mathcal{A}$ would have a maximum flow time $m-1$ while the optimum offline algorithm will have value 2. Our construction involves defining a gadget $G_i(t)$ as follows

(i) At time t, a job is released which can be scheduled on machine i or $i+1$ only.
(ii) For all times $t, t+1 \ldots, t+m-1$, two jobs are released one of which can go only on machine i and the other only on machine $i+1$.
(iii) At time $t+m$, we release a job which can go only to the machine on which $\mathcal{A}$ schedules the job released in step 1. Note that $\mathcal{A}$ must have scheduled the job by time $t+m-1$ or else it would have a flow time more than m.

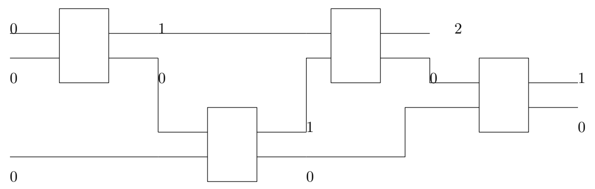

Fig. 4. Composing gadgets to increase load

The following properties of $G_i(t)$ are immediate from the construction

(i) Jobs are released from time t to $t + m$.
(ii) An offline algorithm which had no unfinished jobs on machines $i, i+1$ at time t can schedule all jobs released in $G_i(t)$ within 2 time units of their release. Further, the offline algorithm would have no unfinished jobs at time $t+m+1$.
(iii) Suppose $\mathcal{A}$ has a unfinished jobs on machine i and b unfinished jobs on machine $i+1$ at time t. Then at time $t+m+1$, machine i (respectively $i+1$) has either $a+1$ (respectively $\max(0, b-1)$) or $\max(0, a-1)$ (respectively $b+1$) unfinished jobs.

Note that if a machine i has a unfinished jobs at time t in $\mathcal{A}$, then we can ensure that it continues to have a unfinished jobs at time $t' > t$ by releasing a job which can be assigned only to machine i at each time instant from t to $t' - 1$. This idea is used while composing gadgets to create an instance for which some job has a large flow time in $\mathcal{A}$.

We shall use the following statement by induction on the number of machines: given k machines numbered $1, \ldots, k$, there is an instance such that time a certain time t_k, for every $i, 0 \leq i \leq k-1$, there is a machine with i unfinished jobs in $\mathcal{A}$. For the base case ($k = 2$), we only need the gadget $G_1(0)$ and t_2 is then $m + 1$. Now assume that the statement is true for k machines, and we will prove it for $k + 1$ machines.

Using induction hypothesis, and relabeling of machines, we assume that at time t_k the machine i has $k - i$ unfinished jobs in $\mathcal{A}$, for $1 \leq i \leq k$. Note that machine $k+1$ has 0 unfinished jobs at time t_k. The gadget $G_k(t_k)$, which releases jobs for machines $k, k+1$ in the interval $[t_k, t_k + m]$, ensures that one of the machines $k, k+1$ has one unfinished job. There is no loss of generality in assuming that the number of unfinished jobs on the lower numbered machine increases by 1. With this assumption, we create gadgets $G_i(t_k + (m+1)(k-i))$ which ensure that at time $t_k + (m+1)(k-i+1)$, machine i has $k-i+1$ unfinished jobs. Thus at time $t_k + (m+1)k = t_k^1$, machine 1 has k unfinished jobs in $\mathcal{A}$ (see Figure 4).

However, since machine i is part of gadgets $G_i(\cdot)$ and $G_{i-1}(\cdot)$, the number of unfinished jobs on machine i at time t_k^1 is the same as that at time t_k. This implies that while machine 1 has k unfinished jobs, machines $2, 3, \ldots, k+1$ have one less unfinished job than desired. To correct this, we repeat the construction on machines $2, \ldots, k+1$ from time t_k^1 to time $t_k^2 = t_k^1 + (m+1)(k-1)$ and on machines $3, \ldots, k+1$ from time t_k^2 to time $t_k^3 = t_k^2 + (m+1)(k-2)$ and so on.

Hence at time $t_k^{k+1} = t_k + (m+1)(k+1)k/2 = t_{k+1}$, algorithm $\mathcal{A}$ would have $k+1-i$ unfinished jobs on machine i.

To complete the proof of Theorem 3, note that at time t_m, $\mathcal{A}$ would have $m-1$ unfinished jobs on machine 1, which implies that some job would have a flow time of $m-1$. Further, the composition of these gadgets and the release of the intermediate jobs does not increase the maximum flow time of the off-line optimum. □

D Proof of Theorem 5

Proof. We will prove a stronger statement: given s and c above, and an on-line algorithm $\mathcal{A}$ (depending on s and c), we will construct an instance $\mathcal{I}(s,c)$, such that the value of the optimal off-line solution will be 2, whereas the objective value of $\mathcal{A}$ will be at least $2c$ even if each of the machines has *average* speed of $(s+1)/2$ during the time period 0 to $T(s,c)$. Here, $T(s,c)$ is the time by which any c-competitive algorithm must finish all jobs in $\mathcal{I}(s,c)$, i.e., $\max_j(r_j + 2cw_j)$, because the off-line optimum value will be 2.

We will prove this theorem by induction on s. We first show the base case for $s=2$, i.e., each machine is allowed average speedup of $3/2$. Since c will remain fixed throughout the proof, we will not parameterize various quantities by c.

Base Case: For the sake of contradiction, assume that $\mathcal{A}$ is c-competitive even when we give each of the machines average speedup of $3/2$ on instance $\mathcal{I}(s,c)$ described below. We have two kinds of jobs: a type 0 job has weight $8c$ and size $\frac{1}{8c}$, and a type 1 job has weight and size both 1. We first describe a gadget $G(t)$: here t denotes the starting time for this gadget. The gadget $G(t)$ has 6 machines. At time t we release 6 type 1 jobs — each of these jobs can go on exactly one of the 6 machines. Further, during $(t, t+1)$ we release 5 type 0 jobs after every $\frac{1}{8c}$ time. This completes the description of the gadget.

Before we give the actual construction, we note a useful property of the gadget. Let the machines in $G(t)$ be numbered from 1 to 6.

Claim. Consider any on-line algorithm $\mathcal{B}$ which incurs weighted flow-time of at most $2c$ for each job in $G(t)$. Assume that at time t, for each machine i, we release extra b_i volume of type 1 jobs which can only go on machine i. Further, suppose machine i does s_i amount of processing during $(t, t+1)$ (s_i could be bigger than 1 because we are allowing speedup). Then, at time $t+1$, there must exist some machine i, such that at least $\frac{13}{8} + b_i - s_i$ volume of type 1 jobs which can only go on machine i remain unfinished.

Proof. Each of the type 0 jobs must have weighted flow-time at most $2c$, and so must finish within $1/4$ units after its release date. So the type 0 jobs released during $(t, t+\frac{3}{4})$ must finish during $(t, t+1)$. During $(t, t+\frac{3}{4})$, we release $\frac{15}{4}$ volume of type 0 jobs — since these must be done during $(t, t+1)$ on the 6 machines, it leaves us with $\sum_i s_i - \frac{15}{4}$ amount of time for processing the type 1 jobs. So, we must have $\sum_i b_i + 6 - \left(s_i - \frac{15}{4}\right) = \frac{39}{4} + \sum_i b_i - \sum_i s_i$ amount of unfinished volume of type 1 jobs at time $t+1$. Now we claim that some machine i must have at least $\frac{13}{8} + b_i - s_i$ amount of unfinished type 1 jobs at time $t+1$. Indeed, if this is not the case, then at time $t+1$, the total amount of unfinished type 1 jobs will be less than $6 \cdot \frac{13}{8} + \sum_i (b_i - s_i) = \frac{39}{4} + \sum_i (b_i - s_i)$, a contradiction. $\square$

Now, we give the actual construction of the instance $\mathcal{I}(s,c)$. The instance will have M machines, where $M = 6^{30c}$. Our instance will release jobs during $(0, 30c)$ — let $s_i(t)$ be the amount of processing that machine i does during $(t, t+1)$. Again, note that $s_i(t)$ can be quite large — we are only giving a bound on the average speed of a machine.

We will maintain the following invariant at every integral time $t = 0, \ldots, 30c$ — at the beginning of time t, there will be a set $M(t)$ of $\frac{M}{6^t}$ machines, such that for each of these machines i, the algorithm $\mathcal{A}$ will have at least $\frac{13t}{8} - \sum_{t'=0}^{t-1} s_i(t')$ volume of unfinished type 1 jobs which can only be assigned to i. All jobs released after time t will only go on one of the machines in $M(t)$. Further, at time t, the off-line algorithm would not have any unfinished jobs on these machines.

Clearly, this invariant holds at time 0. Suppose it holds at the beginning of time t. Let $M(t)$ denote the set of these $\frac{M}{6^t}$ machines. We group these machines into disjoint sets of 6 machines each — for each such group, we construct a copy of the gadget $G(t)$. So, let these gadgets be $G_1(t), \ldots, G_r(t)$, where $r = \frac{M}{6^{t+1}}$. Consider a gadget $G_u(t)$ — Claim D implies that there must exist a machine, call it $u(t)$, such that it will have $\frac{13(t+1)}{8} - \sum_{t'=0}^{t} s_{u(t)}(t')$ amount of unfinished type 1 job (we use $b_{u(t)} = \frac{13t}{8} - \sum_{t'=0}^{t-1} s_{u(t)}(t')$ using the invariant at time t). The set of machines $u(t), 1 \leq u \leq r$, form the set $M(t+1)$. This proves that the invariant holds at time $t+1$ as well.

It is easy to check that $M(t+1) \subseteq M(t)$ for all t, and hence, after time $t+1$, we will never assign any jobs to a machine outside $M(t)$. The optimum off-line algorithm has no unfinished volume on machines in $M(t)$ at time t (by invariant). Now, for each of the gadgets $G_u(t)$, it will process the type 1 job released on the machine $u(t)$ during $(t, t+1)$ and all type 0 jobs released during $(t, t+1)$ will be processed on the remaining 5 machines in this gadget. The 5 type 1 jobs (other than the one which can be processed on $u(t)$) will be processed on the corresponding machines during $(t+1, t+2)$ — note that these machines will be idle after time $t+1$, and so this processing can always be done. Thus, all jobs corresponding to this gadget have weighted flow-time of at most 2. Further, the optimum algorithm finishes all jobs which can go on $u(t)$ by time $t+1$.

Therefore, at time $30c+1$, there is some machine i which has more than $\frac{13(30c+1)}{8} - \sum_{t'=0}^{30c} s_i(t')$ amount of unfinished type 1 jobs. Notice that $T(s,c) = 30c + 2c \max_j w_j = 32c$, and so machine i is only allowed total of $\frac{3}{2} \cdot 32c = 48c$ amount of processing during $(0, 32c)$. So, $\sum_{t'=0}^{30c} \leq 48c$. Since $\frac{13(30c+1)}{8} - 48c > 0$, some type 1 job must remain unfinished at time $32c$. This contradicts the fact that $\mathcal{A}$ is c-competitive.

Remarks: Before we go to the induction step, we write down some more invariants about the instance $\mathcal{I}(s,c)$ — it is easy to check that they hold at $s = 2$. First of all, the instance $\mathcal{I}(s,c)$ is constructed with reference to an on-line algorithm $\mathcal{A}$ — so we may refer to it as $\mathcal{I}^{\mathcal{A}}(s,c)$. Further, the jobs released at any time t depend on the following: the speed profile of each of the machines until time t,

and the amount of processing done on all the jobs released before t. In particular, the instance does not depend on the average speedup of the machines. Further, the number of machines and the duration of the instance do not depend on $\mathcal{A}$ — so we will refer to these quantities as $M(s,c)$ and $T(s,c)$ respectively. Also, jobs are released at epochs which are multiples of a parameter $\varepsilon = \frac{1}{8c}$. In all of these instances, the optimum off-line value will be 2.
Induction Step: Suppose the induction hypothesis is true for s and c. We show it is true for $(s+1)$. Fix an on-line algorithm $\mathcal{A}$. We will first construct a gadget G. The gadget G will be constructed depending on how $\mathcal{A}$ behaves. In addition, we will also build another on-line algorithm $\mathcal{B}$ and the corresponding instance $\mathcal{I}^{\mathcal{B}}(s,c)$. G will have $lM(s,c)$ machines, where $l = 3s$. For each machine $i \in \mathcal{I}^{\mathcal{B}}(s,c)$, we will identify l of the machines in G — call these $A(i)$; these sets are disjoint for different i. Further, whenever a job j gets released in $\mathcal{I}^{\mathcal{B}}(s,c)$, we will release $(l-1)$ identical jobs in G – call these $C(j)$. If a job j can go on a set of machines S in $\mathcal{I}^{\mathcal{B}}(s,c)$, then we allow a job in $C(j)$ to go on the machines $\cup_{i \in S} A(i)$ in G. We shall call these jobs type C jobs. Besides these jobs, we will have jobs of type D in G — these jobs will not have any analogues in $\mathcal{I}^{\mathcal{B}}(s,c)$. Each job of type D will have size $T(s,c)$ and weight $\frac{1}{T(s,c)}$.

Let us now construct the gadget G and the instance $\mathcal{I}^{\mathcal{B}}(s,c)$ along with the algorithm $\mathcal{B}$. At time 0, if $\mathcal{I}^{\mathcal{B}}(s,c)$ releases a set of jobs, then we release the corresponding set of jobs in G as described above. Further, we release $(l-1)M(s,c)$ type D jobs at time 0 in G — each of these jobs can go on exactly one of the machines in G.

Now suppose we have constructed the gadget and the algorithm $\mathcal{B}$ until time $T\varepsilon$ for some integer $T \geq 0$. During a time $t \in (T\varepsilon, (T+1)\varepsilon)$, if a machine i' in G processes jobs of type C at rate $x_{i'}(t)$, then we run a machine $i \in G$ at speed $\frac{\sum_{i' \in A(i)} x_{i'}(t)}{l-1}$ at time t. Hence, during this period, if $\mathcal{A}$ processes a job $j' \in C(j)$ on machine $i' \in A(i)$, then $\mathcal{B}$ processes the job j on i at $1/(l-1)$ of the rate at which j' gets processed on i'. Note that we will not process a job j in $\mathcal{I}^{\mathcal{B}}(s,c)$ for more than p_j amount of time. Thus, we have described $\mathcal{B}$ until time $(T+1)\varepsilon$, and so depending on which jobs get released at $\mathcal{I}^{\mathcal{B}}(s,c)$ at this time, we release corresponding jobs in G. This completes the description of G. We now prove the analogue of Claim D.

Claim. Suppose the algorithm $\mathcal{A}$ runs machine i at average speed of s_i in G (during $(0, T(s,c))$). Further, suppose at time 0, for each machine i, we have released $b_i T(s,c)$ volume of type D jobs which can only go on machine i. If $\mathcal{A}$ incurs weighted flow-time of at most $2c$ on all type C jobs, then there exists a machine i for which we have at least $T(s,c)\left(b_i + \frac{1}{4} + \frac{s+2}{2} - s_i\right)$ unfinished volume of type D jobs at time $T(s,c)$.

Proof. If $\mathcal{A}$ incurs weighted flow-time of at most $2c$ on all type C jobs, then $\mathcal{B}$ is c-competitive on $\mathcal{I}^{\mathcal{B}}(s,c)$. So, by the induction hypothesis, there exists a machine $i \in \mathcal{I}^{\mathcal{B}}(s,c)$ which runs at average speed at least $(s+1)/2$. So, if we consider the machines in $A(i)$, then they spend $(l-1)(s+1)T(s,c)/2$ amount of time processing type C jobs. So, the total amount of time for which they can process a job of type

D is at most $\left(\sum_{i' \in A(i)} s_{i'} - \frac{(l-1)(s+1)}{2}\right) T(s,c)$. So, there must exist a machine $i' \in A(i)$ which processes type D jobs for at most $\left(s_{i'} - \frac{(l-1)(s+1)}{2l}\right) T(s,c)$ amount of time (since $|A(i)| = l$). So the unfinished volume of type D jobs on this machine is $b_{i'}T(s,c) + T(s,c) - \left(s_{i'} - \frac{(l-1)(s+1)}{2l}\right) T(s,c)$. The claim follows because

$$1 + \frac{(l-1)(s+1)}{2l} \geq \frac{s+2}{2} + \frac{1}{4}. \qquad \square$$

The rest of the proof is as in the base case. We copy the same proof verbatim with suitable changes. We construct the instance $\mathcal{I}(s+1,c)$. The number of machines will be $M(s+1,c) = (lM(s,c))^{30cs}$. We will divide time into epochs of size $T(s,c)$. We will be releasing jobs during $(0, 30cs \cdot T(s,c))$. Let $s_i(e)$ be the average speed of machine i during epoch e, i.e., $(e \cdot T(s,c), (e+1) \cdot T(s,c))$. We shall use $G(e)$ to refer to the gadget G starting at time $e \cdot T(s,c)$.

We will maintain the following invariant at every epoch $e = 0, \ldots, 30cs$ — at the beginning of time $eT(s,c)$, there will be a set $M(e)$ of $\frac{M(s+1,c)}{(lM(s,c))^e}$ machines, such that for each of these machines i, the algorithm $\mathcal{A}$ will have at least $T(s,c)\left(\frac{e}{4} + \frac{(s+2)e}{2} - \sum_{e'=0}^{e-1} s_i(e')\right)$ volume of unfinished type D jobs which can only be assigned to i. All jobs released after time $e \cdot T(s,c)$ will only go on one of the machines in $M(e)$. Further, at the beginning of epoch e, the off-line algorithm would not have any unfinished jobs on these machines.

Clearly, this invariant holds at time 0. Suppose it holds at the beginning of epoch e. Let $M(e)$ denote the set of these $\frac{M(s+1,c)}{(lM(s,c))^e}$ machines. We group these machines into disjoint sets of $lM(s,c)$ machines each — for each such group, we construct a copy of the gadget $G(e)$. So, let these gadgets be $G_1(e), \ldots, G_r(e)$, where $r = \frac{M(s+1,c)}{(lM(s,c))^{e+1}}$. Consider a gadget $G_u(e)$ — Claim D implies that there must exist a machine, call it $u(e)$, such that it will have $T(s,c)\left(\frac{e+1}{4} + \frac{(s+2)(e+1)}{2} - \sum_{e'=0}^{e} s_i(e')\right)$ amount of unfinished type D job (we use $b_{u(t)} = T(s,c)\left(\frac{e}{4} + \frac{(s+2)e}{2} - \sum_{e'=0}^{e-1} s_i(e')\right)$ using the invariant at epoch e). The set of machines $u(e), 1 \leq u \leq r$, form the set $M(e+1)$. This proves that the invariant holds at the beginning of epoch $e+1$ as well.

It is easy to check that $M(e+1) \subseteq M(e)$ for all e, and hence, after epoch e, we will assign all jobs to a machine in $M(e)$ only. The optimum off-line algorithm has no unfinished volume on machines in $M(e)$ at time beginning of epoch e (by invariant). Now, for each of the gadgets $G_u(e)$, it will process the type D job released on the machine $u(e)$ during this epoch and all type C jobs released during this epoch will be processed on the remaining machines in this gadget. This can be done since by the induction hypothesis, the off-line algorithm can finish all jobs by time $T(s,c)$ in the instance $\mathcal{I}^{\mathcal{B}}(s,c)$. So the off-line algorithm can do the same in the gadget $G_u(e)$ — each job in $\mathcal{I}^{\mathcal{B}}(s,c)$ has $l-1$ copies in the gadget $G_u(e)$, but then barring the machine used for type D job, we still have $l-1$ machines corresponding to each machine in $\mathcal{I}^{\mathcal{B}}(s,c)$.

The remaining $lM(s,c) - 1$ type D jobs (other than the one which can be processed on $u(e)$) will be processed on the corresponding machines during $((e+$

$1)\,T(s,c),(e+2)T(s,c))$ — note that these machines will be idle after epoch e, and so this processing can always be done. Thus, all jobs corresponding to this gadget have weighted flow-time of at most 2. Further, the optimum algorithm finishes all jobs which can go on $u(e)$ before the beginning of epoch $e+1$.

Therefore, at time $(30cs+1)\,T(s,c)$, there is some machine i which has more than $T(s,c)\left(\frac{30cs+1}{4}+\ \frac{(s+2)(30cs+2)}{2}-\sum_{e'=0}^{30cs} s_i(e')\right)$ amount of unfinished type D jobs. Notice that $T(s+1,c)=30csT(s,c)+2c\max_j w_j=(30s+2)cT(s,c)$, and so machine i is only allowed total of $\frac{s+2}{2}\cdot 32cT(s,c)=16(s+2)c\cdot T(s,c)$ amount of processing during $(0,T(s+1,c))$. So, $\sum_{t'=0}^{30c}\leq(15s+1)c(s+2)\cdot T(s,c)$. Since

$$\frac{30cs+1}{4}+\frac{(s+2)(30cs+2)}{2}-(15s+1)c(s+2)>0,$$

some type D job must remain unfinished at time $T(s+1,c)$. This contradicts the fact that $\mathcal{A}$ is c-competitive.

Now, note that $M(s+1,c)=(lM(s,c))^{30cs}$. This implies that $M(s,c)$ is at most $(20s)^{30cs^2}$. □