Minimizing Maximum (Weighted) Flow-Time on Related and Unrelated Machines

S. Anand[1], Karl Bringmann[2,⋆], Tobias Friedrich[3], Naveen Garg[1,⋆⋆], and 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. 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:

1. 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).
2. 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.
3. 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

⋆ Karl Bringmann is a recipient of the *Google Europe Fellowship in Randomized Algorithms*, and this research is supported in part by this Google Fellowship.

⋆⋆ Naveen Garg is supported by the Indo-German Max Planck Center for Computer Science (IMPECS).

F.V. Fomin et al. (Eds.): ICALP 2013, Part I, LNCS 7965, pp. 13–24, 2013.

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 [2,5,10,11] 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 $O(\varepsilon^{-1})$-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. [2] 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 $O(P^{1/2})$-competitive algorithm for a single

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.

	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	$(1+\varepsilon, O(\varepsilon^{-1}))$		

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. Our main results are:

1. 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 $\mathrm{O}(\varepsilon^{-4})$-competitive.
2. 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.
3. 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.

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. [3] 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 3 we consider the problem of minimizing maximum weighted flow time on related machines and unrelated machines, respectively. Section 4 considers the problem of minimizing maximum flow time on unrelated machines. Section 5 presents a lower bound for the L_p norm of the stretches.

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 below. 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)

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 $\mathtt{opt}(\mathcal{I}) \le T$, 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 $\mathtt{opt}(\mathcal{I}) \le T$. 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

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

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 as follows.

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)

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 full version of the paper.

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 full version of this paper, 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 full version of the paper.

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 $\varepsilon \leq 1$, 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

$$\frac{|\Pi_i| + T}{1+\varepsilon} \leq \frac{T/\varepsilon + T}{1+\varepsilon} = \frac{T}{\varepsilon}.$$

□

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 full version of the paper.

Theorem 5. *Given any large enough parameter c, integer $s \geq 1$, 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 $m + m/2 + m/4 + \ldots = O(m)$ 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.

References

1. Ambühl, C., Mastrolilli, M.: On-line scheduling to minimize max flow time: An optimal preemptive algorithm. Oper. Res. Lett. 33(6), 597–602 (2005)
2. Anand, S., Garg, N., Kumar, A.: Resource augmentation for weighted flow-time explained by dual fitting. In: 23rd Symp. Discrete Algorithms (SODA), pp. 1228–1241 (2012)
3. Anand, S., Garg, N., Megow, N.: Meeting deadlines: How much speed suffices? In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part I. LNCS, vol. 6755, pp. 232–243. Springer, Heidelberg (2011)
4. Azar, Y., Naor, J., Rom, R.: The competitiveness of on-line assignments. J. Algorithms 18(2), 221–237 (1995)
5. Bansal, N., Pruhs, K.: Server scheduling in the ℓ_p norm: A rising tide lifts all boats. In: 35th Symp. Theory of Computing (STOC), pp. 242–250 (2003)
6. Bansal, N., Pruhs, K.: Server scheduling in the weighted ℓ_p norm. In: Farach-Colton, M. (ed.) LATIN 2004. LNCS, vol. 2976, pp. 434–443. Springer, Heidelberg (2004)
7. Bender, M.A., Chakrabarti, S., Muthukrishnan, S.: Flow and stretch metrics for scheduling continuous job streams. In: 9th Symp. Discrete Algorithms (SODA), pp. 270–279 (1998)
8. Bender, M.A., Muthukrishnan, S., Rajaraman, R.: Improved algorithms for stretch scheduling. In: 13th Symp. Discrete Algorithms (SODA), pp. 762–771 (2002)
9. Chekuri, C., Moseley, B.: Online scheduling to minimize the maximum delay factor. In: 20th Symp. Discrete Algorithms (SODA), pp. 1116–1125 (2009)
10. Golovin, D., Gupta, A., Kumar, A., Tangwongsan, K.: All-norms and all-ℓ_p-norms approximation algorithms. In: 28th Conf. Foundations of Software Technology and Theoretical Computer Science (FSTTCS), pp. 199–210 (2008)
11. Im, S., Moseley, B.: An online scalable algorithm for minimizing ℓ_k-norms of weighted flow time on unrelated machines. In: 22nd Symp. Discrete Algorithms (SODA), pp. 95–108 (2011)
12. Phillips, C.A., Stein, C., Torng, E., Wein, J.: Optimal time-critical scheduling via resource augmentation. Algorithmica 32(2), 163–200 (2002)