Minimizing Weighted ℓ_p-Norm of Flow-Time in the Rejection Model

Anamitra Roy Choudhury[1], Syamantak Das[2], and Amit Kumar[2]

1 **IBM Research, India**
anamchou@in.ibm.com
2 **IIT Delhi, India**
{sdas,amitk}@cse.iitd.ernet.in

Abstract

We consider the online scheduling problem to minimize the weighted ℓ_p-norm of flow-time of jobs. We study this problem under the *rejection model* introduced by Choudhury et al. (SODA 2015) – here the online algorithm is allowed to not serve an ε-fraction of the requests. We consider the restricted assignments setting where each job can go to a specified subset of machines. Our main result is an immediate dispatch non-migratory $1/\varepsilon^{O(1)}$-competitive algorithm for this problem when one is allowed to reject at most ε-fraction of the total weight of jobs arriving. This is in contrast with the speed augmentation model under which no online algorithm for this problem can achieve a competitive ratio independent of p.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Approximation algorithms, Flow time, Scheduling problem, Rejection model

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2015.25

1 Introduction

The problem of minimizing average flow-time, also known as response-time or waiting time, is of central importance in the scheduling literature [20, 5, 15, 2]. In the online setting of this problem, jobs arrive over time and need to be scheduled on machines, which may have varying characteristics. The flow-time of a job is defined as the difference between its completion time and release date, and we would like the jobs to have small flow-time. One way of measuring this is to take ℓ_p-norm of the flow-time of jobs, where the parameter p could vary depending on the particular application – varying p would mean balancing the trade-off between fairness and average response time. In this paper, we shall consider the well-studied subset-parallel (i.e., the restricted assignment) model, where each job j specifies a processing requirement p_j, but can be processed on a subset of the machines only.

The framework of competitive analysis for such problems turns out to be too pessimistic – it is known that there is no online algorithm for minimizing the average flow-time of jobs in the restricted assignment setting (even if we restrict all job sizes to 1) [16]. One popular approach toward handling this negative result is by providing the online algorithm slightly more power than the off-line adversary. Kalyanasundaram and Pruhs [19] introduced the speed-augmentation model where the machines of the online algorithm have slightly more speed than those of the offline algorithm. The speed augmentation model has been very successful in analyzing performance of natural algorithms for minimizing average flow-time in various scheduling settings. Anand et al. [4] showed that a natural greedy algorithm is constant competitive for minimizing average (weighted) flow-time in the restricted assignment setting

35th IARCS Annual Conf. Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015).
Editors: Prahladh Harsha and G. Ramalingam; pp. 25–37

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(and in the more general unrelated machines setting) if we provide the online algorithm with $(1+\varepsilon)$-extra speed. Im and Moseley [18] extended this result to give an $O(p/\varepsilon^2)$-competitive algorithm for minimizing the ℓ_p-norm of weighted flow-time. However, they showed that the linear dependence on p was necessary (for any immediate dispatch algorithm) even if we allow constant-speedup and all weights are 1. In the extreme case when p becomes infinity, it is known that one cannot obtain better than $O(\log m)$-competitive algorithm (where m denotes the number of machines) even if we allow constant speed-up.

To address the apparent inability of the speed-augmentation model to handle large values of p, Choudhury et al. [13] considered a different job rejection model. Here, we allow the online algorithm to reject an ε-fraction of the jobs, where ε is an arbitrary small positive constant, whereas the off-line optimum is required to schedule all the jobs. This model seems to give more power to the online algorithm than that by the speed augmentation model – the latter model gives *uniformly* extra speed to all machines, whereas in the former model, we could trade-off across machines (by rejecting more jobs assigned to a particular machine at the expense of fewer rejected jobs to other machines). Choudhury et al. [13] formalized this intuition by giving a constant competitive algorithm for the problem of minimizing weighted ℓ_∞-norm of flow-time if the online algorithm is allowed to reject ε-fraction of the weight of the jobs. In this paper, we extend this result by showing that one can get a constant competitive algorithm even for the problem of minimizing weighted ℓ_p-norm of flow-time of jobs, if we are allowed to reject ε-fraction of the weight of the jobs. Note that the competitive ratio has no dependence on p, and so, we get a stronger result as compared to the speed-augmentation model (though we seem to provide more power to the online algorithm).

Our algorithm is based on reducing the problem of minimizing ℓ_p-norm to that of minimizing ℓ_∞-norm (with some more job rejections). But this requires one to track the *average* ℓ_p-norm of jobs released so far (in an off-line optimum algorithm). This turns out to be non-trivial, as this quantity could go up or down with time, and tracking it while not exceeding the optimal value at any time forms the heart of our algorithm. We state the main theorem of the paper as follows.

▶ **Theorem 1.** *If the online algorithm is allowed to reject ε-fraction of the weight of the jobs arrived so far, then there exists an $O(1/\varepsilon^{12})$-competitive algorithm for the problem of minimizing weighted ℓ_p-norm of the flow-time of jobs in the restricted assignments setting.*

Organization of the paper. In Section 2, we formally describe the problems considered in this paper. For sake of clarity, we give details of the special case when p is 1 (i.e., the average flow-time), and all weights are 1. The extension to the weighted case carries over using ideas in Choudhury et al. [13], and the result for ℓ_p-norm for arbitrary p follows without much changes – details of these changes are described in Section 7. In Section 4, we give an overview of the new ideas in this paper. The scheduling algorithm is described in Section 5. The algorithm is split in two parts – algorithm $\mathcal{A}$ first ensures that all queues are bounded, and subsequently, we give algorithm $\mathcal{B}$ which uses $\mathcal{A}$ to construct the actual schedule. Analysis of these algorithms is given in Section 6.

2 Problem Statement

We consider the online problem of scheduling jobs over multiple machines in the subset-parallel (i.e., the restricted assignment) setting. Here, jobs arrive over time. Each job j specifies a processing requirement p_j and a subset S_j of the machines on which it can be processed. Let r_j denote the release date (i.e., the arrival time) of job j. In the weighted

version of the problems, each job j also specifies a non-negative weight w_j. In this paper, we consider algorithms which follow the immediate-dispatch policy: when a job j arrives at time r_j, it is dispatched to one of the machines. Recall that we do not allow migration of jobs across machines, and so, the job gets processed on the machine to which it gets dispatched.

Given a schedule $\mathcal{S}$, the flow-time of a job in the schedule $\mathcal{S}$, $F^{\mathcal{S}}(j)$ is defined as the difference between its completion time in $\mathcal{S}$ and r_j. The goal of our algorithm is to minimize the weighted ℓ_p-norm of the flow-time of jobs, defined as $(\sum_j w_j F^{\mathcal{S}}(j)^p)^{1/p}$. We allow the online algorithm to reject an ε-fraction of the total weight of the jobs – note that the algorithm could reject a job immediately on its arrival, or much later after dispatching the job to a machine. Here ε is a positive (small enough) constant. Thus, we only consider the total flow-time of jobs which do not get rejected by the online algorithm. However, the optimal off-line algorithm is required to schedule all the jobs.

In this paper, we give details of the more special `SumFlowTime` problem, where we seek to minimize the total sum of the flow-time of jobs (i.e., all job weights are 1, and p is 1). The extension to the general case follows along predictable lines and is outlined in Section 7.

3 Related Work

There has been considerable work on scheduling with the objective of minimizing a suitable norm of the flow-time of jobs. For the objective of average flow-time of jobs, a logarithmic competitive algorithm in the identical machines setting is known [20, 5]. Garg and Kumar [15] and subsequently Anand et al. [2] extended this result to the related machines setting. Garg and Kumar [16] showed that the problem becomes considerably harder in the restricted assignment setting and no online algorithm with bounded competitive ratio is possible. Bansal and Pruhs [8] showed that the competitive ratio can be as high as $\Omega(n^c)$ for the problem of minimizing ℓ_p (for any $1 < p < \infty$) norm, where n is the number of jobs, even for a single machine. For minimizing the maximum flow-time in the identical machines model, Ambühl and Mastrolilli [1] gave a simple 2-competitive algorithm. However, Anand et al. [3] showed that the competitive ratio of any online algorithm for the restricted assignment setting is as high as $\Omega(m)$, where m is the number of machines.

The speed augmentation was first proposed by Kalyanasundaram and Pruhs [19] who used it to get an $O(1/\varepsilon)$-competitive algorithm for minimizing flow time on a single machine in the non clairvoyant setting. Bansal and Pruhs [8] proved that several natural scheduling algorithms are $O(1/\varepsilon)$-competitive algorithm for minimizing ℓ_p norm (for any $1 < p < \infty$) of flow-time of jobs in the single machine setting. Golovin et al. [17] extended this result to parallel machines setting. Chekuri et al. [12] showed that the immediate dispatch algorithm of Avrahami and Azar [5] is also $O(1/\varepsilon)$-competitive for all ℓ_p norms ($p \geq 1$).

In the general setting of unrelated machines with speed augmentation, Chadha et al. [10] gave an $O(1/\varepsilon^2)$-competitive algorithm for minimizing the sum of flow-time of jobs, which was improved and extended to the case of ℓ_p norm of flow-time by Im and Moseley [18] and Anand et al. [4]. Im and Moseley [18] present an $O(p/\varepsilon^{2+2/p})$ immediate dispatch and non-migratory algorithm for minimizing the ℓ_p norm of weighted flow-time in unrelated machine; they also show that any immediate dispatch non migratory online algorithm with speed $s > 1$ has competitive ratio $\Omega(p/s)$. Anand et al. [3] showed that for the problem of minimizing weighted ℓ_p norm of flow time of jobs, one cannot obtain competitive ratio better than $\Omega\left(\frac{p}{\varepsilon^{1-O(1/p)}}\right)$ even with non-immediate dispatch. The last two lower bounds hold even in the restricted assignment model.

For minimizing the maximum (unweighted) flow time on unrelated machines, Anand et al. [3] gave a $O(1/\varepsilon)$-competitive, $(1+\varepsilon)$-speed algorithm; however their algorithm is not an

immediate dispatch algorithm. In fact, Azar et al. [6] showed that any immediate dispatch algorithm with constant speedup is $\Omega(\log m)$-competitive in the restricted assignment setting. In the maximum weighted flow-time case, this lower bound holds even if we allow non-immediate dispatch [3].

Scheduling with Rejection. There has been considerable work on online scheduling with job rejections in the prize collecting setting where one incurs an extra cost for non-scheduled jobs (see e.g. [9, 14, 7, 11]).

4 Our Techniques

Here we outline the main ideas of our algorithm (which we call algorithm $\mathcal{A}$). The first idea is to start with the result of Choudhury et al. [13]. They consider the same setting as ours – jobs arriving online in the subset parallel setting. Given parameters ε and T, their schedule processes all but ε-fraction of the jobs. Assuming that there a schedule for which the ℓ_∞-norm of the flow-time of jobs is at most T, they give an online algorithm where the flow-time of all the non-rejected jobs is at most $O(T/\varepsilon^2)$.

For our problem, let us assume that we know the number of jobs n, and the optimal value $T_1^\star$ of the total flow-time of these jobs. From this it follows that at least $(1-\varepsilon)$-fraction of the jobs will have flow-time at most $T^\star = T_1^\star/(\varepsilon n)$. Conversely, if we can have a schedule which ensures that all but ε-fraction of jobs have flow-time at most $T^\star$, then the total flow-time of jobs which are not rejected is $O(T_1^\star/\varepsilon)$. Thus, we have converted the problem of minimizing the ℓ_1-norm to that of minimizing the ℓ_∞-norm of flow-time. So it seems natural to apply the result of Choudhury et al. mentioned above to our problem. However there are two main issues:

- The result of [13] assumes that the parameter T is such that there is a schedule for which the maximum flow-time of *all* jobs is at most T. For us, we have a parameter $T^\star$ such that there is a (off-line) schedule for which the maximum flow-time of all but ε-fraction of the jobs is at most $T^\star/\varepsilon$. We prove a generalization of the result of [13], where the online algorithm can reject 7ε-fraction of the jobs, whereas we compare it with an offline schedule for minimizing maximum flow-time of all but ε-fraction of the jobs. This requires going through the calculations of [13] and making some subtle changes to accommodate the changed settings.
- The more serious issue turns out to be the fact that we really do not know the values $T^\star$. In [13], as is usual in such problems, one starts with a small guess T of $T^\star$ – whenever the algorithm rejects more than ε-fraction of the jobs, they double the guess T and start afresh. This ensures that the T will never go beyond twice the optimal value $T^\star$. Here, we cannot adopt this strategy. Suppose the input consists of two *phases*: an initial phase with lot of jobs such that they have high objective value (both in terms of ℓ_1 and ℓ_∞ norms), and a second phase where jobs arrive over a much longer period of time, and so their flow-time are small. Our algorithm will need to increase the value of T during the first phase. However, it cannot work with a high value of T during the second phase – otherwise, it may allow the jobs in the second phase to last much longer than in an optimal solution. The problem arises from the fact that $T^\star$ is defined as the ratio of two parameters, both of which change (in fact, increase) with time, and so, if we are trying to keep track of $T^\star$, we will need to both increase and decrease the estimate T.

We now describe the details about how we handle the second problem above. As mentioned above, we maintain a variable T which is supposed to track the value $T^\star = T_1^\star/(\varepsilon n)$. We

start with a slightly weaker goal: we want to maintain a schedule such that at all times the queue size (i.e., the total remaining processing time of the jobs in the queue) on any machine remains bounded by $T/\varepsilon^{O(1)}$. We divide the execution of our algorithm into *phases* – during a phase P, we shall not change the value of estimate T (denoted by $\mathbf{est}(P)$). During a phase, we shall work with the algorithm of Choudhury et al. [13] (by supplying it the estimate $T = \mathbf{est}(P)$). The phase will be terminated by one of the following two events:

Case 1: We reject too many jobs in this phase (i.e., at least $O(\varepsilon)$ times the number of job arrivals): here, we show that if N jobs arrive during this phase, then the optimal value of the ℓ_1 norm of these jobs is $\Omega(NT\varepsilon^3)$. This lower bound allows us to pay for the flow-time of these jobs incurred by our algorithm (which will be $NT/\varepsilon^{O(1)}$). Further, we can end this phase, and start a new phase P' with $\mathbf{est}(P') = cT$, where $c = 2/\varepsilon$. We call such phases *good* phases. Whenever a good phase ends at a time t, we push its state on a stack $\mathbf{S}$. More formally, we push a new entry e on the stack $\mathbf{S}$, where e is the tuple $(T, Q(t), J)$. Here, $Q(t)$ denotes the jobs which are waiting to be processed on the queues of one of the machines, and J is a set of jobs for which we can argue that the optimal value is large. Note that we have *shelved* the jobs $Q(t)$ on the stack, and we will start with empty queues in the next phase.

Case 2: A lot of jobs arrive during this phase: this is the worrying case, because if N jobs arrive during this phase, then we may have paid $\Omega(NT)$ for their total flow-time, but the optimum value could be much less. The only way to pay for these jobs would be to charge to the lower bound obtained from previous good phases (stored in the stack $\mathbf{S}$). Whenever we charge to a phase in the top of the stack, we pop it so that it does not get charged again. Now, we would like to end this phase and start a new phase with a smaller value of the estimate T. We face several obstacles here:

- The first obstacle is that if t denotes the current time, then we would need to reduce the number of jobs in the queues of the machines. Our analysis requires that for any phase P, the queue sizes remain bounded by about $\mathbf{est}(P)/\varepsilon^{O(1)}$. Therefore, we are going to reduce the value of $\mathbf{est}(P)$, then we may need to reduce the queue sizes as well. This would mean rejecting jobs in the queues of the machines. Now this can be done provided we do not reject too many jobs. Assuming this is the case, we can start a new phase with a reduced estimate of T/c. However, note that in beginning of the new phase, we will still have non-empty job queues. Therefore, Case 1 (for the new phase) above needs to take these jobs into account as well.
- In the discussion above, when we are trying to reduce the queue sizes at time t, suppose we are not able to do so (because we would end up rejecting too many jobs). Here, we argue that even the optimal ℓ_1-norm of the flow-time of jobs in this phase will be $\Omega(NT\varepsilon^3)$. Thus, this phase again behaves like a good phase, and we save its state on $\mathbf{S}$. Further, we start a new phase P' with estimate T again.

5 The scheduling algorithm

We first describe the algorithm $\mathcal{A}$ and then extend it to the actual scheduling algorithm. We now give all the details of $\mathcal{A}$. We maintain a variable T during the algorithm. The variable T will change in powers of a constant c. For a phase P, we shall use $s(P)$ and $e(P)$ to denote its starting and ending time. We also have a stack $\mathbf{S}$ which is initially empty, and the variable $\mathbf{e}_{\mathtt{top}}$ will denote the entry at the top of the stack. Each entry e in the stack will be a tuple corresponding a phase P: $(\mathbf{est}(P), Q(e(P)), J)$, where $Q(t)$ denotes the jobs in

Algorithm DispatchJob(Job j):

If j is T-big
Reject the job
Else
Let j be of class k.
If for all $i \in S_j$, $\texttt{load}_{i,k}(r_j) + p_j \geq \alpha \cdot T$.
Reject the job
Else
Dispatch j to machine $i \in S_j$ for which $\texttt{load}_{i,k}(r_j)$ is minimum.

Figure 1 Algorithm for dispatching a job.

the queues of all the machines at time t, and J will be a set of jobs (for which we will argue that the optimal value is also large).

We first describe the job dispatch rule. Some definitions first. Let β denote a constant (which will be roughly $O(1/\varepsilon)$). We say that a job j is of class k if p_j lies in the interval $[\beta^k, \beta^{k+1})$. For a machine i, time t, and class k, let $Q_{i,k}(t)$ denote the jobs of class k waiting in the queue of machine i at time t; and define the $\texttt{load}_{i,k}(t)$ as the total remaining processing time of the jobs in $Q_{i,k}(t)$. The job dispatch rule is described in Figure 1. A job is said to be T-*big* if its size is at least $T \cdot (\varepsilon/2)$ and T-*small* otherwise. Thus, the algorithm just considers the queue sizes on each machine corresponding to the class to which j belongs. If all such queues are already full to their limit αT (where $\alpha = O(1/\varepsilon^3)$), we reject the job, else we dispatch it to the one with the smallest load on it.

We now describe the rule according to which jobs are processed on a machine. This is identical to that in [13], but we give it here (in Figure 2) for sake of completeness. It tries to balance two aspects: (i) process small jobs first, and (ii) process jobs from that class for which the corresponding queue is close to its allowable limit. Finally, we describe the algorithm $\mathcal{A}$ in Figure 3. Let P denote the current phase, and P^{prev} denote the previous phase. The algorithm distinguishes two cases: (i) P^{prev} was a good phase, i.e., $\mathbf{est}(P^{prev}) \leq \mathbf{est}(P) = T$, or (ii) P^{prev} was a bad phase, i.e., $\mathbf{est}(P^{prev}) > \mathbf{est}(P)$. If the former case happens, the phase P begins with empty queues of all machines, whereas in the latter case, it begins with non-empty queue sizes. The variable P' is meant to be P^{prev} in the latter case, whereas it is undefined (or empty) in the former. The variable $A(P)$ keeps track of the set of jobs arrived so far in P, whereas the variable $A'(P)$ keeps track of the set of jobs arrived in both P and P'. The variable $R(P)$ denotes the set of jobs which get rejected during the current phase. The variables $a(P)$, $a'(P)$ and $r(P)$ respectively denote the cardinality of the sets $A(P)$, $A'(P)$ and $R(P)$. Let $P^{\mathbf{top}}$ denote the phase corresponding to the entry in the top of the stack $\mathbf{S}$. In case $\mathbf{est}(P^{\mathbf{top}})$ is T or T/c, we define $Q^{\mathbf{top}}$ to be $Q(e(P^{\mathbf{top}}))$, i.e., the jobs which were shelved to the stack during this phase. Otherwise $Q^{\mathbf{top}}$ is set empty.

We now discuss the various steps in $\mathcal{A}$. We start with the estimate T to be 0, and P as the current phase. Recall that P' denotes the previous phase if the previous phase was a bad phase, else it is empty. When a job j arrives, we will increment the counters $a(P)$, $a'(P)$ which counts the number of jobs arrived so far in P and $P \cup P'$ respectively. Normally, we will just call **DispatchJob(j)**. However, this procedure will reject j if j happens to T-big. Now if very few (i.e., $1/\varepsilon$) jobs have arrived so far, then we do not want to reject any

Algorithm ProcessJob(i,t):

$k^\star := \mathbf{argmax}_k \frac{\texttt{load}_{i,k}(t)}{\beta^k}$.
Process the earliest released job from the queue $Q_{i,k^\star}(t)$
(use a fixed tie-breaking rule).

Figure 2 Algorithm for deciding which job gets processed at time t on a machine i.

job. Thus, if this case happens in the initial period of this phase (when not many jobs have arrived), we simply end this phase, and start a new phase with a much higher estimate – this phase will be a good phase because this job's processing time serves as a good lower bound. Note one subtlety – we will consider job j again in the next phase (because we haven't called **DispatchJob**(j) yet).

In the algorithm, we define two procedures – **EndGoodPhase** and **EndBadPhase**. The first one assumes that the current phase has ended as a good phase, while the latter one assumes otherwise. The procedure **EndGoodPhase** just ends the current phase by pushing a new entry on the stack, resetting the value of T, and initializing all queues to empty. The second procedure simply resets the value of T, but does not disturb the queues – the jobs in these queues carry over to the next phase.

Finally, we have a procedure **QueuedJobs**(v), where v is a parameter. This procedure finds the minimal collection of jobs which need to be removed from each of the queues $Q_i(k,t)$, where t denotes the current time, in the reverse order in which they were added to these queues, such that the total remaining processing time of jobs in each of the queues is at most v. It returns the set of such jobs.

As discussed before, algorithm $\mathcal{A}$ tries to maintain a schedule such that for all machines i, class k and time t, the queues $Q_{i,k}(t)$ remains bounded by $\mathbf{est}(P)/\varepsilon^{O(1)}$, where P denotes the phase containing time t. Note that $Q_{i,k}(t)$ only counts the jobs which arrive over this phase (and may be the jobs which were present initially in the queues of the machines if the previous phase was a bad phase). It does not count the jobs which have been shelved in the stack $\mathbf{S}$. We will however prove a stronger property: for any processing class k and an estimate T, let J_T^k be the set of jobs of class k which arrived during phases P for which $\mathbf{est}(P)$ was T; and let $J_T = \cup_k J_T^k$ denote the set of all such jobs. Then, at any time t and machine i, the total remaining processing time of jobs in J_T^k which were dispatched to machine i remains bounded by $T/\varepsilon^{O(1)}$. The fact, however, by itself is not sufficient to guarantee that we can finish any job of J_T within $T/\varepsilon^{O(1)}$ time. We now present our actual algorithm (algorithm $\mathcal{B}$) which ensures that the flow time of every job of J_T (with some additional rejections over algorithm $\mathcal{A}$) is bounded to at most $T/\varepsilon^{O(1)}$.

The scheduling algorithm $\mathcal{B}$. We here state our final scheduling algorithm. We will be using the result of Choudhury et al. [13] for the `GenWtdMaxFlowTime` problem. In the `GenWtdMaxFlowTime` problem, a job j has two weights associated with it, the rejection-weight $w_j^{(r)}$ and flow-time-weight $w_1^{(f)}$; the first one is used for counting the rejection weight of rejected jobs, while the second one is used in the weighted flow-time expression. The objective of the problem is to minimize the maximum over all jobs j of $w_j^{(f)} F_j$, where F_j denotes the flow-time of job j in a schedule; and we are allowed to reject jobs of rejection-weight at most ε times the total rejection-weight of all the jobs. The objective value is compared with the offline optimum which is not allowed to reject any job. In order to describe their

Algorithm $\mathcal{A}$:

Initialize:
$T \leftarrow 0$, $P' \leftarrow \emptyset$; $a(P), a'(P), r(P) \leftarrow 0$; $A(P), A'(P), R(P) \leftarrow \emptyset$.

Phase(P):

1. When a job j arrives at current time t
   (i) If j is T-big and $a(P) \leq 1/\varepsilon$,
   call **EndGoodPhase**$(T, Q(t), A(P) \cup \{j\}, \lceil p_j \rceil)$.
   (ii) Else
   – Update $A(P) \leftarrow A(P) \cup \{j\}, a(P) \leftarrow a(P) + 1$
   – Update $A'(P) \leftarrow A'(P) \cup \{j\}, a'(P) \leftarrow a'(P) + 1$
   – call **DispatchJob**(j)
   – if this job gets rejected,
   update $R(P) \leftarrow R(P) \cup \{j\}, r(P) \leftarrow r(P) + 1$
2. If $r(P) \geq 7\varepsilon \cdot a'(P)$
   (i) call **EndGoodPhase**$(T, Q(t), A'(P), cT)$.
3. If $a(P) \geq (|Q(s(P)|) + |Q^{\mathbf{top}}|)/\varepsilon$
   (i) Reject the jobs in $Q(s(P)) \cup Q^{\mathbf{top}}$.
   (ii) If $\mathbf{est}(P^{\mathbf{top}}) = T$ or T/c, pop the stack $\mathbf{S}$.
   (iii) Let $J \leftarrow$ **QueuedJobs**(T/c).
   (iv) If $|J| \leq 7\varepsilon a(P)$
   Reject all jobs in J and Call **EndBadPhase**(T/c).
   (v) Else Call **EndGoodPhase**$(T, Q(t), A(P), T)$.

EndGoodPhase(T_1, J_1, J_2, T_2)**:**

1. Push (T_1, J_1, J_2) on the stack $\mathbf{S}$.
2. Update $T \leftarrow$ smallest value of c^k above or equal to T_2, for integer k.
3. Initialize all queues to empty.
4. Start a new phase P with
   $A(P), A'(P), R(P) \leftarrow \emptyset, a(P), a'(P), r(P) \leftarrow 0$.

EndBadPhase(T_1)**:**

1. Update $T \leftarrow T_1$.
2. The queues on all machines remain unchanged.
3. Set P' to be the current phase P.
4. Start a new phase P with $A'(P) \leftarrow A(P')$, $A(P) \leftarrow \emptyset$.
   $a'(P) \leftarrow |A'(P)|, R(P) \leftarrow \emptyset, r(P), a(P) \leftarrow 0$.

Figure 3 Algorithm $\mathcal{A}$.

result, we also need to define *flow-time-weight class* and *rejection-weight-density class*. A job j with processing time p_j, rejection-weight $w_j^{(r)}$ and flow-time-weight $w_j^{(f)}$ is of flow-time-weight class k if $2^k \leq w_j^{(f)} < 2^{k+1}$. Similarly, it is of rejection-weight-density class k if $2^k \leq w_j^{(r)}/p_j < 2^{k+1}$. For a job j with remaining processing time p'_j at some time during a schedule, its remaining weighted processing time is simply $w_j^{(f)} \cdot p'_j$. Then we have

▶ **Theorem 2** ([13]). *Suppose there is an immediate dispatch schedule for an instance of the* `GenWtdMaxFlowTime` *problem with the following property: for every flow-time-weight class k and rejection-weight-density-class k', time t and machine i, the total remaining weighted processing times of such jobs waiting in the queue of machine i at time t is at most T, for a parameter T. Then, one can construct another immediate dispatch schedule which dispatches each job to the same machine as in the given schedule, and which may reject some jobs of rejection weight $O(\varepsilon)$ times the total rejection weight of all jobs, such that the weighted flow-time of every job is at most T/ε^4.*

We now present algorithm $\mathcal{B}$. This algorithm first maps our instance $\mathcal{I}$ to an instance $\mathcal{I}'$ of the `GenWtdMaxFlowTime` problem and then invokes the above theorem to get a schedule for $\mathcal{I}'$. We show that the corresponding schedule for $\mathcal{I}$ has the desired properties.

Our algorithm $\mathcal{B}$ will emulate algorithm $\mathcal{A}$ – when a job j arrives, it is dispatched according to $\mathcal{A}$: if $\mathcal{A}$ rejects this job, $\mathcal{B}$ also rejects it; and if $\mathcal{A}$ dispatches it to machine i, then $\mathcal{B}$ also dispatches this job to i. Further, if j does not get rejected, $\mathcal{B}$ adds the job to the instance $\mathcal{I}'$ with the same release date and processing requirement. If the current time (at which j is released) belongs to a phase P of schedule $\mathcal{A}$, then we set $w_j^{(f)}$ to $1/\mathbf{est}(P)$. We set $w_j^{(r)}$ to 1. Now we invoke the theorem above to build a schedule for $\mathcal{I}'$. This schedule may reject some more jobs, but dispatches jobs to the same machine as in $\mathcal{B}$. Thus, we can use the same schedule for $\mathcal{I}$ as well. This completes the description of our scheduling algorithm.

6 Analysis

We here give the analysis of the schedules $\mathcal{A}$ and $\mathcal{B}$.

Algorithm $\mathcal{A}$. We first show that algorithm $\mathcal{A}$ does not reject too many jobs.

▶ **Lemma 3.** *Algorithm $\mathcal{A}$ rejects $O(\varepsilon)$-fraction of the jobs.*

Proof. We argue that the total number of jobs rejected in a phase P is at most $O(\varepsilon)$ times the total number of jobs released during this phase and the previous phase. Summing over all phases, this will prove the desired result.

Algorithm $\mathcal{A}$ employs the following job rejections within a particular phase:

(a) Whenever a job arrives, the **DispatchJob** routine may reject the job - by Step 2 of the algorithm, the total number of such jobs is at most $7\varepsilon a'(P) + 1 \leq 8\varepsilon a'(P)$, because we know that $a'(P) \geq 1/\varepsilon$ (indeed, if $a'(P) < 1/\varepsilon$, then it is easy to check that **DispatchJob** will reject a job j only is it is T-big. But then we should have terminated this phase in Step 1(i) of the algorithm.)

(b) Rejection of jobs in Step 3: Note that steps 3(i) and 3(iv) can be executed at most once during a phase, because after these steps, we will end this phase. The conditions in Step 3 clearly state that the number of such job rejections is $O(\varepsilon a'(P))$. ◀

Next we state the main technical property of algorithm $\mathcal{A}$. This is where we show that for good phases, we get a lower bound on the optimal solution as well. Suppose the entry $(\mathbf{est}(P), Q, S)$ be pushed to the stack $\mathbf{S}$ at the end of a good phase P. Then the following lemma 4 gives a lower bound on the optimal flow-time of the jobs of S. Since $A(P) \subseteq S$, the lower bound also holds for the jobs arriving in the phase P. The proof of this lemma is deferred to the full version; the proof uses ideas from [13], but requires many new details as well.

▶ **Lemma 4.** *Suppose the entry* $(\mathbf{est}(P), Q, S)$ *be pushed to the stack* **S** *at the end of a good phase* P*. The total flow-time of any (off-line) algorithms on the set of jobs in* S *is at least* $\varepsilon^2 \cdot |S| \cdot \mathbf{est}(P)/c$*.*

Recall that for any class k, and parameter T, J_T^k denotes the set of jobs of class k which arrived during those phases P for which $\mathbf{est}(P)$ was T; and J_T denotes $\cup_k J_T^k$.

▶ **Lemma 5.** *Algorithm* $\mathcal{A}$ *ensures that at any time* t*, machine* i *and class* k*, the total remaining processing time of jobs of* J_T^k*, which have been dispatched to a machine* i*, at a time* t *is at most* $O(\alpha T)$*.*

Proof. We shall prove some invariant properties of the stack **S**. The lemma will follow from these properties. Note that an entry in the stack was pushed in Steps 1(i), or 2(i) or 3(v). For each such entry e, let T_e denote the estimate for the corresponding phase. For the sake of writing down the invariants, we define a related quantity, T'_e as follows: if e was pushed on the stack due to Steps 1(i) or 2(i), we define $T'_e = T_e$. Else we set $T'_e = T_e/c$. Consider a time t, and let P denote the current phase, and T be $\mathbf{est}(P)$. Let the entries in the stack (from top to bottom order) be $e_1, \ldots, e_k$. Then, the following property holds: $T > T'_{e_1} > \ldots > T'_{e_k}$.

We prove this by induction on t. For $t = 0$, there is no entry in the stack, and so this statement is true trivially. Now, suppose this is true for some time t during a phase P, and again, let T denote $\mathbf{est}(P)$. If this phase ends in Steps 1(i) or 2(i), then we push another entry e in the stack with $T_e = T'_e = T$. Further, the estimate for the next phase is strictly larger than T. Therefore, the condition continues to hold in the next phase.

Now, suppose the current phase P ends in Step 3(iv). If the stack top entry e satisfied $T'_e = T/c$, then T_e would be T or T/c. Hence, we would have popped such an entry in Step 3(ii). So when this phase ends, the top entry e in the stack would have $T'_e \leq T/c^2$. The next phase would have estimate equal to T/c. Therefore, the invariant continues to hold in the next phase as well.

Finally, suppose the current phase ends in Step 3(v). As argued above, we will pop out any entry with $T'_e = T/c$. Further, we push a new entry e with $T'_e = T/c$, and the estimate for the next phase is T. Thus the invariant holds in this case as well.

The statement of the lemma is clearly true for jobs released in a particular phase (by the properties of the **DispatchJob** algorithm. Now the above invariant implies that for any parameter T, the jobs from J_T which are still alive (i.e., waiting to be processed) can come from at most two phases. Thus, the lemma is true. ◀

Algorithm $\mathcal{B}$. We now discuss the properties of algorithm $\mathcal{B}$. We first show that the flow-time of the jobs in a particular phase is bounded by the estimate for that phase.

▶ **Lemma 6.** *Algorithm* $\mathcal{B}$ *ensures that the flow time of every job of* J_T *(which is not rejected) is at most* $O(\alpha T/\varepsilon^4)$*.*

Proof. We use the notation while discussing algorithm $\mathcal{B}$. Since the rejection weights are all 1, the rejection-weight-density class essentially becomes the same as the definition of class based on processing time only. Assuming c is a power of 2, it follows that each flow-time-weight-class corresponds to a particular value of the estimate T (since the estimates are powers of c). Now, Lemma 5 shows that for a particular class k and estimate T, and a time t and machine i, the total remaining processing time of jobs from J_T^k at time t on machine i is at most $O(\alpha T)$. Hence, their total remaining weighted-processing time is $O(\alpha)$ – note that this bound holds for all jobs (irrespective of k and T). Applying Theorem 2, we get that the

schedule $\mathcal{B}$ incurs weighted flow-time of $O(\alpha/\varepsilon^4)$ for such jobs, and so, the actual flow-time is $O(\alpha T/\varepsilon^4)$. ◀

Having bounded the flow-time of jobs in our schedule, we now give the lower bound of the corresponding quantity for the off-line optimum. Our main technical lemma 4 already lower bounds the optimum value for a specific good phase. The following claim follows easily from this result.

▶ **Claim 7.** *The total flow-time of jobs released during good phases is at most $O(1/\varepsilon^{11})$ times the optimal value.*

Proof. The proof directly follows from Lemma 4 and Lemma 6, and the fact that $|A'(P)| \leq |S| \leq |A'(P)|$, where the entry $(\mathbf{est}(P), Q, S)$ is pushed to the stack $\mathbf{S}$ at the end of the good phase P, and $|A'(P)|$ is at most the number of jobs released during P and the previous phase. ◀

It remains to bound the flow-time of jobs released during bad phases. Before this, we make an important observation.

▶ **Claim 8.** *For any phase P, the set of jobs in $Q(s(P))$, i.e., the jobs waiting in the queues of the machines at the beginning of this phase, could have only arrived during the previous phase.*

Proof. Let P' be the phase preceding to P. If P' is a good phase, then P will start with empty queues, so there is nothing to prove. If P' is a bad phase, it will remove the jobs in $Q(s(P'))$ (in Step 3(i)), and so, the only jobs which carry over to phase P must have been released during P'. ◀

▶ **Lemma 9.** *The total flow-time of jobs released during bad phases is at most $O(1/\varepsilon^{12})$-times the optimal value.*

Proof. Let $B_1, \ldots, B_l$ be a maximal sequence of bad phases, and let G_0 denote the good phase preceding B_1. In Step 3 of the algorithm $\mathcal{A}$, each of the phases B_i may pop an entry from the stack – let this entry correspond to a good phase G_i. Note that G_0 is same as G_1. We know that $\mathbf{est}(G_i) \geq \mathbf{est}(B_i)/c$. For phase P, let $N(P)$ denote the number of jobs released during that phase. For phase B_i, we must have ended it when the condition in Step 3 was satisfied. Therefore, $N(B_i)$ is at most $(N(G_i) + N(B_{i-1}))/\varepsilon$, if $i \geq 1$ (using the above claim). Lemma 6 shows that the total flow-time of jobs during $B_1, \ldots, B_l$ is at most $\sum_{i=1}^{l} N(B_i) \cdot \frac{\alpha \mathbf{est}(B_i)}{\varepsilon^4}$. If T denotes $\mathbf{est}(B_1)$, then $\mathbf{est}(B_i) = \frac{T}{c^{i-1}}$. Therefore, the total flow-time of the jobs released during $B_1, \ldots, B_l$ is at most

$$\sum_{i=1}^{l} N(B_i) \cdot \frac{\alpha T}{c^{i-1}\varepsilon^4}. \tag{1}$$

We also know that $N(B_i) \leq 1/\varepsilon \cdot (N(G_i) + N(B_{i-1}))$. Since $c = 2/\varepsilon$, we get

$$N(B_i) \cdot \frac{\alpha T}{c^{i-1}\varepsilon^4} - N(B_{i-1}) \cdot \frac{\alpha T}{2c^{i-2}\varepsilon^4} \leq N(G_i) \cdot \frac{\alpha T}{2c^{i-2}\varepsilon^4}.$$

Summing the above for all i, we get

$$\sum_i N(B_i) \cdot \frac{\alpha T}{c^{i-1}\varepsilon^4} \leq 2c \sum_i N(G_i) \cdot \frac{\alpha T}{c^{i-1}\varepsilon^4}.$$

Now observe that $\mathbf{est}(G_i) \geq \frac{T}{c^i}$, and Lemma 4 implies that the total flow-time of the jobs released during $G_1, \ldots, G_l$ is at least $\sum_i \Omega(\varepsilon^3 \cdot N(G_i) \cdot \mathbf{est}(G_i))$, which is at least $\Omega(\varepsilon^{12})$ times the total flow-time of the jobs released during $B_1, \ldots, B_l$. Now, we sum this up over all maximal sequences of bad phases, and observe that G_i is uniquely determined by B_i(a stack entry once popped never gets pushed back again). ◀

7 Extension to weighted ℓ_p norm

We first outline the steps needed to extend our results to the case where each job j has a weight w_j. Let W denote the total weight of arriving jobs, and let $T_1^\star$ denote the optimal value $T_1^\star$ of the total weighted flow-time of these jobs. It is easy to see that jobs of total weight at least $(1-\varepsilon)W$ will have (unweighted) flow-time at most $T^\star = T_1^\star/(\varepsilon W)$. Thus, we modify algorithm $\mathcal{A}$ to maintain a variable T which is supposed to track the value $T^\star = T_1^\star/(\varepsilon W)$.

We say a job j with processing time p_j, and weight w_j is of *density class* k if $\beta^k \leq \frac{w_j}{p_j} < \beta^{k+1}$, for some constant β (which will be roughly $O(1/\varepsilon)$). The job dispatch and job processing rule is same as the algorithm for the unweighted case, with the only difference that now $Q_{i,k}(t)$ and $\mathtt{load}_{i,k}(t)$ are defined for every density class k. Rest of the details of $\mathcal{A}$ remain unchanged, except for the fact that $a(P), a'(P), r(P)$ now keep track of the total weight of corresponding jobs. The algorithm $\mathcal{B}$ remains unchanged except for the fact that for a job j, it sets $w_j^{(r)}$ to its weight w_j.

We now show how our results extend to the problem of minimizing the ℓ_p norm of the flow time of the jobs, for some positive constant p. For sake of clarity, we argue about the unweighted case only, though the weighted case follows similarly. Let us assume that we know the total number of the jobs released n, and the optimal value $T_1^\star$ of the ℓ_p norm of the flow-time of these jobs. From this it follows that at least $(1-\varepsilon)$-fraction of the total number of jobs will have (unweighted) flow-time at most $T^\star = T_1^\star/(\varepsilon n)^{1/p}$. Thus in algorithm $\mathcal{A}$, we maintain a variable T which is supposed to track the value $T^\star = T_1^\star/(\varepsilon n)^{1/p}$. Also, we increase or decrease T in factors of $c = (2/\varepsilon)^{1/p}$. The rest of the algorithms $\mathcal{A}$ and $\mathcal{B}$ remains as it is for the problem of minimizing the ℓ_1 norm of the flow time of jobs.

References

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. *Algorithms for flow time scheduling.* PhD thesis, Indian Institute of Technology, Delhi, December 2013.

3 S. Anand, Karl Bringmann, Tobias Friedrich, Naveen Garg, and Amit Kumar. Minimizing maximum (weighted) flow-time on related and unrelated machines. In *ICALP (1)*, pages 13–24, 2013.

4 S. Anand, Naveen Garg, and Amit Kumar. Resource augmentation for weighted flow-time explained by dual fitting. In *SODA*, pages 1228–1241, 2012.

5 Nir Avrahami and Yossi Azar. Minimizing total flow time and total completion time with immediate dispatching. In *SPAA*, pages 11–18, 2003.

6 Yossi Azar, Joseph (Seffi) Naor, and Raphael Rom. The competitiveness of on-line assignments. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 203–210, 1992.

7 Nikhil Bansal, Avrim Blum, Shuchi Chawla, and Kedar Dhamdhere. Scheduling for flow-time with admission control. In *Proc. ESA, 2003*, 2003.

8 Nikhil Bansal and Kirk Pruhs. Server scheduling in the l_p norm: a rising tide lifts all boat. In *STOC*, pages 242–250, 2003.

9 Yair Bartal, Stefano Leonardi, Alberto Marchetti-Spaccamela, Jiri Sgall, and Leen Stougie. Multiprocessor scheduling with rejection. *SIAM J. Discrete Math.*, 13(1):64–78, 2000.

10 Jivitej S. Chadha, Naveen Garg, Amit Kumar, and V. N. Muralidhara. A competitive algorithm for minimizing weighted flow time on unrelatedmachines with speed augmentation. In *STOC*, pages 679–684, 2009.

11 Ho-Leung Chan, Sze-Hang Chan, Tak Wah Lam, Lap-Kei Lee, and Jianqiao Zhu. Non-clairvoyant weighted flow time scheduling with rejection penalty. In *SPAA*, pages 246–254, 2012.

12 Chandra Chekuri, Ashish Goel, Sanjeev Khanna, and Amit Kumar. Multi-processor scheduling to minimize flow time with epsilon resource augmentation. In *STOC*, pages 363–372, 2004.

13 Anamitra Roy Choudhury, Syamantak Das, Naveen Garg, and Amit Kumar. Rejecting jobs to minimize load and maximum flow-time. In *SODA*, pages 1114–1133, 2015.

14 Leah Epstein and Hanan Zebedat-Haider. Preemptive online scheduling with rejection of unit jobs on two uniformly related machines. *J. Scheduling*, 17(1):87–93, 2014.

15 Naveen Garg and Amit Kumar. Better algorithms for minimizing average flow-time on related machines. In *ICALP (1)*, pages 181–190, 2006.

16 Naveen Garg and Amit Kumar. Minimizing average flow-time : Upper and lower bounds. In *FOCS*, pages 603–613, 2007.

17 Daniel Golovin, Anupam Gupta, Amit Kumar, and Kanat Tangwongsan. All-norms and all-ℓ_p-norms approximation algorithms. In *FSTTCS*, pages 199–210, 2008.

18 Sungjin Im and Benjamin Moseley. Online scalable algorithm for minimizing ;k-norms of weighted flow time on unrelated machines. In *SODA*, pages 95–108, 2011.

19 Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. In *FOCS*, pages 214–221, 1995.

20 Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines. *J. Comput. Syst. Sci.*, 73(6):875–891, 2007.