Minimizing the Flow Time without Migration [*]

Baruch Awerbuch [†] Yossi Azar [‡] Stefano Leonardi [§]
Oded Regev [¶]

September 18, 2001

Abstract

We consider the classical problem of scheduling jobs in a multiprocessor setting in order to minimize the flow time (total time in the system). The performance of the algorithm, both in offline and online settings, can be significantly improved if we allow preemption: i.e., interrupt a job and later continue its execution, perhaps migrating it to a different machine. Preemption is inherent to make a scheduling algorithm efficient. While in case of a single processor, most operating systems can easily handle preemptions, migrating a job to a different machine results in a huge overhead. Thus, it is not commonly used in most multiprocessor operating systems. The natural question is whether migration is an inherent component for an efficient scheduling algorithm, in either the online or offline setting.

[*]A preliminary version of this paper appears in the proceedings of STOC'99, pp. 198-205.

[†]Johns Hopkins University, Baltimore, MD 21218, and MIT Lab. for Computer Science. E-mail: baruch@blaze.cs.jhu.edu. Supported by Air Force Contract TNDGAFOSR-86-0078, ARPA/Army contract DABT63-93-C-0038, ARO contract DAAL03-86-K-0171, NSF contract 9114440-CCR, DARPA contract N00014-J-92-1799, and a special grant from IBM.

[‡]Department of Computer Science, Tel Aviv University, Tel-Aviv, 69978, Israel. E-Mail: azar@math.tau.ac.il. Research supported in part by the Israel Science Foundation and by the US-Israel Binational Science Foundation (BSF).

[§]Dipartimento di Informatica Sistemistica, Università di Roma "La Sapienza", via Salaria 113, 00198-Roma, Italia. This work was partly supported by EU ESPRIT Long term Research Project ALCOM-IT under contract n. 20244, and by Italian Ministry of Scientific Research Project 40% "Algoritmi, Modelli di Calcolo e Strutture Informative". E-Mail: leon@dis.uniroma1.it.

[¶]Department of Computer Science, Tel Aviv University. E-Mail: odedr@math.tau.ac.il

Leonardi and Raz (STOC'97) showed that the well known algorithm, *shortest remaining processing time* (SRPT), performs within a logarithmic factor of the optimal offline algorithm. Note that SRPT must use *both* preemption and migration to schedule the jobs. It is not known if better approximation factors can be reached and thus SRPT, although it is an online algorithm, becomes the best known algorithm in the off-line setting. In fact, in the on-line setting, Leonardi and Raz showed that no algorithm can achieve a better bound.

Without migration, no (offline or online) approximations are known. This paper introduces a new algorithm that does not use migration, works *online*, and is just as effective (in terms of approximation ratio) as the best known *offline* algorithm that uses migration.

1 Introduction

Objectives. One of the most basic performance measures in multiprocessor scheduling problems is the overall time the jobs are spending in the system. This includes the delay of waiting for service as well as the actual service time. This measure captures the overall quality of service of the system. Multiprocessor scheduling problems arise, for example, in the context of server farms accommodating requests for retrieving Web contents over the Internet.

We consider the classical problem of minimizing the flow time in a multiprocessor setting with jobs which are released over time. The performance of the algorithm, both in offline and online settings, can be significantly improved if we allow preemption: i.e., interrupt a job and later continue its execution, perhaps migrating it to a different machine. As shown below, preemption is inherent to make a scheduling algorithm efficient.

While in case of a single processor, most operating systems can easily handle preemptions, migrating a job to a different machine results in a huge overhead. Thus, it is not commonly used in most multiprocessor operating systems. The natural question is whether migration is an inherent component for an efficient scheduling algorithm, in either online or offline setting.

Existing work. Surveys on approximation algorithms for scheduling can be found in [4, 7]. In the non-preemptive case it is impossible to achieve a "reasonable" approximation. Specifically, even for one machine one cannot achieve an approximation factor of $O(n^{\frac{1}{2}-\epsilon})$ unless $NP = P$ where n is the number of jobs [6]. For $m > 1$ it is impossible to achieve $O(n^{\frac{1}{3}-\epsilon})$ approximation factor unless $NP = P$ [8]. Thus, preemptions really seem to be essential.

Existing work: single processor. Minimizing the flow time on one machine with preemption can be done optimally in polynomial time using the natural algorithm shortest remaining processing time (SRPT) [1].

Existing work: multiple processors with migration. For more than one machine the preemptive problem becomes NP-hard [2]. Only very recently, Leonardi and Raz [8] (STOC'97) showed that SRPT achieves logarithmic approximation for the multiprocessor case, showing a tight bound of $O(\log(\min\{n/m, P\}))$ on $m > 1$ machines with n jobs, where P denotes the ratio between the processing time of the longest and the shortest jobs.

Note that SRPT must use *both* preemption and migration to schedule the jobs. In the offline setting, it is not known if better approximation factors can be reached. In fact, in the on-line setting SRPT is optimal, i.e., no algorithm can achieve a better bound up to a constant factor[8]. For the easier problem of minimizing the total completion time a constant approximation can be achieved [3].

Existing work: multiple processors without migration. In a recent paper by Kalyanasundaram and Pruhs [5] an algorithm is shown that converts any multiprocessor preemptive schedule to a non-migratory one. However, it is assumed that the number of processors available is a constant factor more than in the original schedule. An important property of their algorithm is that the jobs are scheduled to begin later than their original start time and are completed before their original completion time. Thus, the total flow time is not increased by their transformation. By combining their methods with a migratory algorithm such as $SRPT$, one can achieve a logarithmic factor approximation with the assumption that a constant factor more machines are available.

Our result: multiple processors without migration. Without migration, no (offline or online) approximations are known (without increasing the number of processors). We present in Section 2 a new algorithm for minimizing flow time that uses local preemption, but does not migrate jobs between machines. We show in Section 3 that our algorithm performs as well as the best known *offline* algorithm (SRPT) for the preemptive problem that uses migration. More specifically, our algorithm guarantees, on all input instances, a small performance gap in comparison to the optimal *offline* schedule that allows *both* preemption and migration. Denote by P the ratio between the processing time of the longest and the shortest jobs. Then our algorithm performs by at most $O(\min\{\log P, \log n\})$ factor of the optimal flow time of any (possibly migratory) schedule. The algorithm can be easily implemented in polynomial time in the size of the input instance.

Our algorithm is also on-line. We note that in the proof of the $\Omega(\log P)$, $\Omega(\log(n/m))$ lower bounds of [8] for on-line algorithms, the optimal algorithm does not use migration. Hence, the (randomized) lower bound holds also for a non-migratory algorithm. This implies that our algorithm is optimal with respect to the parameter P, i.e., no on-line algorithm can achieve a better bound both when the off-line algorithm is or is not allowed to migrate jobs (the first claim is obviously stronger), while there is still a small gap between the $O(\log n)$ upper bound and the $\Omega(\log(n/m))$ lower bound for large m. An $\Omega(\log n)$ competitive lower bound for our algorithm is also proved in Section 4 for any m.

Our algorithm and its analysis draw on some ideas from the SRPT algorithm [8]. Moreover, we eliminated several difficulties in the analysis by basing the analysis on classes of jobs. Basically, our algorithm prefers short jobs over long jobs. However, unlike SRPT, our algorithm may continue to run a job on a machine even when a shorter job is waiting to be processed. This seems essential in the non-migratory setting since being too eager to run a shorter job may result in an unbalanced commitment to machines. A non-migratory algorithm has to trade off between the commitment of a job to a machine and the decrease in the flow time yielded by running a shorter job. Our algorithm runs the job with the shortest remaining processing time among all the jobs that were already assigned to that machine, and a new job is assigned to a machine if its processing time is considerably shorter than the job that is currently running.

The model: We are given a set J of n jobs and a set of m identical machines. Each job j is assigned a pair (r_j, p_j) where r_j is the release time of the job and p_j is its processing time. In the preemptive model a job that is running can be preempted and continued later on any machine. Our model allows preemption but does not allow migration, i.e., a job that is running can be preempted but must later continue its execution on the same machine on which its execution began. The scheduling algorithm decides which of the jobs should be executed at each time. Clearly a machine can process at most one job in any given time and a job cannot be processed before its release time. For a given schedule define c_j to be the completion time of job j in this schedule. The flow time of job j for this schedule is $F_j = c_j - r_j$. The total flow time is $\sum_{j \in J} F_j$. The goal of the scheduling algorithm is to minimize the total flow time for each given instance of the problem. In the off-line version of the problem all the jobs are known in advance. In the on-line version of the problem each job is introduced at its release time and the algorithm bases its decision only upon the jobs that were already released.

2 The algorithm

A job is called alive at time t for a given schedule if it has already been released but has not been completed yet. Our algorithm classifies the jobs that are alive into classes according to their remaining processing times. A job j whose remaining processing time is in $[2^k, 2^{k+1})$ is in class k for $-\infty < k < \infty$. Notice that a given job changes its class during its execution. The algorithm holds a pool of jobs that are alive and have not been processed at all. In addition, the algorithm holds a stack of jobs for each of the machines. The stack of machine i holds jobs that are alive and have already been processed by machine i. The algorithm works as follows:

- Each machine processes the job at the top of its stack.
- When a new job arrives the algorithm looks for a machine that is idle or currently processing a job of a higher class than the new job. In case it finds one, the new job in pushed into the machine's stack and its processing begins. Otherwise, the job is inserted into the pool.
- When a job is completed on some machine it is popped from its stack. The algorithm compares the class of the job at the top of the stack with the minimum class of a job in the pool. If the minimum is in the pool then a job that achieves the minimum is pushed into the stack (and removed from the pool).

Clearly, when a job is assigned to a machine it will be processed only on that machine and thus the algorithm does not use migration. In fact, the algorithm bases its decisions only on the jobs that were released up to the current time and hence is an on-line algorithm. Note that it may seem that the algorithm has to keep track of all the infinite number of classes through which a job evolves. However, the algorithm recalculates the classes of jobs only at arrival or completion of a job.

3 Analysis

We denote by A our scheduling algorithm and by OPT the optimal off-line algorithm that minimizes the flow time for any given instance. For our analysis we can even assume that OPT may migrate jobs between machines. Whenever we talk about time t we mean the moment after the events of time t happened. For a given scheduling algorithm S we define $V^S(t)$ to be the volume of a schedule at a certain time t. This volume is the sum of all the remaining processing times of jobs that are alive. In addition, we define $\delta^S(t)$ to be the number of jobs that are alive. $\Delta V(t)$ is defined to be the volume difference between our algorithm and the optimal algorithm, i.e., $V^A(t) - V^{OPT}(t)$. We also define by $\Delta\delta(t) = \delta^A(t) - \delta^{OPT}(t)$ the alive jobs difference at time t between A and OPT. For a generic function f (V, ΔV, δ or $\Delta\delta$) we use $f_{\geq h, \leq k}(t)$ to denote the value of f at time t when restricted to jobs of classes between h and k. Similarly, the notation $f_{=k}(t)$ will represent the value of function f at time t when restricted to jobs of class precisely k.

Let $\gamma^S(t)$ be the number of non-idle machines at time t. Notice that because our algorithm does not migrate jobs, there are situations in which $\gamma^A(t) < m$ and $\delta^A(t) \geq m$. We denote by $\mathcal{T}$ the set of times in which $\gamma^A(t) = m$, that is, the set of times in which none of the machines is idle. Denote by P_{min} the processing time of the shortest job and by P_{max} the processing time of the longest job. Note that $P = P_{max}/P_{min}$. Denote by $k_{min} = \lfloor \log P_{min} \rfloor$ and $k_{max} = \lfloor \log P_{max} \rfloor$ the classes of the shortest and longest jobs upon their arrival.

We start by observing the simple fact that the flow time is the integral over time of the number of jobs that are alive (for example, see [8]):

Fact 3.1 *For any scheduler S,*

$$F^S = \int_t \delta^S(t) dt.$$

First we note that the algorithm preserves the following property of the stacks:

Lemma 3.2 *In each stack the jobs are ordered in a strictly increasing class order and there is at most one job whose class is at most k_{min}.*

Proof: At time $t = 0$ the lemma is true since all the stacks are empty. The lemma is proved by induction on time. The classes of jobs in the stacks change in one of three cases. The first is when the class of the currently processed job decreases. Since the currently processed job is the job with the lowest class, the lemma remains true. The second case is when a new job arrives. In case it enters the pool there is no change in any stack. Otherwise it is pushed into a stack whose top is of a higher class which preserves the first part of the lemma. Since the class of the new job is at least k_{min}, the second part of the lemma remains true as well. The third case is when a job is completed on some machine. If no job is pushed into the stack of that machine the lemma remains easily true. If a new job is pushed to the stack then the lemma remains true in much the same way as in the case of the arrival of a new job. ■

Corollary 3.3 *There are at most $2 + \log P$ jobs in each stack.*

Proof: The number of classes of jobs in each stack is at most $k_{max} - k_{min} + 1 \leq 2 + \log P$. ■

We look at the state of the schedule at a certain time t. First let's look at $t \notin \mathcal{T}$:

Lemma 3.4 *For $t \notin \mathcal{T}$, $\delta^A(t) \leq \gamma^A(t)(2 + \log P)$.*

Proof: By definition of $\mathcal{T}$, at time t at least one machine is idle. This implies that the pool is empty. Moreover, all the stacks of the idle machines are obviously empty. So, all the jobs that are alive are in the stacks of the non-idle machines. The number of non-idle machines is $\gamma^A(t)$ and the number of jobs in each stack is at most $2 + \log P$ according to corollary 3.3 ■

Now, assume that $t \in \mathcal{T}$ and let $\hat{t} < t$ be the earliest time for which $[\hat{t}, t) \subset \mathcal{T}$. We denote the last time in which a job of class more than k was processed by t_k. In case such jobs were not processed at all in the time interval $[\hat{t}, t)$ we set $t_k = \hat{t}$. So, $\hat{t} \leq t_{k_{max}} \leq t_{k_{max-1}} \leq ... \leq t_{k_{min}} \leq t$.

Lemma 3.5 *For $t \in \mathcal{T}$, $\Delta V_{\leq k}(t) \leq \Delta V_{\leq k}(t_k)$.*

Proof: Notice that in the time interval $[t_k, t)$, algorithm A is constantly processing on all the machines jobs whose class is at most k. The off-line algorithm may process jobs of higher classes. Moreover, that can cause jobs of class more than k to actually lower their classes to k and below therefore adding even more to $V_{\leq k}^{OPT}(t)$. Finally, the release of jobs of class $\leq k$ in the interval $[t_k, t)$ is not affecting $\Delta V_{\leq k}(t)$. Therefore, the difference in volume between the two algorithms cannot increase. ■

Lemma 3.6 *For $t \in \mathcal{T}$, $\Delta V_{\leq k}(t_k) \leq m2^{k+2}$.*

Proof: First we claim that at any moment $t_k - \epsilon$, for any $\epsilon > 0$ small enough, the pool does not contain jobs whose class is at most k. In case $t_k = \hat{t}$, at any moment just before t_k there is at least one idle machine which means the pool is empty. Otherwise, $t_k > \hat{t}$ and by definition we know that a job of class more than k is processed just before t_k. Therefore, the pool does not contain any job whose class is at most k.

At time t_k jobs of class at most k might arrive and fill the pool. However, these jobs increase both $V_{\leq k}^{OPT}(t_k)$ and $V_{\leq k}^{A}(t_k)$ by the same amount, so jobs that arrive exactly at t_k do not change $\Delta V_{\leq k}(t_k)$ and can be ignored.

Since the jobs in the pool at time t_k can be ignored, we are left with the jobs in the stacks. Using Lemma 3.2, $\Delta V_{\leq k}(t_k) \leq m(2^{k+1} + 2^k + 2^{k-1} + \ldots) \leq m2^{k+2}$. ■

Lemma 3.7 *For $t \in \mathcal{T}$, $\Delta V_{\leq k}(t) \leq m2^{k+2}$.*

Proof: Combining Lemma 3.5 and 3.6, we obtain $\Delta V_{\leq k}(t) \leq \Delta V_{\leq k}(t_k) \leq m2^{k+2}$ ■

The claim of the following lemma states a property that will be used in the proof of both the $O(\log P)$ and the $O(\log n)$ approximation results.

Lemma 3.8 *For $t \in \mathcal{T}$, for $k_{min} \leq k_1 \leq k_2 \leq k_{max}$, $\delta^A_{\geq k_1, \leq k_2}(t) \leq 2m(k_2 - k_1 + 2) + 2\delta^{OPT}_{\leq k_2}(t)$.*

Proof: $\delta^A_{\geq k_1, \leq k_2}(t)$ can be expressed as:

$$\begin{aligned}
\sum_{i=k_1}^{k_2} \delta^A_{=i}(t) &\leq \sum_{i=k_1}^{k_2} \frac{\Delta V_{=i}(t) + V^{OPT}_{=i}(t)}{2^i} \\
&\leq \sum_{i=k_1}^{k_2} \frac{\Delta V_{\leq i}(t) - \Delta V_{\leq i-1}(t)}{2^i} + 2\delta^{OPT}_{\geq k_1, \leq k_2}(t) \\
&\leq \frac{\Delta V_{\leq k_2}(t)}{2^{k_2}} + \sum_{i=k_1}^{k_2-1} \frac{\Delta V_{\leq i}(t)}{2^{i+1}} - \frac{\Delta V_{\leq k_1-1}(t)}{2^{k_1}} + 2\delta^{OPT}_{\geq k_1, \leq k_2}(t) \\
&\leq 4m + \sum_{i=k_1}^{k_2-1} 2m + \delta^{OPT}_{\leq k_1-1}(t) + 2\delta^{OPT}_{\geq k_1, \leq k_2}(t) \\
&\leq 2m(k_2 - k_1 + 2) + 2\delta^{OPT}_{\leq k_2}(t).
\end{aligned}$$

The first inequality follows since 2^i is the minimum processing time of a job of class i. The second inequality follows since the processing time of a job of class i is less than 2^{i+1}. The fourth inequality is derived by applying Lemma 3.7, observing that $\Delta V_{\leq k_1-1}(t) \geq -V^{OPT}_{\leq k_1-1}(t)$ and that 2^{k_1} is the maximum processing time of a job of class at most $k_1 - 1$. The claim of the lemma then follows. ■

The following corollary of Lemma 3.8 is used in the proof of the $O(\log P)$ approximation ratio of Theorem 3.10

Corollary 3.9 *For $t \in \mathcal{T}$, $\delta^A(t) \leq 2m(4 + \log P) + 2\delta^{OPT}(t)$.*

Proof: We express

$$\begin{aligned}\delta^A(t) &= \delta^A_{\leq k_{max}, \geq k_{min}}(t) + \delta^A_{<k_{min}}(t)\\ &\leq 2m(k_{max} - k_{min} + 2) + 2\delta^{OPT}(t) + m\\ &\leq 2m(4 + \log P) + 2\delta^{OPT}(t)\end{aligned}$$

The second inequality follows from the claim of Lemma 3.8 when $k_2 = k_{max}$ and $k_1 = k_{min}$, and from the claim of Lemma 3.2 stating that the stack of each machine contains at most one job of class less than k_{min}. The third inequality is obtained since $k_{max} - k_{min} + 5/2 \leq \log P + 4$. ■

Theorem 3.10 *$F^A \leq 2(5 + \log P) \cdot F^{OPT}$, that is, algorithm A has a $2(5 + \log P)$ approximation factor even compared to the flow time of the (possibly migratory) schedule of the optimal off-line algorithm.*

Proof:

$$\begin{aligned}F^A &= \int_t \delta^A(t)dt\\ &= \int_{t\notin\mathcal{T}} \delta^A(t)dt + \int_{t\in\mathcal{T}} \delta^A(t)dt\\ &\leq \int_{t\notin\mathcal{T}} \gamma^A(t)(2 + \log P)dt + \int_{t\in\mathcal{T}} (2m(4 + \log P) + 2\delta^{OPT}(t))dt\\ &\leq (2 + \log P)\int_{t\notin\mathcal{T}} \gamma^A(t)dt + 2(4 + \log P)\int_{t\in\mathcal{T}} mdt + 2\int_{t\in\mathcal{T}} \delta^{OPT}(t)dt\\ &\leq (8 + 2\log P)\int_t \gamma^A(t)dt + 2\int_t \delta^{OPT}(t)dt\\ &\leq 2(5 + \log P) \cdot F^{OPT}\end{aligned}$$

The first equality is from the definition of F^A. The second is obtained by looking at the time in which none of the machines is idle and the time in which at least one machine is idle separately. The third inequality uses Lemma 3.4 and Corollary 3.9. The fifth inequality is true since $\gamma^A(t) = m$ when $t \in \mathcal{T}$. Finally, $\int_t \gamma^A(t)dt$ is the total time spent processing jobs by the machines which is exactly $\sum_{j\in J} p_j$. That sum is upper bounded by the flow time of OPT since each job's flow time must be at least its processing time.

■

We now turn to prove the $O(\log n)$ approximation ratio of the algorithm. A different argument is required to prove this second bound. The main idea behind the proof of the

$O(\log P)$ approximation ratio was to bound for any time $t \in \mathcal{T}$, the alive jobs difference between A and OPT by $O(m \log P)$. A similar approach does not allow to prove the $O(\log n)$ approximation ratio: It is possible to construct instances where the alive jobs difference is $\Omega(n)$. The proof that follows uses ideas drawn from the proof given by Leonardi and Raz [8] of the $O(\log(n/m))$ approximation ratio for SRPT when migration is allowed. The idea behind the proof is that the worst case ratio between the algorithm's flow time and the optimal flow time can be raised only if a "big" alive jobs difference is kept for a "long" time period.

We need to introduce more notation. Recall that $\mathcal{T}$ is defined to be the set of times in which $\gamma^A(t) = m$. We denote by $T = \int_{t \in \mathcal{T}} dt$ the size of set $\mathcal{T}$. For any $t \in \mathcal{T}$, define by $\delta^{A,P}(t)$ the number of jobs in the pool of algorithm A, i.e., not assigned to a machine, at time t, and by $\Delta\delta^P(t) = \delta^{A,P}(t) - 2\delta^{OPT}(t)$ the difference between the number of jobs in the pool of algorithm A and twice the number of jobs not finished by the optimal algorithm. For any machine l, time t, define by $\delta^{A,l}(t)$ the number of jobs assigned to machine l at time t in the schedule of algorithm A. Moreover, define by $\mathcal{T}^l = \{t | \delta^{A,l}(t) > 0\}$, the set of times when machine l is assigned with at least one job, and by $T^l = \int_{t \in \mathcal{T}^l} dt$ the size of set $\mathcal{T}^l$.

Lemma 3.11 *For any time $t \in \mathcal{T}$, if $\Delta\delta^P(t) \geq 2mi$, for $i \geq 3$, then the pool of algorithm A contains at least $2m$ jobs of remaining processing time at most $\frac{V^A(t)}{m2^{i-3}}$.*

Proof: Let k_{high} be the maximum integer such that $\delta^{A,P}_{\geq k_{high}}(t) \geq 2m$ and let k_{low} be the maximum integer such that $\delta^{A,P}_{<k_{low}}(t) < 2m$. Note that both numbers are well defined and $k_{low} \leq k_{high}$ since there are at least $6m > 2m$ jobs in the pool. Then,

$$2m \leq \delta^{A,P}_{\geq k_{high}}(t) \quad \leq \quad \delta^{A}_{\geq k_{high}}(t) \leq \frac{V^A(t)}{2^{k_{high}}}$$

thus yielding $2^{k_{high}} \leq \frac{V^A(t)}{2m}$. In particular, the last inequality follows since $2^{k_{high}}$ is the minimum processing time of a job of class k_{high}.

By the definition of k_{high}, we have:

$$\begin{aligned}
\Delta\delta^P_{\leq k_{high}}(t) &= \delta^{A,P}_{\leq k_{high}}(t) - 2\delta^{OPT}_{\leq k_{high}}(t) \\
&= \delta^{A,P}(t) - \delta^{A,P}_{>k_{high}}(t) - 2(\delta^{OPT}(t) - \delta^{OPT}_{>k_{high}}(t)) \\
&= \Delta\delta^P(t) - \delta^{A,P}_{>k_{high}}(t) + 2\delta^{OPT}_{>k_{high}}(t) \\
&\geq 2m(i-1).
\end{aligned}$$

where the last inequality follows since $\delta^{A,P}_{>k_{high}}(t) < 2m$.

From Lemma 3.8, we get:

$$\begin{aligned}
\Delta\delta^{P}_{\leq k_{high}}(t) &= \delta^{A,P}_{\leq k_{high},\geq k_{low}}(t) + \delta^{A,P}_{<k_{low}}(t) - 2\delta^{OPT}_{\leq k_{high}}(t) \\
&\leq \delta^{A}_{\leq k_{high},\geq k_{low}}(t) + \delta^{A,P}_{<k_{low}}(t) - 2\delta^{OPT}_{\leq k_{high}}(t) \\
&\leq \delta^{A,P}_{<k_{low}}(t) + 2m(k_{high} - k_{low} + 2),
\end{aligned}$$

thus yielding $\delta^{A,P}_{<k_{low}}(t) \geq 2m(i + k_{low} - k_{high} - 3)$. Therefore, we get that $2m > 2m\ (i + k_{low} - k_{high} - 3)$ and thus $k_{low} \leq k_{high} - i + 3$. It follows that there exist at least $2m$ jobs of class at most $k_{low} \leq k_{high} - i + 3$ in the pool of the algorithm. The remaining processing time of these $2m$ jobs is bounded by $2^{k_{high}-i+4} \leq \frac{V^A(t)}{m2^{i-3}}$, thus proving the claim. ■

Lemma 3.12 *For any machine l, time $t \in \mathcal{T}^l$, if $\delta^{A,l}(t) \geq i$ for $i \geq 1$ then there exists a job with remaining processing time at most $\frac{T^l}{2^{i-2}}$ assigned to machine l at time t.*

Proof: For any time $t \in \mathcal{T}^l$, there is at most one job assigned to machine l for every specific class. Assume k to be the highest class of a job assigned to machine l, obviously satisfying $T^l \geq 2^k$. If $\delta^{A,l}(t) \geq i$ then there is a job of class at most $k - i + 1$ assigned to machine l, with processing time at most $2^{k-i+2} \leq \frac{T^l}{2^{i-2}}$. ■

A this stage of the exposition we give a brief overview of the proof. Roughly speaking, the two Lemmas 3.11 and 3.12 show that if the alive jobs difference between the algorithm and the optimum is order of mi, then the remaining processing time of the jobs on execution is proportional to $1/2^i$. This implies, as we will show in Lemmas 3.14 and 3.15, that one new job is released for every non-idle machine in every interval of size proportional to $1/2^i$. This fact is exploited in Theorem 3.17 to argue that the integral of the alive jobs difference between A and OPT (e.g. order of mi) is logarithmic in the number of jobs that are released (i.e. order of 2^i).

We partition the set of time instants $\mathcal{T}$ when no machine is idle into a collection of disjoint intervals $I_k = [t_k, r_k)$, $k = 1, \ldots, s$, and associate an integer i_k with each interval, such that for any time $t \in I_k$, $2m(i_k - 1) < \Delta\delta^P(t) < 2m(i_k + 2)$.

Each maximal time interval $[t_b, t_e)$ contained in $\mathcal{T}$ is dealt with separately. Assume we already dealt with all times in $\mathcal{T}$ which are smaller than t_b, and we have created $k-1$ intervals. We then define $t_k = t_b$. Given t_k we choose $i_k = \lfloor \frac{\Delta\delta^P(t_k)}{2m} \rfloor$. Given an interval's t_k and i_k, we define $r_k = min\{t_e, t | t > t_k, \Delta\delta^P(t) \geq 2m(i_k + 2)$ or $\Delta\delta^P(t) \leq 2m(i_k - 1)\}$, that is, r_k is the first time $\Delta\delta^P(t)$ reaches the value $2m(i_k+2)$, or the value $2m(i_k-1)$. As long as $r_k < t_e$, we continue with the next interval beginning at $t_{k+1} = r_k$.

Observation 3.13 *When an interval k begins, $2mi_k \leq \Delta\delta^P(t_k) < 2m(i_k+1)$. When it ends, either $\Delta\delta^P(r_k) \leq 2m(i_k-1)$, $\Delta\delta^P(r_k) \geq 2m(i_k+2)$ or $\Delta\delta^P(r_k) \leq 0$.*

Proof: The first part is clear by the way we choose i_k. The second part is clear when r_k is not equal to some t_e, that is, $t_k \in \mathcal{T}$. Otherwise, $r_k \notin \mathcal{T}$ and therefore $\delta^{A,P}(r_k) = 0$ and $\Delta\delta^P(r_k) \leq 0$. ■

Denote by $x_k = r_k - t_k$ the size of interval I_k, and define $\mathcal{T}_i = \{\cup I_k | i_k = i\}$, $i \geq 1$, as the union of the intervals I_k with $i_k = i$. We indicate by $T_i = \int_{t \in \mathcal{T}_i} dt$ the size of set $\mathcal{T}_i$. We also denote by $D = \max\{T, \max_{t\in\mathcal{T}}\{V^A(t)/m\}\}$. The following lemma relates the number of jobs, n, and the values of T_i.

Lemma 3.14 *The following lower bound holds for the number of jobs:*

$$n \geq \frac{m}{8D} \sum_{i\geq 4} T_i \; 2^{i-5}.$$

Proof: We prove the lemma by associating with every interval a set of jobs that have been, during the interval itself, either released, or completed by OPT, or moved to execution or completed by algorithm A. Consider an interval I_k, with a corresponding $i_k \geq 4$. According to Observation 3.13, when the interval starts $\Delta\delta^P(t_k)$ is between $2mi_k$ and $2m(i_k+1)$. This interval ends when $\Delta\delta^P(r_k)$ goes above $2m(i_k+2)$ or below $2m(i_k-1)$ (it might also reach 0 but $0 < 2m(i_k-1)$). In the first case we have the evidence of at least m jobs finished by OPT (recall that $\Delta\delta^P(t) = \delta^{A,P}(t) - 2\delta^{OPT}(t)$). In the second case we have the evidence of at least $2m$ jobs that either leave the pool to be assigned to a machine by algorithm A or arrive to both A and OPT. In both cases we charge $n_k \geq m$ jobs to interval I_k. We can then conclude with a first lower bound $n_k \geq m$ on the number of jobs charged to any interval $I_k \in \mathcal{T}_i$, $i_k \geq 4$.

Next, we give a second lower bound, based on Lemma 3.11, stating that during an interval $I_k = [t_k, r_k)$ there exist in the pool $2m$ jobs with remaining processing time at most $\frac{D}{2^{i_k-4}}$, since $\Delta\delta^P(t) > 2m(i_k-1)$ for any $t \in [t_k, r_k)$. This implies that all the m machines are processing jobs with remaining processing time at most $\frac{D}{2^{i_k-5}}$. We look at any subinterval of I_k of length $\frac{D}{2^{i_k-5}}$. For each machine, during this subinterval, either a job is finished by the algorithm or a job is preempted by a job of lower class. Therefore, we can charge at least m jobs that are either released or finished with any subinterval of size $\frac{D}{2^{i_k-5}}$ of I_k. A second lower bound on the number of jobs charged to any interval is then given by $n_k \geq m\lfloor\frac{x_k 2^{i_k-5}}{D}\rfloor$.

Observe now that each job is charged at most 4 times, when it is released, when it is assigned to a machine by A, when it is finished by A and when it is finished by OPT. Then,

$$\frac{m}{2D}\sum_{i\geq 4} T_i\ 2^{i-5} \leq \sum_{k|i_k\geq 4} m\ \max\{1, \lfloor\frac{x_k 2^{i_k-5}}{D}\rfloor\} \leq 4n,$$

where the first inequality is obtained by summing over I_k's instead of over T_i's and the simple fact that $\alpha \leq \max\{1, \lfloor 2\alpha\rfloor\}$ and the second inequality follows from the lower bounds we have shown on the charged jobs. The lemma easily follows. ■

We next bound the number of jobs that are assigned to a machine l during the time instants of $\mathcal{T}^l$. We partition the set of time instants $\mathcal{T}^l$ into a set of disjoint intervals $I_k^l = [t_k^l, r_k^l)$, $k = 1, \ldots, s^l$, and associate an integer i_k^l with each interval, such that for any time $t \in I_k^l$, $\delta^{A,l}(t) = i_k^l$. Consider a maximal interval of times $[t_b^l, t_e^l)$ contained in $\mathcal{T}^l$. Assume $t_k^l = t_b^l$. Given t_k^l and i_k^l, define $r_k^l = min\{t_e^l, t | t > t_k^l, \delta^{A,l}(t) \neq i_k^l\}$. In case $r_k^l < t_e^l$, we set $t_{k+1}^l = r_k^l$. Denote by $x_k^l = r_k^l - t_k^l$ the size of interval I_k^l. Define by $\mathcal{T}_i^l = \{\cup I_k^l | i_k^l = i\}$, $i \geq 1$, the union of the intervals I_k^l when the number of jobs assigned to machine l is exactly i, and by $T_i^l = \int_{t\in\mathcal{T}_i^l} dt$ the size of set $\mathcal{T}_i^l$.

Lemma 3.15 *The following lower bound holds for the number of jobs assigned to machine l:*

$$n^l \geq \frac{1}{4T^l}\sum_{i\geq 1} T_i^l\ 2^{i-2}.$$

Proof: For each interval I_k^l we charge at least one job. Every job will be charged at most twice, when it is assigned to a machine by A and when it is finished by A.

Consider the generic interval I_k^l, with the corresponding i_k^l. The interval starts when $\delta^{A,l}(t_k)$ reaches i_k^l, from above or from below. The interval ends when $\delta^{A,l}(r_k)$ reaches $i_k^l + 1$ or $i_k^l - 1$. In the first case we have the evidence of one job that is assigned by A to machine l and in the second case of one job that is finished by A. In both cases we charge one job to interval I_k^l.

Next we give a second lower bound, based on Lemma 3.12. Lemma 3.12 states that during an interval $I_k^l = [t_k^l, r_k^l)$, machine l is constantly assigned with a job of remaining processing time at most $\frac{T^l}{2^{i_k^l-2}}$. We look at any subinterval of I_k^l of length $\frac{T^l}{2^{i_k^l-2}}$. During this subinterval, either a job is finished by the algorithm or a job is preempted by a job of a lower class. In any case, a job that is assigned or finished during any subinterval of size $\frac{T^l}{2^{i_k^l-2}}$ is charged. A second lower bound on the number of jobs charged to any interval is then given by $n_k \geq \lfloor \frac{x_k^l 2^{i_k^l-2}}{T^l} \rfloor$.

Observe now that each job is considered at most twice, when it is assigned to machine l and when it is finished by A. Then, from the following inequalities:

$$\frac{1}{2T^l} \sum_{i \geq 1} T_i^l \; 2^{i-2} \leq \sum_{k | i_k^l \geq 1} \max\{1, \lfloor \frac{x_k^l 2^{i_k^l-2}}{T^l} \rfloor\} \leq 2n^l,$$

the claim follows. ■

Before completing the proof, we still need a simple mathematical lemma:

Lemma 3.16 *Given a sequence $a_1, a_2, ...$ of non-negative numbers such that $\sum_{i \geq 1} a_i \leq A$ and $\sum_{i \geq 1} 2^i a_i \leq B$ then $\sum_{i \geq 1} i a_i \leq \log(4B/A)A$.*

Proof: Define a second sequence, $b_i = \sum_{j \geq i} a_j$ for $i \geq 1$. Then it is known that $A \geq b_1 \geq b_2 \geq ....$ Also, it is known that $\sum_{i \geq 1} 2^i a_i = \sum_{i \geq 1} 2^i (b_i - b_{i+1}) = \frac{1}{2} \sum_{i \geq 1} 2^i b_i + b_1$. This implies that $\sum_{i \geq 1} 2^i b_i \leq 2B$.

The sum we are trying to upper bound is $\sum_{i \geq 1} b_i$. This can be viewed as an optimization problem where we try to maximize $\sum_{i \geq 1} b_i$ subject to $\sum_{i \geq 1} 2^i b_i \leq 2B$ and $b_i \leq A$ for $i \geq 1$. This corresponds to the maximization of a continuous function in a compact domain and any feasible point where $b_i < A, b_{i+1} > 0$ is dominated by the point we get by replacing b_i, b_{i+1} with $b_i + 2\epsilon, b_{i+1} - \epsilon$. Therefore, it is upper bounded by assigning $b_i = A$ for $1 \leq i \leq k$ and $b_i = 0$ for $i > k$ where k is large enough such that $\sum_{i \geq 1} 2^i b_i \geq 2B$. A choice of $k = \lceil \log(2B/A) \rceil$ is adequate and the sum is upper bounded by kA from which the result follows. ■

Theorem 3.17 *$F^A = O(\log n) F^{OPT}$, that is, algorithm A has an $O(\log n)$ approximation ratio even compared with the flow time of the (possibly migratory) schedule of the optimal off-line algorithm.*

Proof:

$$F^A = \int_t \delta^A(t) dt$$

$$
\begin{aligned}
&= \sum_{l=1}^{m} \int_t \delta^{A,l}(t)dt + \int_{t\in\mathcal{T}} \delta^{A,P}(t)dt \\
&= \sum_{l=1}^{m} \int_{t\in\mathcal{T}^l} \delta^{A,l}(t)dt + \int_{t\in\mathcal{T}} (2\delta^{OPT}(t) + \Delta\delta^{P}(t))dt \\
&\le \sum_{l=1}^{m} \int_{t\in\mathcal{T}^l} \delta^{A,l}(t)dt + 2F^{OPT} + \sum_i \int_{t\in\mathcal{T}_i} 2m(i+2)dt \\
&\le \sum_{l=1}^{m} \sum_{i\ge 1} i\ T_i^l + 2F^{OPT} + 2m\sum_i (i+2)\ T_i \\
&= \sum_{l=1}^{m} \sum_{i\ge 1} i\ T_i^l + 2F^{OPT} + 2m\sum_i (i-3)\ T_i + 2m\sum_i 5\ T_i \\
&\le \sum_{l=1}^{m} \sum_{i\ge 1} i\ T_i^l + 2F^{OPT} + 2m\sum_{i\ge 4} (i-3)\ T_i + 10F^{OPT} \\
&\le \sum_{l=1}^{m} \sum_{i\ge 1} i\ T_i^l + 12F^{OPT} + 2m\sum_{i\ge 1} i\ T_{i+3}
\end{aligned}
$$

The second equality is obtained by separately considering the contribution of every machine l, and the contribution of the jobs in the pool. The fourth inequality is obtained by partitioning $\mathcal{T}$ into the $\mathcal{T}_i$'s, such that at any time $t \in \mathcal{T}_i$, $\Delta\delta^P(t) < 2m(i+2)$. The seventh inequality is obtained by observing that $m\sum_i T_i \le \sum_j p_j \le F^{OPT}$, since all machines are busy processing jobs at any time $t \in \mathcal{T}$.

We upper bound $\sum_{i\ge 1} i\ T_{i+3}$ under the constraint $\sum_{i\ge 1} T_{i+3} \le T \le D$, and the constraint on n given by Lemma 3.14. By choosing $a_i = T_{i+3}$ we see that $\sum_{i\ge 1} a_i \le D$ and $\sum_{i\ge 1} 2^i a_i \le 32\frac{n}{m}D$. These two bounds together with Lemma 3.16 result in the upper bound $\sum_{i\ge 1} ia_i \le (7+\log\frac{n}{m})D$. A similar argument is used for the other sum. We choose $a_i = T_i^l$ instead. First note that $\sum_{i\ge 1} a_i \le T^l$. Then, according to Lemma 3.15, $\sum_{i\ge 1} 2^i a_i \le 16n^l T^l$. The upper bound by Lemma 3.16 is $\sum_{i\ge 1} ia_i \le (6+\log n^l)T^l$.

We finally express the total flow time of algorithm A as:

$$
\begin{aligned}
F^A &\le \sum_{l=1}^{m} \sum_{i\ge 1} iT_i^l + 12F^{OPT} + 2m\sum_{i\ge 1} i\ T_{i+3} \\
&\le \sum_{l=1}^{m} O(T^l \log n^l) + 12F^{OPT} + O(mD\log\frac{n}{m}) \\
&= O(\log n)\sum_{l=1}^{m} T^l + 12F^{OPT} + O(\log\frac{n}{m})F^{OPT} \\
&= O(\log n)F^{OPT}.
\end{aligned}
$$

The third equality follows since $mD \leq \max\{mT, \max_{t \in \mathcal{T}}\{V^A(t)\}\} \leq \sum_{j \in J} p_j \leq F^{OPT}$. The fourth equality follows since $\sum_{l=1}^{m} T^l = \sum_j p_j \leq F^{OPT}$. ■

4 Tightness of the Analysis

In this section we present an $\Omega(\log n)$ lower bound on the competitive ratio of the algorithm analyzed in the previous section. The number of machines is an arbitrary $m \geq 2$ but note that the lower bound only uses the first two machines. While presenting the lower bound we assume an order between jobs with same release time. The algorithm deals with jobs released at the same time following the specified order. [1] Moreover, we assume that the algorithm arbitrarily decides if to schedule a job on an idle machine or on a machine processing a job of higher class. The latter assumption is crucial in proving the lower bound and it agrees with the definition of the algorithm.

We choose P, the ratio between the maximum and the minimum processing time to be a power of 2. The maximum processing time is $P/2$ and the minimum is $1/2$. At each time $r_i = P(1 - \sum_{j=i}^{\frac{1}{2}\log P - 1} \frac{1}{2^{2j+1}})$, for $i = 0, \ldots, \frac{1}{2}\log P - 1$, three jobs are released; one job of processing time $\frac{P}{2^{2i+1}}$ followed by 2 jobs of processing time $\frac{P}{2^{2i+2}}$. Finally, 2 jobs of size $\frac{1}{2}$ are released every $1/2$ time unit between time $r_{\frac{1}{2}\log P} = P$ and $2P - 1/2$.

Let us consider the behaviour of the algorithm and of the optimal solution on this instance. At time r_0 the released job of processing time $P/2$ is assigned to machine 1, a job of processing time $P/4$ is assigned to machine 2 while the second job of processing time $P/4$ preempts the job on execution on machine 1. The two jobs of processing time $P/4$ are finished at time $r_0 + P/4$ where the job of size $P/2$ is taken again on execution on machine 1. In general, at time r_i, $i = 1, .., \frac{1}{2}\log P$, i jobs are in the queue of machine 1, the smallest of remaining processing time $\frac{P}{2^{2i}}$, while no job is on execution on machine 2. The job on execution on machine 1 is preempted by the job of size $\frac{P}{2^{2i+1}}$ that is released at time r_i. Immediately afterwards it is preempted by a job of size $\frac{P}{2^{2i+2}}$ that is released at time r_i while the other job of size $\frac{P}{2^{2i+2}}$ released at time r_i is scheduled on machine 2.

At time P, $\frac{1}{2}\log P$ jobs are in the queue of machine 1, the smallest of size 1, while no job is in the pool or assigned to the other machines. The 2 jobs of size $1/2$ released every $1/2$ time unit between time P and $2P - 1/2$ are scheduled on machines 1 and 2. Observe that $\frac{1}{2}\log P$ jobs are waiting on queue 1 between time P and $2P$ thus leading to a flow time of $\Omega(P \log P)$ for the algorithm on this instance of $n = \frac{3}{2}\log P + 4P$ jobs.

On the other hand, an optimal solution schedules the job of size $\frac{P}{2^{2i+1}}$ released at time r_i for $i = 0, .., \frac{1}{2}\log P - 1$ on machine 1 and the two jobs of size $\frac{P}{2^{2i+2}}$ released at

[1] This can be forced by releasing some jobs slightly after previous ones in the specified order and decreasing their processing times appropriately.

time r_i on machine 2. Observe that in this schedule machines 1 and 2 are idle at time r_i. Thus, the jobs of size $\frac{1}{2}$ are scheduled at their release times on machines 1 and 2 and are completed before the next two jobs of size $\frac{1}{2}$ are released. The flow time of the optimal solution is then $O(P)$.

It follows that the ratio between the flow time of the algorithm and the flow time of the optimal solution is $\Omega(\log P)$ that is also $\Omega(\log n)$.

5 Conclusions

In this paper we considered the problem of finding a preemptive schedule that optimizes the total flow time of a set of jobs released over time when job migration is not allowed. We presented a new on-line algorithm that is almost as effective as the best known algorithm that uses migration [8].

In our algorithm jobs are kept in a pool since the release time until they are assigned to a machine. An interesting open problem is to devise an efficient algorithm that assigns jobs to machines at the time of release. A challenging open problem is also to devise a constant off-line approximation algorithm for optimizing total flow time even if both preemption and migration are allowed.

References

[1] K.R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, 1974.

[2] J. Du, J. Y. T. Leung, and G. H. Young. Minimizing mean flow time with release time constraint. *Theoretical Computer Science*, 75(3):347–355, 1990.

[3] L. Hall, D. Shmoys, and J. Wein. Scheduling to minimize average completion time: off-line and on-line approximation algorithms. In *Mathematics of Operations Research* 22, pp. 513-544, 1997.

[4] L.A. Hall. Approximation algorithms for scheduling. In D.S. Hochbaum, editor, *Approximation algorithms for NP-hard problems*, pages 1–45. PWS publishing company, 1997.

[5] B. Kalyanasundaram and K. Pruhs. Eliminating Migration in Multi-Processor Scheduling. In *Journal of Algorithms*, Vol. 38, No. 1, Jan 2001, pp. 2-24.

[6] H. Kellerer, T. Tautenhahn and G.J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. *SIAM Journal on Computing*, Vol. 28, Number 4, pp. 1155-1166, 1999.

[7] E.L. Lawler, J.K Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. Sequencing and scheduling: algorithms and complexity. In *Handbooks in operations research and management science*, volume 4, pages 445–522. North Holland, 1993.

[8] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 110–119, El Paso, Texas, 1997.