On-line Scheduling to Minimize Max Flow Time: An Optimal Preemptive Algorithm*

Christoph Ambühl and Monaldo Mastrolilli†
IDSIA Galleria 2, CH-6928 Manno, Switzerland

October 22, 2004

Abstract

We investigate the problem of on-line scheduling jobs on m identical parallel machines where preemption is allowed. The goal is to minimize the maximum flow time among all jobs, where the flow time of a job is its completion time minus its release time. We derive an on-line algorithm with competitive ratio $2 - 1/m$. Moreover, when the algorithm must schedule jobs one by one in the order they are released, we prove that there is no randomized on-line algorithm with a better competitive ratio. Finally, we investigate the non-preemptive case and show that the *First In First Out* heuristic achieves the best possible competitive ratio when $m = 2$.

Keywords: Scheduling; Preemption; On-line algorithms; Max flow time.

1 Introduction

The m-machine scheduling problem is one of the most widely-studied problems in computer science, with an almost limitless number of variants (see [3] for an excellent survey until 1998). The most common objective function is the *makespan*, which is the length of the schedule, or equivalently the time when the last job is completed. This objective function formalizes the viewpoint of the owner of the machines. If the makespan is small, the utilization of his machines is high; this captures the situation when the benefit of the owner is proportional to the work done. If we turn our attention to the viewpoint of a user, the time it takes to finish individual jobs may be more important; this is especially true in interactive environments. Thus, if many jobs that are released

*Supported by Swiss National Science Foundation project 200021-104017/1, "Power Aware Computing", and by the Swiss National Science Foundation project 200021-100539/1, "Approximation Algorithms for Machine scheduling Through Theory and Experiments".

†Corresponding author. IDSIA - Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, Galleria 2, CH-6928 Manno, Switzerland. Email: monaldo@idsia.ch

early are postponed at the end of the schedule, it is unacceptable to the user of the system even if the makespan is optimal.

Another well-studied objective function is the *total flow time* [1, 6, 8]. The *flow time* of a job is the time the job is in the system, i.e., the completion time minus the time when it becomes first available. The above mentioned objective function is the sum of these flow values over all jobs. The *Shortest Remaining Processing Time* (SRPT) heuristic [5] produces a schedule with optimum total flow time when there is a single processor and preemption is allowed. Unfortunately, this heuristic may lead to *starvation*. That is, some jobs may be delayed to an unbounded extent. Inducing starvation is an inherent property of the total flow time objective function. In particular, there exist inputs where any optimal schedule for total flow time forces the starvation of some job [2]. From the discussion above, it is natural to conclude that in order to avoid starvation, one should bound the flow time of each job. This motivates the study of the minimization of the *maximum flow time*.

Problem definition: We study the problem of preemptively scheduling on-line a set of jobs on m identical machines. Each job J_j comes with its *release time* r_j and its *processing time* p_j. At any time, each machine can process at most one job and each job can be processed by at most one machine. *Preemption* is permitted, which allows to partition the jobs into smaller pieces and to schedule these pieces on different machines. The jobs are presented on-line one-by-one as they appear in a list $(J_1, J_2, \ldots)$ sorted according to jobs release times ($r_1 \leq r_2 \leq \ldots$). As soon as J_j appears, it has to be assigned irrevocably to one or more machines for p_j time units in all. No job can be scheduled before its release time. Job J_j becomes known only when J_{j-1} has already been scheduled. Let C_j be the completion time of job J_j in a given schedule. Then the flow time F_j of job J_j is equal to $F_j = C_j - r_j$ and the *maximum flow time* F_{max} of the schedule is defined as $F_{max} = \max_j F_j$. The objective is to find a schedule that minimizes the maximum flow time.

About the on-line model: In the studied model, the jobs arrive in a sequence and need to be scheduled immediately; however the sequence is sorted according to the release times. The model studied in this paper appears to be new, and it is an unusual blend of two standard models [11]. The first is that of jobs arriving *over time*, i.e., the jobs become known at their release times (all at once if there are more such jobs), and at any time the algorithm can arbitrarily choose among the available jobs - if a job is not started at its release, its assignment may depend on jobs released in the future. In the second model the jobs arrive *over a list*, i.e., the jobs are ordered in some list and are presented one by one according to this list. As soon as the job is presented we know all its characteristics. The information on the next job is not known until the current job is assigned. As soon as a job has been assigned, it cannot be moved again. In general, there is no restriction on the order - the time in the schedule as well as release times can be totally independent of the order of the list.

Scheduling jobs as soon as they are released, seems a very natural model to incorporate release times in the model of scheduling the jobs one by one. In addition, if there is no restriction on the order, we will prove (Section 5) that no on-line algorithm can be competitive. Finally, any algorithm that works in this new model also extends to the standard model of jobs arriving over time.

Previous work and our contribution: The off-line version of this problem, where all the jobs information, such as the total number of jobs, their release dates and processing times, is fully known in advance, can be solved to optimality in polynomial time [7]. The only known result about the on-line version is that the *First In First Out* (FIFO) heuristic (that is, scheduling jobs in the order they arrive to the machine on which they will finish first) is a $(3 - 2/m)$-competitive algorithm (a ρ-competitive algorithm is an on-line algorithm that finds a solution within a ρ factor of the optimum). This bound holds for both the preemptive [9] and non-preemptive [2] setting. We observe that in [2], the authors consider the scheduling over time model when preemption is not allowed, and claim a lower bound of $3/2$ on the best possible competitive ratio. However, it is easy to check that the proposed construction gives only a lower bound of $4/3$ (but the arguments in [2] give a lower bound of $3/2$ when the on-line model is the same as the one proposed in this paper). When the jobs release times are identical, the problem reduces to the classical makespan minimization problem. In this case, it is known [4] that there does not exist any on-line algorithm with competitive ratio better than $m^m/(m^m - (m-1)^m)$, and this bound is valid even for randomized algorithms [11]. Interestingly, there is a deterministic algorithm matching this lower bound [4].

In this paper we derive a $(2 - 1/m)$-competitive algorithm for the preemptive identical machines max flow time minimization problem (Section 2), and prove its *optimality* within the proposed on-line model (Section 3). Indeed, we show that there is no randomized on-line algorithm, working against an oblivious adversary, with a better competitive ratio. We observe that the provided lower bound of $2 - 1/m$ on the competitive ratio holds also for the non-preemptive case. Finally, we further refine this bound when the number of machines is two. Precisely, we provide a lower bound of 2 on the competitive ratio for deterministic non-preemptive algorithms for two machines (Section 4), which shows that FIFO is optimal for $m = 2$.

2 The on-line algorithm

The proposed Algorithm 1 splits any job into equal parts among the machines.

Algorithm 1 *Number the machines from* 1 *to* m*. Whenever a new job arrives, divide it into* m *pieces of equal size. In the order of increasing machine index* i *(*$i = 1, \ldots, m$*) and starting from* $i = 1$*, schedule piece* i *as soon as possible as a whole after all previously scheduled pieces on machine* i*.*

In a practical implementation, one would probably try to take advantage on idle times in the current schedule to schedule the pieces. As long as this does not increase the completion time of any of the pieces compared to Algorithm 1, this obviously cannot worsen the competitive ratio of an instance. But for the worst case competitive ratio, it does not help.

In order to analyze Algorithm 1, we use two lower bounds on the optimal solution value. The first one is the length of the largest job. For the second lower bound, we allow that pieces belonging to the same job are scheduled in parallel. In this model, the optimal algorithm is similar to Algorithm 1: Whenever a job arrives, we break it into m pieces and schedule all of them in parallel, as soon as possible. Obviously, all pieces of a job complete at the same time. Hence we can define c_j to be the common completion time of the pieces of job J_j. In order to see the optimality of our algorithm in the new model, consider the schedule after j jobs have been scheduled. The makespan of this schedule is c_j and obviously, this is optimal. The flow time of job J_j, defined as $f_j := c_j - r_j$, is a lower bound for the max flow time among the first j jobs, since there must be at least one job that finishes not before c_j, and its release time cannot be later than r_j. Using our algorithm in the relaxed model, the flow time of any job J_j is exactly f_j, and therefore it is optimal.

We are now ready to analyze Algorithm 1. Remember that in contrast to the previous paragraph, pieces belonging to the same job cannot be scheduled in parallel. For what follows, we need the following definition.

$$p_{max}^{(k)} := \max_{i=1,\ldots,k} p_i.$$

Lemma 1 *Consider the schedule after j jobs have been scheduled using Algorithm 1. Then the completion time of machine i is at most $c_j + (i-1)p_{max}^{(j)}/m$ and therefore, the flow time of piece i of a job J_j is at most $f_j + (i-1)p_{max}^{(j)}/m$.*

Proof. We give an inductive proof. Obviously, the statement is true if no jobs have been scheduled. For the inductive step, let us assume that $j-1$ jobs have been scheduled. Let job J_j have length l. If $l \leq p_{max}^{(j-1)}$, the completion time on machine i is at most $\max(c_{j-1}, r_j) + (i-1)p_{max}^{(j-1)}/m + l/m = c_j + (i-1)p_{max}^{(j)}/m$. If on the other hand $l > p_{max}^{(j-1)}$, then the completion time of machine i is at most $\max(c_{j-1}, r_j) + i \cdot l/m = c_j + (i-1)p_{max}^{(j)}/m$. ■

Theorem 2 *Algorithm 1 is $2 - 1/m$ competitive.*

Proof. Let n denote the number of submitted jobs and let $F^* = \max\{f_1, f_2, \ldots, f_n\}$, which is a lower bound on the optimal max flow time. The flow time of any job J_j is bounded by

$$f_j + (m-1)\frac{p_{max}^{(n)}}{m} \leq F^* + (m-1)\frac{F^*}{m} = (2 - \frac{1}{m})F^*.$$

■

3 A matching lower bound

In this section we prove that no on-line algorithm with a competitive ratio smaller than $2 - 1/m$ exists, even if we allow randomization. In proving lower bounds for randomized algorithms, one must be careful to delimit the adversary's access to the random bits used by the algorithm. In our setting, we consider the *oblivious adversary* (see e.g. [10]), who knows only the algorithm but not the coin tosses. The oblivious adversary models the situation where a randomized algorithm receives a problem instance and must produce a solution; the random choices it makes have no effect on the input it sees.

Theorem 3 *No randomized on-line algorithm for preemptive scheduling of identical machines, working against an oblivious adversary, has competitive ratio smaller than* $2 - 1/m$.

Proof. We prove the theorem by giving a lower bound of $2 - 1/m - \varepsilon$ on the competitive ratio, for any fixed $\varepsilon > 0$. We apply Yao's Minimax Principle [12]: the competitiveness $C_{\mathcal{P}}^{\mathcal{A}}$ of the optimal deterministic algorithm $\mathcal{A}$ for an arbitrarily chosen input distribution $\mathcal{P}$ is a lower bound on the competitiveness $C_{\mathcal{R}}^{obl}$ of the best randomized algorithm $\mathcal{R}$ against the oblivious adversary. Thus, we can establish a lower bound on $C_{\mathcal{R}}^{obl}$ by giving a probability distribution $\mathcal{P}$ and giving a lower bound on $C_{\mathcal{P}}^{\mathcal{A}}$ for any deterministic on-line algorithm $\mathcal{A}$.

The input distribution $\mathcal{P}$ is as follows. Jobs are released in rounds, starting with round 1. Let $r(i)$ denote the common release time of jobs at round i. Set $r(1) = 0$, whereas the remaining release times are defined as follows. At round i the following set of jobs are submitted in the order.

- $m(m-1)$ unit jobs; set $r(i+1) \leftarrow r(i) + (m-1)$;
- with probability 1/2: submit an additional job of length m and increase the next release date by 1, i.e. $r(i+1) \leftarrow r(i+1) + 1$.

Observe that in the optimal solution there are no idle times between two consecutive rounds. Let $F_{max}^{\mathcal{A}}(i)$ and $F_{max}^{*}(i)$ denote the max flow time of jobs released within the first i rounds and when they are scheduled according to algorithm $\mathcal{A}$ and the optimal off-line algorithm, respectively. Let

$$k_1 \geq \frac{2m^2}{\epsilon} \text{ and choose } k_2 \text{ s.t. } \left(\frac{1}{2}\right)^{k_2} \leq \frac{\epsilon m - \epsilon}{m} \text{ and } k = k_1 + k_2.$$

In the remainder, we show that within k rounds the expected value of $F_{max}^{\mathcal{A}}(k)$ is at least $2m - 1 - m\epsilon$, whereas $F_{max}^{*}(k) \leq m$ and we are done. With this aim, we need to define the notion of *overload*. The overload at round t, denoted by H_t, is the total work that is conducted after the starting time $r(t)$ of round t in the on-line schedule, where only jobs that are released before $r(t)$ are considered. Let I_i be the sum of machine idle times between time $r(i)$ and $r(i) + m - 1$ before the job of length m is, in case, submitted during the i-th

round. Let X_i be an indicator random variable which is equal to one if the large job of size m is submitted at round i, and zero otherwise. Observe that $H_t \geq \sum_{i=1}^{t-1} I_i(1 - X_i)$ and $F_{max}^{\mathcal{A}}(t) \geq \frac{m(m-1)+H_t}{m}$. Therefore, after k rounds either

(a) there are at least k_1 rounds with $I_i \geq \epsilon$, or

(b) there are at least k_2 rounds with $I_i < \epsilon$.

In case (a) we have $E[F_{max}^{\mathcal{A}}(k)] \geq \frac{m(m-1)+m^2}{m} = 2m-1$ by noting that $E[H_k] \geq m^2$, and we are done. In case (b) there are at least k_2 rounds with $I_i < \epsilon$. During these k_2 rounds, if a job of size m is submitted, its starting time cannot be smaller than $m - 1 - \epsilon$. Hence we can conclude that

$$\begin{aligned} E[F_{max}(k)] &> m - 1 - \epsilon + (1 - (1/2)^{k_2})m \\ &\geq 2m - 1 - m\epsilon. \end{aligned}$$

■

4 On the non-preemptive problem

Note that the lower bound given by Theorem 3 holds also for the non-preemptive problem (this immediately follows from the proof by observing that there is an optimal solution of the given instance which is non-preemptive). However, for the non-preemptive case, one can provide a lower bound of two, which matches the competitive ratio of FIFO on two machines [2].

Theorem 4 *On two machines, FIFO is an optimal deterministic non-preemptive algorithm.*

Proof. The lower bound works in rounds. Every round starts by releasing a set of jobs. Depending how the online algorithm processes these jobs, further jobs might be released to complete the round. There are two different types of rounds, which are further subdivided into 5 variations in total, as described in Figure 1. For every variation, Figure 1 shows the on-line and off-line behavior. All rounds have a starting time, which it denoted by r_i. By r_{i+1}, we denote the time by which the optimal off-line algorithm completes the processing of the jobs of the current round. This is also the starting time of the subsequent round. The *overload* at the beginning of a round is the total amount of time for which the machines of the on-line algorithm are busy processing jobs of the previous rounds after time r_i. It is denoted by the grey area in Figure 1. An overload is called *flat* if it is equally partitioned on both machines; this in contrast to a *one-sided* overload, where the whole overload is concentrated on one machine.

The first type of round starts from a flat overload of size o, and ends with a one-sided overload of size $(1 + o)/2$. There are two variations of this type, denoted by (1a) and (1b) in Figure 1. In both variations, first two small jobs of size 1/2 are released. Whether or not the large job of size 1 is released depends

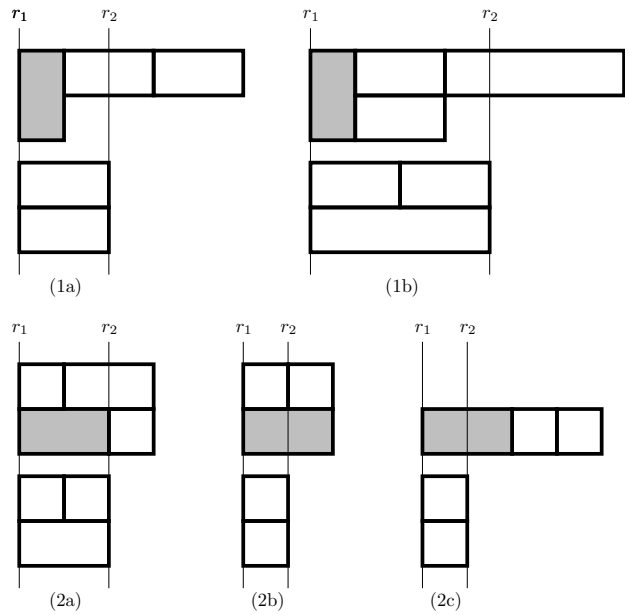

Figure 1: 2-machines lower bound.

on how the on-line algorithm places the small jobs. The release time of all jobs released in this round is r_i. The starting time of the upcoming round is denoted by r_{i+1}.

The second type of round, denoted by (2a–c) in Figure 1, starts from a one-sided overload, and either ends with a one-sided overload of size $(3/2)o$, or turns the overload into a flat one, without increasing it. As for the previous cases, first the two small jobs of size $o/2$ are released. The large job of size o is released only if the two jobs are not placed on the same machine.

The first round will be of type (1), since there is no overload yet. The type of round $i+1$ depends on the type of the overflow after round i finished.

After at most $2k$ rounds, the overload will be at least $1-(1/2)^k$. At round $2k$ or $2k+1$, the overload can be made one-sided, and as soon it happens two jobs of size 1 are released. At least one of these jobs will have flow time $2-(1/2)^k$. It is easy to see from Figure 1 that the adversary can preserve a maximum flow time of 1 for all jobs. Therefore, by choosing k large enough, we can reach $2-\varepsilon$ for any $\varepsilon > 0$. ■

5 On scheduling over a list with release times

In scheduling over a list, the jobs are ordered in some list and are presented one by one according to this list. In the following we observe that if there is no restriction on the order of the list, then no algorithm can be competitive.

Theorem 5 *In the on-line model of scheduling over a list, if release times can be totally independent of the order of the list, then no on-line algorithm can be competitive when minimizing the max flow time.*

Proof. The claim follows by an easy adaptive adversary argument. Consider the single machine case. By contradiction, let us assume that there exists an on-line algorithm A that is k-competitive, for any fixed constant $k \geq 1$. A set of n unit jobs are submitted in the order $J_1, J_2, \ldots,$ such that job J_j is released at time $r_j = j-1$. Depending on the behavior of the on-line algorithm A, the adversary may submit a final unit job whose release time is specified in the following. Starting from $i = 0, 1, 2, \ldots$, if the next $2k$ jobs $J_{2ik+1}, \ldots, J_{2(i+1)k}$ are scheduled by algorithm A one after the other, and without leaving any unit gap between these jobs, then the adversary submits the last unit job $J_{2(i+1)k+1}$, with release time $r_{2(i+1)k+1} = 2ik$. In this case the optimal max flow time is 2, whereas job $J_{2(i+1)k+1}$ is completed not before time $2(i+1)k+1$. Therefore, A returns a solution whose value is at least $2k+1$, which contradicts our assumption that A is k-competitive. Otherwise, every $2k$ scheduled jobs there is at least one unit gap, and after $2k(k+1)$ submitted jobs there are at least at least $k+1$ unit gaps from time 0 up to time $2k(k+1)$. The latter means that at time $2k(k+1)$, there are still at least $k+1$ jobs which have been released before that time, and that have not yet been completed. It is easy to observe that among these jobs there is at least one job with flow time at least $k+1$, whereas the optimal max flow time is 1, which again contradicts our assumption that A is k-competitive. ■

Finally, we would like to point out that the result of Theorem 5 should be put in contrast with the makespan problem with release times, where a competitive algorithm can be easily obtained even when there is no restriction on the order of the list.

6 Open problems

There are several open problems in the area of max flow time scheduling. Concerning the off-line version, FIFO is still the best known non-preemptive algorithm if the number of machines is not a constant. (When the number of machines is a constant, there is a fully polynomial time approximation scheme [9].) As to on-line algorithms, it would be nice to improve on FIFO. While our preemptive algorithm is optimal for identical parallel machines in the proposed on-line model, it would be interesting to investigate on the scheduling over time model. Finally, apparently it does not generalize easily to uniformly related parallel machines.

References

[1] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. In *In Proceedings of the 31st Annual ACM Symposium*

on Theory of Computing (STOC'99), pages 198–205, 1999.

[2] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'98)*, pages 270–279, 1998.

[3] B. Chen, C. Potts, and G. Woeginger. A review of machine scheduling: Complexity, algorithms and approximability. *Handbook of Combinatorial Optimization*, 3:21–169, 1998.

[4] B. Chen, A. van Vliet, and G. J. Woeginger. An optimal algorithm for preemptive on-line scheduling. *Operations Research Letters*, 18:127–131, 1995.

[5] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In M. J. Atallah, editor, *Handbook of Algorithms and Theory of Computation*. CRC Press, 1997.

[6] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *In Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC'96)*, pages 418–426, 1996.

[7] E. Lawler and J. Labetoulle. On preemptive scheduling of unrelated parallel processors by linear programming. *Journal of the ACM*, 25:612–619, 1978.

[8] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proc. 28th Annual ACM Symposium on the Theory of Computing (STOC'96)*, pages 110–119, 1997.

[9] M. Mastrolilli. Scheduling to minimize max flow time: Off-line and on-line algorithms. *International Journal of Foundations of Computer Science*, 15:385–401, 2004.

[10] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, 1995.

[11] J. Sgall. On-line scheduling – a survey. In *A. Fiat and G. Woeginger, editors, On-Line Algorithms, Lecture Notes in Computer Science*, 1997.

[12] A. C. Yao. Probabilistic computations : Toward a unified measure of complexity. In *18th Annual Symposium on Foundations of Computer Science (FOCS'77)*, pages 222–227. IEEE Computer Society Press, 1977.