On-line multi-threaded scheduling

TU/e technische universiteit eindhoven
/department of mathematics and computing science

Memorandum COSOR 99-21

On-line multi-threaded scheduling

E. Feuerstein, M. Mydlarz, L. Stougie

Eindhoven, December 1999
The Netherlands

On-line multi-threaded scheduling

Esteban Feuerstein * Marcelo Mydlarz† Leen Stougie‡

December 1, 1999

Abstract

This paper presents results on on-line scheduling problems with multiple threads. The jobs are organized in a number of sequences called threads. Each job becomes available (is presented) only at the moment a scheduling decision is made on all the preceding jobs in the same thread. Thus, apart from decisions on scheduling of jobs also decisions are required on the order of exploring the threads.

We consider two different on-line paradigms. The first paradigm can be regarded as constructing, in an on-line way, a schedule of the jobs which is to be executed later, a sort of batch process. The other paradigm models a real-time planning situation, in which the jobs are immediately executed at the moment they are assigned to a machine.

We study two classical objective functions: the *makespan* defined as the maximum completion time of the jobs, and the *average completion time* of the jobs, also called the *latency* of the jobs.

We establish a fairly complete set of results for these on-line multi-threaded scheduling problems. One of the highlights of our results is that List Scheduling is an optimal algorithm for the makespan problem under the real-time on-line model if the number of machines does not exceed the number of threads. Another one is that for the latency problem on a single machine under both on-line paradigms there exists a non-trivial algorithm that is best possible and has competitive ratio equal to the number of threads.

*Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, and Instituto de Ciencias, Universidad de General Sarmiento, Argentina. Partly supported by the KIT program of the European Community under contract n. 131 (DYNDATA), by UBA project EX070/J "Algoritmos eficientes para problemas on-line con aplicaciones" and UBA project EX209 "Modelos y técnicas de optimización combinatoria". e-mail: efeuerst@dc.uba.ar

†Department of Computer Science, Rutgers University, Piscataway, New Jersey 08854-8019. e-mail: marcem@cs.rutgers.edu

‡Combinatorial Optimization Group, Faculty of Mathematics, Technical University Eindhoven, P.O. Box 513, 5600MB Eindhoven, The Netherlands. Supported by the TMR Network DONET of the European Community ERB TMRX-CT98-0202 and by the KIT program of the European Community under contract n. 131 (DYNDATA). e-mail: leen@win.tue.nl

Keywords: on-line algorithms, multiple threads, competitive analysis, scheduling problems.

1 Introduction

Scheduling problems appear and have appeared in literature and practice in an enormous variety. The common characteristic of problems of this type is that they require decisions on the order in which jobs are to be processed on machines. In this paper we consider on-line scheduling problems with multiple threads. On-line scheduling problems are scheduling problems of which the input is not (completely) known in advance: the information about the jobs to be processed (or even the jobs themselves) becomes available during the decision process. In multi-threaded on-line scheduling the jobs are organized in a number of sequences called *threads*. Each job becomes available (is presented) only at the moment a scheduling decision is made on all the preceding jobs in the same thread. Thus, apart from decisions on scheduling of jobs also decisions are required on the order of exploring the threads.

The general setting of scheduling problems is that jobs are to be scheduled for processing on machines such as to obtain a schedule that is optimal according to some objective. Jobs are characterized by, possibly among others, their processing times. Therefore in almost all scheduling problems *time* plays a central role. E.g., almost all well-known objectives in scheduling are functions of time. In off-line scheduling this time factor is however not a *real-time* factor: a schedule is to be found for the given jobs on the given machines, and that schedule is to be executed in the future.

In on-line scheduling, time may play that same role or a different one. The first situation arises in the cases in which the characteristics of jobs are revealed one by one. Jobs are to be scheduled on the machines directly and irrevocably, without knowledge of jobs that will be presented later. In such cases again time comes into the model only via the processing times of the jobs and in the objective. But time does not really elapse; the decision process is a sort of *batch* process. In other settings time comes to play a crucial role during the decision process: information about jobs is seen while time flows; and time flows while the previously scheduled jobs are being processed. At any point in time some jobs may be processed and/or machines may be idle. Newly seen jobs must not necessarily be processed immediately. Sometimes waiting is necessary (e.g. in case all machines are busy at the moment the new job comes in and preemption of jobs is not allowed), in other cases waiting may be a choice of the algorithm. Thus, the scheduling process can be seen as a *real-time* process.

In this paper we consider on-line scheduling problems of both types. We define two on-line paradigms, related to the different roles that time may play.

Multi-threaded on-line optimization problems have been proposed first in [5]. In contrast to ordinary on-line optimization, which could be seen as sin-

gle threaded or unthreaded problems, information is contained in one or more threads. The threads imply a certain order in which information becomes available. As soon as the information at the top of a thread has been processed the next information in that thread becomes available. In the on-line scheduling problems that we consider this means that we see the jobs on the top of the threads, and as soon as we have scheduled a job the next job on that thread will be revealed to us. It is possible to recognize if a thread is or has become empty.

This framework models situations in which independent clients present their requests of jobs to be processed. The set of jobs of each client is given in a sequential way, and may be seen as a thread. An example of this (corresponding to the real-time paradigm) is given by an operating system that has to schedule a certain number of sequential processes that must be executed at one or more processors. A situation in which multi-threaded scheduling is a batch process appears in the usage of redirectors for web server load balancing: a redirector is a machine that receives HTTP requests from clients and (re)directs them to different web servers, with the goal of, for example, minimizing the response time to the clients. Yet another example is provided by a shop schedule. Each job consists of a number of sequential operations. In this case the threads correspond to the jobs (see [10]). References on other multi-threaded on-line optimization problems are [6], [7], [2].

For on-line optimization problems in general and for on-line scheduling problems in particular usually no strategy can be found that gives the optimal solution for any input sequence. It has become standard to measure the performance quality of algorithms for on-line optimization problems through *competitive analysis*, in which the quality of an algorithm is measured as the worst-case ratio of the value of its solution and that of the optimal off-line solution. For an overview of the literature on on-line optimization we refer to [4]. An up-to-date survey on on-line scheduling problems is given in [12].

In this paper we study two classical objective functions: the *makespan* defined as the maximum completion time of the jobs, and the *sum of completion times* of the jobs, also called the *latency* of the jobs. This will be done within a model where the information is given in the form of multiple threads.

Now follows an overview of the results comprised in the table that can be found in Section 2.4, after we have specified all aspects of the models we studied more precisely in the preceding subsections of Section 2. We study the makespan problem only on more than one machine, since the single machine case is trivial. We do distinguish between the existence of a single thread and of multiple threads. The results are proven in Section 3.1.

We show that the single and the multi-threaded on-line makespan problems under the batch-model are equivalent from a point of view of competitiveness to the ordinary on-line makespan problem without threads.

Under the real-time model we can show easily that there exists an on-line algorithm for the single threaded problem that yields the off-line optimal solution,

i.e., being 1-competitive. The multiple threaded case is more interesting. We prove that Graham's classical List Scheduling rule [8] (combined with a round robin exploration of the threads) is a best possible algorithm in case the number of machines does not exceed the number of threads by more than 1. Otherwise it is nearly best possible.

For the latency problem the single machine version is far from trivial, though the combination of a single machine and a single thread leads to a trivial problem. In Section 3.2 we prove that the competitive ratio of any algorithm for the on-line latency problem is bounded from below by the number of threads under both the batch and the real-time model. An algorithm is designed that is best possible for this problem under both on-line models.

Finally, in Section 3.3 we prove that for the latency problem on multiple machines no algorithm can have a constant competitive ratio, unless under the real-time model jobs are organized in a single thread, in which case the optimal off-line solution is easily achieved.

2 Problem definitions and results

In this section we specify the three components constituting the problems that we study. First, we define the scheduling aspect. The second subsection concerns various appropriate on-line models for scheduling problems and the third one models for threads in on-line optimization. We conclude the section with a table of results.

2.1 Scheduling

The basic characteristic common to all scheduling problems is that jobs are to be scheduled for processing on one or more machines. Depending on properties of machines, restrictions on jobs and particular objective functions, an enormous variety of scheduling problems emerges. A classification of almost all scheduling problems is given in [11]. We concentrate on the problems that we study here.

As a machine environment we distinguish between the situations of a single machine and multiple parallel identical machines. Any machine can process any job, and a feasible schedule requires that any job is processed on one machine and at any time any machine processes at most one job.

Each job j is completely characterized by its processing time p_j, $j = 1, 2, ...$ Thus, we do not consider problems in which jobs have time windows. We also do not allow preemption of jobs. We actually do consider precedence relations between jobs, but only to the extend in which they emerge from the order in which jobs appear on the threads. We come back to this later in our discussion on the various thread models.

As objectives we consider to minimize the well-known makespan, i.e., the maximum job completion time, and the latency, i.e., the sum of the job com-

pletion times. Since minimizing makespan on a single machine has only trivial solutions, even in on-line settings, we disregard this combination.

2.2 On-line paradigms

In the survey of Sgall [12] various on-line paradigms are distinguished for scheduling problems. We stick to models in which the running time of each job is known at the moment the job is seen by the on-line algorithm. In this paper we make a main distinction between two paradigms, regarding the role that time plays in the process of information (new jobs) becoming available.

In the first paradigm, time does not play any role apart from being part of the input (running times of jobs) and the objective function. The scheduling process is a *batch* process in which jobs are assigned to the machines (processors) and time slots in which they will be executed at a later stage. The on-line nature of the problem is given by the fact that the jobs are to be scheduled on the machines irrevocably and with incomplete information. From now on we will refer to this paradigm as BATCH.

This paradigm has two variations. In the first one the jobs are only assigned to a machine and each next job assigned to a particular machine is scheduled for processing directly after the jobs already assigned to that machine. In the second variation a job is not only allocated to a machine but also assigned a time slot on that machine equal in length to its processing time. Of course, the restriction of one job per machine at any time must remain obeyed.

In this paper we do not consider the latter variation. For minimizing makespan the order of jobs on one particular machine is irrelevant, whence the two variations coincide in their best possible solutions. For minimizing latency they are essentially different and the second one is (at least theoretically) interesting. However, we have not studied the various problems given under that definition.

In contrast with the previous paradigm, in the second paradigm scheduling is a *real-time* process in which, while some jobs are being processed, time is really elapsing and new jobs may be seen. In our multi-threaded setting, the next job in a thread becomes available only at the moment the previous jobs in the same thread have been started, i.e., not before the moment a machine becomes available to process the immediately preceding job in the thread. We call this paradigm REAL-TIME.

The off-line counterpart of the REAL-TIME model would be a scheduling problem with certain precedence restrictions corresponding to the order of the jobs on the threads.

A variation of the REAL-TIME paradigm in which the processing time of a job becomes known only at the moment of its completion has not been considered here as announced above.

Table 1: Makespan

	BATCH		REAL-TIME	
	l.b.	u.b.	l.b.	u.b.
$w = 1$	1.8520	1.9230	1	1
$w \geq m - 1$			$2 - 1/m$	$2 - 1/m$
$w < m - 1$			$2 - 1/(m + 1 - \lceil \frac{m-1}{w} \rceil)$	$2 - 1/m$

2.3 Threads

When jobs are presented on-line in threads, the threads determine the order in which the jobs will be seen. We distinguish between the two cases in which jobs are presented in a single thread and in which jobs are presented in multiple threads. Within the BATCH model the existence of a single thread is exactly the same as the usual model without threads, which has been studied quite extensively in the literature (see [12]).

Within the REAL-TIME model introduced above, however, the existence of a single thread is different from the situation with no threads at all. In the most natural model with no threads, jobs arrive over time independently of any scheduling decision about previously presented jobs. The off-line counterpart is then a scheduling problem with release times of the jobs. In our REAL-TIME model with a single thread we assume that all jobs are there but we see them only if all previous jobs have been started processing. The off-line counterpart is then a scheduling problem with chain-like precedence constraints on the start times of the jobs.

Again the problem with no threads has been studied in the literature (see [12]).

2.4 Results

An overview of the results is given in the Tables 1 and 2. The numbers found in the column of the BATCH model in Table 1 concern bounds for any number of machines. We prove and comment on the results in the tables in the following section.

3 Competitive analysis

First we give two simple lemmas that settle all the 1-competitiveness results in Tables 1 and 2. In the next three subsections we present the other results

Lemma 3.1 *There exist 1-competitive algorithms for each one of the on-line scheduling problems under the REAL-TIME model with a single thread.*

Table 2: Latency

	BATCH		REAL-TIME	
	l.b.	u.b.	l.b.	u.b.
any $w, m = 1$	w	w	w	w
$w = 1$ and $m \geq 2$	∞	–	1	1
$w \geq 2$ and $m \geq 2$	∞	–	∞	–

PROOF. At any point in time there is only one job available for scheduling. Therefore List Scheduling, which schedules each job as soon as a machine becomes available, yields the optimal solution. □

Lemma 3.2 *There exist 1-competitive algorithms for each one of the on-line single machine scheduling problems under the BATCH model with a single thread.*

PROOF. There is only one feasible solution for such a problem. □

In the next three subsections we present the other results

3.1 Makespan

Given the fact that a single thread is not different from no thread within the BATCH model, the corresponding results can be found in the literature [1, 3, 9]. The numbers found in Table 1 are the currently known bounds for the best possible competitive ratio for any number of machines (see [1]). The precise value of the best competitive ratio is unknown.

These bounds actually hold for an infinite number of machines. Closing the gap between these bounds is a challenging research question. For fixed small numbers of machines, better bounds are known, and for $m = 2$ and $m = 3$ List Scheduling is known to be a best possible algorithm yielding competitive ratio's of $3/2$ and $5/3$, respectively (see [12]).

To show that all known bounds hold for the corresponding problems with multiple threads in the BATCH model as well, we first make the following observation.

Lemma 3.3 *A lower bound on the competitive ratio of any on-line algorithm for a problem with a single thread must be a lower bound on the competitive ratio of any on-line algorithm for the same problem with multiple threads.*

PROOF. This is easily seen from the fact that in a w-threaded problem it is always possible to have $w-1$ threads empty or, if this is not appreciated, filled with jobs with negligibly small processing times. □

The following theorem shows that within the BATCH model, for the makespan problem the competitive ratio of the multi threaded problem is at most that of the single threaded one.

Theorem 3.1 *Given any algorithm A for the single threaded BATCH makespan problem having competitive ratio ρ, there exists a ρ-competitive algorithm A_w for the multi-threaded version.*

PROOF. Given algorithm A for the single-threaded problem we construct algorithm A_w simply by considering the input data of the w threads in a *round robin* fashion, i.e., given an arbitrary numbering of the threads it applies algorithm A to the sequence given by the first job of the first thread, the first job of the second thread, and so on until the first job of the last thread, followed by the second job of the first thread, the second job of the second thread and so on, skipping threads that have become empty. Thus, A_w ignores the extra information available in comparison to the single threaded case.

Suppose that A_w is not ρ-competitive. This means that there is a multi-threaded input σ_w such that the cost of A_w on σ_w is greater than ρ times the optimal cost on σ_w. Obviously, A_w's cost on σ_w is exactly the same as A's cost on the single thread σ consisting of the jobs in the order that A_w treats them. On the other hand, the optimal off-line solution on σ is the same as that of the multi-threaded problem on σ_w, and therefore, A is not ρ-competitive, a contradiction.

That the optimal solutions for the single thread σ and the multi-threaded input σ_w coincide is true because in the BATCH model any assignment of jobs to machines can be obtained independently of the order in which the jobs are presented, and for makespan the order of jobs on each single machine is irrelevant. □

We notice that the competitive ratios of algorithms for w-threaded problems can be much larger than those of their single threaded counterparts as is shown in [6] for the paging problem and in this work (Section 3.2) for the latency problem on 1 machine.

The results for the REAL-TIME model with $w > 1$ threads are given in the following theorems.

Theorem 3.2 *Any algorithm for the w-threaded REAL-TIME makespan problem is at least $2 - \frac{1}{m+1-\lceil \frac{m-1}{w} \rceil}$-competitive, for $m > 1$.*

PROOF. We define the following problem instance constructed by the adversary. There are $m-1$ jobs with processing time B, the B-jobs. Each of the w threads starts either with $\lceil \frac{m-1}{w} \rceil$ or with $\lfloor \frac{m-1}{w} \rfloor$ B-jobs. After having scheduled all B-jobs on $w-2$ threads two threads remain with B-jobs. The other threads are now empty. Let us call the remaining non-empty threads Thread 1 and Thread 2. Let Thread 1 be the thread that contains the B-job that is scheduled last by the on-line algorithm and let b_1 be the total number of B-jobs on Thread 1. Threads 1 and 2 are continued with jobs with processing time $S < B$, S-jobs. S is chosen such that $(m - \lceil \frac{m-1}{w} \rceil)S = B$.

Suppose that a job J of size S or B is scheduled on a machine that has already been assigned a B-job. Now, the adversary can present a job of size $M = B + S$, the M-job, on the same thread immediately behind J. Since J starts at time B, the M-job cannot start before time B, yielding a makespan of at least $B+M$, which will be shown to be too high.

Thus, let us suppose that neither an S-job nor a B-job is scheduled on a machine that has already a B-job assigned to it. In this case the adversary places $m-1-b_1$ S-jobs on Thread 1, and 1 S-job on Thread 2.

Consider the moment just after all $m-1$ B-jobs have been scheduled on $m-1$ different machines. Notice that at this moment none of the S-jobs of Thread 1 can have been scheduled. We can consider the following three situations. In the first one the S-job on Thread 2 has not been scheduled yet. In the second one the S-job on Thread 2 has been scheduled to the only machine that does not have a B-job. In the third one the S-job on Thread 2 has been scheduled before a B-job (coming from Thread 1).

In the first two situations, all S-jobs (of Threads 1 and 2) will be scheduled to the same machine (the only one that does not have a B-job), yielding for that machine a total processing time of $(1+(m-1-b_1))S \geq (m-\lceil \frac{m-1}{w} \rceil)S = B$.

In the third situation, the $m-1-b_1$ S-jobs on Thread 1 cannot start before time S, since a B-job of Thread 1 starts after the S-job of THread 2. Again the S-jobs of Thread 1 must be scheduled on the only machine that has no B-job. The last S-job on this machine will not finish before time $S + (m-1-b_1)S \geq (m - \lceil \frac{m-1}{w} \rceil)S = B$.

In each of these three situations, the adversary presents an M-job behind the job scheduled last on the same thread. The resulting schedule has makespan $B + M = 2M - S$.

Clearly, an optimal off-line algorithm schedules all jobs of the thread that contain the M-job first. Notice that, in all situations, there are at most m jobs on this thread. The remaining jobs are then scheduled in a list scheduling way afterwards. This results in a schedule with makespan M. Thus, the competitive ratio is at least $(2M-S)/M = 2-S/M$. The adversary's choice of $M = B+S = (m - \lceil \frac{m-1}{w} \rceil + 1)S$, yields the theorem. □

As a corollary we obtain a lower bound of $2 - \frac{1}{m}$ on the competitive ratio in

case $w \geq m-1$.

Corollary 3.1 *Any algorithm for the w-threaded REAL-TIME makespan problem is at least $2-\frac{1}{m}$-competitive if $w \geq m-1$.*

PROOF. In case $w \geq m-1$, we have that $\lceil \frac{m-1}{w} \rceil = 1$ by which the corollary follows from the previous theorem. □

In the following theorem will be revealed that Graham's famous List Scheduling algorithm [8] combined with a *round robin* exploration of the threads proves to be best possible $2-\frac{1}{m}$-competitive in case $w \geq m-1$. List Scheduling considers the jobs in any arbitrary order and assigns each next job to the machine that is becoming available first. The resulting combination of round robin and List Scheduling we call RRLS.

Theorem 3.3 *RRLS is a $2-\frac{1}{m}$-competitive algorithm for the multi-threaded REAL-TIME makespan problem.*

PROOF. It is known that List Scheduling is $2-\frac{1}{m}$-competitive for any on-line makespan problem, independently of the order in which the jobs are considered for scheduling on the machines, as long as no idle time on any of the machines occurs, i.e., as long as no machine is not busy while there are still jobs available for processing. Clearly, List Scheduling combined with a round robin routine for dealing with the threads gives a schedule that satisfies this property. □

We have not been able to find an algorithm that has a competitive ratio equal to the lower bound of Theorem 3.1 in case $w < m-1$. It is not difficult to devise an input that shows that also in this case RRLS has a competitive ratio of at least $2-\frac{1}{m}$.

It can be shown that the lower bound in Theorem 3.1 can be improved in some cases. We can make a small improvement in case $\frac{m-1}{w} \geq 2$ to a lower bound of $2-\frac{1}{m+1-\lfloor \frac{m-1}{w} \rfloor}$ using essentially the same input sequence.

We used jobs with only three different processing times in the proof. That more different processing times might be useful is revealed by the lower bound of $7/4$ for the case $m=4$, establishing that RRLS is also best possible in this case.

Lemma 3.4 *Any algorithm for the multi-threaded REAL-TIME makespan problem on 4 machines is at least $2-\frac{1}{4}$-competitive.*

PROOF. That $7/4$ is the lower bound on the competitive ratio of algorithms in case $w \geq 3$, is already given in Theorem 3.1. Therefore, we give the proof for $w=2$. Moreover, threads can always be left empty, so that the lower bound

using two threads is also a lower bound for a problem with more than two threads.

We just give the adversarial thread construction. Both threads start with a job with processing time 3, followed by a job with processing time 2, on its turn followed by a job with processing time 1. Finally, there is a job with processing time 4 hidden behind the job that is scheduled last. It is a simple matter of going through all possibilities to see that no on-line algorithm cam avoid that the starting time of this last job becomes at least 3, leading to the ratio of 7/4. □

We do not expect, however, that $2 - 1/m$ is the right lower bound for all cases.

3.2 Latency on a single machine

In this subsection we prove the results on the latency problem defined on a single machine. The following trivial lemma allows us to prove results for both on-line paradigms simultaneously.

Lemma 3.5 *Lower and upper bounds on competitiveness of on-line algorithms for single machine problems under the BATCH and REAL-TIME models coincide.*

PROOF. It is obvious that the extra freedom there is in the BATCH model compared to the REAL-TIME model cannot be exploited on a single machine: any feasible solution within the BATCH model can also be obtained in the REAL-TIME model. □

First we derive the lower bound of w on the competitive ratio of the single machine problem with w threads, followed by an algorithm that is w-competitive. We will use the REAL-TIME model.

Theorem 3.4 *No algorithm for the w-threaded on-line single machine latency problem is better than w-competitive under the REAL-TIME model.*

PROOF. The proof for the single-threaded case ($w = 1$) is trivial. To prove the theorem for $w \geq 2$ we present the following adversarial input sequence. The w threads have each of them a job with processing time 1 at the first position. Then only one of the threads has after this job a sequence of n "small" jobs with processing time ϵ each. The adversary will "hide" this sequence of small jobs in the thread that will be explored last by the on-line algorithm. Notice that any on-line algorithm should schedule one of the first jobs in the threads last (among all the other first jobs in the threads).

For this sequence the optimal strategy is obviously to start processing the first job of the thread that contains the n small jobs. Then these n jobs with processing time ϵ are processed and afterwards the remaining $w-1$ jobs with processing time 1, leading to a sum of completion times

$$\begin{aligned} z^{OPT} &= 1+n+\frac{1}{2}n(n+1)\epsilon+(1+n\epsilon)(w-1)+\frac{1}{2}(w-1)w \\ &= n+\frac{1}{2}w+\frac{1}{2}w^2-\frac{1}{2}n\epsilon+\frac{1}{2}n^2\epsilon+\frac{1}{2}nw\epsilon. \end{aligned}$$

The sum of completion times of the on-line algorithm, that processes the w jobs with processing time 1 first, is given by

$$\begin{aligned} z^{OL} &= \frac{1}{2}w(w+1)+nw+\frac{1}{2}n(n+1)\epsilon \\ &= nw+\frac{1}{2}w+\frac{1}{2}w^2+\frac{1}{2}n\epsilon+\frac{1}{2}n^2\epsilon. \end{aligned}$$

Therefore, the ratio between the two tends to w if $\epsilon = o(n^{-2})$ and n tends to infinity. □

Before we present a w-competitive algorithm we will show that another very natural algorithm fails to be w-competitive. The off-line problem without threads is well-solved by the *Shortest Processing Time first* rule (see [13]): At each step among the unscheduled jobs the one with smallest processing time is selected to be scheduled next.

This suggests for the multi-threaded on-line problem the *Shortest Available Processing Time first* rule (SAPT): At each step, from among the unserved jobs at the head of the threads, schedule the one with the shortest processing time first.

The following instance shows that there is no constant c such that SAPT is c-competitive: there are two non-empty threads, one containing n jobs with processing time $1-\epsilon$ and the other one starting with a job with processing time 1 followed by n^2 jobs with processing time $\epsilon < 1$.

SAPT will first schedule all the jobs in the first thread and then all the jobs in the second thread. This yields

$$\begin{aligned} Z^{SAPT} &= \frac{1}{2}n(n+1)(1-\epsilon)+n(1-\epsilon)+1+n^2(n(1-\epsilon)+1)+\frac{1}{2}n^2(n^2+1)\epsilon \\ &\geq n^2n(1-\epsilon) = n^3(1-\epsilon) \end{aligned}$$

The optimal schedule processes all jobs of the second thread first, yielding

$$\begin{aligned} Z^{OPT} &= 1+n^2+\frac{1}{2}n^2(n^2+1)\epsilon+n(1+n^2\epsilon)+\frac{1}{2}n(n+1)(1-\epsilon) \\ &\leq 4n^2+2n^4\epsilon. \end{aligned}$$

Therefore, for any $c > 0$ we can choose n large enough and ϵ small enough such that $Z^{SAPT}/Z^{OPT} > c$.

The problem with SAPT is that it persists in exploring the same thread even if each job appearing at the head of it does not have a great advantage over the jobs on the head of other threads. The next algorithm avoids this anomalous behavior by balancing the total work done on each of the threads. We call the algorithm BWT (for Balance Work done per Thread).

- At each iteration a job on the head of one of the non-empty threads is processed. The job selected is the one for which the sum of its own processing time and the processing times of all preceding jobs in the same thread is minimal. Ties are broken arbitrarily.

We will show that BWT is best possible for the multi-threaded on-line single machine latency problem. To facilitate the exposition we introduce some notation. Let $S_n(i)$ denote the sum of the processing times of the jobs from thread i that have been processed before or at iteration n, $i = 1, ..., w$. Let $J_n(i)$ denote the job at the head of thread i at the beginning of iteration $n+1$, and $p_n(i)$ its processing time, $i = 1, ..., w$. Moreover, let J_n be the job that is processed at the n-th iteration.

In terms of this notation BWT processes at iteration $n+1$ job $J_{n+1} = J_n(i)$ if $p_n(i)+S_n(i) \leq p_n(j)+S_n(j)$, for all $j = 1, ..., w$ with thread j non-empty.

The following preliminary lemma brings us close to the proof of w-competitiveness of BWT.

Lemma 3.6 *If $J_{n+1} = J_n(i)$ then $S_{n+1}(i) \geq S_{n+1}(j)$, $j = 1, ..., w$, for all $n > 0$.*

PROOF. We will prove the lemma by induction on n. For $n = 0$ we have that if the first job selected to be processed is the one in front of thread i (i.e. $J_1 = J_0(i)$), then $S_1(i) = p_0(i) > 0$, and $S_1(j) = S_0(j) = 0$ for $j \neq i$.

We assume that the lemma is true up to iteration n. Now, consider the $(n+1)$-st iteration. Suppose that $J_{n+1} = J_n(i)$. Thus,

$$S_{n+1}(i) = S_n(i) + p_n(i), \tag{1}$$

$$S_{n+1}(j) = S_n(j), \ j \neq i. \tag{2}$$

We distinguish two cases.

- At iteration n a job has been processed from thread i as well. In that case

$$S_{n+1}(i) \geq S_n(i) \geq S_n(j) = S_{n+1}(j), \ j = 1, ..., w,$$

where the second inequality follows from the induction hypothesis.

- At iteration n a job has been processed from thread $k \neq i$. In this case

$$\begin{aligned} S_{n+1}(i) &= S_n(i) + p_n(i) \\ &= S_{n-1}(i) + p_{n-1}(i) \\ &\geq S_{n-1}(k) + p_{n-1}(k) = S_n(k) = S_{n+1}(k) \\ &\geq S_n(j) = S_{n+1}(j) \ \forall j \neq i, k. \end{aligned}$$

  where the first inequality comes from BWT's decision in the n-th iteration and the last inequality follows from the induction hypothesis.

□

Theorem 3.5 *BWT is w-competitive for the w-threaded on-line single machine latency problem.*

PROOF. Let the h-th job of the i-th thread be scheduled at iteration n of BWT. Its completion time is then given by

$$C_{ih}^{BWT} = \sum_{k=1}^{w} S_n(k) \leq w S_n(i)$$

due to Lemma 3.6. Besides, $S_n(i) \leq C_{ih}^{OPT}$, because also in the optimal solution at least the 1st up to the $h-1$-st job of the i-th thread must have been processed before the h-th job of this thread. Thus, for every individual job we have

$$C_{ih}^{BWT} \leq w S_n(i) \leq w C_{ih}^{OPT}.$$

The theorem follows since both optimal and BWT solution values are the sum of the individual completion times. □

We notice that BWT is a polynomial time algorithm.

3.3 Latency on multiple machines

We now prove the results on the latency problem defined on more than one machine. Apart from the problem with a single thread in the REAL-TIME model, which has competitive ratio 1, for all other problems on multiple machines no algorithm exists with constant competitive ratio. We will show it first for the BATCH model.

Theorem 3.6 *No on-line algorithm with constant competitive ratio exists under the BATCH model for the latency problem on more than one machine.*

PROOF. Consider the following adversarial sequence. Let $w-1$ threads be empty and one thread begin with a job with processing time h^2 followed by h jobs with processing time 1, for some $h \geq 1$.

Suppose that the on-line algorithm processes all the jobs on a unique machine. Its cost would be

$$h^2 + \sum_{i=1}^{h}(h^2+i) = (h+1)(h^2+h/2) \geq h^3.$$

An optimal algorithm would process the jobs on two machines yielding a cost of

$$h^2 + \sum_{i=1}^{h} i = h^2 + h(h+1)/2 \leq 2h^2.$$

Therefore, the ratio between the two algorithms cannot be bounded by any constant.

Thus, any on-line algorithm that could have a constant competitive ratio must schedule the given set of jobs on at least 2 machines. Suppose without loss of generality that it uses (among others) the machines 1 and 2.

Then, the adversary continues the thread with k groups of jobs (k will be defined later), leaving all other threads empty. The i-th group will be composed of n^i jobs, each one with processing time n^{-2i}. Group $i+1$ follows group i. From each group i at least n^i/m jobs must be assigned to the same machine. Each machine receiving at least n^i/m jobs from some group i will be called "*loaded*". Machines 1 and 2 will be called loaded, too. Now k is chosen as the first index for which the on-line algorithm loads a machine that was already loaded before; i.e. the adversary stops the sequence as soon as the on-line algorithm loads any machine for the second time. Thus, at least n^k/m jobs from group k are assigned to an already loaded machine. As each group loads at least one machine, loading all the machines (without re-loading any machine twice) will take at most $m-2$ groups. Therefore, $k \leq m-1$.

The contribution to the cost by the n^t/m jobs (at least) of group t that load a machine will be at least

$$\sum_{i=1}^{n^t/m} in^{-2t}, \quad t = 1, \ldots, k-1.$$

As group k will reload at least one machine already loaded, the cost of scheduling the n^k/m jobs (at least) of group k will be at least

$$\sum_{i=1}^{n^k/m}\left(\frac{n^{k-1}}{m}n^{-2(k-1)} + in^{-2k}\right) = \sum_{i=1}^{n^k/m}\left(\frac{1}{mn^{k-1}} + in^{-2k}\right),$$

where $\frac{1}{mn^{k-1}}$ is the earliest possible starting time of the set of jobs that loads a machine for the second time. In case machine 1 or 2 is reloaded the earliest

possible starting time of the reloading set of jobs is at least 1, yielding a cost of at least

$$\sum_{i=1}^{n^k/m}(1+in^{-2k}) > \sum_{i=1}^{n^k/m}(\frac{1}{mn^{k-1}}+in^{-2k}),$$

As a result, any on-line algorithm will have a cost of at least

$$\begin{aligned} & h^2+\frac{h}{m}+\sum_{t=1}^{k}\sum_{i=1}^{n^t/m} in^{-2t}+\sum_{i=1}^{n^k/m}\frac{n^{k-1}}{m}n^{-2(k-1)} \\ > & \sum_{i=1}^{n^k/m}\frac{n^{k-1}}{m}n^{-2(k-1)}=\frac{n}{m^2}. \end{aligned}$$

We notice that in case there are only two machines, $m = 2$, necessarily the first group must reload directly machine 1 or 2, $k = 1$, yielding a cost of at least $(n^k/m) > n/m^2$.

A better schedule is obtained by assigning the first $h+1$ jobs (the first with processing time h^2 and the subsequent h with processing time 1) to machine 1, and each of the subsequent groups to an empty machine. This is possible because, as we argued before, $k \leq m-1$. Its cost, which is obviously greater than or equal to the optimal cost, is

$$\begin{aligned} & h^2+\sum_{i=1}^{h}(h^2+i)+\sum_{t=1}^{k}\sum_{i=1}^{n^t} in^{-2t} \\ = \; & h^2+h^3+\frac{h(h+1)}{2}+\sum_{t=1}^{k}\frac{n^t+1}{2n^t} \\ \leq \; & 4h^3+\sum_{t=1}^{k}\frac{2n^t}{2n^t}=4h^3+k. \end{aligned}$$

Thus, any c-competitive algorithm will have $c \geq \frac{n}{m^2(4h^3+k)}$, which tends to infinity when n tends to infinity. □

Finally we prove the same result for the REAL-TIME model when the number of threads is greater than 1.

Theorem 3.7 *No on-line algorithm with constant competitive ratio exists under the REAL-TIME model with more than one thread for the latency problem on more than one machine.*

PROOF. Consider the following instance. There are 2 non-empty threads and $w-2$ empty threads. Both non-empty threads start with a job with processing time 1. One of the two jobs has to be scheduled first. The thread to which this job belongs, say thread 1, contains no further jobs. The other thread, the second thread, continues with $m-2$ jobs with processing time 1 and nm jobs with processing time ϵ.

After having scheduled the job of the first thread on one machine, an on-line algorithm cannot do better than scheduling the first $m-1$ jobs of the second thread on the remaining $m-1$ machines. Afterwards, the best the on-line algorithm can do is distributing the nm jobs with processing time ϵ evenly over the m machines, such that n of them are scheduled to each machine.

In this way, the cost of the on-line algorithm is:

$$m + m\sum_{i=1}^{n}(1+i\epsilon) = m(n+1) + m\frac{n(n+1)}{2}\epsilon$$

On the other hand, an optimal off-line algorithm, schedules all jobs of the second thread first and then the job of the first thread. Its cost is bounded from above by

$$m - 1 + \sum_{i=1}^{nm} i\epsilon + (nm\epsilon + 1) = m + \frac{nm(nm+3)}{2}\epsilon$$

Therefore, a lower bound on the competitiveness of an on-line algorithm is:

$$\frac{m(n+1) + m\frac{n(n+1)}{2}\epsilon}{m + \frac{nm(nm+3)}{2}\epsilon} = \frac{2(n+1) + n(n+1)\epsilon}{2 + n(nm+3)\epsilon},$$

which tends to infinity when $\epsilon = o(n^{-2})$ and n tends to infinity. □

4 Conclusions

We have provided a fairly complete picture of the competitiveness of on-line multi-threaded scheduling problems on parallel machines with objectives minimizing makespan and latency. One open question is if it is possible to find an on-line algorithm that has a competitive ratio which matches the lower bound for minimizing makespan under the REAL-TIME model in case the number of machines exceeds the number of threads by more than 1.

Given the adversarial instances one might expect that randomized algorithms can do substantially better for most of the problems. We leave this as an interesting subject for further studies.

A related problem that could be studied occurs in case of infinite multiple threads; i.e., each thread contains an infinite sequence of information. In that case different competitive measures can be defined, e.g., stating that an algorithm has to be competitive at every point in the decision process, or in the limit. In In [6, 7] it is shown that infinite multiple threads may yield competitiveness that is different from the case with finite threads. Moreover, both in the case of finite and infinite threads, fairness restrictions may be imposed on the way of processing jobs from the various threads.

Another variation of the problem is obtained when the processing times of the jobs become known only after completion.

5 Bibliography

References

[1] S. Albers. Better bounds for on-line scheduling. In *Proc. of the 29th Annual ACM Symp. on Theory of Computing (STOC)*, 1997, 130-139.

[2] H. Alborzi, E. Torng, P. Uthaisombut, S. Wagner, The k-client Problem. In *Proc. of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1997)*, pp.73–82.

[3] Y. Bartal, A. Fiat, H. Karloff, R. Vohra. New algorithms for an ancient scheduling problem. *J. Comput. Syst. Sci. 51*, 359-366, 1995, 359–366.

[4] A. Borodin and Ran El-Yaniv. *On-line Computation and Competitive analysis.* Cambridge University Press, Boston, 1998.

[5] E. Feuerstein. *On-line paging of structured data and multi-threaded paging.* Ph.D-Thesis, Università di Roma "La Sapienza", 1995.

[6] E. Feuerstein, A. Strejilevich de Loma. On multi-threaded paging. In *Proc. Seventh Annual International Symposium on Algorithms and Computation (ISAAC'96), Lecture Notes in Computer Science 1178*, Springer-Verlag, 1996, 417–426.

[7] E. Feuerstein, A. Strejilevich de Loma. *Different competitiveness measures for multi-threaded paging*, International Workshop on On-line Algorithms, OLA98, Udine, Italy, 1998.

[8] R.L. Graham, Bounds for certain multiprocessing anomalies. *Bell System Technical Journal 45*, 1966, 1563–1581.

[9] D.R. Karger, S.J. Philips, E. Torng. A better algorithm for an ancientll scheduling problem. *J. of Algorithms 20*, 1996, 400–430

[10] T. Kimbrel. *Online and offline preemptive two-machine job shop scheduling.* IBM Research Report RC 21520, 1999.

[11] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, D.B. Shmoys. Sequencing and scheduling: algorithms and complexity. In S.C. Graves, A.H.G. Rinnooy Kan, P.H. Zipkin (eds.). *Handbooks in Operations Research, Vol. 4.* North Holland, 1993.

[12] J. Sgall. On-line scheduling. In *On-line Algorithms - The state of the Art*, A. Fiat and G. Woeginger (eds.), Lecture Notes in Computer Science 1442, Springer-Verlag, 1998, 196–231.

[13] W.E. Smith. Various optimizers for single-stage production. *Naval Res.Logist. Quart. 3*, 1956, 59–66.