Optimal and online preemptive scheduling on uniformly related machines

Tomáš Ebenlendr [*] Jiří Sgall[†]

December 16, 2003

Abstract

We consider the problem of preemptive scheduling on uniformly related machines. We present a semi-online algorithm which, if the optimal makespan is given in advance, produces an optimal schedule. Using the standard doubling technique, this yields a 4 competitive deterministic and $e \approx 2.71$ competitive randomized online algorithms. In addition, it matches the performance of the previously known algorithms for the offline case, with a considerably simpler proof. Finally, we study the performance of greedy heuristics for the same problem.

1 Introduction

We consider the scheduling problem denoted $Q|pmtn|C_{\max}$ in the three-field notation. We are given m uniformly related machines, each characterized by its speed, and a sequence of jobs, each characterized by its processing time. If a job with processing time p is assigned to a machine of speed s it requires time p/s. Allowing preemption means that any job may be divided into several pieces that may be processed on several machines; in addition, the time slots assigned to different pieces must be disjoint. The goal is to

[*]Faculty of Mathematics and Physics, Charles University, Praha. E-mail ebik@ucw.cz

[†]Mathematical Institute, AS CR, Žitná 25, CZ-11567 Praha 1, Czech Republic. E-mail sgall@math.cas.cz. Partially supported by Institute for Theoretical Computer Science, Prague (project LN00A056 of MŠMT ČR) and grant 201/01/1195 of GA ČR.

minimize the length of the schedule (makespan), i.e., the time when all jobs are finished.

In the online problem $Q|online\text{ - }list,pmtn|C_{\max}$ the jobs arrive in a sequence and we have to assign each job immediately, without any knowledge of the future requests. This means that the algorithm has to determine immediately all the machines and all the time slots in which the current job is scheduled. The quality of an online algorithm is measured by the competitive ratio, which is the worst case ratio of the length of the produced schedule to the optimal (minimal) length. We also consider the semi-online variant in which an online algorithm is given in advance the value of the optimal makespan.

Finally, we also study the performance of two well-known greedy heuristics, LIST and LPT. LIST (LIST scheduling) is an online algorithm which schedules each coming job so that it finishes as soon as possible. For preemptive scheduling, it means that at each time the job is scheduled on the fastest available machine. LPT (Largest Processing Time first) uses the same strategy, but the jobs are sorted and processed from the largest one; i.e., it is no longer an online algorithm.

Preemptive scheduling on uniformly related machines is one of classical scheduling problems, yet it did not receive much attention in the online version. One motivation for research is the expectation that, similarly as for identical machines, the problem should be tractable, as the structure of the optimum is well understood, and at the same time it could provide a useful insight for constructing efficient randomized algorithms for the non-preemptive version.

We describe known results and our contribution in each area separately.

1.1 Optimal offline and semi-online algorithms

For offline preemptive scheduling the optimal solution was given already by Horvath *et al.* [14] and Gonzales and Sahni [13]. The algorithm of Gonzales and Sahni is more efficient: First, the total number of preemptions in the schedule is $2(m-1)$, which is the best possible bound for schedules with the optimal makespan. Second, its running time is $O(n + m \log m)$ and this is also best possible: the term $m \log m$ is caused by sorting the largest m jobs, which is necessary to obtain an optimal schedule.

A semi-online algorithm with the optimal makespan known in advance generating the optimal solution was previously known only for two machines,

see Epstein [8].

Our results. We give an optimal (1-competitive) semi-online algorithm for the studied problem. It generates at most $2(m-1)$ preemptions and runs in time $O(n + m \log m)$, thus it is as efficient as the offline algorithm of Gonzales and Sahni. In addition it has the advantage of being semi-online, i.e., the jobs can be scheduled in an arbitrary order after computing the optimal makespan.

Since the value of the optimal makespan can be easily computed, our algorithm can be also used as an efficient offline algorithm instead of the algorithm of Gonzales and Sahni. The efficiency is the same and we believe that our algorithm is conceptually significantly simpler and easier to understand.

1.2 Online algorithms

For non-preemptive scheduling on uniformly related machines, $Q|$*online - list*$|C_{\max}$, the first constant competitive algorithm was given by Aspnes *et al.* [1]; it is deterministic and its competitive ratio is 8. This was improved by Berman *et al.* [4]; they present a 5.828 competitive deterministic and a 4.311 competitive randomized algorithms. For an alternative very nice presentation see [3].

These algorithms can also be used for preemptive scheduling. Woeginger [19] observed that the optimal non-preemptive makespan is at most twice the optimal preemptive makespan for uniformly related machines. Consequently, the previous algorithms that do not use preemption are also 10.657 competitive deterministic and 8.622 competitive randomized algorithms for $Q|$*online - list,pmtn*$|C_{\max}$. No better preemptive online algorithms were known before.

All these algorithms are based on a semi-online algorithm and a doubling strategy for guessing the optimal value. Such doubling strategies are a very common tool in computer science. In the area of online algorithms, it leads to optimal algorithms for search on a line (also known as the cow-path problem) and its generalizations, both for deterministic and randomized algorithms, see [2, 16, 15]. In the context of online scheduling, it was used the first time in [17, 1].

The lower bounds are 2.438 for deterministic algorithms for $Q|$*online - list*$|C_{\max}$ [4] and 2 for randomized algorithms for $Q|$*online - list*$|C_{\max}$ as well as

for deterministic and randomized algorithms for $Q|\textit{online - list,pmtn}|C_{\max}$ [11].

Our results. Using our optimal (1-competitive) semi-online algorithm and the same doubling strategy as in the previous results, we obtain a 4 competitive deterministic and $e \approx 2.7183$ competitive randomized algorithms for $Q|\textit{online - list,pmtn}|C_{\max}$.

1.3 Greedy algorithms

Both LIST and LPT were previously studied for the non-preemptive case, $Q|\textit{online - list}|C_{\max}$. The competitive ratio of LIST is not constant, it is asymptotically $\Theta(\log m)$, see [6, 1] for the lower bound and the upper bound, respectively. This is very far from the optimum, however, sorting the jobs improves the performance dramatically. The competitive ratio of LPT is between 1.52 and 1.66 [12]; a better upper bound of 1.58 is claimed in [7], but the proof appears to be incomplete.

Our results. We show that with preemption, $Q|\textit{online - list,pmtn}|C_{\max}$, the situation is similar. The competitive ratio of LIST is $\Theta(\log m)$ and the competitive ratio of LPT is 2. More precisely, it is between $2 - 2/(m+1)$ and $2 - 1/m$.

1.4 Special cases: Two machines and identical machines

Let us conclude by two special cases in which we know the exact competitive ratio for preemptive scheduling from previous results.

The first case is that of identical machines (i.e., all the speeds are equal to 1), denoted by $P|\textit{online - list,pmtn}|C_{\max}$. Chen *et al.* [5] gives an optimal deterministic algorithm and a matching lower bound which works even for randomized algorithms. The optimal competitive ratio is $4/3$ for $m = 2$ and increases to $e/(e-1) \approx 1.582$ as $m \to \infty$.

For the special case of two related machines the optimal competitive ratio for preemptive scheduling, $Q2|\textit{online - list,pmtn}|C_{\max}$, was given independently by Wen and Du [18] and Epstein *et al.* [10] for any combination of speeds. If the ratio of the two speeds is $s \geq 1$, the optimal competitive ratio is $1 + s/(s^2 + s + 1)$ (this is equal to $4/3$ for $s = 1$ and decreases to 1

as $s \to \infty$); randomization does not help here either. The semi-online deterministic case of $Q2|$*online - list,pmtn*$|C_{\max}$ with jobs arriving sorted was completely analyzed in [9].

2 Preliminaries

Let M_i, $i = 1, \ldots, m$, denote the m machines and let $s_i \geq 0$ be the speed of machine M_i. We assume, w.l.o.g., that the machines are sorted so that $s_1 \geq s_2 \geq \ldots$. The input sequence of jobs is denoted $J = (p_j)_{j=1}^n$, where n is the number of jobs and $p_j \geq 0$ is the processing time of jth job.

Let OPT be the makespan of the optimal schedule. There are two easy lower bounds on OPT. First, OPT is bounded by the total work that can be done on all machines. Thus

$$OPT \geq \frac{\sum_{j=1}^n p_j}{\sum_{i=1}^m s_i}. \tag{1}$$

Second, OPT is bounded by the optimal makespan of any k largest jobs. An optimal schedule of k jobs uses only k fastest machines: if it used a slower machine, some faster one would be idle at the same time. Thus, for all $k = 1, \ldots, m$,

$$OPT \geq \frac{\sum_{i=1}^k \bar{p}_i}{\sum_{i=1}^k s_i}, \tag{2}$$

where $\bar{p}_i$ is the ith largest processing time. It is known that the actual value of OPT is the minimal value satisfying the conditions (1) and (2) [14, 13]. In particular, we can compute OPT in time $O(n + m \log m)$.

3 An optimal semi-online algorithm

We present the semi-online algorithm, which, given T, generates a schedule with makespan at most T, if some such schedule exists.

The idea of the algorithm is to schedule each job on two adjacent machines using one preemption so that it spans over the whole interval $[0, T)$, i.e., it is always running on one of the machines. Thus at each time exactly one of these machines remains idle. Such a pair of machines can be thought as one *virtual machine* with possibly changing speed. For the subsequent jobs, such virtual machines are used in place of real ones. See Figure 1 for an example

of such a step. If a job is too small, we create a machine with zero speed, to fit this scheme. To prove that this outline works, it remains to check that if a job is too long to fit on a machine, T is smaller than OPT, as one of the conditions (1) and (2) is violated.

3.1 Preliminaries

We assume in this section, w.l.o.g., that $p_j > 0$ for all j; the algorithm can skip the jobs with zero processing times. We define machines $M_{m+1}, M_{m+2}, \ldots$ as machines with zero speed.

We define a *virtual machine* as a set of adjacent machines, such that exactly one of them is idle at any given time in $[0, T)$. Let V_i denote the ith virtual machine. Scheduling a job on V_i at time t means that we schedule it on the machine in V_i that is idle at time t. The speed of virtual machine V_i is denoted $v_i(t)$; it is defined to be the speed of the unique machine in V_i which is idle at time t. Let $W_i = \int_0^T v_i(t)dt$ be the total work which can be done on V_i. Note that a virtual machine is defined so that all this work can be used by a single job.

3.2 Algorithm InTime

The algorithm is defined to schedule in the interval $[\mathit{offset}, \mathit{offset} + \mathit{time})$ instead of $[0, \mathit{time})$. This is used later in the online variants of the algorithm.

Invariants: The algorithm works with sets V_i and numbers W_i. The following properties of the virtual machines are invariants of the algorithm:

1. Sets V_i are virtual machines. Every real machine belongs to exactly one virtual machine. $W_i = \int_0^T v_i(t)dt$.
2. For all i and t, $v_i(t) \geq v_{i+1}(t)$. This also implies $W_i \geq W_{i+1}$.
3. Each job that is already processed is scheduled on machines that belong to a single virtual machine. For every i, there are exactly $|V_i| - 1$ jobs that are scheduled on the machines that belong to V_i.

Algorithm $InTime(T)$

- Initialization:

  procedure $InitInTime(offset, time)$

  $T := time$; $o := offset$

  For all i **do** $V_i := \{M_i\}$; $W_i := s_i \cdot T$

- Step: (add a job with processing time p to the schedule)

  function $DoInTime(p)$

  1. **Find** i **such that** $W_i \geq p > W_{i+1}$ **or return FALSE**
  2. **Find** t **such that** $\int_0^t v_i(\tau)d\tau + \int_t^T v_{i+1}(\tau)d\tau = p$
  3. **Schedule job** p **on** V_i **in time interval** $[o, o+t)$ **and on** V_{i+1} **in** $[o+t, o+T)$**.**
  4. $V_i := V_i \cup V_{i+1}$; $W_i := W_i + W_{i+1} - p$;
     For all $j > i$ **do** $V_j := V_{j+1}$; $W_j := W_{j+1}$
  5. **return TRUE**

- Main body:

  $InitInTime(0, T)$

  for $i := 0$ **to** n **do if not** $DoInTime(p_i)$ **fail**

Theorem 3.1 *Algorithm InTime is correct and generates a schedule with makespan at most T whenever such a schedule exists.*

Proof. All three invariants are satisfied after the initialization. Now we show that the function $DoInTime()$ maintains all the invariants and in addition it can fail only in line 1.

In line 2, t exists because the left-hand side of condition is continuous in t, and p lies between values of the left-hand side for 0 and T (i.e., W_{i+1} and W_i). The value t can be easily computed since the function $v_i(t)$ changes its value at most m times (see below for a more precise analysis).

Line 3 is correct, because virtual machines are idle by definition. The real schedule can be generated because the algorithm knows the mappings to determine $v_i(t)$.

Line 4 merges the two half-used virtual machines to one virtual machine that satisfy invariant 1. Invariant 2 is not broken because the two virtual

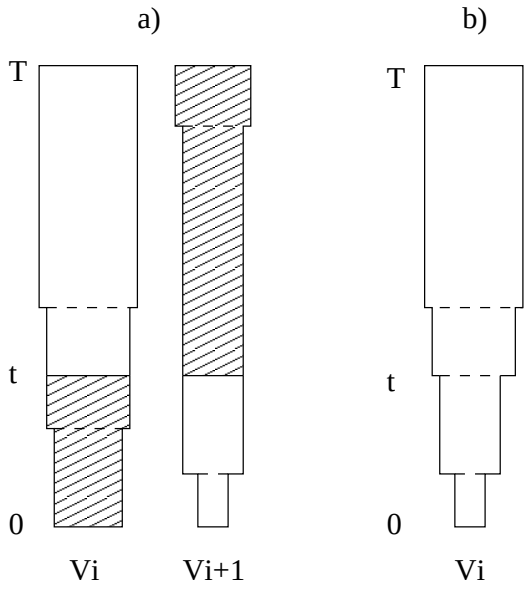

Figure 1: One step of the algorithm InTime. The vertical axis is the time. The width of the columns is proportional to the speed of the machines and thus the area is proportional to the work done on a given job. (a) Two machines with one job that spans over $[0, T)$. (b) The new virtual machine obtained by merging the two idle parts. There is a new postponed preemption at time t.

machines are adjacent. There are k real machines and $k-1$ jobs in V_i and l machines and $l-1$ jobs in V_{i+1}. We create one virtual machine with $k+l$ real machines and $k+l-1$ jobs by scheduling the actual job and merging the two virtual machines and invariant 3 is valid as well.

Consequently, when $DoInTime()$ does not returns FALSE in line 1, it generates a correct schedule for given job and returns TRUE.

If $DoInTime()$ returns FALSE it means that $p > W_1$ (since we always have a machine of speed zero). Let j be the index of this job. We know that $V_1 = \{M_1, \ldots, M_k\}$, for some k. In case $k \geq m$ we know that $\sum_{i=1}^{j} p_i > T \cdot \sum_{i=1}^{m} s_i$. By (1), $T < OPT$ and thus no schedule exists. In case $k < m$ we know that there are $k-1$ jobs scheduled on the machines of V_1. So, including the new job, we have k jobs that are together larger than the total work that can be done on the k fastest machines before time T. By (2), $T < OPT$ and no schedule exists again. □

Our algorithm can also be used as an optimal offline algorithm. As noted above, the exact preemptive optimum can be computed using conditions (1) and (2). Using the computed value as T in $InTime(T)$, the previous theorem guarantees that the produced schedule is optimal.

3.3 Efficiency of the algorithm

The number of preemptions There are two types of preemptions in the algorithm. *Immediate preemptions* are created by dividing a job between two virtual machines. *Postponed preemptions* are generated by scheduling on the virtual machine as its speed changes. It is clear that every immediate preemption generates at most one postponed preemption.

Define zero virtual machine as a set of real machines with zero speed. When scheduling on a non-zero virtual machine and a zero virtual machine, no immediate preemption occurs because the job is completed on the non-zero one. On the other hand, after scheduling on two non-zero machines, the number of non-zero machines decreases. Because we have m non-zero virtual machines after the initialization, the algorithm creates at most $m-1$ immediate preemptions and thus no more than $2m-2$ preemptions overall.

The time complexity and implementation To make the algorithm efficient, we use search trees to store both the list of machines and the lists of preemptions on the machines. Specifically, we use 2-3-trees, i.e., perfectly balanced trees with inner nodes with 2 or 3 successors, with items stored at leaves only. This standard data structure with N items needs in the worst case time $O(\log N)$ to perform search, insertion, delete, or merge of two trees such that all the keys in one tree precede all the keys in the other tree. In addition, we assume that the leaves are connected in a bidirectional list; in particular we always have a direct link to the first and last nodes in the tree and we can test in constant time if our key is smaller or larger than all the keys in the tree.

The non-zero virtual machines are stored in the 2-3-tree using the value of W_i as a key. The zero machines are not represented, when scheduling on the last represented machine, algorithm acts as if the next virtual machine exists and it is the zero one.

The virtual machine V_i is represented by its W_i, the preemption offset ω_i and a 2-3-tree of preemptions. Each preemption is represented by its time $\pi_{i,j}$, which is also used as a search key, its relative work $w_{i,j}$ and the index

of the corresponding real machine. The relative work $w_{i,j}$ is defined so that $w_{i,j} + \omega_i$ is the actual work that can be done on V_i before time $\pi_{i,j}$.

The algorithm needs to know the actual work that can be done before any given postponed preemption. The more complicated representation with ω_i is chosen so that the updates are more efficient. In particular, if the actual work of many preemptions needs to be updated by the same number the algorithm changes preemption offset ω_i instead of changing all the values $w_{i,j}$.

In line 1, the algorithm searches for V_i, but at first it looks at the last non-zero virtual machine in constant time. The search is done in $O(\log m)$. There are at most $m-1$ searches, as the number of virtual machines decreases after each search, and the total cost is $O(n + m \log m)$.

If $W_{i+1} = 0$, in line 2 the algorithm looks on the first preemption of V_i before doing search, so if job can be done before the first preemption the lines 2 to 4 take only constant time. If not, the algorithm searches for the appropriate value of t in time $O(\log m)$. For all jobs, the total time is at most $O(m \log m)$ because we delete at least one preemption if the job is longer than the first preemption. The complexity of scheduling a job in lines 3 and 4 is a constant plus $O(\log m)$ for each removed preemption. Thus the total complexity of this case for all jobs is at most $O(n + m \log m)$.

If $W_{i+1} > 0$, the algorithm is scheduling on two non-zero machines. Let Π_i be the current number of preemptions of V_i.

In line 2, the algorithm searches for the indices k and l satisfying $\pi_{i,k} \leq t < \pi_{i,k+1}$ and $\pi_{i+1,l} \leq t < \pi_{i+1,l+1}$. After this it computes the value of t. We claim that this can be done in time $O(\log \Pi_i \log \Pi_{i+1})$. Assume that $\Pi_i \leq \Pi_{i+1}$, otherwise use a symmetric argument. For any time τ, we may compute in time $O(\log \Pi_{i+1})$ the work that can be performed on V_{i+1} after time τ (using binary search on the preemptions of V_{i+1}). We perform a binary search for the correct value of k. In time $O(\log \Pi_i)$ we find a preemption on V_i which is approximately in the middle of the tree (e.g., the leftmost leaf in the second subtree). In time $O(\log \Pi_{i+1})$ we determine if t needs to be before or after this preemption and recurse. After at most two iterations the height of the remaining 2-3-tree decreases, thus after $O(\log \Pi_i)$ iterations we are done. After computing k, it is easy to compute l in time $O(\log \Pi_{i+1})$ and t in constant time. This finishes the proof of the claim.

In line 3, the time to schedule the job is a constant plus $O(\log m)$ for each removed preemption. Thus the total cost is at most $O(m \log m)$.

In line 4, the algorithm needs to merge the lists of preemptions. Note that

all the remaining preemptions on V_i occur after all the remaining preemptions on V_{i+1}. The algorithm first, in constant time, updates the offset ω_i so that $\omega_i + w_{i,j}$ is the actual work before any remaining preemption that can be done on the new merged virtual machine. Next it updates the offset and the relative work of preemptions in the smaller list so that the actual work does not change and $\omega_i = \omega_{i+1}$; this takes time $O(min(\Pi_i, \Pi_{i+1}))$. Finally, the 2-3-trees of preemptions are concatenated and the list of machines is updated, in time $O(\log m)$ or total $O(m \log m)$ for all occurrences of this case.

It remains to bound the total of the contributions $O(\log \Pi_i \log \Pi_{i+1} + min(\Pi_i, \Pi_{i+1}))$. We claim that the total of these contribution when a virtual machine with a non-zero real machines is created is at most $T(a) = C \cdot a \log_2 a$, for some constant C. When merging adjacent machines with a and b real machines, $a \geq b \geq 1$, the number of preemptions on them is at most a and b and thus the new contribution is, for some sufficiently large C', C'' and C, at most

$$\begin{aligned} C'(b + \log_2 b \log_2 a) &= C'(b + \log_2 b \log_2 b + \log_2 b \log_2 \frac{a}{b}) \\ &\leq C'' \cdot b(1 + \log_2 \frac{a}{b}) \leq C \cdot b \log_2(1 + \frac{a}{b}) \end{aligned}$$

This yields total time for creating a virtual machine with $a+b$ real machines at most

$$\begin{aligned} T(a) + T(b) + C \cdot b \log_2(1 + \tfrac{a}{b}) &= C \cdot \left(a \log_2 a + b \log_2 b + b \log_2 \left(\tfrac{a+b}{b}\right)\right) \\ &= C \cdot (a \log_2 a + b \log_2(a+b)) \\ &\leq C \cdot (a+b) \log_2(a+b), \end{aligned}$$

and the induction is complete. The algorithm possibly ends with more then one non-zero virtual machine, but $T(a) + T(b) \leq T(a+b)$ and there are only m non-zero real machines, thus the total contribution is at most $T(m) = O(m \log m)$. Summarizing, the total time complexity of the algorithm is $O(n + m \log m)$.

3.4 Generalizations

It is easy to generalize our algorithm so that the real machines change their speeds over time arbitrarily. In fact, this is what our virtual machines do. It

is necessary to preprocess the speed profiles so that at each time the machines are sorted according to their speeds; this is easy to do using additional preemptions to "glue" the initial virtual machines from pieces of different real machines. The same lower bounds (1) and (2) then hold for the optimum and the algorithm gives a matching schedule. Naturally, the time complexity and the number of preemptions will now depend on the speed profiles of the machines.

4 Doubling online algorithms

When the optimal makespan is not known in advance, we guess it and if the guess turns out to be too small, we double it. As mentioned above, this is a standard technique, and it can be improved using randomization. It is well known that the exponential distribution is the best choice for the initial guess and it is also not hard to optimize the multiplicative constants.

Algorithm *Double*()

- Initialization: $G := p_1/s_1$; $B := 0$; $InitInTime(B, G)$
- Step i: **while not** $DoInTime(p_i)$ **do** $B := B + G$; $G := 2 \cdot G$; $InitInTime(B, G)$

Theorem 4.1 *The algorithm Double is a* 4 *competitive deterministic online algorithm for preemptive scheduling on uniformly related machines.*

Proof. We divide the run of the algorithm in phases so that every phase begins with a call to $InitInTime(B, G)$. This ensures that the algorithm schedules the jobs in the interval $[B_p, B_p + G_p)$ in phase p. Intervals do not overlap because $B_{p+1} = B_p + G_p$.

The algorithm stops at the latest when $G \geq OPT$ because $DoInTime(p_i)$ does not fail then (every subsequence of J can be scheduled in time OPT). Let G_f be the last value of G. This yields $G_f \leq 2 \cdot OPT$ and $B_f = \sum_{p=1}^{f-1} G_p < \sum_{i=1}^{\infty} 2^{-i} \cdot G_f = G_f$. All jobs end before the time $B_f + G_f < 2 \cdot G_f \leq 4 \cdot OPT$. □

Algorithm *DoubleRand*()

- Initialization:

  $r := rand([0,1])$; (r is uniformly distributed in $[0,1]$)
  $G := e^r \cdot p_1/s_1$; $B := 0$; $InitInTime(B,G)$

- Step i: **while not** $DoInTime(p_i)$ **do** $B := B + G$; $G := e \cdot G$; $InitInTime(B,G)$

Theorem 4.2 *The algorithm DoubleRand is an e competitive randomized algorithm for preemptive scheduling on uniformly related machines.*

Proof. Define $k = \ln(OPT \cdot s_1/p_1)$, note that k is a constant and $k \geq 1$. The algorithm multiplies G no more than $\lceil k-r \rceil$ times, because $e^{r+\lceil k-r \rceil} \cdot p_1/s_1 \geq e^k \cdot p_1/s_1 = OPT$. Denote $z = \lceil k-r \rceil - (k-r)$, i.e., the fractional part of $k-r$. Then $z \in [0,1]$ is a random variable with the uniform distribution. The algorithm stops with $G_f \leq e^{r+\lceil k-r \rceil} \cdot p_1/s_1 = e^z \cdot OPT$. The expected makespan of the produced schedule is at most $\mathbb{E}[B_f + G_f] \leq \sum_0^\infty e^{-i} \cdot \mathbb{E}[G_f] \leq \frac{e}{e-1} \cdot \mathbb{E}[e^z] \cdot OPT = e \cdot OPT$. □

5 Greedy algorithms

The greedy rule for scheduling on related machines instructs us to schedule each coming job so that it ends as soon as possible. For preemptive scheduling, this is achieved by scheduling the job from time 0 on the fastest idle machine (if there is any), for every time t, until it is completed. Thus the first job is scheduled on the fastest machine. The second job on the second fastest, with preemption at the end of first job (if it is not completed earlier), and then on the fastest machine, etc. See Figure 2 for an example. This algorithm is called LIST scheduling. If, in addition, the jobs arrive ordered so that their sizes are non-increasing, the algorithm is called LPT (Largest Processing Time first).

We prove that both of these algorithms have asymptotically the same competitive ratio as in the non-preemptive case. However, note that this is not a straightforward consequence of the non-preemptive case, as the preemptive and non-preemptive versions of LIST can generate very different schedules on the same instance.

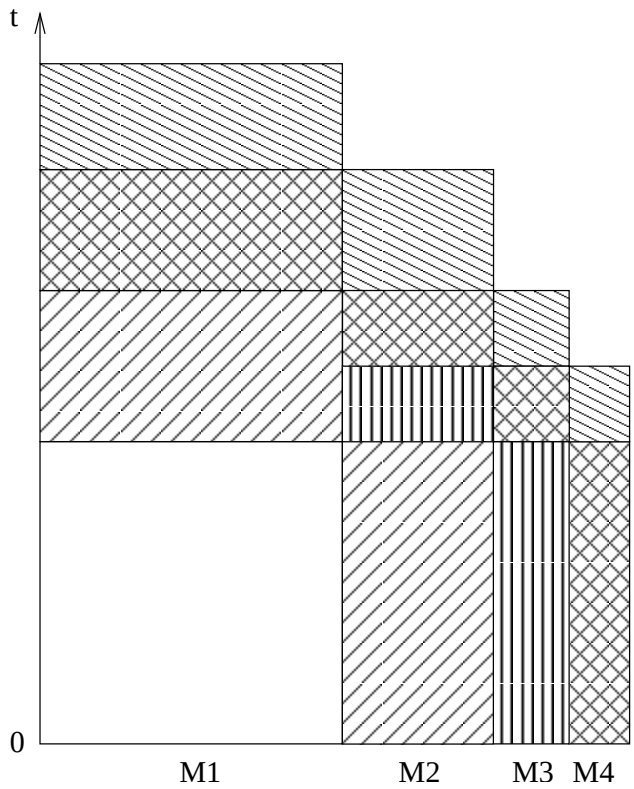

Figure 2: An example of a schedule generated by the LIST algorithm. Similarly shaded regions correspond to the same job.

Notation. For a given instance, $NOPT$ denotes a non-preemptive optimal schedule and its makespan. It is known that $NOPT \leq 2 \cdot OPT$. this is proved as an upper bound on the non-preemptive LPT algorithm in [19]; we use the same proof in Theorem 5.7.

For input sequences J and J', $J' \subseteq J$ denotes that J' is a subsequence of J (an arbitrary subset of jobs in the same order, not necessarily a contiguous segment). We say that J dominates J' if $p_i \geq p'_i$ for all i. In both cases trivially $OPT(J') \leq OPT(J)$.

5.1 Analysis of LIST

LIST is a simple online algorithm. However, we show that its competitive ratio is $\Theta(\log m)$ and thus it is not very good. Let $LIST(J)$ denote both the schedule generated by LIST on the input sequence of jobs J and its makespan.

We start by the upper bound. First we show that decreasing the size of jobs can only improve the schedule. This implies that removing short jobs decreases the makespan by a constant multiple of OPT; doing this in logarithmic number of phases we obtain the desired bound.

Lemma 5.1 *Let J and J' be such that J dominates J'. Then $LIST(J) \geq LIST(J')$.*

Proof. Consider the schedule after scheduling job p_j and let $t_{i,j}$, $i \leq j$, denote the ith smallest completion time of a job. Note that the sequence $T_j = (t_{i,j})_{i=1}^{j}$ is non-decreasing. Define $t_{0,j} = 0$. The job p_{j+1} is scheduled on machine M_{j+1-i} in the interval $[t_{i,j}, t_{i+1,j})$ and on M_1 after $t_{j,j}$; of course this hold only until it is completed (it may be too short to reach M_1 or even slower machines). The corresponding times for J' are denoted by $t'_{i,j}$ and the sequences T'_j.

We prove by induction on j that for all i, $t'_{i,j} \leq t_{i,j}$, i.e., T'_j is pointwise smaller than or equal to T_j. The induction assumption for zero jobs is trivial. The induction step says that $t'_{i,j} \leq t_{i,j}$ implies $t'_{i,j+1} \leq t_{i,j+1}$ assuming that $p'_{j+1} \leq p_{j+1}$.

By the induction assumption, the job p'_{j+1} in $LIST(J')$ is, at every time t, processed by a machine that is at least as fast as the machine that processes the job p_j in $LIST(J)$ at the same time. Moreover, $p'_{j+1} \leq p_{j+1}$. So the job p'_{j+1} must be completed earlier than or at the the same time when p_{j+1} is. The sequence T'_{j+1} is obtained from T'_j by inserting the completion time of p'_j, while T_{j+1} is obtained from T_j by inserting the completion time of p_j. Since a smaller or equal number is inserted into a pointwise smaller or equal sorted sequence T'_j, it necessarily remains pointwise smaller than or equal to T_j and the inductive claim follows. □

Lemma 5.2 *Let $J' \subseteq J$. Let t be a time such that all jobs $j \in J \setminus J'$ are completed at time t in $LIST(J)$. Then $LIST(J) \leq t + LIST(J')$.*

Proof. Let J'' be obtained from J' by replacing each job p_j by p'_j which is equal to the part of p_j not processed in $LIST(J)$ by time t. Then the schedule $LIST(J'')$ is exactly the same as the interval $[t, LIST(J))$ of the schedule $LIST(J)$, except for the shift by t. By its definition, J'' is dominated by J'. Using Lemma 5.1 we get $LIST(J) = t + LIST(J'') \leq t + LIST(J')$. □

Lemma 5.3 *The jobs with length at most p_{max}/m are completed by time $3 \cdot OPT$.*

Proof. First we claim that all jobs are started in $LIST$ by the time OPT. Otherwise all machines are busy until some time $t > OPT$, the total work processed is $\sum_{j=1}^{n} p_j \geq t \cdot \sum_{i=1}^{m} s_j > OPT \cdot \sum_{i=1}^{m} s_j$, a contradiction with (1).

Second, we claim that all machines with speed $s \leq s_1/m$ are idle at and after time $2 \cdot OPT$. Let I be the indices of the slow machines, $I = \{i : s_i \leq s_{max}/m\}$. The total capacity of the slow machines is small, namely $\sum_{i \in I} s_i \leq (m-1)s_1/m < s_1 \leq \sum_{i \notin I} s_i$ and thus $2 \sum_{i \notin I} s_i > \sum_{i=1}^{m} s_i$. Suppose some machine from I is not idle at time $2 \cdot OPT$. Then all machines from $M \setminus I$ are busy for all the time from 0 till $2 \cdot OPT$ and the total size of jobs processed on them is at least $2 \cdot OPT \cdot \sum_{i \notin I} s_i > OPT \cdot \sum_{i=1}^{m} s_i$, which is a contradiction with (1) as well.

It follows that after time $2 \cdot OPT$, each job is scheduled and moreover it is scheduled only on machines faster than s_1/m. If $p_j \geq p_{max}/m$, then the job p_j is completed by time $2 \cdot OPT + p_j/(s_1/m) \leq 2 \cdot OPT + p_{max}/s_1 \leq 3 \cdot OPT$. □

Lemma 5.4 *All the jobs with $p_j \leq 2p_{min}$ are completed by time $3 \cdot NOPT$.*

Proof. Let M_k denote the slowest machine used in a non-preemptive optimal schedule $NOPT$. Then $p_{min}/s_k \leq NOPT$. We also know that is M_k idle in $LIST$ at time $NOPT$, as otherwise $LIST$ would schedule more total work of jobs than $NOPT$. Then the any job with $p_j \leq 2p_{min}$ is completed by time $NOPT + p_j/s_k \leq NOPT + 2p_{min}/s_k \leq 3 \cdot NOPT$. □

Theorem 5.5 *$LIST$ is a $(9 + 6 \log_2 m)$ competitive for preemptive scheduling on related machines.*

Proof. Let J be the input sequence, let $k = \lceil \log_2 m \rceil$. We define job sequences $J_k \subseteq \cdots \subseteq J_1 \subseteq J_0 \subseteq J$; $p_j^{(i)}$ and $p_{min}^{(i)}$ then refer to the processing

times in the sequence J_i. Define J_0 as the sequence J without jobs with $p_j \leq p_{max}/m$. Define J_{i+1} as the sequence J_i without jobs with $p_j^{(i)} \leq 2p_{min}^{(i)}$. It follows that J_k is an empty sequence and $LIST(J_k) = 0$. By Lemmas 5.3 and 5.2, $LIST(J) \leq 3 \cdot OPT + LIST(J_0)$. By Lemmas 5.4 and 5.2, $LIST(J_i) \leq 3 \cdot NOPT + LIST(J_{i+1}) \leq 6 \cdot OPT + LIST(J_{i+1})$. Putting this together, we have $LIST(J) \leq 3 \cdot OPT + LIST(J_0) \leq 3 \cdot OPT + 6k \cdot OPT + LIST(J_k) \leq (9 + 6 \log_2 m) \cdot OPT$. □

Now we turn to the lower bound. The instance uses groups of machines with geometrically decreasing speeds, but the number of machines increases even faster so that the total capacity of slow machines is larger than the capacity of the fast machines.

Theorem 5.6 *The competitive ratio of LIST is at least* $\Omega(\log m)$.

Proof. Let us construct the hard instance. Choose integers a, b, g such that $a \geq 2b > 4$ and g is arbitrary. The set of machines consists of groups $G_0, \ldots, G_g$, where the group G_i consists of a^i machines with speed b^{-i}. The jobs are in similar groups named J_i, each containing a^i jobs of length b^{-i}. The input sequence is a concatenation of these groups starting with the smallest job, that is, $J = J_g, \ldots, J_0$. We name the phases by the groups of jobs processed in each phase (i.e., we start by phase J_g, note that the indices of the groups are decreasing).

By scheduling each J_k to G_k we get $OPT = 1$, so it remains to prove that $LIST \geq \Omega(\log m)$.

For $k = 1, \ldots, g$, let $i_k = \sum_{l=0}^{k-1} a^l$ be the number of processors in groups $G_0, \ldots, G_{k-1}$. The choice of a guarantees that the number of jobs in I_k is $a^k \geq 2i_k$.

To prove a lower bound on $LIST(J)$, we construct a sequence J' dominated by J. Each group J_k, for $k = g, \ldots, 1$, is replaced by a corresponding group J'_k, defined inductively below. The last group J_0 with a single job is unchanged, so the sequence J' is defined as the concatenation of the groups $J'_g, \ldots, J'_1, J_0$.

To modify the group J_k, consider the schedule $LIST(J'_g, \ldots, J'_{k+1}, J_k)$. All the jobs in J_k have the same length, thus their completion times are non-decreasing. We construct a group J'_k by shortening the last i_k jobs in J_k so that their completion times are equal. Denote this common completion time τ_k. For $k = 0$, the sequence is not modified and τ_0 is the completion time of the single job in J_0. Define also $\tau_{g+1} = 0$.

We prove by induction that, for each $k = g, \ldots, 0$, (i) $1 \geq \tau_k - \tau_{k+1} \geq \Omega(1)$ and (ii) in the schedule $LIST(J'_g, \ldots, J'_{k+1}, J'_k)$, all the i_k processors in groups $G_0, \ldots, G_{k-1}$ are busy until time τ_k and all the machines are idle after time τ_k.

To start, note that (ii) for $k = g+1$ holds trivially. Using (ii) for $k+1$, it is feasible to schedule all jobs in J_k on machines G_k starting from time τ_{k+1} without preemptions and completing at time $1 + \tau_{k+1}$. The greedy schedule may schedule the jobs on faster processors which can only decrease their completion times and thus the first inequality of (i) holds. Using (ii) for $l > k$, it follows that the work done on any job in J_k before time τ_{k+1} is at most

$$\sum_{l=k+1}^{g} (\tau_l - \tau_{l+1})\, b^{-l} \quad \leq \quad \sum_{l=k+1}^{\infty} b^{-l} \quad = \quad \frac{1}{b-1} b^{-k}.$$

Consequently, all the completion times of jobs in J_k are larger than τ_{k+1}; thus $\tau_k > \tau_{k+1}$ and (ii) holds for k by the structure of $LIST$ schedule. Since the first i_k jobs from I_k are not shortened, the work done on the machines in $G_0, \ldots, G_{k-1}$ between τ_{k+1} and τ_k is at least

$$i_k \left(b^{-k} - \frac{1}{b-1} b^{-k} \right) \geq \frac{b-2}{b-1} b^{-k} a^{k-1}.$$

The total capacity of the machines in $G_0, \ldots, G_{k-1}$ is $\sum_{l=0}^{k-1} a^l b^{-l} < a^k b^{-k}$, using $a/b \geq 2$. Thus

$$\tau_k - \tau_{k+1} \geq \frac{\frac{b-2}{b-1} b^{-k} a^{k-1}}{a^k b^{-k}} = \frac{(b-2)}{a(b-1)} = \Omega(1);$$

this finishes the proof of (i) and the whole induction.

Using Lemma 5.1, $LIST(J) \geq LIST(J') = \tau_0 = \sum_{k=0}^{g} (\tau_k - \tau_{k+1}) \geq g \cdot \Omega(1) = \Omega(\log m)$. □

5.2 The analysis of LPT

LPT is a simple approximation algorithm (no longer online), thus it is interesting to know its performance. We show that the approximation ratio of LPT for preemptive variant is between $2 - \frac{2}{m+1}$ and $2 - \frac{1}{m}$. The proof of the upper bound is same as proof for non-preemptive case, which is also described in [19]. Non-preemptive LPT is there used as an upper bound on NOPT in comparison to OPT.

Theorem 5.7 *Both preemptive and non-preemptive LPT produce schedules with makespan at most* $2 - \frac{1}{m}$ *times (preemptive) OPT.*

Proof. Given an arbitrary $\alpha > 1$, consider an instance with $LPT \geq \alpha \cdot OPT$ such that the number of jobs n is minimal. We may assume that $m \leq n$ as otherwise removing the $m - n$ slowest machines leads to the same schedules. Let L_i be the load of the machine M_i (the sum of processing times of jobs assigned to it) before scheduling the last job p_n. We have $p_n + \sum_{i=1}^{m} L_i = \sum_{i=1}^{n} p_i \geq np_n$, since p_n is the smallest job. From the minimality of the counterexample we know that scheduling p_n on any machine leads to its completion no earlier than at time $\alpha \cdot OPT$. Thus $L_i + p_n \geq s_i \alpha \cdot OPT$ for all i. Summing over all i we get

$$(m-1)p_n + \sum_{i=1}^{n} p_i \quad = \quad \sum_{i=1}^{m} (L_i + p_n) \quad \geq \quad \alpha \cdot OPT \cdot \sum_{i=1}^{m} s_i \quad \geq \quad \alpha \sum_{i=1}^{n} p_i$$

and thus $(m-1)p_n \geq (\alpha - 1) \sum_{i=1}^{n} p_i \geq (\alpha - 1)np_n \geq (\alpha - 1)mp_n$. This implies $\alpha \leq 2 - \frac{1}{m}$. □

An almost matching lower bound is shown by an instance consisting of m identical machines and $m + 1$ identical jobs. Assuming unit jobs and machines, LPT produces a schedule of makespan 2 (no preemptions are used), while the optimal preemptive makespan is $\frac{m+1}{m}$. This yields a lower bound of $2 - \frac{2}{m+1}$ on the competitive ratio.

Conclusions

Our main result is an improvement of online algorithms for scheduling on uniformly related machines, $Q|online\text{ - }list,pmtn|C_{\max}$, by a factor of more than 2. Still, some gap still remains. Our intuition is that, similarly as for identical machines, $P|online\text{ - }list,pmtn|C_{\max}$, randomization should not help for preemptive online scheduling and thus the deterministic 4 competitive algorithm can be improved.

Acknowledgment We are grateful to Yossi Azar for useful discussions.

References

[1] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. *Journal of the ACM*, 44:486–504, 1997.

[2] R. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, 106:234–252, 1993.

[3] A. Bar-Noy, A. Freund, and J. Naor. New algorithms for related machines with temporary jobs. *Journal of Scheduling*, 3:259–272, 2000.

[4] P. Berman, M. Charikar, and M. Karpinski. On-line load balancing for related machines. *Journal of Algorithms*, 35:108–121, 2000.

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

[6] Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9:91–103, 1980.

[7] G. Dobson. Scheduling independent tasks on uniform processors. *SIAM Journal on Computing*, 13:705–716, 1984.

[8] L. Epstein. Bin stretching revisited. *Acta Informatica*, 39:97–117, 2003.

[9] L. Epstein and L. M. Favrholdt. Optimal preemptive semi-online scheduling to minimize makespan on two related machines. *Operations Research Letters*, 30:269–275, 2002.

[10] L. Epstein, J. Noga, S. S. Seiden, J. Sgall, and G. J. Woeginger. Randomized on-line scheduling for two related machines. *Journal of Scheduling*, 4:71–92, 2001.

[11] L. Epstein and J. Sgall. A lower bound for on-line scheduling on uniformly related machines. *Operations Research Letters*, 26(1):17–22, 2000.

[12] D. K. Friesen. Tighter bounds for LPT scheduling on uniform processors. *SIAM Journal on Computing*, 16:554–560, 1987.

[13] T. F. Gonzales and S. Sahni. Preemptive scheduling of uniform processor systems. *Journal of the ACM*, 25:92–101, 1978.

[14] E. Horwath, E. C. Lam, and R. Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM*, 24:32–43, 1977.

[15] M.-Y. Kao, Y. Ma, M. Sipser, and Y. Yin. Optimal constructions of hybrid algorithms. *Journal of Algorithms*, 29:142–164, 1998.

[16] M.-Y. Kao, J. H. Reif, and S. R. Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. *Information and Computation*, 131:63–79, 1996.

[17] D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on-line. *SIAM Journal on Computing*, 24:1313–1331, 1995.

[18] J. Wen and D. Du. Preemptive on-line scheduling for two uniform processors. *Operations Research Letters*, 23:113–116, 1998.

[19] G. J. Woeginger. A comment on scheduling on uniform machines under chain-type precedence constraints. *Operations Research Letters*, 26:107–109, 2000.