Scheduling Parallel Machines On-line

David B. Shmoys* Cornell University
Joel Wein† MIT
David P. Williamson‡ MIT

Abstract

We study the problem of scheduling jobs on parallel machines when the existence of a job is not known until an unknown release date and the processing requirement of a job is not known until the job is processed to completion. We demonstrate two general algorithmic techniques for converting existing polynomial-time algorithms that require complete knowledge about the input data into algorithms that need less advance knowledge. We also prove information-theoretic lower bounds on the lengths of on-line schedules for several basic parallel machine models, then show that our algorithms construct schedules with lengths that either match or come within a constant factor of the lower bounds.

1 Introduction

The scheduling of a set of tasks on parallel machines is a basic problem in combinatorial optimization, with a number of increasingly important applications. There is a rich literature on parallel machine scheduling, and on deterministic scheduling in general, but the overwhelming majority of these results assume that a complete specification of the instance is available before the algorithm begins to construct a schedule. This fails, however, to capture many scheduling problems that arise in practice. Consider, for example, the allocation of jobs to the processing units of a multiprocessor: the scheduler does not in advance have complete knowledge of a job's running time, or of what jobs will be created and require processing in the future. In this paper we will study on-line algorithms – algorithms that work without any clairvoyant assumptions – for the most basic types of parallel machine models. Our algorithms are based on two rather general techniques that allow us to convert algorithms that need more complete knowledge of the input data into ones that need less advance knowledge.

When on-line scheduling has been studied in the past, the models that have been considered were typically of the following form: the existence of a job is unknown until a certain *release date*, at which point the processing requirement for that job is completely specified. We will consider more realistic models, where the processing requirement of a job is also unknown when it starts processing, and can only be determined by processing the job and observing how long it takes to be completed. In fact, our results show that the traditional sort of on-line scheduling problem is provably not much harder than its off-line analogue whereas the lack of knowledge about job sizes can drastically affect the quality of solutions that can be obtained.

We will study the three basic types of parallel machine models [10]. In each, there are n jobs to be scheduled on m machines. Each machine can process at most one job at a time, and each job must be processed in an uninterrupted fashion on one of the machines. In the most general setting, the machines are *unrelated*: job j takes $p_{ij} = p_j/s_{ij}$ time units when processed by machine i, where p_j is the *processing requirement* (or *size*) of job j and s_{ij} is the speed of machine i on job j. If the machines are *uniformly related*, then each machine i runs at a given speed s_i for all jobs j, and the processing time p_{ij} is given by p_j/s_i. Finally, for *identical* machines, we assume that $s_i = 1$ for each machine i. If C_j denotes the time at which job j completes processing in a schedule, then the *makespan* or *length* of the schedule is $C_{\max} = \max_j C_j$. For a given instance $\mathcal{I}$, our objective is to find a schedule of minimum length $C^*_{\max}(\mathcal{I})$.

In an off-line setting, these three types of parallel machine models have been studied extensively. The associated scheduling problems are all strongly $\mathcal{NP}$-

*Research partially supported by NSF PYI Award CCR-89-96272 with matching support from UPS and Sun, and by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through NSF grant DMS-8920550.

†Research partially supported by an ARO graduate fellowship, by NSF PYI award CCR-89-96272, with matching funds from Sun Microsystems, and UPS, and by DARPA Contract N00014-89-J-1988.

‡Partially supported by an NSF graduate fellowship, by DARPA Contracts N00014-89-J-1988 and N00014-87-K-825, and Air Force Contract AFOSR-89-0271.

hard [7], and polynomial approximation schemes are known when the machines are either identical or uniformly related [12, 13]. For unrelated machines, obtaining a solution better than $(3/2)C^*_{\max}$ is $\mathcal{NP}$-hard, whereas a schedule of length at most $2C^*_{\max}$ can be found in polynomial time [20]. We will also consider the *preemptive* versions of these models, in which a job may be interrupted on one machine and continued later (possibly on another machine) without penalty. In each of these three models, there is a polynomial-time off-line algorithm to find an optimal preemptive solution [22, 14, 19].

We shall evaluate on-line algorithms in terms of their *competitive ratio* [26]. Let $C^A_{\max}(\mathcal{I})$ be the makespan of a deterministic on-line algorithm A on instance $\mathcal{I}$. Algorithm A is said to have *competitive ratio* c (or is said to be *c-competitive*) if $C^A_{\max}(\mathcal{I}) \leq c \cdot C^*_{\max}(\mathcal{I}) + O(1)$ for all problem instances $\mathcal{I}$. If A is a randomized algorithm, then A is said to have competitive ratio c (or is said to be c-competitive) if $E[C^A_{\max}(\mathcal{I})] \leq c \cdot C^*_{\max}(\mathcal{I}) + O(1)$ for all instances $\mathcal{I}$, where the expectation is taken over all random choices of the algorithm A. Although these notions apply to algorithms without any restrictions on their running times, we will focus on polynomial-time on-line algorithms, rather than the purely information-theoretic analogue. Nonetheless, our lower bounds are based on information-theoretic arguments.

In a non-preemptive model, it may be unrealistic to assume that once a job is started, it must be run until its (unknown) completion time, without any form of recourse. A central aspect of our models is that we introduce the notion of *restarts*: a job may be canceled and later started again, but it is started again from scratch. For example, in the uniformly related machine model, we may wish to cancel a job that is taking longer than "anticipated", and then start it again on a faster machine.

The results of this paper are as follows. We introduce two general techniques to convert off-line algorithms into algorithms that require less initial information. Using the first technique, we show that we can focus on the case when all jobs are available at time 0, since the situation in which there are unknown job arrivals and unknown processing times can be reduced, with only a factor of 2 increase in the competitive ratio, to one in which there are only unknown processing times. This result also holds when comparing a model in which there are only unknown arrivals and its off-line equivalent. As a consequence, we consider the situation in which all jobs (of unknown size) are available to be scheduled from the start. For both uniformly related and unrelated machines, we use our second technique to convert off-line algorithms into algorithms that need not be given the processing time of each job. Nonetheless, the resulting on-line algorithms do not suffer too great a degradation in the quality of the solution produced.

It is quite simple to obtain tight bounds for the identical machine case: one of the oldest results in scheduling theory is an on-line algorithm of Graham [8], which produces a schedule of length at most $(2 - \frac{1}{m})C^*_{\max}$; we give a straightforward proof that this is exactly the best possible ratio. We also give an identical tight bound on the competitive ratio obtainable in the preemptive variant. This has the important consequence that, although complexity theory shows that there is a fundamental difference between the preemptive and non-preemptive models, this difference disappears when scheduling jobs on-line. We also show that randomization is of little help to the scheduler, proving that no randomized algorithm can achieve competitive ratio better than $(2 - O(\frac{1}{\sqrt{m}}))$, even against an oblivious adversary. This result is in sharp contrast to other recent work in on-line algorithms, in which randomness has been shown to significantly increase the performance of the algorithms [18, 27].

We then show that on-line scheduling on uniformly related machines is much harder than on identical machines. This is also quite different from the the off-line setting, where results for identical machines have typically extended to the case where machines run at different speeds. In our on-line model, we show that this generalization does make the problem significantly harder: we prove that the optimal competitive ratio is $\Theta(\log m)$. We generalize this model to unrelated machines by assuming that for each job, the relative speeds of the machines are known, but its size is unknown. In this setting, we can also obtain an on-line algorithm with an $O(\log n)$ competitive ratio. Once again, we also give identical results for the preemptive variants of these models. For uniformly related machines, we also show how to take advantage of the situation that the relative speeds of the machines are not too different, and give an $O(\log R)$-competitive algorithm, where R is the ratio of the fastest-to-slowest machine speeds. Finally, we can show that this bound is tight, in the following sense: we prove a lower bound of $\Omega(\log R)$ on the competitive ratio of any deterministic on-line scheduling algorithm for the scenario in which the ratio of machine speeds is equal to R, $R < m$.

On-line algorithms have been studied for a vari-

ety of problem domains. Some of the oldest of these results are for the bin-packing problem. When the number of bins is fixed, on-line bin-packing can be interpreted as a type of on-line scheduling, where the jobs are given in a list and scheduled in turn. The job currently being scheduled is completely specified, but the jobs later in the list are completely unknown. This model, however, is rather different from the ones we consider. There are many recent results on on-line algorithms for problem domains that range from classic problems in combinatorial optimization [15, 18, 27], to various problems in data and memory management [17, 26], to the k-server problem [6, 21].

Relatively little work has been done on the model of on-line scheduling that we consider. Chandra, Karloff, and Vishwanathan [2] proposed studying on-line scheduling with unknown processing times, and analyzed the problem of minimizing the average completion time on a single machine with preemption. In addition to the algorithms for identical machines given by Graham [8], the only other work for parallel machines known to the authors is that of Jaffe [16] and Davis and Jaffe [4]. Davis and Jaffe show that in a restricted model without restarts, an on-line algorithm for non-preemptive scheduling of uniformly related machines cannot have competitive ratio better than $\Omega(\sqrt{m})$. Jaffe gives an algorithm for this case with competitive ratio $O(\sqrt{m})$. Very recently, Feldmann, Sgall, and Teng studied the problem of scheduling jobs of unknown processing requirement on a mesh of processors, where each job requires a submesh of a specified size [5].

2 Unknown Release Dates

Our model of on-line scheduling includes both unknown release dates for jobs and unknown job sizes. In this section we will show that, with respect to minimizing schedule length, the first element is much less important than the second element. We will show that if the release dates are unknown, then we can assume that all jobs *are* always available, and repeatedly use an algorithm that works in this environment; the feasible schedules produced by this simulation are of only somewhat lesser quality than can be obtained in the special case. This result does not depend on the remaining specifics of the scheduling environment; in particular, it allows us to use off-line algorithms to obtain algorithms that can handle unknown release dates (but where the processing times are known once released), as well as allowing us to focus on on-line algorithms in the case when all jobs are released at time 0.

Theorem 2.1 Let A be a polynomial-time scheduling algorithm which works in an environment in which each job to be scheduled is available at time 0 and always produces a schedule of length at most $\rho C^*_{\max}$. For the analogous environment in which the existence of a job is unknown until its release date, there exists another polynomial-time algorithm A' that works in this more general setting, and produces a schedule of length at most $2\rho C^*_{\max}$.

Proof: Let $\mathcal{I}$ be an instance including jobs with unknown release dates, and let S_0 be the set of jobs available at time 0. The scheduler applies algorithm A and schedules the jobs in S_0, finishing at time F_0. Let S_1 be the set of jobs released in time $(0, F_0]$. The scheduler now, at time F_0, applies algorithm A to schedule S_1, finishing at time F_1. In general let S_{i+1} be the set of jobs released in $(F_{i-1}, F_i]$, and let F_i be the point in time when the schedule for S_i completes. At time F_i, the scheduler uses algorithm A to schedule the jobs in S_{i+1}. Let F_k be the finishing point of the entire schedule. (See Figure 1.)

To analyze the length of the resulting schedule, consider the modified problem instance $\mathcal{I}'$ where the jobs in S_k are released at time F_{k-2}. Since these jobs are released at an earlier point in time in $\mathcal{I}'$ than in $\mathcal{I}$, certainly $C^*_{\max}(\mathcal{I}') \leq C^*_{\max}(\mathcal{I})$.

Now note that $F_{k-2} + F_k - F_{k-1} \leq \rho C^*_{\max}(\mathcal{I}')$, since the jobs in S_k are not released until F_{k-2} and the properties of algorithm A guarantee that $F_k - F_{k-1}$ is within a factor of ρ of the shortest schedule for S_k. Similarly, $F_{k-1} - F_{k-2} \leq \rho C^*_{\max}(\mathcal{I})$. Therefore $F_k \leq 2\rho C^*_{\max}(\mathcal{I}') \leq 2\rho C^*_{\max}(\mathcal{I})$. ■

This theorem is very general, in that it can be applied to a number of different types of scheduling scenarios. In particular, it shows that to produce an on-line algorithm for our full on-line model, we can modify an algorithm for the case in which all jobs are available at time 0 and processing times are unknown, increasing the competitive ratio of the algorithm by only a factor of 2. Further, the theorem applies not only to problems of parallel machine scheduling but also to the entire class of shop scheduling problems, including open shop, flow shop and job shop [25]. In addition, it applies to the scheduling model of Feldmann, Sgall and Teng [5]; their algorithms only worked when all jobs were available at time 0. Our theorem generalizes their result to a $\Theta(\sqrt{\log \log m})$ on-line algorithm even when jobs have unknown release dates.

Finally, using [13] and [20], we obtain the following corollary:

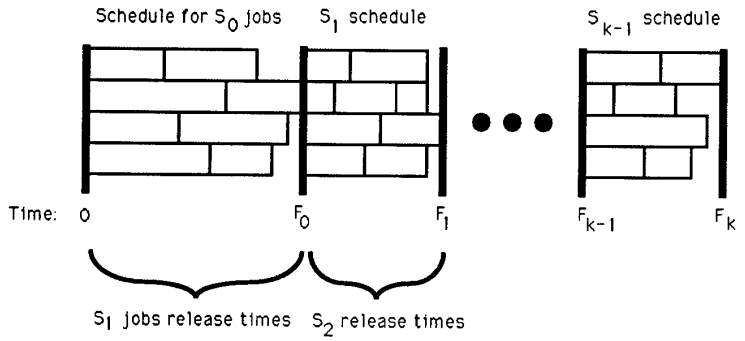

Figure 1: Scheduling with unknown release dates

Corollary 2.2 If job release dates are unknown, but once a job arrives its size is known, there is a polynomial-time *on-line* algorithm for scheduling uniformly related machines that comes within a factor of $(2+\epsilon)$ of optimal and a polynomial-time *on-line* algorithm for scheduling unrelated machines that comes within a factor of 4 of optimal.

The case of identical machines is somewhat easier, since it is not hard to show that with both unknown release dates and processing times, a natural extension of the on-line algorithm of Graham produces a schedule of length at most $(2 - \frac{1}{m})C^*_{\max}$ (see, for example, [11]).

Despite the fact that unknown release dates do not make a scheduling problem *much* more difficult, we can show that they do make it more difficult to schedule machines near-optimally.

Theorem 2.3 There is no on-line algorithm for non-preemptive scheduling of identical machines with unknown release dates but known processing requirements with competitive ratio better than 10/9.

Proof: (Omitted). Note that this is not the case when preemption is allowed [24].

In light of the results in this section, for the remainder of this paper we shall focus on scheduling environments in which all jobs are available to be scheduled at time 0.

3 Algorithms for On-Line Scheduling

In this section we will present on-line scheduling algorithms for the basic parallel machine models. We first note that in the case of identical machines, the well-known list scheduling algorithm of Graham [8] always comes within a factor $(2 - \frac{1}{m})$ of the optimal length schedule, and comes within the same bound of the optimal preemptive schedule length. In list scheduling, the scheduler takes any list of jobs and, whenever a machine becomes available, places the next job on the list on that machine. Since list scheduling does not depend on the sizes of the jobs, list scheduling is an on-line algorithm, and so we see that this yields a $(2 - \frac{1}{m})$ competitive ratio.

For the other machine models, we will present a general technique which yields four algorithms: an $O(\log m)$-competitive algorithm for both preemptive and non-preemptive uniformly related machines, and an $O(\log n)$-competitive algorithm for both preemptive and non-preemptive unrelated machines. We will also present an algorithm for non-preemptive uniformly related machines which has a competitive ratio of $O(\min(\log m, \log(s_1/s_m))$, assuming $s_1 \geq s_2 \geq \cdots \geq s_m$.

3.1 The General Technique

Our general algorithm depends on the existence of either polynomial-time algorithms or polynomial-time *ρ-approximation algorithms* for scheduling in the various machine models. A ρ-approximation algorithm produces a schedule no longer than ρ times the length of the optimal schedule. Throughout this paper we use log to refer to the logarithm base 2.

Theorem 3.1 Suppose that there is a ρ-approximation algorithm A for the [non-preemptive/preemptive] [uniformly related/unrelated] machine problem, and let $\mathcal{I}$

be an instance of this problem. Then there is an on-line scheduling algorithm which produces a schedule no longer than $(4\rho \log n + 4\rho \log 2\rho + 1)C^*_{\max}(\mathcal{I})$.

Proof: The on-line algorithm works by repeatedly applying algorithm A to the jobs, after guessing the size of each job. Given a schedule produced by the algorithm A, our on-line algorithm will run each job at the particular time interval and on the particular machine specified by the schedule. In the preemptive case, the job may not have finished all its processing by the end of the schedule, in which case we preempt the job. In the non-preemptive case, we cancel the job if it is not completely processed in the time allotted.

For simplicity of notation, we will assume without loss of generality that the speeds are normalized so that for each job j the fastest machine $i(j)$ has speed $s_{i(j)j} = 1$. One result of this assumption is that any job of size p_j takes time p_j to complete on the machine that processes it fastest.

The complete on-line algorithm is below.

Step 1 Pick any job j and run it to completion on a machine $i(j)$ such that $s_{i(j),j} = 1$. Let the time that this takes be denoted by Δ.

Step 2 Let $q = \Delta/\rho n$.

Step 3 Construct a schedule with algorithm A for all jobs that have not yet completed, using speeds s_{ij} and setting $p_j \leftarrow q$ for all remaining jobs j. Run the jobs in this schedule, preempting or cancelling all jobs that do not complete in the time allotted to them by the schedule.

Step 4 If any jobs have not yet completed, set $q \leftarrow 2q$ and go to Step 3.

Let $C^*_{\max}$ be the length of the optimal schedule. We will now analyze the algorithm and the length of the schedule it produces. First, in Step 1, the time Δ taken by job j on machine $i(j)$ is less than $C^*_{\max}$, since the optimal schedule can be no shorter than the time taken by any job running on the machine which processes it the fastest.

Next, we show that the first iteration of Step 3 produces a schedule no longer than $\Delta \leq C^*_{\max}$. We could construct a schedule by assigning each of the n jobs on the machine that processes it the fastest. In the worst case, all n jobs would be assigned to the same machine, and this schedule would have length $nq = \Delta/\rho$. Since the schedule produced by algorithm A is no longer than ρ times optimal, it must produce a schedule of length no longer than $\Delta \leq C^*_{\max}$.

In addition, future iterations of Step 3 must produce schedules of length no longer than $2\rho C^*_{\max}$. Suppose the algorithm performs an iteration of Step 3 in which the jobs are assigned size q. Since the algorithm did not finish processing all jobs in the previous iteration, we know that the instance being scheduled must have some jobs j such that $p_j > q/2$. An optimal schedule for this subset of jobs must take time no greater than $C^*_{\max}$. Ensuring that each of these jobs gets processed for q units can increase the optimal schedule for these jobs by no more than a factor of 2. Finally, the algorithm A will find a schedule for these jobs that is no more than ρ times as long as the optimal schedule, so that the schedule can be no longer than $2\rho C^*_{\max}$.

To derive our $O((\log n)C^*_{\max})$ bound on the length of the schedule, we show that we essentially need to consider only the last $\log(2\rho n)$ iterations of Step 3. Suppose there are f iterations of Step 3. Simple upper and lower bounds on schedule length show that length of the schedule produced in iteration $f - i$ (when the job size is q) is at least 2^ℓ times as long as the length of the schedule produced in iteration $f - i - \ell \log(2\rho n)$ (when the job size is $q/(2\rho n)^\ell$). It follows that the total length of all prior iterations is no more than the total length of the last $\log(2\rho n)$ iterations. Since each of these iterations has length no longer than $2\rho C^*_{\max}$, the overall length of the schedule is at most $\Delta + 2(\log(2\rho n))(2\rho C^*_{\max}) = (4\rho \log n + 4\rho \log 2\rho + 1)C^*_{\max}$, which is $O((\log n)C^*_{\max})$. ■

By the results in [19] and [20] we have the following two corollaries.

Corollary 3.2 There is a polynomial-time on-line algorithm for scheduling preemptive unrelated machines that has competitive ratio $4(\log n) + 5$.

Corollary 3.3 There is a polynomial-time on-line algorithm for scheduling non-preemptive unrelated machines that has competitive ratio $8(\log n) + 17$.

We can do somewhat better with uniformly related machines. By applying a list scheduling algorithm until there are at most m unfinished jobs, we obtain the following lemma.

Lemma 3.4 The number of jobs in any uniformly related machine problem instance can be reduced on-line from n to m, while increasing the competitive ratio by 1.

The preceding theorem and lemma, combined with the results in [14] and [13], yield the following corollaries.

Corollary 3.5 There is a polynomial-time on-line algorithm for scheduling preemptive uniformly related machines that has competitive ratio $4(\log m) + 6$.

Corollary 3.6 There is a polynomial-time on-line algorithm for scheduling non-preemptive uniformly related machines that has competitive ratio $(4+\epsilon)(\log m)+(4+\epsilon)\log(2+\epsilon)+2$.

3.2 An Improved Algorithm for Non-Preemptive Uniformly Related Machines

In the case of non-preemptive uniformly related machines, we can obtain an even better bound when the ratio between the speeds of the fastest and slowest machines is less than m. Let $R = s_1/s_m$. We will give an algorithm with competitive ratio $O(\min(\log R, \log m))$.

First we give a new (off-line) 2-relaxed decision procedure for uniformly related machines that will be the basis of our on-line algorithm. The notion of a *ρ-relaxed decision procedure* was used by Hochbaum and Shmoys [12]: given a deadline d, such a procedure either produces a schedule of length ρd or verifies that there exists no schedule of length d. By using binary search, a ρ-relaxed decision procedure can be converted into a ρ-approximation algorithm.

The 2-relaxed decision procedure is as follows. Each machine has an associated queue. Each job is placed into the queue of the slowest machine m_k such that $p_j \leq s_k d$; that is, the slowest machine that can complete the job within the given deadline. If for some job there is no such machine it is clear that there does not exist a schedule of length d. To construct a schedule, whenever a machine is idle, it starts processing a new, unprocessed job from its queue. If a machine's queue is empty, it takes a job to process from the queue of the fastest machine that is slower than it and that has a nonempty queue. If all such queues are empty, then the machine remains idle. If the schedule constructed has $C_{\max} > 2d$, output **no**. Otherwise we have produced a schedule of length at most $2d$.

We must prove that when the procedure outputs **no** there is no schedule of length d. Consider a job j that was not finished by time $2d$. Since jobs are only processed by machines on which they take less than d units of time, this job must have started after time d; thus it was on the queue of some machine m_k until time d. This implies that until time d machines $m_1, \ldots m_k$ were all busy processing jobs that could not have completed on machines $m_{k+1}, \ldots m_m$. Therefore in a schedule of length d there is no possibility to process all of these jobs and job j. Thus there is no schedule of length d.

We next use this relaxed decision procedure to prove the following theorem.

Theorem 3.7 Let $\mathcal{I}$ be an instance of the scheduling problem for non-preemptive uniformly related machines. Then there is a polynomial-time on-line scheduling algorithm which produces a schedule no longer than $16(\log R)C^*_{\max}(\mathcal{I})$.

Proof sketch: We round machine speeds down to the nearest power of two: when a machine finishes processing a job it pretends to keep processing it long enough so that it seems to have been processed at the lesser speed. When we interpret the schedules produced by an on-line algorithm for these rounded instances as schedules for the actual problem, the competitive ratio can be increased by at most a factor of two. Since the s_i are all powers of two, and all the s_i are within a factor of R of s_1, it immediately follows that there are at most $\log R$ different machine speeds. Let $M_1 = \{m_i | s_i = s_1\}$, $M_2 = \{m_i | s_i = s_1/2\}, \ldots, M_{\log R} = \{m_i | s_i = s_1/2^{\log R}\}$. We would like to apply the off-line decision procedure to this instance. Note that instead of queueing jobs on machines $m_1, \ldots, m_m$, we can instead queue jobs on sets of machines $M_1, \ldots, M_{\log R}$.

As in the general technique, we will repeatedly run the relaxed decision procedure to either schedule a job, or else update the estimate of its size. In this way, we convert the decision procedure above into an on-line decision procedure. We initially assign all jobs to the $M_{\log R}$ queue, and run the off-line decision procedure, though starting no new jobs after time d. If all jobs finish processing by time $2d$, we are done. If either of the following two conditions hold, then the jobs can't be processed in time d: first, if some machine in M_1 is still processing a job at time $2d$, and second, if any machine has a job j in its queue at time d. Otherwise, all remaining jobs are being processed at time $2d$ on machines in M_i, $i > 1$. We cancel these jobs and put them on the queue of M_{i-1}. We can perform this last step at most $\log R$ times; by this time we must either have completed all jobs or have discovered that the jobs can't be processed in time d.

The length of the schedule or partial schedule produced by the on-line decision procedure is at most $2d \log R$. We omit the proof of the correctness of the procedure due to its similarity to the off-line proof. We also omit details on how to convert this on-line relaxed decision procedure into an on-line algorithm.

The general idea is that we initially set the deadline d to some lower bound $\Delta \leq C^*_{\max}$, and double d each time the decision procedure returns the answer "no". When the decision procedure finally schedules all jobs, the length of the schedule due to previous calls to the decision procedure will be relatively insignificant compared to the length due to the last call. Hence the total length will be $O((\log R)C^*_{\max})$. ■

We now note that if the algorithm uses only the k fastest machines, where k is defined as the smallest k such that such that $\sum_{i=1}^{k} s_i \geq \frac{1}{2}\sum_{i=1}^{m} s_i$, then $s_1 \leq ms_k$. In time $2C^*_{\max}$ we can process on-line all but k of the jobs by processing jobs arbitrarily on machines $m_1, \ldots, m_k$ until the first moment in time at which at most k jobs have not yet been completely processed. The amount of time it takes until this point is bounded above by $(\sum_{j=1}^{n} p_j)/(\frac{1}{2}\sum_{i=1}^{m} s_i) \leq 2C^*_{\max}$, since none of the k machines is idle. We will only need machines $m_1, \ldots, m_k$ to process these last k jobs. Thus if we then produce a schedule of length l for the last k jobs on these machines, the entire schedule will be of length $2C^*_{\max} + l$, and the machine speeds will satisfy $R \leq m$.

Corollary 3.8 There is an on-line algorithm for scheduling non-preemptive uniformly related machines that has competitive ratio $\min(16 \log R, 16 \log m + 2)$.

4 Lower Bounds

4.1 Identical Machines

As with other on-line algorithms, on-line scheduling can be viewed as a game against an adversary who is allowed to determine the information that is revealed incrementally to the algorithm. Therefore, our lower bound arguments will often be phrased in terms of a strategy for an adversary, who attempts to reveal information in such a way as to force the competitive ratio to be as large as possible. In this paper, we will consider two possible types of adversaries: the *adaptive adversary*, who knows in advance both the scheduling algorithm and the result of any coin tosses of the algorithm, and an *oblivious adversary*, who knows only the algorithm but not the coin tosses [23, 1]. For deterministic algorithms this distinction is clearly irrelevant.

We begin with a lower bound on the competitive ratio of any on-line algorithm for scheduling identical machines.

Theorem 4.1 The competitive ratio of any deterministic on-line algorithm for scheduling identical machines, with no preemption allowed, is at least $(2 - \frac{1}{m})$.

Proof: For any m, let $n = m(m-1)+1$. Each of the first $m(m-1)$ jobs is of size 1, while the last job is of size m; that is, $p_1 = \cdots = p_{n-1} = 1, p_n = m$. This instance is due to Graham [8]. The optimal schedule is of length m, and consists of scheduling the last job on a machine by itself, and scheduling m of the single unit jobs on each of the remaining $m-1$ machines. The length of a schedule for this instance is determined by the starting time of the job of size m; therefore the adversary wishes to make it start as late as possible. Each of the first $n-1$ jobs that the scheduler allows to run for at least one unit of time will be fixed by the adversary to be jobs of size 1. Given this strategy of the adversary, it is not difficult to see that by time i, $1 \leq i \leq m-1$, at most im jobs are either completely processed or currently being processed. Hence by time $m-1$ there must be one job that has not been completely processed and is not currently being processed. The adversary sets this job to be of size m. If this job starts at time $m-1$ the fastest the schedule can complete is by time $2m-1$, which is $(2-\frac{1}{m})$ times as long as the optimal schedule. ■

In contrast to the nonpreemptive model an optimal preemptive schedule can be found off-line in polynomial time [22]. Interestingly enough, an argument similar to the previous proof shows that the on-line worst-case characterization of both models is the same.

Theorem 4.2 The competitive ratio of any deterministic on-line algorithm for the preemptive scheduling of identical machines is at least $(2 - \frac{1}{m})$.

The essence of these deterministic lower bounds is that there is one large job whose starting time determines the length of the schedule, and the adversary can force the scheduler to start that job late in the schedule. When we move to randomized algorithms it is true that an *adaptive* adversary can force the randomized scheduler to do as poorly as the deterministic scheduler. One might imagine, however, that a randomized algorithm A working against an *oblivious* adversary might, with significant probability, select and schedule the large jobs earlier, thus doing better. (It is known, for example, that an algorithm that schedules jobs in nonincreasing size order produces a schedule of length no longer than $\frac{4}{3}C^*_{\max}$ [9].) We will prove, however, that randomness is of little help to a non-preemptive on-line scheduler for this problem.

Theorem 4.3 The competitive ratio of any randomized on-line algorithm for the non-preemptive scheduling of identical machines, working against an oblivious adversary, is at least $(2 - O(\frac{1}{\sqrt{m}}))$.

Our strategy to prove this theorem is as follows. We will first define the notion of a *reasonable* randomized algorithm for scheduling identical machines. We will then show that for any c-competitive unreasonable algorithm, there exists a reasonable algorithm that has a competitive ratio no greater than c and that always chooses the next job to schedule uniformly. Finally, we will provide an instance for which the competitive ratio of such a strategy has worst case expected value $(2 - O(\frac{1}{\sqrt{m}}))C^*_{\max}$.

Definition 4.4 A *reasonable* randomized algorithm for scheduling identical machines is an algorithm that does not restart any job and does not leave any machine idle so long as there is some job that has not yet been started.

Lemma 4.5 For any unreasonable algorithm A there is a reasonable algorithm A' whose worst-case expected performance is at least as good as that of A.

Lemma 4.6 A reasonable randomized algorithm A is equivalent to an algorithm that, whenever a machine becomes idle, picks one of the unstarted jobs with a certain probability distribution which may depend on the schedule constructed up to that point.

Proof: Since a reasonable randomized algorithm constructs a schedule with no restarts and no idle time, it must be the case that it schedules some unstarted job whenever a machine becomes idle. The probability distribution for its next choice cannot depend on information that the algorithm does not have at that point; thus, it can depend only on the schedule constructed until that particular choice of a job. ■

We will now argue that the adversary can always force the scheduler to do as poorly as it would have done had it always made its choices according to the uniform distribution.

Lemma 4.7 The competitive ratio of a reasonable randomized algorithm A can be no less than that of the reasonable algorithm U that always picks the next job to process uniformly from among the remaining jobs.

Proof: We note that the adversary's strategy can be described as choosing the sizes of the jobs and then choosing some permutation of the jobs. If the adversary chooses the permutation randomly and uniformly, then the probability of the algorithm A selecting any particular job is uniform over all remaining jobs, no matter what probability distribution A uses. Let $\mathcal{E}$ be the expected performance of algorithm A against this adversary, where the expectation is taken over the random choices of both A and the adversary. Note that the adversary can always choose some permutation of jobs such that the expected performance of A, taken over just the choices of A, is no better than $\mathcal{E}$. Since the expected performance of the algorithm U that chooses uniformly is $\mathcal{E}$ no matter which permutation is used, algorithm A can have competitive ratio no better than algorithm U. ■

We complete the proof of theorem 4.3 by showing that scheduling by choosing the next job uniformly can do quite poorly.

Lemma 4.8 There is a problem instance for scheduling identical machines on which a uniform choice of the next job to process produces a schedule with expected length $(2 - O(\frac{1}{\sqrt{m}}))C^*_{\max}$.

Proof: We will consider the problem instance with k jobs of size m ("big" jobs) and $m(m-k)$ jobs of size 1 ("small" jobs). The optimum length schedule for this instance is of length m. The expected length of the schedule is then $m + E_s$, where E_s is the expected start time of the last big job in the schedule. E_s will be at least $\frac{k}{k+1}(m-k)$; maximizing this expression over k yields $k = O(\sqrt{m})$ and thus $E_s = (2 - \frac{2}{\sqrt{m}} + o(\frac{1}{\sqrt{m}}))C^*_{\max}$, which implies the stated result. ■

4.2 Uniformly Related Machines

In the case of uniformly related machines the situation becomes significantly more difficult for the scheduler. We will show that the adversary can force any deterministic scheduler to construct a schedule of length $\Omega(\log m)$ times the length of the optimal schedule, whether or not the scheduler is allowed to preempt jobs.

Theorem 4.9 The competitive ratio of any on-line scheduling algorithm for uniformly related machines, whether or not preemption is allowed, is $\Omega(\log m)$.

Our proof will be in terms of a preemptive scheduler, but it is not hard to see that a similar argument will work when no preemption is allowed.

To prove this theorem we use a family of instances $\mathcal{I}_k$ for uniformly related machines given by Cho and Sahni [3] in a somewhat different context. Let $k = (\log_2(3m-1)+1)/2$. We restrict ourselves to values of m such that k is integral. The instance $\mathcal{I}_k$ has k sets of machines G_i and k sets of jobs T_i, $1 \le i \le k$. Each machine in G_i has a speed of 2^i and each job in T_i has size 2^i. Finally, $|G_i| = |T_i| = 2^{2k-2i-1}$ for $1 \le i < k$, and $|G_k| = |T_k| = 1$. It is easy to see that

$C^*_{\max} = 1$, since each job of size p_j can be scheduled by itself on a machine of speed p_j.

Let X_i be the time when, in a schedule for $\mathcal{I}_k$, the last job in T_i finishes.

Lemma 4.10 The adversary can always force the scheduler to construct a schedule for $\mathcal{I}_k$ in which $X_1 \leq X_2 \leq \ldots \leq X_k$.

Proof: Assume that the adversary is competing against a scheduler who somehow knows the job sizes in advance, but doesn't know which size belongs to which job. Certainly if the adversary can force this type of scheduler to do badly, the adversary can force a scheduler with no knowledge of job sizes to do badly. We introduce the idea of the adversary *committing* to a set of jobs. At time t, let $J(t)$ be the set of jobs that have not yet completed. The scheduler has a corresponding set $L(t)$ of the sizes of the jobs which have not yet completed. The adversary is not committed to any job in $J(t)$ if, given the amount of time that the jobs in $J(t)$ have been running, the scheduler cannot infer any information about which job in $J(t)$ is associated with which size in $L(t)$. More formally, the adversary is *not committed to any job in* $J(t)$ if, at time t, any bijective mapping from $J(t)$ to $L(t)$ is valid given the schedule thus far. Let $R(i,t)$ be the total amount of processing that has been done on the job that is running on machine i at time t. If the adversary is not committed to any job in $J(t)$ at time t then

$$R(i,t) \leq \min_{j \in J(t)} p_j,\ 1 \leq i \leq m.$$

The adversary's strategy is to avoid being committed to any job in $J(t)$. The adversary can do this, if, at any point in time t' such that $R(i,t') = \min_{j \in J(t')} p_j$ for some i, the adversary allows the smallest job in $J(t')$ to complete on machine i. If the equality holds true for more than one machine i or more than one job j, then the smallest indexed job j completes on the smallest indexed machine i and so forth. The adversary continues to complete jobs until the inequality $R(i,t') < \min_{j \in J(t')} p_j$ holds again. It is clear that this strategy yields a schedule satisfying the condition of the lemma. ■

Lemma 4.11 The adversary can always force the scheduler to produce a schedule for $\mathcal{I}_k$ in which $X_i - X_{i-1} \geq \frac{1}{4}$, $1 \leq i \leq k$, $X_0 = 0$.

Proof: The adversary uses the same strategy as in the previous proof. Consider the status of the jobs in T_{i+1} at time X_i. None of them have been completed; in fact, no more than 2^i of the 2^{i+1} units of each job have been processed. This is because until all the jobs in T_i were completed (at time X_i), any job that had 2^i units processed was designated by the adversary to be in T_i and was thus finished. Therefore there are $|T_{i+1}|$ jobs, each with remaining work of at least 2^i units each. How quickly might these all complete?

Since there are more than $|T_{i+1}|$ machines in the sets $G_{i+1}, G_{i+2}, \ldots G_k$, in an optimal schedule there is no need to run one of the jobs in T_{i+1} on a machine in G_i or slower. At best, processing all the jobs from T_{i+1} on all the machines in G_{i+1} and faster must take time at least the sum of the remaining processing requirements of the jobs in T_{i+1} over the sum of the processing speeds of processors in G_{i+1} or faster. A straightforward calculation shows this quotient is at least $\frac{1}{4}$. ■

The $\Omega(\log m)$ lower bound follows directly from these lemmas. Note that the problem instance utilized in the proof has a ratio of machine speeds $R = \Omega(m)$. We can prove a more general lower bound on the competitive ratio when the ratio of speeds is smaller than m.

Theorem 4.12 The competitive ratio of any deterministic on-line algorithm for scheduling uniformly related machines with $R = s_1/s_m < m$ is $\Omega(\log R)$, whether or not preemption is allowed.

Acknowledgements

We are grateful to Howard Karloff for suggesting on-line scheduling as a fruitful area of research. We also thank Nabil Kahale for a useful discussion on probability, and Cliff Stein and Lisa Hellerstein for comments on a draft of this paper.

References

[1] S. Ben-David, A. Borodin, R.M. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 379–386, May 1990.

[2] B. Chandra, H.J. Karloff, and S. Vishwanathan. Private communication, 1991.

[3] Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9(1):91–103, February 1980.

[4] E. Davis and J.M. Jaffe. Algorithms for scheduling tasks on unrelated processors. *Journal of the ACM*, 28:712–736, 1981.

[5] A. Feldmann, J. Sgall, and S. Teng. Dynamic scheduling on parallel machines. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, October 1991.

[6] A. Fiat, Y. Rabani, and Y. Ravid. Competitive k-server algorithms. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, 1990.

[7] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

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

[9] R.L. Graham. Bounds on multiprocessing anomalies. *SIAM Journal of Applied Mathematics*, 17:263–269, 1969.

[10] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[11] L. Hall and D. B. Shmoys. Approximation schemes for constrained scheduling problems. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 134–141. IEEE, October 1989.

[12] D.S. Hochbaum and D.B. Shmoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of the ACM*, 34:144–162, 1987.

[13] D.S. Hochbaum and D.B. Shmoys. A polynomial approximation scheme for machine scheduling on uniform processors: using the dual approximation approach. *SIAM Journal on Computing*, 17:539–551, 1988.

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

[15] S. Irani. Coloring inductive graphs on-line. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 470–479, 1990.

[16] J.M. Jaffe. Efficient scheduling of tasks without full use of processor resources. *Theoretical Computer Science*, 12:1–17, 1980.

[17] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[18] R. Karp, U.V. Vazirani, and V.V. Vazirani. Online algorithms for bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 352–358, 1990.

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

[20] J.K. Lenstra, D.B. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[21] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for on-line problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 322–333, May 1988.

[22] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6:1–12, 1959.

[23] P. Raghavan and M. Snir. Memory vs. randomization in on-line algorithms. In *Proceedings of the 1989 ICALP Conference*, 1989.

[24] S. Sahni and Y. Cho. Nearly on line scheduling of a uniform processor system with release times. *SIAM Journal on Computing*, 8:275–285, 1979.

[25] D. B. Shmoys, C. Stein, and J. Wein. Improved approximation algorithms for shop scheduling problems. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pages 148–157, January 1991.

[26] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[27] S. Vishwanathan. Randomized online coloring of graphs. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science*, pages 464–469, 1990.