LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TM-437

ON-LINE SCHEDULING OF PARALLEL MACHINES

Joel Wein
David P. Williamson

November 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

On-line Scheduling of Parallel Machines

(Extended Abstract)

Joel Wein*
Department of Mathematics and
Laboratory for Computer Science
MIT
Cambridge, MA 02139

David P. Williamson†
Laboratory for Computer Science
MIT
Cambridge, MA 02139

November 27, 1990

Abstract

We study the problem of scheduling jobs on parallel machines in an *on-line* fashion, where the processing requirement of a job is not known until the job is completed. Despite this lack of knowledge of the future, we wish to schedule so as to minimize the completion time of the entire set of jobs. In general, the performance of an on-line algorithm is measured by its *competitive ratio:* the worst case ratio of its performance to that of an optimal algorithm with total prior knowledge. We study two fundamental models for this problem, that of *identical machines*, where all the machines run at the same speed, and *uniformly related machines*, where the machines run at different speeds. Our results include:

- Matching upper and lower bounds on the competitive ratio for the case of identical machines.
- Upper and lower bounds that differ by a constant factor for uniformly related machines.
- A lower bound for randomized algorithms for identical machines that nearly matches the deterministic upper bound.
- Several upper and lower bounds for variations on these models.

Keywords: On-line algorithms, scheduling algorithms, parallel machines, combinatorial optimization.

*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.

1 Introduction

Recently there has been a great deal of interest in the field of on-line algorithms. An on-line algorithm produces a reasonable solution to a problem despite the fact that the entire specification of the problem is only revealed incrementally. On-line algorithms have been developed for a number of classes of problems, from classic problems in combinatorial optimization [1], [10], [14], [21], to various problems in data and memory management [13], [20], to the k-server problem [6], [17]. A very natural and largely untouched area in which to study on-line algorithms is that of *scheduling*. In many real-world scheduling situations the scheduler does not begin with full knowledge of the amount or type of processing required [12]. Despite this lack of knowledge of the future the scheduler must utilize a strategy that will do as well as possible.

In this paper we study on-line algorithms for *scheduling parallel machines*. We define a realistic and theoretically interesting model of on-line scheduling, and bound the performance of on-line algorithms for several fundamental versions of this problem. This class of problems is fundamental in combinatorial scheduling theory [15], and has inspired a variety of techniques that are of general interest in combinatorial optimization [8], [9], [16]. Further, these problems have important applications to the design of parallel computing systems [4].

The basic model for scheduling parallel machines is as follows. We are given n jobs and m machines. Each job j has a processing requirement of p_j units. Job j can only be processed by one machine at a time, and a job must be processed in an uninterrupted fashion on one of the machines. We specify that the machines either are *identical*, so that processing p units takes time p on any machine, or the machines are *uniformly related*, so that each machine i runs at a certain speed s_i, and processing p units takes time p/s_i on machine i. A solution to an instance of the problem is a *schedule* that for each job j specifies a time interval on one machine during which all p_j units of job j are processed, while ensuring that no machine processes more than one job at the same time. If C_j is the point in time at which job j has finished processing, then the *makespan* or *length* of the schedule is $C_{\max} = \max_j C_j$.

It is important to define a model of on-line scheduling that realistically captures the scheduler's lack of knowledge of the data and yet is not so strong as to leave no room for an algorithm to perform adequately. Traditional off-line models are unrealistic in that all the p_j are known in advance. It would also be unrealistic, however, to force the scheduler to irrevocably assign a job to a machine before learning something about the size of the job. Therefore, we model on-line scheduling by assuming that none of the p_j are known in advance, but that the scheduler can *restart* a job, i.e. cancel it and start it from scratch on another machine. For simplicity's sake, we assume all the jobs are available at the start time. It is possible to extend our results to the more general setting where not all jobs are available at the start time; we discuss this extension to the model in section 7.

On-line algorithms are traditionally evaluated in terms of their *competitive ratio*, the worst-case ratio of their performance to that of an optimal off-line algorithm. For these scheduling problems, the "performance" of an algorithm on a problem instance is the makespan of the schedule constructed by that algorithm. Thus the performance of an on-line scheduling algorithm is characterized by the worst-case ratio of the length of the schedule produced to the length of the shortest possible schedule.

In this paper we give essentially exact characterizations of the power of on-line deterministic schedulers in two fundamental models. Specifically, we prove that a deterministic on-line scheduler for identical machines can not achieve a competitive ratio less than $2 - \frac{1}{m}$; by an algorithm of Graham this is a tight bound [7]. For uniformly related machines we give a scheduling algorithm

with competitive ratio $O(\log m)$ and show that no algorithm can have ratio better than $\Omega(\log m)$.

We then consider how randomness might help the on-line scheduler. We argue that a randomized scheduler working against an adaptive adversary can do no better than a deterministic scheduler. Randomization would seem to be much more useful when the adversary is oblivious, but we show that for identical machines no randomized algorithm can achieve competitive ratio better than $(2 - \Omega(1/\sqrt{m}))$. This result is surprisingly strong: since the bound for deterministic algorithms is $(2 - \frac{1}{m})$, the result shows that randomness does not much improve any algorithm's competitive ratio. 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 [14], [21].

Relatively little work has been done for this model of on-line scheduling. In addition to the algorithms for identical machines given by Graham [7], the only other work known to the authors is that of Jaffe [11] and Davis and Jaffe [3]. Davis and Jaffe show that in a restricted model without restarts, an on-line algorithm for scheduling uniformly related machines cannot have competitive ratio better than $\Omega(\sqrt{m})$. Jaffe gives an algorithm for this case with competitive ratio $O(\sqrt{m})$.

Our paper is organized as follows. In section 2 we define our models precisely and establish some terminology. In section 3, we prove matching lower and upper bounds for identical machines. Section 4 contains the proof of our lower bound for uniformly related machines, while section 5 demonstrates an on-line algorithm for this case with competitive ratio within a constant factor of the lower bound. The bounds for randomized algorithms are established in section 6. We conclude in section 7 by discussing several possible variants of the basic model and giving upper and lower bounds for some of these variants.

2 Preliminaries

In this section we give precise definitions of a *competitive ratio* and an *adversary* before going on to prove our bounds.

For each instance I of each type of on-line scheduling problem given in the introduction there is some *optimal* schedule with a minimum possible makespan. We will denote the length of this optimal schedule by $C^*_{\max}(I)$, or sometimes by $C^*_{\max}$ when there is no danger of confusion. There are exponential-time off-line algorithms for both problems (the decision version for both identical and uniformly related machines is NP-complete); therefore we will compare the makespan of a schedule constructed by an on-line algorithm for instance I to $C^*_{\max}(I)$.

Let $C^A_{\max}(I)$ be the makespan of a deterministic on-line algorithm A on instance I. Algorithm A is said to have *competitive ratio* c (or is said to be *c-competitive*) if $C^A_{\max}(I) \leq c \cdot C^*_{\max}(I) + O(1)$ for all problem instances I. If A is a randomized algorithm, then A is said have competitive ratio c (or is said to be c-competitive) if $E[C^A_{\max}(I)] \leq c \cdot C^*_{\max}(I) + O(1)$ for all instances I, where the expectation is taken over all the random choices of the algorithm A.

As with other on-line algorithms, on-line scheduling algorithms can be viewed as a game against an adversary who is allowed to determine the information that is revealed incrementally to the algorithm. The adversary attempts to reveal information in such a way as to force the competitive ratio to be as large as possible. In the case of on-line scheduling, the adversary is allowed to determine the size p_j of each job j. In this paper, we will consider two possible types of adversaries: an *adaptive adversary*, which is allowed to determine the size of jobs p_j as the scheduler is scheduling; and an *oblivious adversary*, which must fix the size p_j of each job before

the scheduler begins scheduling.

In sections 3, 4, and 5, we will consider deterministic algorithms playing against adaptive adversaries. It is not hard to see that these bounds must hold for deterministic algorithms playing against oblivious adversaries. An oblivious adversary can first run by itself an adaptive adversary against the deterministic algorithm, and then present the algorithm with the job sizes chosen by the adaptive adversary. The competitive ratio of the deterministic algorithm playing against this oblivious adversary must be the same as that obtained by the algorithm against the adaptive adversary. There is, however, a distinction between the two adversaries for randomized algorithms. The bounds given in the next three sections do not depend on the determinism of the scheduling algorithm, and so will hold for randomized algorithms playing against adaptive adversaries. The final case of randomized algorithms playing against oblivious adversaries is discussed in section 6.

3 Identical Machines

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

Theorem 3.1 The competitive ratio of any deterministic on-line algorithm for scheduling identical machines 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 [7]. 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. ■

Now we turn to on-line algorithms for this problem. Graham [7] showed that *list scheduling* for identical machines always comes within a factor $(2 - \frac{1}{m})$ of the optimal length schedule. 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. In fact, Graham shows that list scheduling on identical machines always constructs a schedule no longer than $\frac{1}{m}\sum_{j=1}^{n} p_j + (1 - \frac{1}{m})p_{\max}$, where $p_{\max} = \max_j p_j$. Since both $\frac{1}{m}\sum_{j=1}^{n} p_j$ and $p_{\max}$ are lower bounds on the length of any schedule, we have the following theorem.

Theorem 3.2 [Graham] There is an on-line algorithm for scheduling identical machines that achieves competitive ratio $(2 - \frac{1}{m})$.

4 The Lower Bound for Uniformly Related Machines

In the case of uniformly related machines the situation becomes a good deal 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. Intuitively, the adversary's best strategy would be to tie up the faster machines with small jobs, forcing the large jobs either to run on slower machines or to begin late in the schedule. We first prove that the adversary can follow a strategy very similar to this. Then we show that this strategy implies a $\Omega(\log m)$ lower bound.

Before we state the theorem, we introduce some notation. Given a particular schedule, let $t_s(j)$ and $t_f(j)$ be the starting and finishing times, respectively, of job j in that schedule. We let $m(j)$ be the machine that job j completes on, and assume that jobs are sorted so that $p_1 \leq p_2 \leq \ldots \leq p_n$. We denote the speeds of the machines by $s_1, \cdots, s_m$ with $s_1 \geq s_2 \ldots \geq s_m$. We assume that the speeds of the machines are known to the scheduler.

The following theorem states that the adversary can always force an on-line scheduler to produce a schedule of a certain form, which loosely understood specifies that larger jobs finish later in the schedule and that a small job j_1 only runs on a slower machine than a large job j_2 if thereby it finishes faster than by waiting to start on j_2's faster machine.

Theorem 4.1 Let L be the length of the shortest schedule that satisfies the following three conditions:

1. $t_f(1) \leq t_f(2) \leq \ldots \leq t_f(n)$
2. For $k > 0$, if $m(j+k) < m(j)$, then $t_s(j+k) + p_j/s_{m(j+k)} > t_f(j)$.
3. For $k > 0$, if $m(j+k) > m(j)$, then $t_s(j+k) + p_j/s_{m(j+k)} \geq t_f(j)$.

Then the adversary can always force the scheduler to construct a schedule of length at least L.

Proof: To prove this theorem, it is enough to show that the adversary always has a strategy that generates a schedule meeting these three conditions. Thus the best the scheduler can do is to find the optimal such schedule.

We introduce the idea of the adversary *committing* to a set of jobs. 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. 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, at time t, any bijective mapping from $J(t)$ to $L(t)$ is valid given the schedule thus far. In other words, 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 $L(t)$. Let $T(i,t)$ be the length of time machine i has been running a job at time t. If we forget for a moment the possibility that jobs in $J(t)$ may have been run before and cancelled (yielding some information about their size), then the adversary is not committed to any job in $J(t)$ at time t if

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

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 $T(i,t')s_i = \min_{j \in J(t')} p_j$ for some i, the adversary allows the smallest job in $J(t')$ to complete on machine i. This resets $T(i',t')$ to 0. 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 $T(i,t')s_i < \min_{j \in J(t')} p_j$ holds again.

This strategy immediately implies condition 1, since the smallest job will finish first, the next smallest second, etc. The strategy also takes care of our proviso above about past running times. If the scheduler runs a job j on machine i and cancels it at time t after it has processed for k units of time, the scheduler has learned that $p_j > k \cdot s_i$. This is true for any job in $J(t)$, however, since otherwise the adversary would have made the smallest job in $J(t)$ finish at or before time t on machine i. Therefore, cancelling a job does not force the adversary to make any new commitments.

Now to show that this strategy implies the second and third conditions. We will prove this by contradiction. Suppose that for some $j,k > 0$ $m(j+k) < m(j)$, but $t_s(j+k) + p_j/s_{m(j+k)} \leq t_f(j)$. Then $p_j \leq [t_f(j) - t_s(j+k)]s_{m(j+k)}$. Notice that it must be the case that $T(m(j+k), t_f(j))s_{m(j+k)} < p_j$, or else, since $m(j+k) < m(j)$, job j would have completed on machine $m(j+k)$ instead. But $T(m(j+k), t_f(j)) = t_f(j) - t_s(j+k)$, so that $[t_f(j) - t_s(j+k)]s_{m(j+k)} < p_j$, which contradicts the equation above. The third condition is proven similarly. ■

We use the theorem above to derive the $\Omega(\log m)$ lower bound. To do this, we use a family of instances for uniformly related machines given by Cho and Sahni [2] 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 has k sets of machines G_i and k sets of jobs T_i, $1 \leq i \leq k$. 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 $1 \leq i < k$, and $|G_k| = |T_k| = 1$. It is easy to see that $C^*_{\max} = 1$.

We would like to show that any schedule for this instance that obeys the conditions of Theorem 4.1 will do poorly. One schedule that does obey the conditions of Theorem 4.1 is the following: schedule one job from T_i to start on each machine from G_i at time $(i-1)(\frac{1}{2}+\delta)$, for any $\delta > 0$. Then the entire schedule will complete at time $(k-1)(\frac{1}{2}+\delta)+1 = \frac{k+1}{2} + (k-1)\delta$ (See Figure 1 in the appendix). We will show in a series of lemmas that no other permissible schedule for this instance can finish faster than this schedule: essentially, scheduling every set of jobs T_i will take at least one unit of time, and we will not be able to finish T_i any faster than one-half a unit of time later than the finishing time of T_{i-1}.

We will let j_i^{min} and j_i^{max} stand for the highest and lowest indexed jobs in T_i respectively.

Lemma 4.2 In any schedule obeying the conditions of Theorem 4.1, $t_f(j_i^{max}) - t_s(j_i^{min}) \geq 1$ for any i.

Proof: We will in fact show that $\max_{j \in T_i}[t_f(j_i^{max}) - t_s(j)] \geq 1$. This implies the lemma, since the second and third conditions of Theorem 4.1 together imply that $t_s(j_i^{min}) \leq t_s(j)$ for all jobs j in T_i. Suppose that some job $j \in T_i$ runs on a machine from G_i or slower. Then for this job j, $t_f(j) - t_s(j) \geq 2^i/2^i = 1$. Since by the conditions of Theorem 4.1, $t_f(j_i^{max}) \geq t_f(j)$, the lemma statement holds. Now suppose that no job $j \in T_i$ runs on a machine from G_i or slower. Note that at best, processing all the jobs from T_i on all the machines in G_{i+1} and faster must take time at least the sum of the processing requirements of T_i over the sum of the processing speeds of processors in G_{i+1} or faster. So the time taken is at least

$$\begin{aligned}
\frac{\sum_{j\in T_i} p_j}{\sum_{l\in\cup_{i+1}^{k} G_i} s_l} &= \frac{2^i|T_i|}{\sum_{l=i+1}^{k} s_l|G_l|} \\
&= \frac{2^{2k-i-1}}{\sum_{l=i+1}^{k-1} 2^l \cdot 2^{2k-2l-1} + 2^k} \\
&= \frac{2^{2k-i-1}}{2^k \sum_{r=0}^{k-i-2} 2^r + 2^k} \\
&= \frac{2^{2k-i-1}}{2^k(2^{k-i-1}-1) + 2^k} \\
&= \frac{2^{2k-i-1}}{2^{2k-i-1}} \\
&= 1.
\end{aligned}$$

Hence the lemma statement must hold in this case as well. ■

Lemma 4.3 In any schedule obeying the conditions of Theorem 4.1, $t_f(j_i^{max}) - t_f(j_{i-1}^{max}) \geq \frac{1}{2}$, for any i, $2 \leq i \leq k$.

Proof: Suppose all jobs from T_i run on machines from G_{i+1} or faster. As was shown in Lemma 4.2, the jobs must take at least one time unit to complete. Then there must exist some machine r on which the difference between the finishing time of the last T_i job run on r and the starting time of the first T_i job run on r is at least 1. Call the last job from T_i to run on machine r job $j_i^{r,last}$ and the first job from T_i to run on machine r job $j_i^{r,first}$. So

$$t_f(j_i^{r,last}) - t_s(j_i^{r,first}) \geq 1.$$

Since r is a machine from G_{i+1} or faster, $s_r \geq 2^{i+1}$. Hence

$$t_f(j_i^{r,first}) = t_s(j_i^{r,first}) + \frac{2^i}{s_r} \leq t_s(j_i^{r,first}) + \frac{1}{2}.$$

Because $t_f(j_{i-1}^{max}) \leq t_f(j_i^{r,first})$, it follows that

$$t_s(j_i^{r,first}) - t_f(j_{i-1}^{max}) \geq -\frac{1}{2}.$$

Adding this equation to equation (4) gives us

$$t_f(j_i^{r,last}) - t_f(j_{i-1}^{max}) \geq \frac{1}{2}.$$

Since $t_f(j_i^{r,last}) \leq t_f(j_i^{max})$, it follows that

$$t_f(j_i^{max}) - t_f(j_{i-1}^{max}) \geq \frac{1}{2}.$$

Now suppose that there is some job from T_i than runs on a machine from G_i or slower, call it machine q. We will call this job j_i. If job j_i runs on the same machine q as job j_{i-1}^{max}, then the lemma statement follows since $t_f(j_i^{max}) \geq t_f(j_i) \geq t_f(j_{i-1}^{max}) + \frac{2^{i-1}}{s_q} \geq t_f(j_{i-1}^{max}) + \frac{1}{2}$. If j_{i-1}^{max} runs on some machine other than q, then by the second and third conditions of Theorem 4.1, $t_s(j_i) + \frac{2^{i-1}}{s_q} \geq t_f(j_{i-1}^{max})$. Since $t_f(j_i^{max}) \geq t_f(j_i) = t_s(j_i) + \frac{2^i}{s_q}$, it follows that $t_f(j_i^{max}) \geq t_f(j_{i-1}^{max}) + \frac{2^i}{s_q} - \frac{2^{i-1}}{s_q}$. We know that $s_q \leq 2^i$, so the lemma statement follows. ■

Lemma 4.4 In any schedule obeying the conditions of Theorem 4.1, $t_f(j_k^{max}) \geq \frac{k+1}{2}$.

Proof: Add together the $k-1$ possible inequalities from Lemma 4.3. This yields the equation

$$t_f(j_k^{max}) - t_f(j_{k-1}^{max}) + t_f(j_{k-1}^{max}) - t_f(j_{k-2}^{max}) + \ldots - t_f(j_1^{max}) \geq \frac{k-1}{2}.$$

By collapsing the sum, we obtain

$$t_f(j_k^{max}) - t_f(j_1^{max}) \geq \frac{k-1}{2}.$$

From Lemma 4.2, we know that $t_f(j_1^{max}) - t_s(j_1^{min}) \geq 1$, so that $t_f(j_1^{max}) \geq 1$. Hence we have

$$t_f(j_k^{max}) \geq \frac{k+1}{2}.$$

■

The lower bound follows directly from these lemmas.

Theorem 4.5 The competitive ratio of any on-line algorithm for uniformly related machines is at least $\frac{(\log_2(3m-1)+1)}{4} + \frac{1}{2}$.

5 The Upper Bound for Uniformly Related Machines

In this section we will present an $O(\log m)$-competitive on-line algorithm for scheduling uniformly related machines.

5.1 A Simple (Off-Line) 2-Relaxed Decision Procedure

First we give a simple (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 first introduced by Hochbaum and Shmoys [8]: given a deadline d, such a procedure either produces a schedule of length ρd or verifies that there exists no schedule of length d.

The 2-relaxed decision procedure is as follows. Put each job into the queue of the slowest machine m_k such that $p_j \leq s_k d$. If for some job there is no such machine it is clear that there does not exist a schedule of length d. Machines now process the jobs in their queues. If a machine's queue is empty it takes jobs to process from the queue of the first machine that is slower than it and that has a nonempty queue. 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.

5.2 The On-line Algorithm

In this section we will first give an $O(\log m)$-competitive on-line algorithm for a restricted set of instances of the problem. We will then show through a series of lemmas that any instance can be reduced to one of these restricted instances while increasing the competitive ratio by at most a constant factor. Hence we will have an on-line algorithm for all instances with competitive ratio $O(\log m)$.

First we present the main algorithm.

Theorem 5.1 Let I be an instance of the scheduling problem for uniformly related machines. Suppose that for instance I all machine speeds are powers of 2 and that the fastest machine is no more than m times faster than the slowest machine. Then there is an on-line scheduling algorithm which produces a schedule no longer than $[8(\log m) + 1]C^*_{\max}(I)$.

Proof: Since the s_i are all powers of two, and all the s_i are within a factor of m of s_1, it immediately follows that there are at most $\log m$ different machine speeds. Let $M_1 = \{m_i | s_i = s_1\}$, $M_2 = \{m_i | s_i = s_1/2\}, \ldots, M_{\log m} = \{m_i | s_i = s_1/2^{\log m}\}$. We would like to apply the off-line decision procedure of section 5.1 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 $M_1, ..., M_{\log m}$.

The off-line decision procedure does not immediately lend itself to an on-line algorithm, since the criterion it uses to assign jobs to machine queues utilizes knowledge of the job sizes. To convert this to an on-line algorithm we will initially assign all the jobs to the $M_{\log m}$ queue; when we discover a job could not have completed on a machine in time d it is still possible that it might be able to complete in time d on a faster machine. Thus we move it to the queue of the next fastest set of machines and then try again. After at most $\log m$ iterations of putting jobs in the queues of faster machines we will either have discovered that the job can't be processed in time d or we will have processed it.

A formal description of an on-line relaxed decision procedure is as follows. This procedure will form the heart of our on-line algorithm. The procedure either outputs **no** if there is no schedule of length d or it produces a schedule of length $2d \log m$. Note that even if it answers **no** the procedure may have completely processed some of the jobs in that time.

Input A set of jobs and a deadline d.

Step 0 Put all jobs into the $M_{\log m}$ queue.

Step 1 Run the off-line 2-relaxed decision procedure, with the modification that no jobs are started after time d (that is, when a machine is idle it takes a job to process off of its queue, or, when its queue is empty, off of the first slower machine that has a non-empty queue; etc.)

Step 2

1. **If** all jobs finish processing by time $2d$ we are done.
2. **If** any machine in M_1 is still processing a job at time $2d$ then **there is no schedule of length** d. Output **no**; **return**
3. **If** any set of machines M_k has a job j in its queue at time d then there is no schedule of length d. Output **no**; **return**
4. **If** there are jobs that are being processed at time $2d$, on machines M_i, $i > 1$, stop these jobs and put them on the queue of M_{i-1}. Go to Step 1.

Analysis of procedure

- The length of the schedule or partial schedule produced is no longer than $2d \log m$.
- If the procedure outputs **no** then there is no schedule of length d. If condition 2 is true then an M_1 machine ran a job for more than d time; therefore this job clearly could not have been processed in time d on any of the machines, since no other machine runs at a faster speed. If condition 3 is true, then up until time d all machines in the sets $M_1, \ldots M_k$ must have been busy processing jobs that could not have been processed in time d on machines in $M_{k-1}, \ldots, M_{\log m}$. Therefore, machines in $M_1, \ldots, M_k$ could not have processed all of these jobs and job j as well by time d.

Notice that this procedure relies heavily on our ability to restart jobs.

Our on-line algorithm initially establishes a lower bound Δ on $C^*_{\max}$ by running an arbitrarily chosen job on the fastest processor. Let Δ be the time taken to complete this job; certainly $\Delta \leq C^*_{\max}$. Next, the on-line algorithm calls the procedure on the set of all jobs with $d = \Delta$. If the procedure returns **no**, then we will call it again with $d = 2\Delta$ and the set of jobs which were not completely processed in the first iteration. In general, if the ith iteration fails to produce a schedule, then we will call the procedure again for the $(i+1)$st time with $d = 2^i\Delta$ and all jobs that have not yet been completely processed. Observe that if the ith iteration fails to produce a schedule when called with $d = 2^{i-1}\Delta$, then it proves that $2^{i-1}\Delta < C^*_{\max}$. Suppose that we finally finish processing all jobs in iteration f. Then the total length of the schedule produced is

$$\Delta + (1 + 2 + \cdots + 2^{f-1})(2\Delta \log m) \leq 2^{f+1}\Delta \log m.$$

Since the procedure failed to produce a schedule on iteration $f-1$, we know that $2^{f-2}\Delta < C^*_{\max}$. Therefore the total length of the schedule produced is no greater than $(8(\log m)+1)C^*_{\max}$. ■

We now present a series of lemmas that show how to reduce any instance of the scheduling problem to an instance of the form required by the algorithm above, while increasing the competitive ratio by at most a constant factor.

Lemma 5.2 Any instance of the problem can be reduced on-line to one in which all machine speeds are powers of two, increasing the competitive ratio by a factor of at most 2.

Proof: We effectively round the speeds down to the nearest power of two. When a machine finishes processing a job it holds on to it long enough so that it seems to have been processed at the lesser speed. ■

Lemma 5.3 Any instance of the problem can be reduced on-line to one in which the speed of the fastest machine is no more than m times the speed of the slowest. This reduction increases the competitive ratio by 2.

Proof: Let k be such that $\sum_{i=1}^{k} s_i \geq \frac{1}{2}\sum_{i=1}^{m} s_i$, and $\sum_{i=1}^{k-1} s_i < \frac{1}{2}\sum_{i=1}^{m} s_i$. By this definition of k, $s_1 \leq m s_k$. 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 be as required. ■

6 Randomized Algorithms

We now consider the degree to which randomness can help an on-line scheduler. As is the case with many problems in on-line algorithms, there is a distinction between the two adversary models for a randomized scheduling algorithm. It is easy to see that none of our lower bounds in the previous sections depended on the scheduler being deterministic; they just required the adversary to be able to make decisions while the scheduler was scheduling. Therefore, all of our lower bounds remain valid for a randomized scheduler playing against an adaptive adversary. In the more realistic model of an oblivious adversary, however, the scheduler becomes slightly more competitive with the use of randomization. Nevertheless, the improvement is surprisingly small: we will prove a strong lower bound on the performance of any randomized algorithm for scheduling identical machines.

Theorem 6.1 Any randomized algorithm for scheduling identical machines has worst case expected value of at least $(2 - \Omega(1/\sqrt{m}))C^*_{\max}$.

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 $(2 - \Omega(1/\sqrt{m}))C^*_{\max}$.

Definition 6.2 A *reasonable* randomized algorithm for scheduling identical machines is an algorithm that does not restart any job and does not leave any machine idle as long as there is some job that has not yet been started.

Lemma 6.3 For any unreasonable algorithm $\mathcal{A}$ there is a reasonable algorithm $\mathcal{A}'$ whose worst-case expected performance is at least as good as that of $\mathcal{A}$.

Proof: First we argue that the introduction of idle time into a schedule cannot help the scheduler. Assume that job j is to be started at time t_2 on machine i which is idle from time t_1 to t_2. Now if job j is available at time t_1, it is clearly to the advantage of the scheduler to start job j on machine i at time t_1. If job j is not available at time t_1 then it is running on another machine i'. In this case there is no point in restarting job j on machine i; since the two machines are of identical speed we can switch the future schedules of the two machines without increasing the total length of the schedule.

Now restarting a job j after it has run for $t < p_j$ units of time is equivalent, in terms of the effect on the length of the schedule, to introducing t units of idle time, and thus does not help the scheduler either. ■

Lemma 6.4 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 6.5 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. Suppose the adversary chooses the permutation randomly and uniformly. Consider then the expected performance $\mathcal{E}$ of the randomized algorithm, taken over the random choices of both the adversary and the algorithm. At any particular point at which the algorithm chooses a job to schedule from among the remaining jobs, no matter what probability distribution the algorithm uses, the uniformly random choice of the adversary ensures that the probability of the algorithm selecting any particular job is uniform over all the remaining jobs. Thus the expected performance $\mathcal{E}$ for algorithm A is the same as that of the reasonable algorithm U. In reality, the adversary will pick some fixed permutation but if the expected value over all of the choices of both the adversary and the algorithm is $\mathcal{E}$, the adversary can always pick some permutation of the jobs such that the expected performance of the algorithm A, taken over just the choices of the algorithm, is no better than $\mathcal{E}$. Note that by the argument above, the performance of algorithm U on any permutation will be $\mathcal{E}$. Therefore, the algorithm A can do no better than the algorithm U that makes choices uniformly. ■

We complete the proof of theorem 6.1 by showing that scheduling by choosing the next job uniformly can do quite poorly.

Lemma 6.6 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 $(2 - \Omega(1/\sqrt{m}))C^*_{\max}$.

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 these jobs 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 $E_s = (2 - 2/\sqrt{m} + o(1/\sqrt{m}))C^*_{\max}$, which implies the stated result. ■

7 Other Models and Open Problems

We have defined a natural way to model the problem of scheduling in an on-line fashion, and have given matching or near-matching bounds on the competitive ratios that can be achieved in two fundamental cases of scheduling parallel machines. This work raises a number of interesting open questions. One direction of interest is to incorporate further elements and constraints into the basic models of scheduling parallel machines. We discuss here several additions to the model and their implications for on-line algorithms.

Release Dates: Traditional scheduling models often contain the added constraint that not all the jobs are available for processing at time 0, but that job j arrives at time r_j. In these models the r_j are known in advance; the natural on-line model is that the scheduler does not know that job j exists until time r_j. It is known that list scheduling can be adapted to work with release

times [15], and since our lower bounds still apply, list scheduling is an optimally competitive on-line algorithm for *identical machines with release dates.*

Our on-line algorithm for uniformly related machines can also be extended to handle release dates. Our algorithm consisted of a series of phases, each phase having an associated deadline d. If a job is released in the phase with deadline d the algorithm begins processing it at the start of the next phase (i.e. it is put on the queue of the slowest machine along with all the other jobs that were released in earlier phases and remain unprocessed). If the algorithm would have completed all the jobs in the kth phase, with deadline d, then upon completion of the kth phase it starts a new phase with deadline d and begins processing those jobs that arrived during the kth phase. It is not hard to see that this modification gives an $O(\log m)$ upper bound for this more general problem.

Preemption: In some scheduling scenarios it is a reasonable assumption that the processing of a job can be interrupted and restarted on another machine without losing any work; this is referred to as the *preemptive model.* In contrast to the nonpreemptive model which we have considered in this paper, an optimal schedule can be found off-line in polynomial time when preemption is allowed [18]. Interestingly enough, the on-line worst case characterization of both models is the same.

Theorem 7.1 An on-line algorithm for scheduling identical machines with preemption allowed has competitive ratio at least $(2 - \frac{1}{m})$ and there is an algorithm that achieves this ratio.

Proof: Graham has showed that list scheduling achieves a competitive ratio of $2 - \frac{1}{m}$ for scheduling identical machines with preemption allowed as well. To prove the corresponding lower bound, consider an instance with $n = m+1$ jobs. The adversary allows the scheduler to begin scheduling, and waits until either the scheduler preempts a job for the first time, or 1 time unit has passed, whichever comes first. Call this time t. By time t, at most m jobs can have been started (since scheduler didn't preempt anything until time t). Let job n be a job that was not started. At time t, the adversary sets $p_1 = \cdots = p_{n-1} = t$, and sets $p_n = tm/(m-1)$. The scheduler can clearly complete the entire schedule no sooner than time $t + tm/(m-1)$. The length of the optimal preemptive schedule is known to be $\max(p_{\max}, \sum p_j/m)$. In this case $\max(p_{\max}, \sum p_j/m) = tm/(m-1)$. Therefore the adversary has competitive ratio $[t + tm/(m-1)]/[tm/(m-1)] = 2 - 1/m$. ■

Gang Scheduling: In the models of scheduling we have discussed, each job is viewed as requiring only one processor. A possible extension to the identical machines model would be to view a job as requiring a specific number of processors, i.e. a job j must run on q_j processors simultaneously for p_j units of time. The value q_j is called the *width* of job j. In the literature on multiprocessing this model has been referred to as *gang scheduling*, whereby interacting threads of a computation are required to execute simultaneously [5].

Feitelson and Rudolph [4],[5] argue that it is a realistic assumption that the job sizes are not known beforehand but that the job widths are. They present several results assuming specific input distributions. It is clear that our lower bounds for identical machines apply here since they just use gangs of width 1. We have shown that a simple strategy comes within a factor of $3 - \frac{2}{m}$ of optimal; the proof will appear in the full version of the paper.

Theorem 7.2 There exists an on-line algorithm with competitive ratio $(3 - \frac{2}{m})$ for *gang scheduling* of identical machines.

Precedence Constraints: A standard addition to the models we have discussed is *precedence constraints*, which specify that certain jobs must be processed before others, via a partial order on the jobs. We note in passing that for identical machines list scheduling can be shown to achieve

the same bounds for a problem with precedence constraints [7], so list scheduling remains an optimally competitive on-line algorithm. When precedence constraints are added to the uniform non-preemptive model, the best known approximation algorithm, due to Jaffe [11], produces schedules within an $O(\sqrt{m})$ factor of optimal. This algorithm happens to be an on-line algorithm.

Unrelated Machines: We have assumed that the machines are either identical or uniformly related. A third model that has been studied in the literature is that of *unrelated machines*, where a job j runs at speed s_{ij} on machine i, and the s_{ij} need not be related in any coherent fashion. Davis and Jaffe give an $O(\sqrt{m})$ approximation algorithm for the non-preemptive model that is on-line but does not take advantage of restarts [3]. In related work with Shmoys, we have developed an on-line algorithm for the non-preemptive model with competitive factor $O(\log n)$ [19]; the best known lower bound is our lower bound for the uniformly related case.

It would also be interesting to study on-line scheduling algorithms for different realms of scheduling, such as shop scheduling or single-machine scheduling [15], and/or algorithms that optimize criteria other than the makespan. There is a tremendous amount of literature on these different realms, and relatively little of it translates into useful techniques for on-line algorithms.

Acknowledgements

We are grateful to Howard Karloff for suggesting on-line scheduling as a fruitful area of research. We are also grateful to David Shmoys for suggesting the models and problems in this paper, an important idea in the upper bound of section 5, and many helpful discussions. Finally, we 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] J. Aslam and A. Dhagat. Online algorithms for 2-coloring hypergraphs via chip games. Unpublished manuscript, July 1990.

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

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

[4] D.G. Feitelson and L.R. Rudolph. Mapping and scheduling in a shared parallel environment using distributed hierarchical control. In *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[5] D.G. Feitelson and L.R. Rudolph. Wasted resources in gang scheduling. In *Proceedings of the 5th Jerusalem Conference on Information Technology*, October 1990.

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

[8] 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.

[9] 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.

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

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

[12] S. Jain, K. Barber, and D. Osterfeld. Expert simulation for on-line scheduling. *Communications of the ACM*, pages 54–60, 1990.

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

[14] R. Karp, U.V. Vazirani, and V.V. Vazirani. On-line algorithms for bipartite matching. In *Proceedings of the 22st Annual ACM Symposium on Theory of Computing*, pages 352–358, 1990.

[15] Eugene L. Lawler, Jan Karel Lenstra, Alexander H.G. Rinooy Kan, and David B. Shmoys. Sequencing and scheduling: Algorithms and complexity. Technical Report BS-R8909, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1989. To appear in Handbooks in Operations Research and Management Science, Volume 4: Logistics of Production and Inventory.

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

[17] 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.

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

[19] D.B. Shmoys, J. Wein, and D.P. Williamson. An on-line algorithm for scheduling unrelated machines. Unpublished manuscript.

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

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

8 Appendix

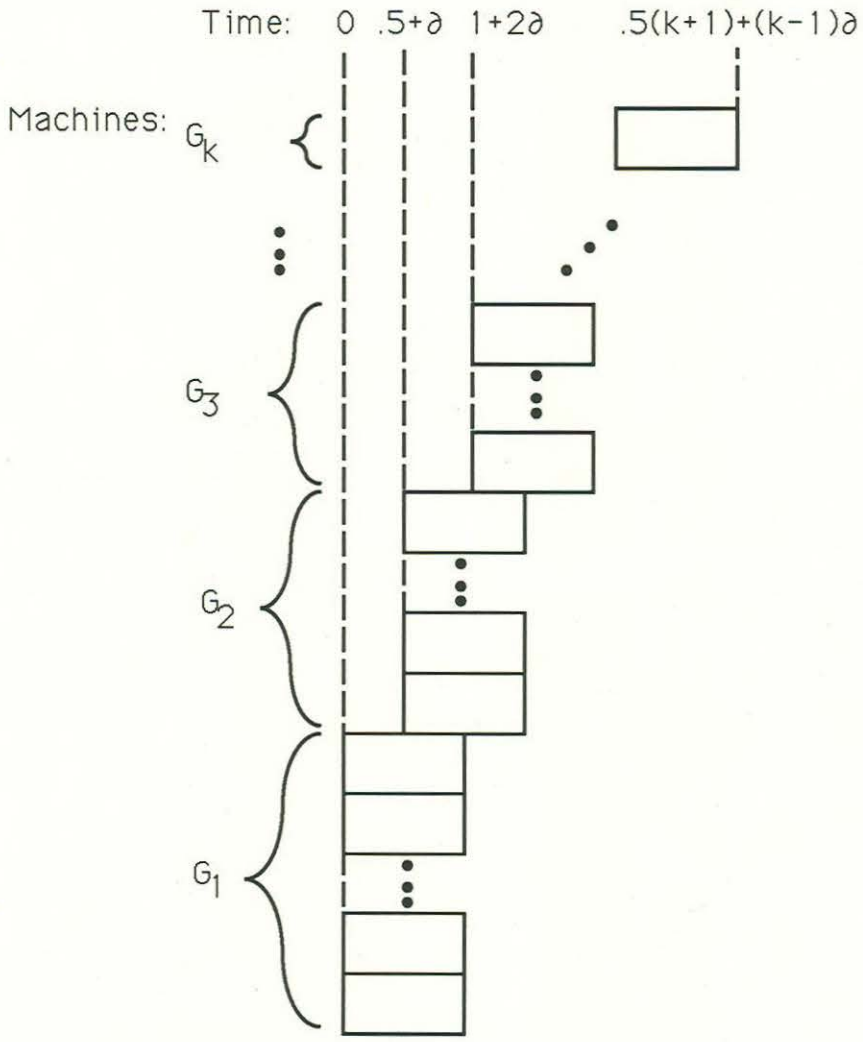

Figure 1: Near optimal schedule for example in section 4