On-Line Scheduling — A Survey

Jiří Sgall

E-mail: sgall@math.cas.cz
http://www.math.cas.cz/~sgall/
Mathematical Institute, AS CR
Žitná 25
115 67 Praha 1
Czech Republic

1 Introduction

Scheduling has been studied extensively in many varieties and from many viewpoints. Inspired by applications in practical computer systems, it developed into a theoretical area with many interesting results, both positive and negative.

The basic situation we study is the following. We have some sequence of jobs that have to be processed on the machines available to us. In the most basic problem, each job is characterized by its running time and has to be scheduled for that time on one of the machines. In other variants there may be additional restrictions or relaxations specifying which schedules are allowed. We want to schedule the jobs as efficiently as possible, which most often means that the total length of the schedule (the makespan) should be as small as possible, but other objective functions are also considered.

The notion of an on-line algorithm is intended to formalize the realistic scenario, where the algorithm does not have the access to the whole input instance, unlike the off-line algorithms. Instead, it learns the input piece by piece, and has to react to the new requests with only a partial knowledge of the input. Such scheduling algorithms are the topic of this survey. (For a general reference on on-line algorithms see upcoming book [13].)

Scheduling have continuously been an active research area, reflecting the changes in the theoretical computer science. When the theory of NP-completeness was developed, many scheduling problems have been shown to be NP-complete: Garey and Johnson [39] give 18 basic NP-complete scheduling problems; since then many new variants were considered and shown to be NP-complete. After the NP-completeness results, the focus shifted to designing approximation algorithms, often using quite non-trivial techniques and insights. There is extensive literature on these subjects, for a recent surveys see e.g. [59, 54].

Many natural heuristics for scheduling are in fact on-line algorithms. Hence when the study of on-line algorithms using competitive analysis became usual, this approach was naturally and quite successfully applied to scheduling.

Organization of the survey

We define some of the variants of scheduling that have been studied in the on-line setting in Section 2. In Section 3 we discuss the early results on on-line scheduling, focusing on Graham's paper [43]. The three sections 4 to 6 survey the results divided according to the three different on-line paradigms described in Section 2.2. In Section 7 we discuss several papers which study various modification of competitive analysis in which the on-line algorithm is less restricted than in the standard situation.

We keep the exposition as informal as possible, focusing on the intuition behind the results. We include several algorithms and proofs that are reasonably simple and illustrate more general techniques. Each section contains its own subsection of open problems, where we describe the open problems which we feel are the most important ones. In addition, there are of course many cases where we do not know the tight bounds on the competitive ratio, or variants that were not studied at all in the on-line setting. We give the appropriate citations for each result or variant of scheduling; whenever we give the citation in the heading of a section, it means that all the results come from the cited paper(s).

2 Taxonomy of on-line scheduling problems

After giving the general definitions, in Sections 2.2 to 2.5 we survey the features important in every variant of scheduling; these include the on-line character of the problem, the objective function, the use of randomization, release times and precedence constraints. Then we define other features which are used only in a few variants.

2.1 General definitions and preliminaries

The *number of machines* is always denoted m, and n stands for the *number of jobs*. Each job is characterized by its *running time*, which we denote t (it is also often denoted p and called processing time), and perhaps other characteristics as required by each variant of scheduling. The scheduling algorithm is asked to produce a schedule, which means that each job is assigned to one or more machines and one or more time slots, according to the variant of scheduling. Each machine is assigned to a single job at any time, and the processing of a job always takes at least its running time.

All scheduling problems we consider ask for minimization of some objective function (performance measure). The performance of an on-line algorithm is measured by the *competitive ratio* (w.r.t. some objective function). An on-line algorithm is σ-competitive if for each input instance the objective value of the schedule produced by the algorithm is at most σ times larger than the optimal objective value. The competitive ratio may depend on m, but should be independent of n, which reflects the fact that the number of jobs is not known to the on-line algorithm.

The minimal makespan for a given instance is denoted T_{opt}. A machine is *busy* at a given time, if it is assigned to some job, otherwise it is *idle*. The *load* of the machine is the total time during which this machine is busy (i.e., the idle time does not contribute to the load of the machine). A job is *available* at a given time, if it was not scheduled yet and can be scheduled in a time slot(s) starting at that time consistently with the restrictions of the given variant of scheduling (e.g., it is after its release time). In many variants of scheduling we encounter the golden ratio, defined by $\phi = (\sqrt{5}+1)/2 \approx 1.6180$.

While most results work with arbitrary real running times, some make technical restrictions, e.g., the minimal running time is specified or the running times and release times are required to be integral. This is important for example for some results on preemptive scheduling, where we want to divide the time of one machine equally among all jobs. Another possibility how to deal with this problem is to use the common more general definition of competitiveness which includes an additive term, an algorithm is then σ-competitive if its objective value is at most some constant plus σ times the optimal objective value. We do not pay much attention to these restrictions, as they are only technical details.

2.2 Different on-line paradigms

For on-line scheduling the most important classification of the on-line problems is according to which part of the problem is given on-line. There are several very different possibilities.

Scheduling jobs one by one. In this paradigm the jobs are ordered in some list (sequence) and are presented one by one according to this list. Each of them has to be assigned to some machine(s) and time slot(s) before the next jobs are seen, consistently with other restrictions given by the problem. As soon as the job is presented we know all its characteristics, including the running time. We are allowed to assign the jobs to arbitrary time slots (i.e., they can be delayed), thus a job can start running later than the successive jobs in the sequence; however, once we see the successive jobs we cannot change the assignment of the previous jobs.

Unknown running times. Here the main on-line feature is the fact that the running time of a job is unknown until the job finishes; an on-line algorithm only knows whether a job is still running or not. Unlike in the previous paradigm, at any time all currently available jobs are at the disposal of the algorithm; any of them can be started now on any machine(s) or delayed further. Also, if preemptions or restarts are allowed (see Section 2.6), the algorithm can decide to preempt or stop any job which is currently running. The jobs may become available over time according to their release times or precedence constraints (see Section 2.5), but the situation when all jobs are available at the beginning plays an important role in this paradigm, too. If there are other characteristics of a job than its running time, they are known when the job becomes available (such characteristics may include for example the number of parallel processors it uses, which has to be known to guarantee that the job is scheduled legally).

Jobs arrive over time. In this paradigm the algorithm has the same freedom as in the previous one, and in addition the running time of each job is also known when that job is available. Thus the only on-line feature is the lack of knowledge of jobs arriving in the future.

Sometimes the algorithm that know the running time of a job as soon as it arrives are called *clairvoyant*, in contrast to *non-clairvoyant* algorithms that correspond to the previous paradigm of unknown running times.

Interval scheduling. All the previous paradigms assume that a job may be delayed. Contrary to that, the paradigm of interval scheduling assumes that each job has to be executed in a precisely given time interval; if this is impossible it may be rejected. This scenario is very different from the previous three. For example, it is meaningless to measure the length of the schedule, as it is essentially fixed; instead we measure the weight (or the number) of accepted jobs. We do not cover this paradigm in this survey. It is studied for example in the papers [83, 62, 32], and it is also related to load balancing [5].

2.3 Objective functions

The most common objective function is the *makespan*, which is the length of the schedule, or equivalently the time when the last job is completed. In one of the variations we allow jobs to be rejected at a certain penalty, in which case we minimize the sum of the makespan and the penalties of rejected jobs. These objective functions formalize the viewpoint of the owner of the machines. If the makespan is small, the utilization of his machines is high; this captures the situation when the benefits of the owner are proportional to the work done. Penalties are intended to capture the situation when this is not true: if a job has a long running time and small benefit (i.e., there is a small penalty for not scheduling it), it is better to reject it.

If we turn our attention to the viewpoint of a user, the time it takes to finish individual jobs may be more important; this is especially true in interactive environments. Thus, if many short jobs are postponed after some long job, it is unacceptable to the user of the system even if the makespan is optimal. For that reason other objective functions are studied, namely the *total completion time*, the *total flow time* (also called response time), and the *total waiting time*. The completion time of a job is the time when this particular job is completed; thus the makespan equals the maximal completion time. The flow time of a job is the time the job is in the system, i.e., the completion time minus the time when it becomes first available. The waiting time is the flow time minus the running time of a job. The objective functions are the sums of these values over all jobs. (Equivalently, we can take the average values, as they always differ by a factor of n.) The competitive ratio w.r.t. the total completion time is at most the competitive ratio w.r.t. the total flow time, as the flow times are smaller by an additive term which is equal for both on-line and optimal schedules; similarly the competitive ratio w.r.t. the total flow time is at most the competitive ratio w.r.t. the total waiting time. Thus the best algorithms are those competitive w.r.t. the total waiting time. The weighted variant of these measures is also studied; in this case each job has its weight and we take the weighted sums. This captures the situation when some jobs are more important than others. Maximal waiting and flow time are also reasonable objective functions, but they were not considered in the context of on-line scheduling algorithms so far.

Another possibility is to consider general L_p norms of the vector of loads of the machines, which is studied for load-balancing, where in particular L_2 norm has a natural interpretation [5, 4].

If we allow preemptions (see Section 2.6), we usually want to minimize the number of times we preempt a job. However, this is always a secondary criterion. This means that we are interested to quantify how many preemptions we need to use to obtain optimal or almost optimal competitive ratio w.r.t. some other objective function.

2.4 Randomization

In most cases we are interested in both deterministic and randomized algorithms. If we allow randomization, we consider the expected objective value, where the expectation is taken over the random choices of the algorithm. A randomized algorithm is σ-competitive if for each instance this expectation is within a factor of σ of the optimal objective value. This corresponds to the so-called oblivious adversary [12], which has to commit to an input instance beforehand, without any knowledge of the random bits or actions of the algorithm.

2.5 Release times and deadlines, precedence constraints and conflicting jobs

Each job may have an individual *release time*, which is the earliest time when it may be scheduled, and in the on-line setting also the time when it becomes known. The dual notion of individual *deadlines* is common in off-line scheduling, but not in the on-line case. The reason is that individual deadlines only define feasible solutions, which is not compatible with the goal of the on-line algorithm which is to do reasonably well on some global measure. This intuition can even be stated formally. In the case of preemptive scheduling of jobs arriving over time with known running times there exists an on-line algorithm which schedules all jobs before their common deadline if this is feasible; as soon as we allow two different deadlines no such on-line algorithm exists [48].

An often considered variant assumes that there are some *precedence constraints* between the jobs. They are generally given by a directed acyclic graph on the jobs; each directed edge indicates that one job has to be scheduled before another one. In the on-line framework a job is known only after all its predecessors in the dependency graph are processed by the on-line algorithm. A related model considers the case of *conflicting jobs* when the jobs may conflict with each other, but the order in which such pairs have to be scheduled is not given; in such a case the conflict graph is undirected.

2.6 Preemptions and restarts

In many problems it is assumed that a running job may be *preempted*, i.e., its processing may be stopped and resume later on the same or different machine(s). In the on-line setting, there is another possibility which is meaningless for the off-line algorithms. Namely, a running job can be stopped and *restarted* later on the same or different machine(s); thus in order to finish, it has to be assigned to the same machine(s) for its whole running time without an interruption.

2.7 Parallel jobs

Parallel jobs are those jobs that have to be scheduled on some number of machines at the same time. They are characterized by two parameters, the running time t and the number of requested machines p. We consider two variants. In the first the jobs are non-malleable, which means that they have to be scheduled on the requested number of machines. On the other hand, malleable jobs may be scheduled on fewer machines, at the cost of increasing the processing time. Most of the time we consider ideally malleable jobs, where the increase of the running time is proportional to the decrease of the number of machines, i.e., scheduling on $q \leq p$ machines takes time tp/q.

2.8 Different speeds of machines

The speeds of the machines can be different. The three most common models of this situation were both studied in the on-line setting.

In the variant of *uniformly related machines* the ith machine has speed $s_i > 0$. If a job with running time t is scheduled on ith machine, its processing takes time t/s_i. (Here we possibly deviate from our convention that processing a job takes always at least its running time; we can always repair this by scaling the speeds so that $s_i \leq 1$.)

The next variant is that of *unrelated machines* where the vector of speeds is possibly different for each job. However, even if the running time is unknown, we assume that the speeds are known for each job (i.e., for each job we know the relative speeds of the machines).

The last variant is the *restricted assignment*. Here the machines have identical speed, but each job can be executed only on some subset of processors. This can be thought of as a special case of unrelated machines, where the speeds are always 1 or infinitely small. This variant is not comparable to uniformly related machines.

2.9 Shop scheduling

In shop scheduling the job has several tasks (operations) that have to be processed on different machines. The running time of each task is a separate parameter. The different tasks of the same job cannot be scheduled at the same time. According to additional restrictions, we distinguish *open shop*, when the different tasks of a job may be scheduled in any order (in addition, if preemption is allowed, different tasks may be even interleaved), *flow shop*, if the order of the tasks is fixed and it is the same for all jobs, and *job shop*, if the order of the tasks is fixed and possibly different for each job.

2.10 Variants not covered in this survey

A very different model of on-line scheduling in real-time systems, where each job has its deadline and value and the goal is to maximize the value of jobs finished before their deadline, is considered in several papers [28, 11, 52, 65].

Other papers study models with additional obstacles. On-line scheduling in the presence of processor faults is studied in [51, 53]. On-line scheduling in presence of delays in communication between processors is studied in [26, 27].

3 General results and history

The first proof of competitiveness of an on-line algorithm for a scheduling problem, and perhaps for any problem, was given by Graham already in 1966 [43]. He studied a simple deterministic greedy algorithm, now commonly called List Scheduling. The studied model is the basic one, where we have m identical machines and a sequence of sequential jobs characterized by their running times. The objective is to minimize the makespan. The algorithm was designed for the case of precedence constraints, but it can be easily modified to handle also release times [46]. Preemption is not used.

Algorithm List Scheduling

(i) The jobs are ordered in arbitrary list (sequence).

(ii) Whenever some machine is idle, we schedule on it the first job on the list which is available (i.e., it is not scheduled yet, the current time is greater than its release time, and all its predecessors in the precedence graph are finished).

This algorithm works in all three on-line paradigms we study. It is already formulated in the model with release times, and since it does not use the information about running times, it carries over to the paradigm or unknown running times. For the paradigm of scheduling jobs one by one we simply present the jobs in the order of the input list, and when a job is presented, we schedule it on a machine with the smallest load so far. This works only for the case with no precedence constraints, but we do not study precedence constraints in this paradigm (cf. Section 4).

Theorem 1 [43, 46]. *The competitive ratio of List Scheduling is* $2 - \frac{1}{m}$.

Proof. First we show that the competitive ratio of List Scheduling is not better than $2-\frac{1}{m}$. Consider the sequence of $m(m-1)$ jobs with running time 1 followed by one job with running time m. There are no precedence constraints or release times (and hence the lower bound is true in any of the three on-line paradigms). List Scheduling schedules this sequence in time $2m-1$, while the optimal schedule has makespan m.

Next we show that List Scheduling is $(2-\frac{1}{m})$-competitive if there are no precedence constraints and no release times. Consider the job that finishes as the last one, suppose it was started at time τ and its running time is T. At all times before τ, all the machines are busy, as otherwise the last job would be scheduled earlier. Hence the optimal makespan $T_{\rm opt}$ is at least $T_{\rm opt} \geq \tau + \frac{T}{m}$, as the optimal schedule has to schedule all the jobs. We certainly have $T_{\rm opt} \geq T$, as the optimal schedule takes time T to process even this single job. Combining these two inequalities we get that the makespan of the on-line solution, which is $\tau + T$, is bounded by $\tau + T = \tau + \frac{T}{m} + (1-\frac{1}{m})T \leq (2-\frac{1}{m})T_{\rm opt}$.

For the general case with precedence constraints and release times we need a different bound on the time when some machine is idle. We define a sequence of jobs $J_1, \ldots, J_k$ inductively as follows. Let J_1 be the job that finishes last. If no predecessor of J_i in the dependency graph finishes after the release time of J_i, we stop. Otherwise J_{i+1} is defined as the latest-finishing predecessor of J_i. It follows that whenever some machine is idle, either one of the jobs $J_1, \ldots, J_k$ is running or it is before the release time of J_k (otherwise one of the jobs J_i would be available and hence scheduled). The optimal schedule has to schedule these jobs sequentially from J_k to J_1, and it can start only after J_k is released. This proves that the total time when some machine is idle is bounded by the optimal makespan, and the rest of the argument is the same as without precedence constraints. □

While we now interpret Graham's result as a proof of competitiveness of List Scheduling, it should be stressed that his analysis was deeper. He gives examples where we can increase the makespan by making the problem easier ("timing anomalies" stands for this paradox), namely by either increasing the number of machines, or decreasing the running time of some job, by relaxing the precedence constraints, or, finally, by reordering the list. He proves that in all of these cases the makespan can change by almost a factor of 2, giving the tight bounds in all cases. The case of reordering the list amounts to the competitiveness analysis given in Theorem 1, as the optimal schedule can be obtained by some particular ordering of the list.

In the follow-up paper [44] Graham shows that the factor of 2 decreases if we modify the algorithm so that some number of long jobs is scheduled first using an optimal schedule, and the rest is scheduled by List Scheduling. Clearly, this algorithm is no longer on-line in any of the paradigms we study.

Other two early papers that contain results about on-line scheduling algorithms are [66, 23]. The first one gives an optimal algorithm for minimizing the makespan of a preemptive schedule on identical machines where jobs arrive over time, and mentions that the algorithm is on-line. The second paper is to our best knowledge the first one that states explicitly a lower bound on the performance ratio of any on-line algorithm for some scheduling problem, namely the bound of $\Omega(\sqrt{n})$ for non-preemptive scheduling jobs with unknown running times on uniformly related machines; the paper even mentions the possible usefulness of restarts, which later indeed proved to be quite useful in this case [75].

Around 1990 new results were discovered concerning many variants of on-line scheduling, both old and new. Most of the results use the makespan as the objective function, consequently our understanding of this measure is most complete. Recently other objective functions are drawing more attention, perhaps also because in many practical applications they are more important.

4 Scheduling jobs one by one

This paradigm corresponds most closely to the standard model of request sequences in competitive analysis. It can be formulated in the language of on-line load balancing as the case where the jobs are permanent and the load is their only parameter corresponding to our running time (cf. [5]).

In this paradigm we do not allow release times and precedence constraints, as these restrictions appear to be unnatural with scheduling jobs one by one. In most of the variants it is also sufficient to assign each job to some machine(s) for some length of time, but it is not necessary to specify the actual time slot(s), in other words it is not necessary or useful to introduce idle time on any machine.

We first give the results considering minimizing the makespan, only in Sections 4.7 and 4.8 we briefly mention results for other objective functions, namely minimizing the L_p norm and the total completion time.

4.1 The basic model

We have m machines and a sequence of jobs characterized by their running times. The jobs are presented one by one, and we have to schedule each job before we see the next one. Performance is measured by the makespan. Each job is assigned to a single machine. There are no additional constraints, preemption is not allowed, all the machines have the same speed, and the objective function is the makespan. In this section we are interested in deterministic algorithms.

By Theorem 1 it follows that the competitive ratio of List Scheduling is $2-\frac{1}{m}$. This is provably the best possible for $m = 2$ and $m = 3$ [31], but for larger m it is possible to develop better algorithms.

From the proof of Theorem 1 it is clear what is the main issue in designing algorithms better than List Scheduling. If all machines have equal loads and a job with long running time is presented, we create a schedule which is almost twice as long as the optimal one. This is a problem if the scheduled jobs are sufficiently small, and the optimal schedule can distribute them evenly on $m-1$ machines in parallel with the last long job on the remaining machine. Thus, to achieve better results, we have to create some imbalance and keep some machines lightly loaded, to be used by long jobs.

Let us suppose that we want to achieve competitive ratio σ. When a job is presented, we can schedule it on any machine such that after this step the

	deterministic			randomized	
m	lower bound	upper bound	LS	lower bound	upper bound
2	1.5000	1.5000	1.5000	1.3333	1.3334
3	1.6666	1.6667	1.6667	1.4210	1.5567
4	1.7310	1.7333	1.7500	1.4628	1.6589
5	1.7462	1.7708	1.8000	1.4873	1.7338
6	1.7730	1.8000	1.8333	1.5035	1.7829
7	1.7910	1.8229	1.8571	1.5149	1.8169
∞	1.8520	1.9230	2.0000	1.5819	–

Table 1. Current bounds for algorithms scheduling jobs one by one with no constraints.

competitive ratio is at most σ. Suppose that we choose always the most loaded of these machines, to create as large imbalance as possible. This seems to be a natural idea to prevent the previous problems, however, it turns out that it does not work, either. If this algorithm is presented with a long list of jobs with the same running time, it distributes them almost evenly on a constant fraction of the machines, with only one job scheduled on each of the remaining machines. Now we can continue with a sequence of long jobs, first making the load distributed evenly on all machines, and then forcing the schedule to be too long. Thus this method cannot give a better competitive ratio than List Scheduling.

To design a good algorithm, we need to avoid both of these extremes. Current results use two different approaches. One is to schedule each job on one of the two currently least loaded machines [37, 17]. This gives better results than List Scheduling for any $m \geq 4$, and achieves the currently best upper bounds for small m. However, for large m, the competitive ratio still approaches 2. This approach leaves at most one lightly loaded machine, hence after two long jobs we get a long schedule and the competitive ratio is at least $2 - \frac{2}{m}$. To keep the competitive ratio bounded away from 2 even for large m it is necessary to keep some constant fraction of machines lightly loaded. Such an algorithm was first developed in [8], later better algorithms based on this idea were designed in [55, 1] to give the currently best upper bounds for large m. The analysis of all these algorithms is relatively complicated.

The current state of our knowledge is summarized in Table 1. For comparison we include also the competitive ratio of List Scheduling. (See Section 4.2 for a discussion of results for randomized algorithms.) The observation that List Scheduling is optimal for $m = 2, 3$ is due to [31]. The other lower bounds for small m are due to [17]. The lower bound for large m is due to [1], improving upon [9]. Very recently R. Chandrasekaran claimed a lower bound of $\sqrt{3} \approx 1.7321$ for $m = 4$, which would significantly decrease the gap in this case.

4.2 Randomized algorithms

Much less is known about randomized algorithms for the basic model studied in Section 4.1. Only for the case of $m = 2$ we know an optimal randomized algorithm. A 4/3-competitive randomized algorithm for two machines was presented in [8]. First we prove that this is best possible.

Theorem 2 [8]. *No randomized algorithm for 2-machine scheduling can be better than* $4/3$*-competitive.*

Proof. Consider the sequence of three jobs with running times 1, 1, and 2. Suppose we have an algorithm which is better than 4/3-competitive. We schedule the first two jobs, order the machines according to their load and consider the expected load of the more loaded machine and of the less loaded one, where the expectation is taken over the random choices of the algorithm. After scheduling the first two jobs, the expected load of the more loaded machine is less than 4/3, as it is equal to the expected makespan and the optimal makespan is 1. Hence the expected load of the less loaded machine is more than 2/3, and even if the last job is always scheduled on the least loaded machine, the expected makespan after scheduling of all three jobs is more than 8/3, contradicting the assumption that the algorithm is better than 4/3-competitive. □

In the proof we can replace the first two jobs by an arbitrary sequence of jobs with total running time 2. Hence the proof actually shows that in any 4/3-competitive algorithm, the expected load of the more loaded machine has to be at least twice as much as the expected load of the other machine at all times. This has to be tight whenever we can partition the jobs into two sets with exactly the same sum of running times. The most natural way to design an algorithm with this in mind is to keep the desired ratio of expected loads at all times, and this in fact works.

Algorithm Random

(i) If possible, schedule the job randomly so that afterwards the expected makespan equals 2/3 of the total running time.

(ii) Otherwise schedule the job always on the less loaded machine.

To implement this algorithm it is necessary to keep track of all possible schedules and their probabilities. The naive way of doing this uses 2^{n-1} configurations after n jobs, but it is possible to implement the algorithm with only n configurations [8].

Theorem 3 [8]. *The algorithm Random is* $4/3$*-competitive for two machines.*

Proof. After scheduling some sequence of jobs, let a be the expected makespan, b be the total running time of all jobs scheduled so far, and let T be the longest running time among all jobs scheduled so far. We prove by induction that at any time the following is true:

$$a \geq \frac{2}{3}b, \text{ and} \tag{1}$$

$$\text{If } a > \tfrac{2}{3}b \text{ then } T \geq \frac{3}{4}a. \tag{2}$$

From this condition it follows that the algorithm is 4/3-competitive, as the optimal makespan is at least $\max(T, b/2)$. Both conditions are trivially true before any job is scheduled.

Let t be the running time of the job that has to be scheduled next, and let $a(t)$ be the expected makespan after scheduling this job deterministically on the less loaded machine. Consider what happens if we change the probability that the job is scheduled on the more loaded machine continuously from 0 to 1. The expected makespan increases continuously from $a(t)$ to $a + t$. By the induction assumption (1), $a + t \geq \frac{2}{3}b + t \geq \frac{2}{3}(b + t)$. Hence if $a(t) \leq \frac{2}{3}(b + t)$, we can schedule the job so that afterwards the expected makespan equals to $\frac{2}{3}(b + t)$, as required in the step (i) of the algorithm, and both (1) and (2) are satisfied.

Thus we only need to consider the case when $a(t) > \frac{2}{3}(b + t)$. In this case the next job is always scheduled on the less loaded machine, and (1) is satisfied since the expected makespan is $a(t)$. It remains to prove the condition (2). We distinguish three cases according to the value of t.

First suppose that $t \leq 2T - b$. This means that $T \geq t + b - T$, i.e., T is larger than the total running time of all the other jobs including the next one. Hence by scheduling the new job on the less loaded machine we do not change the makespan and (2) remains satisfied.

Next suppose that $t \geq b$. This means that the next job has longer running time than all the previous jobs together, and hence the machine on which it is scheduled always becomes the more loaded one. Hence $a(t) = t + b - a$. Since $b - a \leq b/3$ by the induction assumption (1), it follows that $a(t) \leq t + b/3 \leq \frac{4}{3}t$, and (2) remains true, as t is now the longest running time.

Now consider the remaining case when $\max(0, 2T - b) \leq t \leq b$. We prove that in this case $a(t) \leq \frac{2}{3}(b + t)$, hence by the previous considerations the next job is scheduled by the step (i) of the algorithm and the inductive conditions are satisfied. We prove that $a(t) \leq \frac{2}{3}(b+t)$ at the endpoints of the interval allowed for t and that $a(t)$ is a convex function of t; the inequality then follows for every t in the interval, since the right hand side is linear in t. First consider the endpoints of the interval. If $t = b$, we have seen in the previous paragraph that $a(t) \leq t + b/3$, which equals $\frac{2}{3}(b+t)$. If $t = 2T - b \geq 0$, we have $a(t) = a \leq \frac{4}{3}T = \frac{2}{3}(b+t)$, using the induction assumption (2). If $t = 0 > 2T - b$, we know that $T < \frac{1}{2}b \leq \frac{3}{4}a$ by the assumption (1); therefore it has to be the case that $a = \frac{2}{3}b = \frac{2}{3}(b+t)$ if (2) is satisfied. It remains to prove that $a(t)$ is convex. The derivative of $a(t)$ at point t is equal to the probability that after scheduling the job with running time t on the less loaded machine it becomes the more loaded one. It is easy to see that this probability is non-decreasing in t, hence $a(t)$ is convex. (To be more precise, we should notice that the derivative may be undefined at finitely many points. However, this does not change the conclusion.) □

Very recently new randomized algorithms for small m were developed in [71, 70]. It is provably better than any deterministic algorithm for $m = 3, 4, 5$ and better than the currently best deterministic algorithm for $m = 6, 7$. It always assigns the new job on one of the two least loaded machines, similarly to the deterministic algorithms for small m from [37, 17]. Consequently, its competitive ratio approaches two as m grows. The analysis of this algorithm is again difficult, even involving extensive computations to obtain the best results.

The idea of the lower bound for two machines can be extended to arbitrary number of machines [16, 72, 74]. It turns out that for m machines, the expected loads should be in geometric sequence with the ratio $m : (m-1)$, if the machines are always ordered so that their loads are non-decreasing. (For example, for $m = 3$ the ratio of loads is $4 : 6 : 9$.) This leads to a lower bound of $1/(1-(1-1/m)^m)$, which approaches $e/(e-1) \approx 1.5819$ for large m and increases with increasing m.

For any $m > 2$ it is an open question whether there exists an algorithm matching this lower bound. (Seiden [71] demonstrated that his algorithm does not match this bound.) The insight from the proof of the lower bound leads to a natural invariant that should be preserved by any algorithm matching it. Namely, such an algorithm should preserve the ratio of expected loads described above. An algorithm based on this invariant would be a natural generalization of the optimal algorithm for two machines from [8]; it would also follow the suggestion from [18] (see Section 4.3). This faces several problems.

First of all, it is not clear at all that we would be able to handle long jobs similarly as for $m = 2$. A job is long if its running time is more than the $1/(m-1)$ fraction of the sum of running times of all previous jobs (intuitively this means that its running time determines the optimal makespan). For $m = 2$ this means that the running time of a long job is more than the total running time of all previous jobs, therefore we know that no matter on which machine it is scheduled, this machine will become the most loaded one; we used this fact significantly in the proof of Theorem 3. For three machines this is no longer true, and hence the structure of the problem is much more difficult.

Second, it is not clear whether we would be able to preserve the invariant ratio of expected loads even if all jobs are small. In [74] it is demonstrated that even for $m = 3$ it is impossible to preserve this invariant inductively, meaning that there exists a probability distribution on the configurations such that the expected loads have the desired ratio, but after the next job this ratio cannot be maintained; moreover this configuration is reachable so that the ratio is kept invariant at all times, except for the first few jobs. We want to keep the ratio invariant, so this means that for a design of a matching algorithm we should use a stronger inductive invariant; at present we do not know if this is possible.

Third, for $m = 2$ the lower bound proof implies that the ratio of loads has to be at least $2 : 1$ at all times. For $m = 3$ this is only true if the previous jobs can be exactly balanced on two machines, and for $m > 3$ we get even more such balancing conditions. This means that we have some more freedom in the design of the algorithm, and hence it is harder to improve the lower bounds.

To summarize, we have the optimal algorithm for $m = 2$, and an improvement of the deterministic algorithms for small m. However, for $m > 7$ we have no randomized algorithm with a better competitive ratio than known deterministic algorithms; this means that we do not know how to make use of randomization for large m. See Table 1.

4.3 Preemptive scheduling [18]

In this model preemption is allowed. Each job may be assigned to one or more machines and time slots (the time slots have to be disjoint, of course), and this assignment has to be determined completely as soon as the job is presented. It should be noted that in this model the off-line case is easily solved, and the optimal makespan is the maximum of the maximal running time and the sum of the running times divided by m (i.e., the average load of a machine).

It is easy to see that the lower bounds from Section 4.2 hold in this model, too, as they only use the arguments about expected load. This again leads to a lower bound of $1/(1-(1-1/m)^m)$, which approaches $e/(e-1) \approx 1.5819$ for large m, valid even for randomized algorithms. The proof shows that expected loads in the optimal algorithm have to be in geometric sequence with the ratio $m : (m-1)$, if the machines are always ordered so that their loads are non-decreasing and there are no long jobs; in this case this has to be true at all times, as using preemption we can always balance the machines exactly.

Interestingly, for this model there exists a deterministic algorithm matching this lower bound. It essentially tries to preserve the invariant above, with some special considerations for large jobs.

Thus, in this model both deterministic and randomized cases are completely solved, giving the same bounds as the randomized lower bounds in Table 1. Moreover, we know that randomization does not help. This agrees with the intuition. In the basic model randomization can serve us to spread the load of a job among more machines, but we still have the problem that the individual configurations cannot look exactly as we would like. With preemption, we can spread the load as we wish, while still keeping just one configuration with the ideal spread of the total load, and this makes it more powerful than randomization.

4.4 Scheduling with rejections

In this version jobs may be rejected at a certain penalty. Each job is characterized by the running time and the penalty. A job can either be rejected, in which case its penalty is paid, or scheduled on one of the machines, in which case its running time contributes to the completion time of that machine (as usual). The objective is to minimize the makespan of the schedule for accepted jobs plus the sum of the penalties of all rejected jobs. Again, there are no additional constraints and all the machines have the same speed.

The main goal of an on-line algorithm is to choose the correct balance between the penalties of the rejected jobs and the increase in the makespan for the accepted jobs. At the beginning it might have to reject some jobs, if the penalty for their rejection is small compared to their running time. However, at some point it would have been better to schedule some of the previously rejected jobs since the increase in the makespan due to scheduling those jobs in parallel is less than the total penalty incurred. Thus this on-line problem can be seen as a non-trivial generalization of the well-known ski rental problem.

We first look at deterministic algorithms in the case when preemption is not allowed [10]. At first it would seem that a good algorithm has to do well both in deciding which jobs to accept, and on which machines to schedule the accepted jobs. However, it turns out that after the right decision is made about rejections, it is sufficient to schedule the accepted jobs using List Scheduling. This is certainly surprising, as we know that without rejections List Scheduling is not optimal, and hence it is natural to expect that any algorithm for scheduling with rejections would benefit from using a better algorithm for scheduling the accepted jobs.

We can solve this problem optimally for $m = 2$ and for unbounded m, the competitive ratios are ϕ and $1 + \phi$. However, the best competitive ratio for fixed $m \geq 3$ is not known. It certainly tends to $1 + \phi$, which is the optimum for unbounded m, but the rate of convergence is not clear: while the upper bound is $1 + \phi - 1/m$ (i.e., the same rate of convergence as for List Scheduling), the lower bound is only $1 + \phi - 1/O(\log m)$.

The optimal algorithm for two machines is extremely simple: if a job with running time t and penalty p is presented, we reject it if $t \geq \phi p$; otherwise we schedule it using List Scheduling. The optimal algorithm for arbitrary m uses two rules for rejecting jobs: (i) a job is rejected whenever $t \geq mp$, and (ii) a job is rejected if $t \geq \phi(P + p)$, where P is the total penalty of all jobs rejected so far by the rule (ii); accepted jobs are again scheduled by List Scheduling.

The lower bounds for small m from [10] work also for preemptive deterministic algorithms, but for large m yield only a lower bound of 2. An improved algorithm for deterministic preemptive scheduling was designed in [69]. It achieves competitive ratio 2.3875 for all m. The scheme for rejecting jobs is similar as in the previous case, but the optimal algorithm for preemptive scheduling is used instead of List Scheduling. An interesting question is whether a better than 2-competitive algorithm can be found for $m = 3$: we now know several different 2-competitive algorithms even without preemption, but the lower bound does not match this barrier.

Randomized algorithms for this problem were designed in [70, 69]. The general idea is to use modifications of the deterministic algorithms where the thresholds for rejection are parameterized, and certain random choice of these parameters is made. In the non-preemptive case the competitive ratios are 1.5, 1.8358, and 2.0545 for $m = 2, 3$, and 4. With preemption better upper bounds can be achieved. No algorithms better than the deterministic ones are known for large m. The lower bounds for randomized scheduling without rejection (Table 1) clearly apply here (set the penalties infinitely large), and no better lower bounds are known.

m	deterministic lower bounds	deterministic upper bounds		randomized upper bounds	
		non-preemptive	preemptive	non-preemptive	preemptive
2	$\phi \approx 1.6180$	ϕ	ϕ	1.5000	1.5000
3	1.8392	2.0000	2.0000	1.8358	1.7774
4	1.9276	2.1514	2.0995	2.0544	2.0227
5	1.9660	2.2434	2.1581	2.1521	2.0941
∞	$1+\phi \approx 2.6180$	$1+\phi$	2.3875	–	–

Table 2. Current bounds for algorithms scheduling jobs one by one with possible rejection.

The results are summarized in Table 2. The deterministic lower bounds apply both for algorithms with and without preemption, with the exception of arbitrary m where the lower bound is only 2 with preemption.

4.5 Different speeds

For related machines, a simple doubling strategy leads to a constant competitive ratio [2]. We guess an estimate on the makespan, and schedule each job on the slowest machine such that the current makespan does not exceed the estimate; if this fail we double the estimate and continue. The competitive ratio can be improved by using more sophisticated techniques instead of doubling, but its precise value is not known, see [5] for more references.

For the restricted assignment the optimal competitive ratio is $\Theta(\log m)$ both for deterministic and randomized algorithms [6]. For unrelated machines with no restriction it is also possible to obtain $O(\log m)$-competitive deterministic algorithm [2, 60]. By the previous lower bound this is optimal, too.

It is interesting that both for related and unrelated machines the optimal algorithms are asymptotically better than List Scheduling. Here List Scheduling is modified so that the next job is always scheduled on that machine on which it will finish earliest (for the case of identical speed this is clearly equivalent to the more usual formulation that the next job is scheduled on the machine with the smallest load). For unrelated machines the competitive ratio of List Scheduling is exactly n [2]. For related machines the competitive ratio of List Scheduling is asymptotically $\Theta(\log m)$ [22, 2] (the lower and upper bounds, respectively). The exact competitive ratio for $m = 2$ is ϕ and for $3 \leq m \leq 6$ it is equal to $1 + \sqrt{(m-1)/2}$ [22]; moreover for $m = 2, 3$ it can be checked easily that there is no better deterministic algorithm.

For two machines we are able to analyze the situation further [30]. Suppose that the speeds of the two machines are 1 and $s \geq 1$. It is easy to see that List Scheduling is the best deterministic on-line algorithm for any choice of s. For $s \leq \phi$ the competitive ratio is $1 + s/(s+1)$, increasing from 3/2 to ϕ. For

$s \geq \phi$ the competitive ratio is $1 + 1/s$, decreasing from ϕ to 1; this is the same as for the algorithm which puts all jobs on the faster machine. It turns out that this is also the best possible randomized algorithm for $s \geq 2$. On the other hand, for any $s < 2$ randomized algorithms are better than deterministic ones. If we consider deterministic preemptive scheduling, the competitive ratio is better than for non-preemptive randomized scheduling for any $s > 1$, moreover, it is also always better than for the identical machines ($s = 1$), in contrast without preemption the worst competitive ratio (both deterministic and randomized) is achieved for some $s > 1$.

4.6 Shop scheduling [21]

On-line shop scheduling was so far considered mainly for two machines. This variant of scheduling is somewhat different from all the ones we considered before, since here it may be necessary to introduce idle times on the machines. Hence we need to specify also the time slots for each job, not only the machine(s) on which it runs as before. As a consequence, this variant no longer corresponds to load balancing.

For flow shop and job shop scheduling it turns out that no deterministic algorithm is better than 2-competitive. To design 2-competitive algorithm is trivial: just reserve for each job the needed time on both machines. The same lower bound holds even if preemption is allowed. On the other hand, we do not know if it is possible to extend it to randomized algorithms, with or without preemption.

For open shop the situation is more interesting. If preemption is allowed, the optimal competitive ratio is 4/3. As far as randomization is concerned, the situation is similar as in the basic model with preemption: the 4/3-competitive algorithm is deterministic, while the lower bound holds also for randomized algorithms. Without preemption we have a 1.875-competitive deterministic algorithm and a lower bound of $\phi \approx 1.6180$ for deterministic algorithms. Nothing is known about the power of randomization in this case.

Only a few observations are known about open shop scheduling for $m \geq 3$. Joel Wein observed that for preemptive open shop scheduling there exists a 2-competitive algorithm for arbitrary m. Gerhard Woeginger and the author observed that the randomized lower bound from the basic model which approaches $e/(e-1) \approx 1.5819$ (see Table 1) can be modified to work for open shop, too.

4.7 Minimizing the L_p norm [3]

Here we minimize the L_p norm of the load vector, instead of the makespan, which is equivalent to the L_∞ norm. Of special interest is the Euclidean L_2 norm, the square root of the sum of squares of loads, which has a natural interpretation in load balancing [5, 4]. For L_2 norm, List Scheduling is $\sqrt{4/3}$ competitive, and this is optimal. The performance of List Scheduling is not monotone in the number of machines. It is equal to $\sqrt{4/3}$ only for m divisible by 3, otherwise it is strictly better.

More surprisingly, there exists an algorithm which is for sufficiently large m better than $\sqrt{4/3} - \delta$ for some $\delta > 0$, which means that also the optimal competitive ratio is not monotone in m. For a general p, the same approach leads also to an algorithm better than List Scheduling for large m.

4.8 Minimizing the total completion time [36]

In this variant it is necessary to use idle times, as we have to finish the jobs with short running times first to minimize the total completion time. Even on a single machine it is hard to design a good algorithm and the competitive ratio depends on the number of jobs logarithmically. More precisely, there exists a deterministic $(\log n)^{1+\varepsilon}$-competitive algorithm on a single machine without preemptions, but no $\log n$-competitive algorithm exists even if preemption is allowed.

4.9 Open problems

Randomized algorithms We understand very little about the power of randomization in this on-line paradigm. We know that randomization does not help in most of the variations with preemption. It is open whether randomization helps for shop scheduling and in the model with rejections for large m.

In the basic model, we only know the optimal randomized algorithm for $m = 2$, but for large m we even know no better randomized algorithms than deterministic ones. We conjecture that randomized algorithms are provably better than deterministic ones for every m, and also for m tending to infinity. The following two problems seem to be most interesting.

Open Problem 4.1 *Design an optimal randomized algorithm for* 3*-machine scheduling (in the basic model).*

Open Problem 4.2 *Design a randomized algorithm for arbitrary number of machines which is provably better than any such deterministic algorithm.*

Asymptotic behavior of the competitive ratio In the basic model we do not know optimal deterministic algorithms for any fixed $m > 3$. A major open problem is this.

Open Problem 4.3 *Determine the optimal competitive ratio for deterministic scheduling algorithms in the basic model working for arbitrary* m.

Considering this problem, we usually assume that the competitive ratio increases with increasing m, and hence the hardest case is for m large. However, the following problem is open.

Open Problem 4.4 *Prove that the optimal competitive ratio for* m *is less than or equal to the optimal competitive ratio for* $m + 1$.

Not only that, we even cannot prove that the competitive ratio increases if the number of machines for example doubles (this seems to be a more reasonable goal, as we could avoid some anomalies that occur when the increase of the number of machines is small, cf. [43]). The lack of our knowledge is demonstrated by the fact that we even cannot exclude that the maximal competitive ratio is actually attained for some $m < \infty$. The problems about behavior of the competitive ratio as a function of m are equally open for randomized scheduling.

To compare, for scheduling with rejections we know the limiting value for large m, and we also know that if we increase the number of machines exponentially, the competitive ratio actually increases. On the other hand, if we minimize the L_2 norm instead of the makespan, the maximal competitive ratio is achieved for $m = 3$, and the limit for large m is strictly smaller.

5 Unknown running times

In this on-line paradigm the running time of a job is unknown until the job finishes. This is motivated by the situation of a scheduling algorithm which gets the jobs from different users and has no way of saying how long each job will take. We first focus on minimizing the makespan and later in Section 5.4 we discuss other objective functions.

We are interested in the variants with jobs released over time, either at their release times or according to the precedence constraints, but also in the variant of *batch-style scheduling* where all the jobs are given at time 0. The next general reduction theorem explains why the batch-style algorithms are so important.

Theorem 4 [75]. *Suppose that we have a batch-style σ-competitive algorithm (w.r.t. the makespan). Then there exists a 2σ-competitive algorithm which allows release times.*

Proof. Consider an on-line algorithm that works in phases as follows. In each phase all jobs available at the beginning of the phase are scheduled using the batch-style algorithm. The next phase starts immediately after all these jobs are processed, or, if no jobs are available at that time, at the time the next job is released. This describes a legal algorithm, as at the beginning of each phase no job is running and hence we can use the batch-style algorithm.

Now consider a schedule generated by this algorithm. Let T_3 be the time spent in the last phase, T_2 the time spent in the last but one phase, and T_1 be the time of all the previous phases. We know that $T_2 \leq \sigma T_{\text{opt}}$, as the jobs scheduled during T_2 must be scheduled by the optimal schedule, too, and the batch-style algorithm is σ-competitive. Similarly, $T_1 + T_3 \leq \sigma T_{\text{opt}}$, as the jobs scheduled during T_3 can be scheduled in the optimal schedule only after the time T_1 (if they are released earlier, the on-line algorithm would schedule them in one of the earlier phases), and the batch-style algorithm takes at most σ times longer than the optimal one. The theorem now follows. □

The above reduction is completely satisfactory if we are interested only in the asymptotic behavior of the competitive ratio. However, if the competitive ratio is a constant, we may be interested in a tighter result. In [34] it is proved that for a certain class of algorithms the competitive ratio is increased only by 1, instead of the factor of 2 in the previous theorem; this class of algorithms includes all algorithms that use a greedy approach similar to List Scheduling.

The intuition beyond these reductions is that if the release times are fixed, the optimal algorithm cannot do much before the last release time. In fact, if the on-line algorithm would know which job is the last one, it could wait until its release, then use the batch-style algorithm once, and achieve the competitive ratio of $\sigma + 1$ easily.

In the basic model where the only characteristic of a job is the running time, there is not much we can do if we do not know it. Theorem 1 shows that List Scheduling is $2 - \frac{1}{m}$ competitive also if release times are allowed (hence we do not lose anything in the competitive ratio, unlike in the reductions above), the same bound is true even with precedence constraints. This competitive ratio is tight for deterministic algorithms and almost tight for randomized algorithms, even restricted to the batch-style model.

Theorem 5 [75]. *For batch-style scheduling with unknown running times, no deterministic algorithm is better than $(2 - \frac{1}{m})$-competitive and no randomized algorithm is better than $(2 - O(\frac{1}{\sqrt{m}}))$-competitive.*

Proof. In the deterministic case we use the same instance as in Theorem 1. The algorithm is given $m(m-1)+1$ jobs. All of them have running time 1, except the one scheduled last, which has running time m. We can assign the running times in this way, since the algorithm is deterministic and we can simulate it on the instance where all the jobs have running time 1. After we see which job is scheduled last, we change its running time to m. As the algorithm does not see the running times, it has to behave identically. The last job is scheduled at time at least $m-1$, hence the on-line schedule has makespan at least $2m-1$, while the optimal makespan is m.

For the randomized case we consider the instance with $\sqrt{m}$ jobs of running time m and $m(m - \sqrt{m})$ jobs of running time 1, permuted randomly. As the algorithm does not see the running times, whenever it schedules a job, it in fact chooses one of the remaining ones at random. A standard computation shows that the probability that the last $cm^{3/2}$ jobs contain no long jobs is at most α^c for some constant α. Hence the expected time when the last long job is scheduled is at least $m - O(\sqrt{m})$ and the expected makespan of the on-line algorithm is at least $2m - O(\sqrt{m})$, while the optimal makespan is m. □

5.1 Different speeds

Here we consider both variants, uniformly related machines and unrelated machines. In the case of related machines the speed of each machine is the same for all jobs and given in advance. For unrelated machines the speeds are different for each job. However, we assume that the speeds are known for each job, only the running time is not known (i.e., for each job we know the relative speeds of machines).

If no restarts are allowed, a simple example shows that the best competitive ratio is $\Omega(\sqrt{m})$ [23], even for uniformly related machines. Consider the case of m jobs to be scheduled on m machines, one with speed $\sqrt{m}$ and the rest with speed 1. Whenever a job is scheduled on one of the slow machines, we assign it long running time, and scheduling it on the fast machine is $\sqrt{m}$ times faster; if all jobs are scheduled on the fast machine we assign them the same running time, and lose the same factor. A matching, $O(\sqrt{m})$-competitive, algorithm is known even for unrelated machines [23].

Next we consider the case when restarts are allowed, studied in [75]. In this case we can use a similar general reduction as Theorem 4 to convert an arbitrary off-line algorithm into an on-line algorithm. (We can use either the optimal algorithm, or, if we require polynomial time algorithm, we can use the approximation schemes known for the considered problems to obtain the same asymptotic bounds.) Since we do not know the running time, we guess that all jobs have some chosen running time, then run the appropriate schedule. If any job is not finished in the guessed time, we stop it, double the estimate, and repeat the procedure for all such jobs. This method, together with additional improvements, yields for uniformly related machines an algorithm with competitive ratio $O(\min(\log m, \log R))$, where R is the ratio between the largest and smallest speed. A matching lower bound shows that this is optimal. Also in the restricted assignment case there exists $O(\log m)$-competitive algorithm, but it is not known whether this is tight; in fact the best lower bound is only $2 - \frac{1}{m}$. For unrelated machines similar methods yield an $O(\log n)$-competitive algorithm, where n is the number of jobs. It would be interesting to know if there exists an on-line algorithm with a competitive ratio independent of n.

5.2 Parallel jobs

In this variant each job is characterized by its running time and the number of machines (processors) it requests. We consider two variants, batch-style algorithms and algorithms for instances with precedence constraints. While the running times are unknown, the number of machines a job requests is known as soon as it becomes available. All machines have the same speed and no preemptions or restarts are allowed.

Consider the simplest greedy approach for batch-style algorithms: whenever there are sufficiently many machines idle, we schedule some job on as many machines as it requests. This leads to $(2 - \frac{1}{m})$-competitive algorithm, regardless of the rule by which we choose the job to be scheduled (note that here we have a meaningful choice, as we know how many machines each job requests) [35]. This is optimal by Theorem 5, as the basic model corresponds to the special case when each job requests only one machine. Moreover, this algorithm works even for non-malleable jobs.

If we allow precedence constraints, no reasonable on-line algorithm exists for non-malleable parallel jobs. Consider the following situation. At the beginning there are available m jobs requesting one machine; one of them has running time 1 and all other 0. There is one parallel job requesting all machines and running time 0; this job is dependent on one of the jobs with running time 0. The on-line algorithm cannot distinguish the jobs available at the beginning, so that it may happen that the parallel job can be scheduled only after the job with running time 1 is finished. If we iterate this properly, we obtain a lower bound of m on the competitive ratio; a trivial algorithm which at each time schedules only one job achieves this [33]. This argument works also for randomized algorithms, and gives a lower bound of $m/2$ in this case [72].

Hence we turn our attention to ideally malleable jobs. It turns out that the optimal competitive ratio for deterministic algorithms is $1 + \phi \approx 2.6180$, and it is achieved by the following simple algorithm [33].

Algorithm Parallel

(i) If an available job requests p machines and p machines are idle, schedule this job on p machines.

(ii) If less than m/ϕ machines are busy and some job is available, schedule it on all available machines.

Note that this algorithm uses the fact that jobs are malleable only for large jobs. Accordingly, if there is an upper bound on the number of machines a job can use, we can get better algorithms and also algorithms for non-malleable jobs. The tight tradeoffs are given in [33]. It is also interesting that this result improves on the best previously known off-line algorithm, which only achieves an approximation ratio 3 [82].

5.3 Parallel jobs on specific networks

Here we consider a similar model as in the last section with an additional restriction. We require that each parallel job is scheduled on some subset of machines with a specific structure, not an arbitrary subset as before. This is motivated by the situation in which parallel jobs are designed for specific multiprocessor systems and may use the specific properties of the network connecting the individual processors. We consider three topologies of this network. If the network is a hypercube, each parallel job can only be scheduled on a subhypercube of the network (in particular the number of processors a job requests must be a power of two). If the network is a linear array, each job must be scheduled on a contiguous segment of this line. If the network is a two-dimensional mesh, a job must be scheduled on a rectangle of given dimensions. The previous case with no restriction on the set of machines may be viewed as the case of PRAM or a complete graph, where every two machines are directly connected, and therefore there is no preference among the subsets.

Table 3 summarizes the results in this model. The results for deterministic batch-style algorithms are from [35], the results for deterministic algorithm with precedence constraints from [33], and the results on randomized algorithms are from [72, 73].

Network	Batch-style		With precedence constraints	
	Deterministic	Randomized	Deterministic	Randomized
Hypercube	$2 - \frac{1}{m}$	$\leq 2 - \frac{1}{m}$	$O(\frac{\log m}{\log \log m})$	$O(\frac{\log m}{\log \log m})$
Linear array	≤ 2.5	≤ 2.5	$\Theta(\frac{\log m}{\log \log m})$	$\Theta(\frac{\log m}{\log \log m})$
Two-dimensional mesh	$O(\sqrt{\log \log m})$	$O(1)$	$O((\frac{\log m}{\log \log m})^2)$	$O((\frac{\log m}{\log \log m})^2)$

Table 3. Summary of results for scheduling parallel jobs with unknown running times on specific networks.

For batch-style algorithms we have seen in the previous section that with no restriction given by the network the simple greedy approach works. This is no longer true for specific networks; the problem is that a few jobs using one machine can make the whole system unusable for larger jobs. If we allow preemptions or restarts, we can solve this easily by rearranging the jobs into some compact area, but if it is impossible to stop a job, the situation is more difficult. It is always essential to sort the jobs according to their sizes. If we then schedule the jobs greedily from the largest ones (i.e., those requesting most machines), we get the optimal batch-style algorithm for hypercubes. For linear array this leads to 3-competitive algorithm, the 2.5-competitive algorithm needs more careful placement of the jobs.

For two-dimensional mesh the situation is most interesting. It is no longer possible to start from the largest jobs, as for example 10×10 and 5×20 meshes are not comparable. Instead, we divide the jobs into $(\log m)^2$ classes so that the jobs in each class require meshes of similar sizes, and deal with each of them separately. The optimal deterministic algorithm always schedules all of them at once in equal partitions of the mesh, and repeats this procedure several times. Interestingly, any greedy approach that tries from the beginning to use the whole mesh fails and leads to an algorithm whose competitive ratio is the square of the optimal one; to achieve the optimal results we have to start by using only a small portion of the mesh.

In the case of two-dimensional meshes we know that randomization decreases the competitive ratio from non-constant one to a constant. The basic idea is to sample the running times in each class of jobs, and then to schedule each class in an area of the mesh proportional to the estimate of the total work (running time times the number of machines requested) in that class (unlike the deterministic case where we use the same area for each class). To make this work, it is essential to bound the probability that some estimate is wrong by a constant. Since the number of classes is non-constant, this requires a trick: we use the fact that from the classes of jobs requiring less machines we can sample more jobs in the same time and thus we get more accurate estimates.

As in the previous section, all the batch-style algorithms work for non-malleable jobs, but no better algorithms exist even for malleable jobs. Again, with precedence constraints, we always need to use malleable jobs to obtain non-trivial upper bounds.

Even then in the presence of precedence constraints we cannot use the same ideas as for the batch-style algorithms. Even if we process at the beginning all available large jobs, we cannot exclude that later on more of them become available. Thus the best algorithms we can design simply set aside groups of machines for each size of the jobs. For example, for linear array we divide the line of machines into $\log m/\log\log m$ segments, divide the jobs according to the sizes so that in each class the number of requested machines differs by at most a factor of $\log m/\log\log m$, and schedule each class in its segment greedily, using malleability if necessary. Whenever some job is available and not running, one of the segments is fully used, and no job is slowed down by a larger factor than $\log m/\log\log m$ due to malleability; this together gives the upper bound. It is interesting that this simple approach is optimal and even randomization does not help to improve it; the proof of this result is tedious.

Many results in Table 3 are tight, but a few gaps remain. For batch-style algorithms we do not know the exact competitive ratio for the case of linear array. This is interesting, since the case of linear array is an on-line version of the strip packing, where we have to pack given rectangles into a strip of fixed width and as small height as possible. For a long time the best algorithm for this off-line problem gave approximation ratio 2.5 [76], which was matched by the on-line algorithm mentioned in the table. Later, the off-line solution was improved to approximation ratio 2 [77], and it would be interesting to see if this can be achieved by an on-line algorithm, too. The strip packing problem was also studied in the setting equivalent to our paradigm of scheduling jobs one by one as a variant of two-dimensional bin packing, see [38].

For scheduling with precedence constraints on two-dimensional meshes, both deterministic and randomized, there is a gap between the lower bound which follows from the lower bound for linear arrays and is $\Omega(\log m/\log\log m)$ and the upper bound which is the square of the lower bound. For hypercubes there is no non-trivial lower bound, the claim of a tight bound for this case in [33] is incorrect.

5.4 Other objective functions

In this section we consider the competitive ratio w.r.t. the total waiting time and completion time on identical machines. To minimize these objectives off-line, we have to schedule first the jobs with small running times. If there are no preemptions, we clearly cannot do this at all for unknown running times even in batch-style scheduling: consider a sequence of $n-1$ jobs with running time 0 and one job with running time t on a single machine. If we do not know the running time, it may happen that we schedule as the first job the long one (even in the randomized case with probability $1/n$), and the resulting total waiting time is $(n-1)t$, while the optimal schedule has waiting time 0. Thus, we have to use preemption, and even then it is surprising that we can design competitive algorithms at all. We also assume that there is a minimal running time for each job (the above example also shows that this is necessary).

First we consider batch style algorithms for sequential jobs. In turns out that the optimal competitive ratio is obtained by the simple Round Robin algorithm. It cycles through all unfinished jobs and assigns to each of them to one of the machines and time slot of length τ fixed beforehand. This algorithm has competitive ratio 2 [63] even w.r.t. the total waiting time. (More precisely, if τ approaches zero, the competitive ratio approaches 2 from above.) This is optimal even for randomized algorithms, and even if we consider the total completion time instead of waiting time [63].

It is somewhat unsatisfactory that in the previous algorithm the number of preemption is $\Theta(t)$ for a job with running time t. It turns out that for sequential jobs we can increase τ during the Round Robin algorithm so that the values in the successive cycles form a geometric sequence. Then the competitive ratio is the same, approaching 2 from above for small starting τ and a small step of the geometric sequence, and the number of preemption decreases to $\Theta(\log t)$ [63]. This is optimal for deterministic algorithms, as any algorithm with $o(\log t)$ preemptions has competitive ratio at least $\Omega(n)$ [63].

Interestingly, the results for batch-style scheduling can be generalized to parallel jobs. Here we consider only total completion time (note that waiting time is not well defined for malleable parallel jobs). For ideally malleable jobs there exists a 2-competitive deterministic algorithm [25], matching the performance for sequential jobs (consequently, randomization cannot help in this case).

A wide range of types of non-ideally malleable jobs together with various restrictions on the number of preemptions is studied in [29]. It is even possible to obtain algorithm for non-ideally malleable jobs under the restriction that the speedup function is non-decreasing and sublinear, which means that allocating an extra processor cannot decrease the actual processing time and cannot decrease the work done for this job; we can even allow to that this parallelism profile changes over time for each job. Here the simple algorithm which assigns the same number of processors to each unfinished job has competitive ratio at most $2 + \sqrt{3} \approx 3.74$; the number of preemptions is n for each job, but it can be decreased to $\log n$ at the cost of constant factor increase in the competitive ratio [29]. (Note that here the jobs are allowed to change parallelism profile; in other models this would be treated e.g. as a sequence of distinct different jobs and the number of jobs n could increase significantly.) Another type of parallel jobs is studied in [24]. Here each job is represented as a directed graph of sequential (sub)jobs, and the competitive ratio achieved is 4.

For algorithms with release times, we know that there are no good on-line algorithms w.r.t. total flow time even for sequential jobs. The competitive ratio is at least $\Omega(n^{1/3})$ for deterministic algorithms and $\Omega(\log n)$ for randomized algorithms [63].

5.5 Open problems

This on-line paradigm seems to be understood relatively well, including such issues like randomization. Perhaps the most interesting problem concerns the general reduction in Theorem 4.

Open Problem 5.1 *Find a variant of scheduling for which the optimal competitive ratio (w.r.t. the makespan) for algorithms with release times is twice the optimal competitive ratio for batch-style algorithms.*

In particular, this possibility is open for parallel jobs with no specific network, on hypercubes, and on linear arrays, where the best competitive ratios in the presence of release times given by Theorem 4 are 4, 4, and 5, respectively. It would be interesting to improve any of these bounds. Note that if we allow malleable jobs in the case with no specific network, the algorithm Parallel gives the upper bound of $1 + \phi$, which is strictly below the upper bound of 4 obtained by Theorem 4 for this case.

6 Jobs arriving over time

In this paradigm the only feature unknown to the on-line algorithm is the existence of the jobs whose release time did not pass yet. In the results surveyed in this section we will see that from many viewpoints such algorithms can do almost as well as off-line algorithms.

We first consider the objective of minimizing the makespan, and then turn to other objective functions and scheduling of conflicting jobs.

6.1 Minimizing the makespan

Here a batch-style algorithm has full information, and hence it can schedule the jobs optimally. Thus from Theorem 4 we get the following result.

Theorem 6 [75]. *For any variant of on-line scheduling of jobs arriving over time (with all characteristics known), there exists a 2-competitive algorithm w.r.t. the makespan.*

As most of the scheduling variants are NP-hard, algorithms obtained by the previous theorem may not be computationally feasible. However, instead of an optimal algorithm we can use an off-line ρ-approximation algorithm, in which case we obtain a 2ρ-competitive on-line algorithm.

For the basic scheduling problems we can achieve even better results than using this reduction. The optimal, i.e., 1-competitive, on-line algorithms for preemptive scheduling on identical machines is given in [42, 48]. The idea of the algorithm is simple: whenever a new job arrives, we reschedule the unfinished parts of previous jobs and all unscheduled jobs so that they are finished as early as possible. For uniformly related machines with different speeds an optimal on-line algorithm exists if and only if the speeds satisfy $s_{i-1}/s_i \leq s_i/s_{i+1}$, where

s_i is the speed of ith fastest machine [80, 81]. All these algorithms use $\Theta(mn)$ preemptions, and this is actually necessary [80, 81].

For uniformly related machines with arbitrary speeds there exist optimal algorithms that are nearly on-line [66, 58]. This means that at each time we know when the next job will be released, in addition to the running times of already released jobs. Any nearly on-line optimal algorithm can be easily transformed into a $(1+\varepsilon)$-competitive on-line algorithm for arbitrary $\varepsilon > 0$: The on-line algorithm chooses a small $\delta > 0$, changes the instance by introducing new jobs with release dates at each multiple of δ and running time 0 (or sufficiently small) and by delaying each release time of an original job by δ. Now it uses the nearly on-line scheduler on the new instance; it always knows the next release time which is at most δ away. The result is an on-line algorithm which produces a makespan of $T_{\text{opt}} + \delta$, i.e., with only a small additive constant; to achieve a small relative error, choose δ proportional to the running time of the first released job with non-zero running time. Note that this on-line algorithm is 1-competitive if we allow an additive term in the definition of competitiveness, but it is not optimal and not 1-competitive if we allow no additive term. Thus, using the results of [80, 81], for uniformly related machines with certain speed ratios we have a curious situation where $(1+\varepsilon)$-competitive algorithms do exist for any $\varepsilon > 0$, but no optimal on-line algorithm exists.

If we consider scheduling without preemptions, we no longer can get a 1-competitive algorithm, even for identical machines. The best upper bound was obtained for the simple algorithm which always schedules the available job with the longest running time; this algorithm is 1.5-competitive [19, 81]. A lower bound of 1.3473 shows that this is close to the best possible [19, 81].

For open shop scheduling on two machines the greedy algorithm achieves the competitive ratio of 3/2; this is optimal for scheduling without preemptions [20]. With preemption, a 5/4 competitive algorithm exists and this is optimal [20]. Note that the greedy algorithm can also be used in the paradigm with unknown running times; in that case it is optimal even if preemptions are allowed [20].

6.2 Minimizing the total weighted completion time

For minimizing the total completion time, and even total weighted completion time, it is possible to give a similar general theorem as Theorem 6 [47, 45, 14]. More surprisingly, it is possible to design schedules that are close to optimal simultaneously for the total completion time and the makespan [14]. For technical reasons we assume that all running times are at least 1. The algorithm for this general reduction is the following.

Algorithm Greedy-Interval

for $i := 0, 1, \ldots$ **do**

At time $\tau = 2^i$ consider all the jobs released by the time τ and not scheduled yet.

(i) Find a schedule with the optimal makespan for these jobs. If it is shorter than τ, use it starting at time τ (i.e., at time 2τ all the jobs will be finished).

(ii) Otherwise find a schedule with makespan at most τ which schedules the jobs with the largest possible total weight and use it starting at time τ.

Theorem 7 [47, 45, 14]. *For any variant of on-line scheduling of jobs arriving over time, Greedy-Interval is* 4*-competitive w.r.t. the total weighted completion time and simultaneously* 3*-competitive w.r.t. the makespan.*

Proof. First we prove that Greedy-Interval is competitive w.r.t. the total weighted completion time. Fix an optimal schedule. The rule (ii) of the algorithm guarantees that the total weight of the jobs that Greedy-Interval completes by the time 2^{l+1} is at least the total weight of jobs finished by time 2^l in the optimal schedule. Hence the weight of jobs Greedy-Interval finishes by an arbitrary time τ is at least the weight the optimal schedule finishes by time $\tau/4$. The bound now follows as the total completion time can be equivalently expressed as the sum of the weight of unfinished jobs over all times.

Next we prove that Greedy Interval is competitive w.r.t. the makespan. Suppose that the optimal makespan $T_{\rm opt}$ satisfies $2^i \le T_{\rm opt} < 2^{i+1}$. All jobs are released by 2^{i+1}, so in the next iteration Greedy-Interval is able to schedule all jobs and finds the optimal schedule. Hence its makespan is at most $T_{\rm opt} + 2^{i+1} \le 3T_{\rm opt}$. □

For preemptive scheduling on a single machine w.r.t. the total completion time it is easy to construct an optimal schedule on-line by always running the job with shortest remaining processing time. The same rule yields a 2-competitive algorithm on identical machines [64].

In the case of single-machine non-preemptive scheduling it is also possible to get better bounds than in Theorem 7. For minimizing the total completion time 2-competitive algorithms were given [64, 49, 78], moreover this is optimal [49, 78]. (An open question is whether it helps if we allow restarts, a lower bound for deterministic algorithms is 1.112 [81].) For minimizing the total weighted completion time a slight modification of Greedy-Interval yields a competitive ratio of 3 [45]; later this was improved to $1 + \sqrt{2} \approx 2.414$ by α-point scheduling discussed below [40].

As in Theorem 6, it is possible to use an approximation algorithm instead of the infeasible optimal one and obtain accordingly larger competitive ratios. Here the goal of the approximation algorithm is not to schedule all jobs as fast as possible, but it needs to solve a dual problem, namely within a given time, to schedule jobs with as large weight as possible; we then use this algorithm also in the step (i) of Greedy-Interval instead of the optimal makespan schedule. This gives competitive ratio 4ρ w.r.t. both the makespan and the total completion time, if ρ is the approximation ratio of the algorithm we use. Number of results that follows from using such approximation algorithms is described in [47, 45, 14]; for example for minimizing the total weighted completion time on identical machines there exists an $(1+\varepsilon)$-approximation polynomial time algorithm, and hence we obtain $(4+\varepsilon)$-competitive polynomial time algorithm [45].

A general randomization technique can be used to improve upon the deterministic algorithm Greedy-Interval. If we use $\tau = \beta 2^i$ in the algorithm for β chosen uniformly between 1/2 and 1, the competitive ratios will be 2.89 w.r.t. the total completion time and 2.45 w.r.t. the makespan [14]. (Note that since the randomized competitive ratio is actually an expectation, we cannot guarantee that the schedule is actually simultaneously within the given factor of both objective functions. However, this proves e.g. that there always exists a schedule which is within a factor of 2.89 of the optimal total completion time and within a factor of 3 of the optimal makespan.)

One method to obtain deterministic 2-competitive algorithms w.r.t. the total completion time is to take the optimal preemptive schedule (which is easy to compute even on-line) and schedule the jobs in the order of their completion in this auxiliary schedule [64]. This idea led to a generalization which turned to be very useful for off-line approximation algorithms and also for randomized on-line scheduling.

Call an α-point of a job the first time when α fraction of this job is finished. Now schedule the jobs in the order of α-points for some α [47]. (Thus the method of [64] is simply scheduling in the order of 1-points.) After using α-point scheduling for off-line algorithms in [47], it was observed that choosing α randomly, under a suitable distribution and starting from a suitable preemptive schedule that can be computed on-line, leads to new randomized on-line algorithms [15, 40]. These methods generally lead not only to c-competitive algorithms for non-preemptive scheduling, they in fact guarantee that the produced non-preemptive schedule is within the factor c of the optimal preemptive schedule.

In the case of a single machine α-point scheduling leads to a randomized algorithm with competitive ratio w.r.t. the total completion time $e/(e-1) \approx 1.5819$ [15]; this is also optimal [79, 81]. For the total weighted completion time α-point scheduling gives a randomized 2-competitive algorithm for a single machine [40]. Recently, this has been improved to 1.6853-competitive algorithm, using a further modification that the α is chosen randomly not once for the whole schedule but independently for each job [41]. Similar methods can be used also for other problems. If preemption is allowed, a competitive ratio of 4/3 w.r.t. the total weighted completion time for a single machine can be achieved [67]. On parallel identical machines without preemptions a randomized algorithm 2-competitive w.r.t. the total weighted completion time was given in [68].

6.3 Minimizing the total flow time

Minimizing the total flow time is much harder than to minimize the total completion time also with known running times.

Without preemption we have strong lower bound even for a single machine. Clearly, no deterministic algorithm can be better then $n-1$ competitive: consider an instance where one job with running time arrives at time 0 and $n-1$

jobs with running time 0 arrive just after this job was scheduled. Even if the algorithm is randomized, no algorithm is better than $\Omega(\sqrt{n})$ competitive [79, 81], and if a deterministic algorithm is allowed to restart jobs, the lower bound is $\Omega(\sqrt[4]{n})$ [81]. Note that also the off-line problem is hard, it is NP-hard to achieve an approximation ratio $n^{\frac{1}{2}-\varepsilon}$ for any $\varepsilon > 0$ [56].

With preemptions the optimal competitive ratio still depends on the number of jobs; it is $\Theta(\log(n/m))$ [61]. Only in the case when the ratio between the maximum and the minimum running time is bounded by P, we can obtain a bound independent of n, namely $\Theta(\log P)$; this is again tight [61].

6.4 Conflicting jobs

The last variant we consider in this section is very different than all the ones considered before. Here some jobs may conflict with each other, in which case we cannot schedule them at the same time. These conflicts are given by a conflict graph, which means that at any time we are allowed to schedule only an independent set in this graph; we assume that on a given node of the conflict graph there may be more jobs which then have to be scheduled one by one (this can be modeled by a clique of these jobs, but this generalization is important if we consider restricted graphs). We assume that we have infinitely many machines, which allows us to focus on the issue of conflicts, and also corresponds to some practical motivation, cf. [50]. We assume that the jobs have integral release times and all the running times are 1; however, we can clearly relax this to arbitrary times if we allow preemptions, as job with running time t is equivalent to t jobs of time 1 on the same node in the conflict graph.

For the makespan, the optimal competitive ratio is 2, even for randomized algorithms, and it is achieved by the following simple algorithm. At any time we find the coloring of the available jobs by the smallest number of colors and schedule one of these colors. This bound follows directly from Theorem 4, even for arbitrary known running times, and it was rediscovered in [63], who also proved the matching lower bound.

If we consider maximal flow time instead of the makespan, some partial results were obtained by [50]. They show that for conflict graphs that are either interval graphs or bipartite graph, there is an on-line algorithm with the maximal response time bounded by $O(v^3 A^2)$, where A is the optimal objective value and v is the number of vertices in the conflict graph. It would be interesting to obtain a competitive ratio which is a function of only v, i.e., a performance guarantee linear in A. The same paper gives a lower bound of $\Omega(v)$ for the competitive ratio of deterministic algorithms which applies even to the interval and bipartite graphs.

6.5 Open problems

Theorems 6 and 7 give very good general algorithms for the makespan and the total weighted completion time. Moreover, for the total completion time we have tight results at least on a single machine.

To our best knowledge, no lower bounds for total weighted completion time are known that are better than the bounds for the unweighted total completion time, and also no lower bounds for total completion time on identical machines are better than the bounds on a single machine.

Open Problem 6.1 *Prove for some scheduling problem that the competitive ratio on identical machines w.r.t. the total weighted completion time is strictly larger than the competitive ratio on a single machine w.r.t. the total completion time.*

Another open problem is to investigate the other objective functions, total flow time and total waiting time even in some restricted cases (as we have seen that to minimize these objectives is hard in general).

7 Relaxed notions of competitiveness

For several variants of scheduling we have seen quite strong negative results. However, it turns out that if we allow the on-line algorithm to use some additional information or slightly more resources than the off-line algorithm, we can sometimes overcome these problems and obtain reasonable algorithms.

7.1 Algorithms with more computational power

If we allow the on-line algorithm to use machines with speed $1 + \varepsilon$, there exists a $(1 + 1/\varepsilon)$-competitive algorithm for minimizing total flow time with preemptions and unknown running times on one machine [52]; in contrast without the additional power we have seen that the competitive ratio has to depend on the number of jobs. For scheduling of jobs arriving over time with known running times several results of this kind are obtained by [65]; they either use $O(\log n)$ machines instead of one in the non-preemptive one-machine version or increase the speed of the machines by a factor of two in the preemptive m-machine version, and in both cases they obtain the optimal sum of flow times. In the second case they also show that increasing the speed by $1+\varepsilon$ for some small ε is not sufficient. The positive results should again be contrasted with the negative results if we do not allow additional resources.

7.2 Algorithms with additional knowledge

Another way to improve the performance of the on-line algorithms is to give them some additional information. If we use List Scheduling on a sequence of jobs with non-increasing running times arriving one by one, the competitive ratio is $4/3 - 1/(3m)$, an improvement from $2 - 1/m$ [44]. For deterministic scheduling of jobs arriving one by one on two machines several such possibilities are considered in [57]. They show that the competitive ratio decreases from $3/2$ to $4/3$ in any of the following three scenarios. First, we know the total running time of all jobs. Second, we have a buffer where we can store one job (if we allow buffer for more jobs, we do not gain either). Third, we are allowed to produce two solutions and choose the better one afterwards (equivalently, this means that we get one bit of a hint in advance). If the optimum is known, the problem is also called bin-stretching (because we know that the jobs fit into some number of bins of some height, and we ask how much we need to "stretch" the bins to fit the jobs on-line), and is studied in [7]. For two machines once again 4/3 is the correct and tight answer and for more machines a 1.625-competitive algorithm is presented.

8 Conclusions

We have seen a variety of on-line scheduling problems. Many of them are understood satisfactorily, but there are also many interesting open problems. Studied scheduling problems differ not only in the setting and numerical results, but also in the techniques used. In this way on-line scheduling illustrates many general aspects of competitive analysis.

Acknowledgements

I am grateful to Bo Chen, Andreas Schulz, David Shmoys, Martin Skutella, Leen Stougie, Arjen Vestjens, Joel Wein, Gerhard Woeginger, and other colleagues for many useful comments, pointers to the literature, and manuscripts. Without them this survey could not possibly cover as many results as it does. This work was partially supported by grant A1019602 of AV ČR.

References

1. S. Albers. Better bounds for online scheduling. In *Proc. of the 29th Ann. ACM Symp. on Theory of Computing*, pages 130–139. ACM, 1997.
2. J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line load balancing with applications to machine scheduling and virtual circuit routing. *J. ACM*, 44(3):486–504, 1997.
3. A. Avidor, Y. Azar, and J. Sgall. Ancient and new algorithms for load balancing in the L_p norm. To appear in Proc. of the 9th Ann. ACM-SIAM Symp. on Discrete Algorithms, 1998.
4. B. Awerbuch, Y. Azar, E. F. Grove, M.-Y. Kao, P. Krishnan, and J. S. Vitter. Load balancing in the l_p norm. In *Proc. of the 36th Ann. IEEE Symp. on Foundations of Computer Sci.*, pages 383–391. IEEE, 1995.
5. Y. Azar. On-line load balancing. To appear in *On-Line Algorithms*, eds. A. Fiat and G. Woeginger, Lecture Notes in Comput. Sci. Springer-Verlag, 1998.
6. Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. *J. of Algorithms*, 18:221–237, 1995.
7. Y. Azar and O. Regev. On-line bin-stretching. Manuscript, 1997.
8. Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. *J. Comput. Syst. Sci.*, 51(3):359–366, 1995.

9. Y. Bartal, H. Karloff, and Y. Rabani. A new lower bound for m-machine scheduling. *Inf. Process. Lett.*, 50:113–116, 1994.
10. Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, and L. Stougie. Multiprocessor scheduling with rejection. In *Proc. of the 7th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 95–103. ACM-SIAM, 1996. To appear in SIAM J. Disc. Math.
11. S. Baruah, G. Koren, B. Mishra, A. Raghunatan, L. Roiser, and D. Sasha. On-line scheduling in the presence of overload. In *Proc. of the 32nd Ann. IEEE Symp. on Foundations of Computer Sci.*, pages 100–110. IEEE, 1991.
12. S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11:2–14, 1994.
13. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis.* Cambridge University Press, 1998.
14. S. Chakrabarti, C. A. Phillips, A. S. Schulz, D. B. Shmoys, C. Stein, and J. Wein. Improved scheduling algorithms for minsum criteria. In *Proc. of the 23th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 1099*, pages 646–657. Springer-Verlag, 1996.
15. C. Chekuri, R. Motwani, B. Natarajan, and C. Stein. Approximation techniques for average completion time scheduling. In *Proc. of the 8th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 609–618. ACM-SIAM, 1997.
16. B. Chen, A. van Vliet, and G. J. Woeginger. A lower bound for randomized on-line scheduling algorithms. *Inf. Process. Lett.*, 51:219–222, 1994.
17. B. Chen, A. van Vliet, and G. J. Woeginger. New lower and upper bounds for on-line scheduling. *Oper. Res. Lett.*, 16:221–230, 1994.
18. B. Chen, A. van Vliet, and G. J. Woeginger. An optimal algorithm for preemptive on-line scheduling. *Oper. Res. Lett.*, 18:127–131, 1995.
19. B. Chen and A. P. A. Vestjens. Scheduling on identical machines: How good is lpt in an on-line setting? Technical Report Memorandum COSOR 96-11, Eindhoven University of Technology, 1996. To appear in Oper. Res. Lett.
20. B. Chen, A. P. A. Vestjens, and G. J. Woeginger. On-line scheduling of two-machine open shops where jobs arrive over time. *J. of Combinatorial Optimization*, 1:355–365, 1997.
21. B. Chen and G. J. Woeginger. A study of on-line scheduling two-stage shops. In D.-Z. Du and P. M. Pardalos, editors, *Minimax and Applications*, pages 97–107. Kluwer Academic Publishers, 1995.
22. Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM J. Comput.*, 9(1):91–103, 1980.
23. E. Davis and J. M. Jaffe. Algorithms for scheduling tasks on unrelated processors. *J. ACM*, 28(4):721–736, 1981.
24. X. Deng and P. Dymond. On multiprocessor system scheduling. In *Proc. of the 7th Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 82–88. ACM, 1996.
25. X. Deng, N. Gu, T. Brecht, and K. Lu. Preemptive scheduling of parallel jobs on multiprocessors. In *Proc. of the 7th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 159–167. ACM-SIAM, 1996.
26. X. Deng and E. Koutsoupias. Competitive implementation of parallel programs. In *Proc. of the 4th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 455–461. ACM-SIAM, 1993.
27. X. Deng, E. Koutsoupias, and P. MacKenzie. Competitive implementation of parallel programs. To appear in Algorithmica, 1997.

28. M. Dertouzos and A. Mok. Multiprocessor on-line scheduling with release dates. *IEEE Transactions on Software Engineering*, 15:1497–1506, 1989.
29. J. Edmonds, D. D. Chinn, T. Brecht, and X. Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. In *Proc. of the 29th Ann. ACM Symp. on Theory of Computing*, pages 120–129. ACM, 1997.
30. L. Epstein, J. Noga, S. S. Seiden, J. Sgall, and G. J. Woeginger. Randomized online scheduling for two related machines. Work in progress, 1997.
31. U. Faigle, W. Kern, and G. Turan. On the performance of on-line algorithms for partition problems. *Acta Cybernetica*, 9:107–119, 1989.
32. U. Faigle and W. M. Nawijn. Note on scheduling intervals on-line. *Discrete Applied Mathematics*, 58:13–17, 1995.
33. A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng. Optimal online scheduling of parallel jobs with dependencies. In *Proc. of the 25th Ann. ACM Symp. on Theory of Computing*, pages 642–651. ACM, 1993. To appear in a special issue of *J. of Combinatorial Optimization* on scheduling.
34. A. Feldmann, B. Maggs, J. Sgall, D. D. Sleator, and A. Tomkins. Competitive analysis of call admission algorithms that allow delay. Technical Report CMU-CS-95-102, Carnegie-Mellon University, Pittsburgh, PA, U.S.A., 1995.
35. A. Feldmann, J. Sgall, and S.-H. Teng. Dynamic scheduling on parallel machines. *Theoretical Comput. Sci.*, 130(1):49–72, 1994.
36. A. Fiat and G. J. Woeginger. On-line scheduling on a single machine: Minimizing the total completion time. Technical Report Woe-04, Department of Mathematics, TU Graz, Graz, Austria, 1997.
37. G. Galambos and G. J. Woeginger. An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling. *SIAM J. Comput.*, 22(2):349–355, 1993.
38. G. Galambos and G. J. Woeginger. On-line bin packing – a restricted survey. *ZOR – Mathematical Methods of Operations Research*, 42:25–45, 1995.
39. M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP–completeness.* Freeman, 1979.
40. M. X. Goemans. Improved approximation algorithms for scheduling with release dates. In *Proc. of the 8th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 591–598. ACM-SIAM, 1997.
41. M. X. Goemans, M. Queyranne, A. S. Schulz, M. Skutella, and Y. Wang. Manuscript, 1997.
42. T. F. Gonzales and D. B. Johnson. A new algorithm for preemptive scheduling of trees. *J. ACM*, 27:287–312, 1980.
43. R. L. Graham. Bounds for certain multiprocessor anomalies. *Bell System Technical J.*, 45:1563–1581, Nov. 1966.
44. R. L. Graham. Bounds on multiprocessor timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429, 1969.
45. L. A. Hall, A. S. Schulz, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line approximation algorithms. *Mathematics of Operations Research*, 22:513–544, 1997.
46. L. A. Hall and D. B. Shmoys. Approximation schemes for constrained scheduling problems. In *Proc. of the 30th Ann. IEEE Symp. on Foundations of Computer Sci.*, pages 134–139. IEEE, 1989.
47. L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line algorithms. In *Proc. of the 7th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 142–151. ACM-SIAM, 1996.

48. K. S. Hong and J. Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41(10):1326–1331, 1992.
49. J. A. Hoogeveen and A. P. A. Vestjens. Optimal on-line algorithms for single-machine scheduling. In *Proc. of the 5th Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 1084*, pages 404–414. Springer-Verlag, 1996.
50. S. Irani and V. Leung. Scheduling with conflicts, and applications to traffic signal control. In *Proc. of the 7th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 85–94. ACM-SIAM, 1996.
51. B. Kalyanasundaram and K. R. Pruhs. Fault-tolerant scheduling. In *Proc. of the 26th Ann. ACM Symp. on Theory of Computing*, pages 115–124. ACM, 1994.
52. B. Kalyanasundaram and K. R. Pruhs. Speed is as powerful as clairvoyance. In *Proc. of the 36th Ann. IEEE Symp. on Foundations of Computer Sci.*, pages 214–221. IEEE, 1995.
53. B. Kalyanasundaram and K. R. Pruhs. Fault-tolerant real-time scheduling. In *Proc. of the 5th Ann. European Symp. on Algorithms, Lecture Notes in Comput. Sci. 1284*, pages 296–307. Springer-Verlag, 1997.
54. D. Karger, C. Stein, and J. Wein. Scheduling algorithms. To appear in *Handbook of Algorithms and Theory of Computation*, M. J. Atallah, editor. CRC Press, 1997.
55. D. R. Karger, S. J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. *J. of Algorithms*, 20:400–430, 1996.
56. H. Keller, T. Tautenhahn, and G. J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *Proc. of the 28th Ann. ACM Symp. on Theory of Computing*, pages 418–426. ACM, 1996. To appear in SIAM J. Comput.
57. H. Kellerer, V. Kotov, M. G. Speranza, and Z. Tuza. Semi on-line algorithms for the partition problem. To appear in Oper. Res. Lett., 1996.
58. J. Labetoulle, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Preemptive scheduling of uniform machines subject to release dates. In W. R. Pulleyblank, editor, *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, 1984.
59. E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S. C. Graves, A. H. G. Rinnooy Kan, and P. Zipkin, editors, *Handbooks in Operations Research and Management Science, Vol. 4: Logistics of Production and Inventory*, pages 445–552. North-Holland, 1993.
60. S. Leonardi and A. Marchetti-Spaccamela. On-line resource management with application to routing and scheduling. In *Proc. of the 22th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 944*, pages 303–314. Springer-Verlag, 1995. To appear in Algorithmica.
61. S. Leonardi and D. Raz. Approximating total flow time with preemption. In *Proc. of the 29th Ann. ACM Symp. on Theory of Computing*, pages 110–119. ACM, 1997.
62. R. J. Lipton and A. Tomkins. Online interval scheduling. In *Proc. of the 5th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 302–305. ACM-SIAM, 1994.
63. R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Comput. Sci.*, 130:17–47, 1994.
64. C. Philips, C. Stein, and J. Wein. Minimizing average completion time in the presence of release dates. In *Proc. of the 4th Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 955*, pages 86–97. Springer-Verlag, 1995. To appear in Math. Programming.

65. C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. In *Proc. of the 29th Ann. ACM Symp. on Theory of Computing*, pages 140–149. ACM, 1997.
66. S. Sahni and Y. Cho. Nearly on line scheduling of a uniform processor system with release times. *SIAM J. Comput.*, 8(2):275–285, 1979.
67. A. S. Schulz and M. Skutella. Scheduling-LPs bear probabilities: Randomized approximations for min-sum criteria. Technical Report 533/1996, Department of Mathematics, Technical University of Berlin, Berlin, Germany, 1996 (revised 1997).
68. A. S. Schulz and M. Skutella. Scheduling-LPs bear probabilities. In *Proc. of the 5th Ann. European Symp. on Algorithms, Lecture Notes in Comput. Sci. 1284*, pages 416–429. Springer-Verlag, 1997.
69. S. S. Seiden. More multiprocessor scheduling with rejection. Manuscript, 1997.
70. S. S. Seiden. *Randomization in Online Computation*. PhD thesis, University of California, Irvine, CA, U.S.A., 1997.
71. S. S. Seiden. A randomized algorithm for that ancient scheduling problem. In *Proc. of the 5th Workshop on Algorithms and Data Structures, Lecture Notes in Comput. Sci. 1272*, pages 210–223. Springer-Verlag, 1997.
72. J. Sgall. *On-Line Scheduling on Parallel Machines*. PhD thesis, Technical Report CMU-CS-94-144, Carnegie-Mellon University, Pittsburgh, PA, U.S.A., 1994.
73. J. Sgall. Randomized on-line scheduling of parallel jobs. *J. of Algorithms*, 21:149–175, 1996.
74. J. Sgall. A lower bound for randomized on-line multiprocessor scheduling. *Inf. Process. Lett.*, 63(1):51–55, 1997.
75. D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on-line. *SIAM J. Comput.*, 24:1313–1331, 1995.
76. D. D. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Inf. Process. Lett.*, 10:37–40, 1980.
77. A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM J. Comput.*, 26(2):401–409, 1997.
78. L. Stougie. Personal communication, 1995.
79. L. Stougie and A. P. A. Vestjens. Randomized on-line scheduling: How low can't you go? Manuscript, 1997.
80. A. P. A. Vestjens. Scheduling uniform machines on-line requires nondecreasing speed ratios. Technical Report Memorandum COSOR 94-35, Eindhoven University of Technology, 1994. To appear in Math. Programming.
81. A. P. A. Vestjens. *On-line Machine Scheduling*. PhD thesis, Eindhoven University of Technology, The Netherlands, 1997.
82. Q. Wang and K. H. Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM J. Comput.*, 21(2):281–294, 1992.
83. G. J. Woeginger. On-line scheduling of jobs with fixed start and end times. *Theoretical Comput. Sci.*, 130:5–16, 1994.

This article was processed using the LaTeX macro package with LLNCS style