Online Scheduling

Kirk Pruhs * Jiří Sgall † Eric Torng ‡

September 29, 2003

1 Introduction

In this chapter, we summarize research efforts on several different problems that fall under the rubric of online scheduling. In online scheduling, the scheduler receives jobs that arrive over time, and generally must schedule the jobs without any knowledge of the future. The lack of knowledge of the future generally precludes the scheduler from guaranteeing optimal schedules. Thus much research has been focused on finding scheduling algorithms that guarantee schedules that are in some way not too far from optimal.

We focus on problems that arise within the ubiquitous client-server setting. In a client-server system, there are many clients and one server (or a perhaps a few servers). Clients submit requests for service to the server(s) over time. In the language of scheduling, a server is a processor, and a request is a job. Applications that motivate the research we survey include multiuser operating systems such as Unix and Windows, web servers, database servers, name servers, and load balancers sitting in front of server farms.

The area of online scheduling is much too large for a chapter sized unabridged survey. Our goal is to highlight the critical ideas and techniques that have driven much of the recent research and to focus attention on the open problems that appear to be most interesting.

1.1 Online Paradigms and Notation

The idea behind an online algorithm is that the algorithm does not have access to the entire input instance as it makes its decisions. For a thorough introduction to online algorithms that extends beyond online scheduling algorithms, please refer to the book by Borodin and El Yaniv [30] and the collection of surveys [56]. In scheduling, we model a range of different environments which differ in the way the information is released. These are discussed below, as well as some additional notation specific to online problems that we use to supplement the standard three-field notation introduced in Chapter 1.

In the *online-time* paradigm, the scheduler must decide at each time t which job to run at time t. Problems within the *online-time* model typically have release dates, and the scheduler is not aware of the existence of a job until its release date. Once a job is released, we assume that the scheduler learns the processing time of a job. For example, a web server serving static documents might reasonably be modeled by the *online-time* model since the web server can know the size of the requested file. In contrast, in the *online-time-nclv* model, the scheduler is given no information about the processing time of a job at its release date. For example, the process scheduling component of an operating system is better modeled by the *online-time-nclv* model than the *online-time* model since the operating system typically will not know the execution time of a process. This lack of knowledge of the processing time is called nonclairvoyance.

*Computer Science Department. University of Pittsburgh. Pittsburgh, PA 15260 USA. E-mail `kirk@cs.pitt.edu`. Supported in part by NSF grant CCR-0098752, NSF grant ANI-0123705, NSF grant ANI-0325353, and a grant from the United States Air Force.

†Mathematical Institute, AS CR, Žitná 25, CZ-11567 Praha 1, Czech Republic. E-mail `sgall@math.cas.cz`. Partially supported by Institute for Theoretical Computer Science, Prague (project LN00A056 of MŠMT ČR), grant 201/01/1195 of GA ČR, and cooperative grant KONTAKT-ME476/CCR-9988360-001 from MŠMT ČR and NSF.

‡Department of Computer Science and Engineering. Michigan State University. East Lansing, MI 48824 USA. E-mail `torng@msu.edu`. Supported in part by NSF grant CCR-9701679, NSF grant CCR-0105283, and NSF grant EIA-0219229.

If preemption is not allowed for problems in either the *online-time* or *online-time-nclv* model, and jobs can have arbitrary processing time, then there is usually a trivial example that shows that any online scheduler will produce schedules that are far from optimal. This is why server systems, such as operating systems and web servers, generally allow preemption. Thus most research in online scheduling assumes preemption unless all jobs have similar processing times (as could be the case for a name server, for example). In the online setting, there is another possibility, which is meaningless for offline algorithms. Namely, a running job can be stopped and later *restarted* from the beginning on the same or different machine(s). Thus in order to finish, a job has to be assigned to the same machine(s) for its whole running time without an interruption; in the offline case this is equivalent to non-preemptive scheduling as the unfinished parts of a job can simply be removed from the schedule. This possibility will be denoted by *pmtn-restart* in the middle (job) field of the three-field notation.

In the *online-list* paradigm, the jobs are ordered in a list/sequence. As soon as the job is presented, we know all its characteristics, including the processing time. The job has to be assigned to some machine and time slots (consistent with the restrictions of the given problem) before the next job is seen. The scheduling algorithm cannot change this assignment once it has been made. In the *online-list* model, in contrast to the *online-time* and *online-time-nclv* models, the time between when jobs are assigned is irrelevant or meaningless. The *online-list* model might be an appropriate model for a load balancer sitting in front of a server farm.

The notation *online-time*, *online-time-nclv* or *online-list* will be included in the job field of the three-field notation. So, for example, $1|\textit{online-time}, r_j, pmtn|\sum F_j$ represents the problem of minimizing total flow time on identical machines in the *online-time* model with preemption, which models the problem faced by a web server. And $P|\textit{online-list}|C_{\max}$ is the problem of minimizing the makespan on identical machines when jobs are presented one by one.

1.2 Competitive Analysis

Given that an online algorithm has only partial knowledge of the input instance, for most problems, no online algorithm can produce an optimal solution for all input instances. Probably the most obvious method for evaluating the worst-case performance of an algorithm is the worst-case relative error between the quality of the computed solution for an instance and the quality of the corresponding optimal solution. For example, this is the standard technique for evaluating polynomial-time approximation algorithms for NP-hard problems. In the context of online algorithms, this method is called *competitive analysis* [73, 99]. Let $f(A, I)$ denote the objective value of the schedule produced by algorithm A on input instance I where A could be an online or offline algorithm and f be an objective value that we are trying to minimize such as makespan or total flow time. We say that an online algorithm A is c-competitive if $f(A, I) \leq c \cdot f(OPT, I) + b$ for any input instance I for a fixed constant b where OPT is the optimal offline scheduling algorithm for this problem. For most of the problems we consider, we can ignore the additive constant b. This follows from the fact that scheduling problems are typically scalable; by scaling all the jobs so that the objective is arbitrarily large, the possible benefit of the additive constant disappears. The competitive ratio of algorithm A, denoted c_A, is the infimum of c such that A is c-competitive.

The goal in any problem is to find an algorithm with a competitive ratio as small as possible. Ideally, this competitive ratio should be a constant independent of any parameter of the input instance such as the number of jobs faced, but we shall see this is not always possible.

1.3 Worst-case Analysis and Other Alternatives

Competitive analysis allows us to prove lower bounds using the so-called adversary method. This means that a malicious omnipotent adversary uses the partial schedule generated by the online algorithm to decide what further jobs should be generated. If the algorithms considered are deterministic, this process can be simulated beforehand, and thus it provides a lower bound on the competitive ratio.

For many scheduling problems, worst case competitive analysis gives quite strong lower bounds. For example, for the problem of $1|\textit{online-time-nclv}, r_j, pmtn|\sum F_j$, the competitive ratio of every deterministic algorithm is $\Omega(n^{1/3})$, see [80]. Consider for the moment the possibility that we have an $O(n^{1/3})$-competitive algorithm A for this problem. In absence of other information, this is positive evidence of the superiority of A to other possible algorithms with higher competitive ratios. However, given the magnitude of the $O(n^{1/3})$ guarantee on relative error, it is probable that an operating system designer would not take this as strong evidence to adopt A. Such situations have led to the development of many alternative techniques for analyzing online algorithms.

Randomized algorithms. One standard alternative is to consider randomized algorithms that make random choices as they construct a schedule. We say that a randomized algorithm A is c-competitive if $E[f(A, I)] \leq c \cdot f(OPT, I)$ for all input instances I where $E[f(A, i)]$ is the expected cost of algorithm A on input instance I. This corresponds to the so-called oblivious adversary in online algorithms terminology [25, 30]. An oblivious adversary has to commit to an input instance a priori without any knowledge of the random events internal to the algorithm. Intuitively, this takes away the 'unfair' power of the adversary to completely predict the behavior of the algorithm. The assumption of an oblivious adversary is appropriate for scheduling problems where the scheduling decisions do not affect future input. Even in situations where the oblivious adversary assumption is not fully justified, such an analysis might still provide new insights. For some online scheduling problems, the use of randomized algorithms dramatically decreases the competitive ratio. For example, the competitive ratio for $1|\textit{online-time-nclv}, r_j, pmtn|\sum F_j$ drops from $\Omega(n^{1/3})$ to $\Theta(\log n)$ when one allows randomized algorithms against an oblivious adversary [21].

The most common technique for proving a lower bound on the competitive ratio for any randomized algorithm against an oblivious adversary is Yao's technique. In Yao's technique you lower bound the expected competitive ratio of any deterministic algorithm on an input distribution of your choosing. Generally this expected lower bound for deterministic algorithms also then lower bounds the competitive ratio for any randomized algorithm against an oblivious adversary. However, there are some cases, particularly for maximization problems, where one needs to be a bit careful in applying this technique. For more information see [30].

Resource augmentation. Another alternative that has proven especially useful in the context of online scheduling is resource augmentation. The recent popularity of resource augmentation analysis of scheduling problems emanates from a paper by Kalyanasundaram and Pruhs [67]. The term *resource augmentation*, and the associated terminology we use, was introduced by Phillips, Stein, Torng and Wein [83]. In this model, we augment the online algorithm with extra resources in the form of faster processors or extra processors. For now, we focus on faster processors as most resource augmentation results utilize faster processors. Let A_s denote an algorithm that works with processors of speed s where $s \geq 1$. We say that an online algorithm A is an s-speed c-competitive algorithm if $f(A_s, I) \leq c \cdot f(OPT_1, I)$ for all input instances I.

Research with resource augmentation has focused on two primary goals. The first focuses on minimizing the speed subject to the constraint that the competitive ratio is $O(1)$. To understand this goal, we first need to understand how client-server systems typically behave. Figure 1(a) depicts "typical" average performance curves for client-server systems. That is, the average performance at loads below capacity is good, and the average performance above capacity is intolerable. So, in some sense, one can specify the performance of such a system by simply giving the value of the capacity of the system. Note that formally defining load (or capacity) is not easy, but load generally reflects the size of jobs and their rate of arrival over time. We next need to understand what it means to have s-speed processors. An alternative interpretation is that the jobs created have processing time p_j/s. That is, we can interpret the load as shrinking by a factor of s. This means that an s-speed c-competitive algorithm A performs at most c times worse than the optimal performance on inputs with s times higher load. Assuming that the optimal performance does correspond to the curves in Figure 1(a) and is either good or intolerable, a modest c times either good or intolerable still gives you quite good or intolerable. So an s-speed c-competitive algorithm should perform reasonably well up to load $1/s$ of the capacity of the system as long as c is of modest size. Thus an ideal resource augmentation result would be to prove an algorithm is $(1+\epsilon)$-speed $O(1)$-competitive.

We call a $(1+\epsilon)$-speed $O(1)$-competitive algorithm *almost fully scalable* since it should perform well up to almost the peak capacity of the system. For many scheduling problems, there are almost fully scalable algorithms even though there are no $O(1)$-competitive algorithms. The intuition behind this is that if a system's load is near its capacity, then the scheduler has no time to recover from even small mistakes. Note many of the strong lower bounds for online scheduling problems utilize input instances where the load is essentially the capacity of the system.

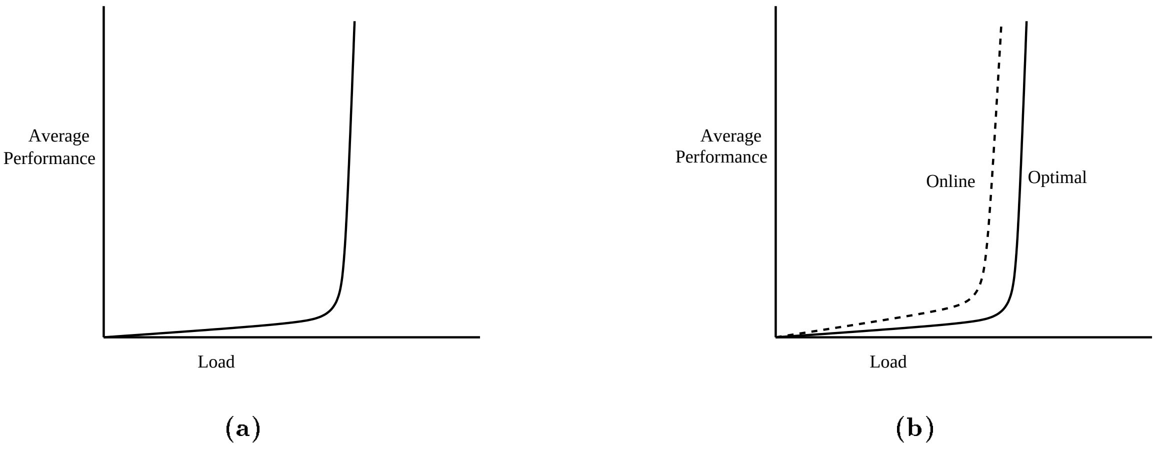

Figure 1: (a) Standard performance curve, and (b) The worst possible performance curve of an s-speed c-competitive online algorithm.

The second goal is to find s-speed 1-competitive algorithms for these problems for as small a value of s as possible. The intuition behind these results is that s represents the tradeoff between extra resources and the partial knowledge that the online algorithm faces. That is, with s times faster processors, the online algorithm is able to overcome its lack of knowledge of the input instance and produce a schedule that is at least as good as the one produced by the optimal offline algorithm.

Semi-online algorithms. One reason for a relatively poor performance of online algorithms is a possibly arbitrarily large variance of job parameters. For example, many greedy algorithms perform badly if they have to handle many jobs of similar size and a few very large jobs. Such inputs may be rare in applications, and we may want to avoid them in the analysis by giving the algorithm some additional knowledge. A semi-online algorithm may know in advance the value of the optimum, the size of the largest job, or the jobs may be required to arrive sorted. Such algorithms are often studied in the *online-list* environment; we shall mention a few examples in that section. While such algorithms may not be implementable in an online environment, the hope is that they will reveal some interesting aspects of the problem not revealed by other analysis. Also, a semi-online algorithm with known optimal makespan can be used to create an online algorithm using the so-called doubling strategy, see Section 3.4. Another related possibility is to make the algorithm general, but study the dependence of the competitive ratio on the variance of some parameter (for example, the ratio of the largest and the smallest processing times).

Average-Case Analysis. Average-case analysis of algorithms is desirable if we have a reasonable approximation of what the input distribution should be. For some client server systems, this is known. For example, traffic for a web server is often modeled using a Poisson distribution for job arrivals and independent identical Zipf distributions for job lengths [31].

While there are many alternative analysis options, we note that worst-case analysis of online algorithms is of fundamental importance. In addition to the standard arguments in its favor (guarantee under any circumstances, etc.), in many online systems, positive feedback appears and thus bad situations may happen more often than one would expect. For example, in many embedded scheduling systems, a request not serviced sufficiently quickly may be reissued.

1.4 History

Many natural heuristics for scheduling are in fact online algorithms; thus some of the early scheduling literature prove bounds on the performance of online algorithms, in current terminology.

The first proof of competitiveness of an online algorithm for a scheduling problem, and perhaps for any problem, was given by Graham in 1966 [62]. It is quite remarkable that this happened at about the same time as Edmonds discovered his famous polynomial time algorithm for matching in graphs, long before notions like polynomial time and NP-hard problems were standard.

Graham [62] studied a simple deterministic greedy algorithm, now commonly called List Scheduling, for $P||C_{\max}$. Each job is scheduled to a machine with currently the smallest load (total size of jobs assigned to it). Graham proved that the job arrival order can change the resulting makespan by a factor of at most $2-1/m$ and that this bound is the best possible. Since this algorithm and a slightly refined analysis works in all three online environments we consider, even with release times and precedence constraints if jobs arrive over time, we get a $(2-1/m)$-competitive online algorithm for $Pm|\textit{online-list}, r_j|C_{\max}$ and $Pm|\textit{online-time-nclv}, \textit{prec}, r_j|C_{\max}$.

Graham [62] even considered the case when the number of machines changes, again giving tight bounds for this algorithm. Today we may view this as a result on resource augmentation. In the follow-up paper [63], Graham shows that the factor of $2-1/m$ decreases to $4/3-1/(3m)$ if we require the jobs to arrive in a sequence sorted according to non-increasing processing times. Thus this is a semi-online algorithm for $Pm|\textit{online-list}|C_{\max}$.

Two other early papers that contain results about online scheduling algorithms are [87, 39]. The first one gives an optimal (1-competitive) algorithm for $P|\textit{online-time}, \textit{pmtn}|C_{\max}$ and explicitly mentions that the algorithm is online. (Actually, the algorithm is not quite online, since it assumes that at any time, the next release time of a job is known. This additional assumption was later removed in [64].) The second paper is, to our best knowledge, the first one that states explicitly a lower bound on the performance ratio of any online algorithm for some scheduling problem, namely the bound of $\Omega(\sqrt{m})$ for $Qm|\textit{online-time-nclv}|C_{\max}$; the paper even suggests that restarts may be helpful, a conjecture later proven to be correct [98].

Around 1990 new results were discovered concerning many variants of online scheduling, both old and new. The development over the last 15 years is the main topic of this chapter.

1.5 Organization of the Chapter

In Section 2, we focus on the problem of scheduling jobs that arrive over time, both clairvoyantly and nonclairvoyantly. We focus our efforts primarily on minimizing total flow time and related objective functions, but we also briefly discuss related results on other objective functions. In Section 3, we focus on the problem of scheduling jobs one by one. As nonclairvoyance does not really make sense in this model, we cover only clairvoyant algorithms. Most research in this area has focused on minimizing the makespan.

1.6 Related Areas

As we stated earlier, we do not provide a complete overview of all research in online scheduling. We admit that our choices are necessarily idiosyncratic.

One particularly interesting and important class of problems that we do not cover is that of real-time scheduling where jobs have deadlines. This topic is covered in Chapter 33 of this book.

We do not cover any of results for minimizing total completion time or total weighted completion time. Some discussion of offline algorithms for these problems can be found in Chapter 13 of this book. Note several of the techniques used for offline algorithms can be adapted to construct online algorithms.

There seem to be only a limited amount of results concerning online shop scheduling, we refer to the survey [34] for some references.

We also do not cover any average-case analysis results in this chapter. More information can be found in part V of this book and in particular in Chapter 38.

2 Jobs that Arrive Over Time

In this section we discuss online scheduling problems in the job model *online-time*.

2.1 Standard Algorithms

Most results in the literature end up analyzing one of a handful of standard algorithms. We now introduce these algorithms, along with their standard abbreviations and some brief comments. The standard clairvoyant algorithms are:

SRPT The algorithm Shortest Remaining Processing Time always runs the job with the least remaining work. It is well known that SRPT is optimal for average flow time on 1 processor.

FIFO The algorithm First In First Out always runs the job with the earliest release time. It is well known that FIFO is optimal for maximum flow time on 1 processor. FIFO is also called First Come First Served in the literature.

SJF The algorithm Shortest Job First always runs the job with the least initial work. For resource augmentation analysis results, SJF is often easier to analyze than SRPT.

HDF The algorithm Highest Density First always runs the job with the highest density, which is the weight of the job divided by the initial work of the job.

The standard nonclairvoyant algorithms are:

RR The algorithm Round Robin devotes an equal amount of processing resources to all jobs. An understanding of RR is important because it is the underlying scheduling algorithm for many technologies. For example, the congestion control protocol within the ubiquitous TCP Internet protocol can be viewed as scheduling connections through a single bottleneck using RR. This algorithm is also called Processor Sharing, or Equi-Partition.

SETF The algorithm Shortest Elapsed Time First devotes all the resources to the job that has been processed the least. In the case of ties, this amounts to RR on the jobs that have been processed the least. While RR perhaps most intuitively captures the notion of fairness, SETF can be seen as fair in an affirmative action sense of fairness.

MLF The algorithm Multi-Level Feedback algorithm can be viewed as mimicking SETF, while keeping the number of preemptions per job to be logarithmic. In most real systems, preemptions take a nontrivial amount of time. In MLF, there are a collection $Q_0, Q_1, \ldots$ of queues. There is a target processing time T_i associated with each queue. Typically, $T_i = 2^{i+1}$, but some results require more slowly growing targets, e.g. $T_i = (1+\epsilon)^{i+1}$. Each job J_j gets processed for $T_i - T_{i-1}$ units of time while in queue Q_i before being promoted to the next queue, Q_{i+1}. MLF maintains the invariant that it is always running the job in the front of the lowest nonempty queue.

It is of natural interest to ask about the scheduling algorithms used by current server technology. Unfortunately, because of the messiness of real software, it is often debatable what the best abstraction of the implemented algorithm is. Let us give a couple of examples. Currently the most commonly used web server software is Apache. The underlying scheduler for the Apache is usually described to be FIFO. But it would probably be more accurate to say that threads are allocated to pending requests on a FIFO basis, and then threads are scheduled using another algorithm. Often it is reported that the underlying process scheduling algorithm for the Unix and Windows NT operating systems is MLF. But in Unix and NT there are only a fixed number of queues, where the lowest priority queue may be scheduled using RR. Thus whether the underlying scheduler is best viewed as MLF or RR depends on the relationship between the job sizes and the largest job quantum.

2.2 Objective Functions

Perhaps the most intuitive measure of Quality of Service (QoS) received by an individual job J_i is the flow time $F_i = C_i - r_i$. The terminology is not standard, and flow time is also called response time, wait time, latency etc. Another intuitive QoS measure for a job J_i is the *stretch* $S_i = (C_i - r_i)/p_i$. If a job has stretch s, then it appears to the client that it received dedicated service from a speed $1/s$ processor. One motivation for considering stretch is that a human user may have some feeling for the work of a job. For example, in the setting of a web server, the user may have some knowledge about the size of the requested document (for example the user may know that video documents are generally larger than text documents) and may be willing to tolerate a larger response time for larger documents.

In order to get a QoS measure for a collection of jobs, one needs to combine the QoS measures for the individual jobs in some way. Almost all the literature uses an l_p norm for some $1 \leq p \leq \infty$. For example, the l_p norm of the flow times is $(\sum_{i=1}^{n} F_i^p)^{1/p}$.

By far the most commonly used QoS measure in the computer systems literature is average flow time, that is, the l_1 norm of flow times. Schedules with good average QoS may still provide very bad QoS to some small number of jobs. In the computer systems literature these jobs are said to be starved. Some scheduling systems such as process scheduling in Unix have mechanisms to try to prevent starvation. To measure how well a schedule avoids starvation, one may use the l_∞ norm. However, an optimal schedule under the l_∞ norm may provide relatively lousy service to the majority of jobs. A common compromise in many settings is the l_p norm, for something like $p = 2$ or $p = 3$. The l_p, $1 < p < \infty$, objective function still considers the average in the sense that it takes into account all values, but because x^p is strictly a convex function of x, the l_p norm more severely penalizes outliers than the standard l_1 norm.

2.3 Notation and Analysis Techniques

If A is an algorithm and I is an input instance, then $A(I)$ will refer to the schedule that A outputs on I. If F is an objective function and S is a schedule, then $F(S)$ is the value of the objective function on that schedule. OPT generally refers to the optimal schedule.

A common difficulty in presenting scheduling results is keeping the notational complexity manageable. One technique to achieve this, which we will adopt, is to drop notation when it is understood. For example, A may refer to the schedule that algorithm A produces on some understood input, as well the value of some understood objective function on that schedule.

The majority of analyses in this area use local competitiveness. Let $A(t)$ be the rate at which the objective function is increasing at time t for the scheduling algorithm A. Generally, $A(t)$ has a nice physical interpretation. For example, if the objective function is total flow time, then $A(t)$ is the number of unfinished jobs at time t.

A local competitiveness argument has the following form. The argument starts by fixing an arbitrary time t. The argument then shows that $A(t) \leq c \cdot OPT(t)$, where $OPT(t)$ is the schedule that minimizes the rate of the increase of the objective function for the specific time t under consideration. So if the objective function was average flow time, $OPT(t)$ would be the schedule that minimizes the number of unfinished jobs at time t. Note that this gives the adversary some advantage since she is not constrained to have any consistency between the schedules $OPT(t)$ and $OPT(t')$ for two different times. But the compensation for the human prover is that the structure of the schedules $OPT(t)$ are often simpler to deal with than globally optimal schedules. It then immediately follows that A is c-competitive in the global sense since

$$A = \int_0^\infty A(t)\, dt \leq \int_0^\infty c \cdot OPT(t)\, dt \leq c \cdot OPT$$

The condition that $A(t) \leq c \cdot OPT(t)$, is called *local c-competitiveness.* Note that in any local c-competitiveness argument, it must be the case that $c \geq 1$. To see this consider an instance consisting of just one job.

Sometimes a local competitiveness argument is not possible because no matter what the online algorithm does, the adversary can get in the lead for at least a short period of time. In general, if a local competitiveness argument is not possible, there is usually a relatively straightforward instance that formally demonstrates this. Also in general, arguments that do not use local competitiveness are more complicated. We will particularly emphasize the analyses in the literature that do not use local competitiveness.

Often our competitive ratios will depend on some parameter. Following standard convention we generally use n to denote the number of jobs, and m to denote the number of processors. We use P_{max} to denote the maximum processing time of any job, P_{min} to denote the minimum processing time of any job, and use P to denote P_{max}/P_{min}.

In the context of resource augmentation results, or other results where there are variable speed processors, the processing time of a job is not fixed. So it no longer makes sense to call p_i processing time. Instead, p_i is usually referred to as the *work* of a job. The time that a job is processed is then its work divided by the average speed at which it is processed.

2.4 Clairvoyant Scheduling to Minimize Average/Maximum Flow/Stretch

In this subsection, we focus on the *online-time* model where the scheduling algorithms know the processing requirements of jobs as soon as they are released. We first cover results for the total flow time and total stretch objective functions that measure average response time for clients. We then discuss results for the max flow and max stretch objective functions that measure server fairness to outlier jobs. In both subsections, we first summarize the results and then highlight some of the key proofs behind these results. We assume that the algorithms are preemptive except when considering maximum flow.

2.4.1 Total Flow Time and Total Stretch

For both single machine scheduling and parallel machine scheduling, SRPT has, within constant factors, the best possible competitive ratio of any online algorithm for minimizing both total flow time and total stretch. On a single machine, SRPT is an optimal algorithm for minimizing total flow time, i.e., $1|online\text{-}time, pmtn, r_j|\sum F_j$. On parallel machines, it is $\Theta(\min(\log P, \log n/m))$-competitive for minimizing total flow time, i.e., $P|online\text{-}time, pmtn, r_j|\sum F_j$, and this is known to be optimal within constant factors [76]. The lower bound applies to randomized as well as deterministic online algorithms. A simpler analysis of SRPT's performance for minimizing flow time is available in [74]. Applying resource augmentation to $P|online\text{-}time, pmtn, r_j|\sum F_j$, Phillips, Stein, Torng, and Wein showed that SRPT is a $(2-1/m)$-speed 1-competitive algorithm for minimizing total flow time [83]. McCullough and Torng improved this result by showing that SRPT is an s-speed $1/s$-competitive for minimizing total flow time for $s \geq 2 - 1/m$ [78]. That is, SRPT "optimally" uses its faster processors. Meanwhile, SRPT is 2-competitive for minimizing total stretch on a single machine, $1|online\text{-}time, pmtn, r_j|\sum S_j$, it is 14-competitive for minimizing total stretch on parallel machines, $P|online\text{-}time, pmtn, r_j|\sum S_j$, and no 1-competitive online algorithm exists for minimizing total stretch on a single machine or parallel machines [81].

While SRPT has essentially the best possible competitive ratio, it utilizes both preemptions and job migrations to achieve its performance. Awerbuch, Azar, Leonardi, and Regev developed an algorithm without job migration (each job is processed on only one machine) that is $(O(\min(\log P, \log n))$-competitive with respect to total flow time [7] and 37-competitive with respect to total stretch [24]. Chekuri, Khanna, and Zhu developed a related algorithm without migration that is $(O(\min(\log P, \log n/m))$-competitive with respect to total flow time and 17.32-competitive with respect to total stretch [33]. If migration is allowed, a modified algorithm is 9.82-competitive for minimizing total stretch [33]. While these algorithms never migrate jobs, they do hold some jobs in a central pool after their release date. Avrahami and Azar developed an algorithm without migration with immediate dispatch (each job is assigned to a machine upon its release) that is $O(\min(\log P, \log n))$-competitive for minimizing total flow time [5]. Chekuri, Khanna, and Kumar have shown that Avrahami and Azar's immediate dispatch algorithm is almost fully scalable for minimizing both total flow time and total stretch [32].

The difference between one machine and parallel machines. We now examine why online algorithms can do well for minimizing total flow time on a single machine but cannot do well for minimizing total flow time on parallel machines. The key observation is that for any time t on a single machine, SRPT has completed as many jobs as any other algorithm.

The following notation will be used throughout this subsection. Let $A(t)$ be both the set of unfinished jobs for algorithm A applied to input instance I at time t as well as the number of jobs in this set. The specific meaning should be clear from context. Furthermore, let $A_j(t)$ be the jth smallest job in $A(t)$ as well as that job's remaining processing time, and let $A^j(t)$ be the jth largest job in $A(t)$ as well as that job's remaining processing time. Again, the specific meaning should be clear from context.

Algorithm	Total Flow Time		Total Stretch	
	Uniprocessor	Parallel Machines	Uniprocessor	Parallel Machines
Best Upper Bound	1	$\Theta(\min(\log P, \log n/m))$	2	9.82
Best Lower Bound	1	$\Theta(\min(\log P, \log n/m))$	1.036	1.093
SRPT	1	$\Theta(\min(\log P, \log n/m))$	2	14
Speed-s SRPT $s \geq 2 - 1/m$	$1/s$	$1/s$	?	?
No migration		$\Theta(\min(\log P, \log n/m))$		17.32
Immediate dispatch		$\Theta(\min(\log P, \log n))$		?
Speed-$(1+\epsilon)$ immediate dispatch		$O(1 + 1/\epsilon)$		$O(1 + 1/\epsilon)$

Table 1: Summary of results for minimizing total flow time and total stretch for single processor and parallel machines.

Lemma 2.1 *Consider any input instance I, and schedule S, and any time t. When we consider a single machine environment, $SRPT(t) \leq S(t)$.*

To prove this, we typically prove a stronger result first.

Lemma 2.2 *Consider any input instance I, any schedule S that never idles the machine unnecessarily, any time t, and any integer $k \geq 0$. When we consider a single machine environment, $\sum_{j=1}^{k} SRPT^j(t) \geq \sum_{j=1}^{k} S^j(t)$.*

Lemma 2.2 implies Lemma 2.1. Consider any schedule S for some input instance I at an arbitrary time t. We first observe that $\sum_{j=1}^{SRPT(t)} SRPT^j(t) = \sum_{j=1}^{S(t)} S^j(t)$ since neither algorithm ever unnecessarily idles the processor. Suppose $S(t) < SRPT(t)$ and let $y = S(t)$. Then it must be the case that $\sum_{j=1}^{y} S^j(t) > SRPT^j(t)$. Since this contradicts Lemma 2.2, it follows that $S(t) \geq SRPT(t)$.

Note that the number of unfinished jobs is the weight of the schedule at any time, and thus SRPT is locally 1-competitive on a single machine. No comparable guarantee can be made in the parallel machine environment, even for randomized algorithms. In particular, no online algorithm can be locally c-competitive for any constant c as demonstrated by the following proof from Leonardi and Raz [76].

Theorem 2.3 *Any randomized online algorithm for $P|\text{online-time}, \text{pmtn}, r_j|\sum F_j$ is $\Omega(\log P)$-competitive. Likewise, any such algorithm is $\Omega(\log(n/m))$-competitive.*

Proof Sketch. We focus on deterministic algorithms but the argument is essentially unchanged for randomized algorithms. The lower bound is obtained by considering a family of input instances composed of repeating phases P_i for $i \geq 0$ followed eventually by a stream of short jobs. The first phase P_0 is organized as follows. At time 0, release a collection of $m/2$ long jobs of size P. At times 0 through $P/2 - 1$, release a collection of m short jobs of size 1. One algorithm A_1 finishes all the short jobs by time $P/2$ by devoting all m machines to the short jobs from time 0 to time $P/2$. A second algorithm A_2 finishes all jobs by time P by devoting $m/2$ of the machines to the long jobs and the other $m/2$ machines to the short jobs.

If an online algorithm A is not locally $(\log P)$-competitive with A_1 at time $P/2$, we introduce m jobs of size 1 for P^2 time units starting at time $P/2$. It can be easily seen that the competitive ratio of any such algorithm A will be $\Omega(\log P)$. Since $n = O(mP^2)$, it follows that the algorithm will also be $\Omega(\log(n/m))$.

Thus, we can focus our attention on algorithms that are locally $(\log P)$-competitive with algorithm A_1 at time $P/2$. Since $A_1(P/2) = m/2$, this means that A must finish all but roughly $m \log P$ short jobs by time $P/2$. This means that at most $m \log P$ time units can be devoted to the long jobs before time $P/2$. If P is sufficiently larger than m, this means that the $m/2$ long jobs will still have remaining processing times of essentially $P/2$ at time P.

Phase P_i for $i \geq 1$ has an initial release time of $R_i = 2P - P/2^{i-1}$ and a "halfway" time of $H_i = R_i + P/2^{i-2}$. The $m/2$ long jobs of length $P/2^i$ are released at time R_i while the short jobs of length 1 are released at times R_i through $H_i - 1$. As before, there is an algorithm A_1 that finishes all the short jobs of phase P_i by time H_i while another algorithm A_2 finishes all the jobs of phase P_i by R_{i+1}. Similar to the analysis of phase P_0, any online algorithm that is not locally $(\log P)$-competitive with algorithm A_1 at time H_i has a competitive ratio of $\Omega(\log P)$ and $\Omega(\log(n/m))$. Thus, we restrict our attention to online algorithms that are locally $(\log P)$-competitive with the algorithm A_1 that finishes all short jobs of phase P_i by time H_i. After phase $P_{\log P - 1}$, such an online algorithm will not have finished any of its long jobs from any phase. Thus, the online algorithm will have $m/2 \log P$ jobs left while OPT will have no jobs left. We now introduce a stream of m jobs of length 1 for P^2 time units, and these algorithms also have competitive ratios of $\Omega(\log P)$ and $\Omega(\log(n/m))$, and the result follows. ■

The structure of SRPT's extra jobs. From the lower bound argument above, we see that unlike in the single machine case, SRPT can idle some of the parallel machines unnecessarily leading to situations where SRPT has many more unfinished jobs than some other schedule for the same input instance. However, Leonardi and Raz were able to show that while SRPT can have arbitrarily more extra jobs, there is a structure to these extra jobs. This leads to an upper bound on SRPT's flow time. We present a brief analysis of SRPT utilizing crucial ideas from [76] and [81].

The concept of volume [76] captures the total remaining work that needs to be completed at any time. For any schedule S, any input instance I, any time t, let the volume $vol(S,t) = \sum_j S_j(t)$ be the sum of remaining processing times of jobs in $S(t)$. Let the volume difference $V'(S,t) = vol(SRPT,t) - vol(S,t)$ be the difference in volume between SRPT's schedule and any other schedule S. We will be interested in focusing on some restricted subsets of jobs when looking at volumes and volume differences. Let $vol(S,t,x)$ be the sum of remaining processing times in $S(t)$ when restricted to jobs of size at most x, and let $V'(S,t,x) = vol(SRPT,t,x) - vol(S,t,x)$.

The following proof from [81] provides a bound on $V'(S,t,x)$.

Lemma 2.4 *For any time t, any input instance I, any real x, and any schedule S, $V'(S,t,x) \leq mx$.*

Proof Sketch. Suppose there are at most m jobs with remaining processing times at most x in $SRPT(t)$. Then clearly $vol(SRPT,t,x) \leq mx$ and the result follows. Thus assume there are more than m jobs with remaining processing times at most x in $SRPT(t)$.

Let t' be the last moment before time t where fewer than m jobs of remaining processing time at most x exist in SRPT's schedule. To simplify the proof description, we also use t' to denote the moment immediately after t' (i.e. the moment where there are now more than m jobs with remaining processing time at most x in $SRPT(t')$). If no such time exists, $t' = 0$. Clearly, during the interval $(t', t]$, SRPT will devote all m machines to jobs with remaining processing times at most x, so we can bound $V'(S,t,x)$ by $V'(S,t',x)$.

We now analyze $V'(S,t',x)$. There were $y \leq m$ jobs of size at most x at time t'. These contribute at most yx work to $vol(SRPT,t',x)$. New jobs with processing times at most x might arrive, but these jobs will contribute to both $vol(SRPT,t',x)$ and $vol(S,t',x)$, so they do not affect $V'(S,t',x)$. Finally, the $m-y$ machines not working on jobs with remaining processing times at most x may create $m-y$ jobs with remaining processing times of x. No other jobs of size at most x can be created at time t' and the result follows. ■

Applying a result from [81], we derive the following characterization of the extra jobs in SRPT's schedule. We first consider the case where S has finished all jobs at time t.

Lemma 2.5 *For any input instance I, for any schedule S, any time t where $S(t) = 0$, and any $i \leq SRPT(t) - 2m$, $SRPT_{2m+i}(t) \geq P_{min}(m/(m-1))^i$ for $i \geq 0$, and the sum of these $2m+i$ smallest jobs in $SRPT(t)$ is at least $mP_{min}(m/(m-1))^i$.*

Proof Sketch. We prove this result by induction on i. We first show the base case for $i = 0$. SRPT can have at most m jobs with remaining processing time less than P_{min} as such a job will never be preempted by a newly arriving job. The next m smallest jobs in SRPT's schedule must have size at least P_{min} and the base case follows. We now assume the result holds for some n and show that it applies for $n + 1$.

By the induction hypothesis, we know that the sum of the $2m + n$ smallest jobs in $SRPT(t) \geq mP_{min}(m/(m-1))^n$. Let y denote the size of the $(2m+n+1)$st smallest job in $SRPT(t)$. From Lemma 2.4, we have that $V'(S, t, y) \leq my$. Since $vol(S, t) = 0$, this means $vol(SRPT, t, y) \leq my$. However, we know that $vol(SRPT, t, y) \geq mP_{min}(m/(m-1))^n + y$. Thus, we derive that $my \geq mP_{min}(m/(m-1))^n + y$ which means that $y = SRPT_{2m+n+1}(t) \geq P_{min}(m/(m-1))^{n+1}$ completing the first part of the induction. Adding this lower bound on $SRPT_{2m+n+1}(t)$ with the lower bound on the sum of the $2m+n$ smallest jobs in $SRPT(t)$ completes the second part of the induction and the result follows. ∎

We can extend this result and eliminate the restriction that $S(t) = 0$ as follows.

Lemma 2.6 *For any input instance I, for any schedule S, any time t, and any $i \leq SRPT(t) - 2m - S(t)$, $SRPT_{2m+i+S(t)}(t) \geq P_{min}(m/(m-1))^i$.*

Proof Sketch. The key observation is that an unfinished job $j \in S(t)$ of size z cannot increase the number of jobs in $SRPT(t)$ by more than one, and this job must have size at least z. Consider for example the job $S_1(t)$ of size z. Despite the existence of job $S_1(t)$, $vol(S, t, y) = 0$ for all $y < z$, and thus $vol(SRPT, t, y) \leq my$. The fact that $vol(S, t, z) = z$ (assuming no other jobs of size z are in $S(t)$) implies $vol(SRPT, t, z) \leq mz + z$ which allows the addition of one job of size at least z to $SRPT(t)$ in addition to the jobs generated by the argument of Lemma 2.6. ∎

With this result, we can now derive the following bound on $SRPT(t)$.

Theorem 2.7 *For any input instance I, any schedule S, and any time t, $SRPT(t) \leq S(t) + m(2 + \ln P)$.*

Proof Sketch. Applying Corollary 2.6, we have that $SRPT(t) \leq S(t) + 2m + \log_{m/(m-1)} P$. Now $(m/(m-1))^m = (1 + 1/(m-1))^m \geq e$ for $m \geq 2$. Thus, $\log_{m/(m-1)} P \leq m \ln P$ and the result follows. ∎

To derive the upper bound on SRPT's flow time, we first observe that the contribution of jobs in $SRPT(t)$ that correspond to jobs in $S(t)$ to SRPT's flow time is at most the total flow time incurred by $S(I)$. We now divide time into two categories: intervals where all m machines are busy and intervals where some machines are idle. SRPT can only have extra jobs during busy times. If we focus only on the active jobs during these busy times, their contribution to SRPT's flow time is at most the sum of processing times of jobs, and this is clearly a lower bound on the optimal flow time. Note there are m active jobs at any time during these busy intervals. Thus, the at most $O(m \log P)$ extra jobs in SRPT's schedule during these busy intervals is at most $O(\log P)$ more than the m active jobs, and it then follows that SRPT is $O(\log P)$-competitive for the problem of minimizing total flow time, $P|\textit{online-time}, pmtn, r_j|\sum F_j$. To prove that SRPT is $O(\log(n/m))$-competitive requires more sophisticated arguments which we omit.

Eliminating migration and immediate dispatch. The key idea in algorithms that eliminate migration is the idea of classifying jobs by size [7, 33, 5]. In [7], jobs are classified as follows: a job j whose remaining processing time is in $[2^k, 2^{k+1})$ is in class k for $-\infty < k < \infty$. Note that jobs change classes as they execute. This class definition reflects the structure of extra jobs in SRPT's schedule first observed in [76].

The algorithm A uses the following data structures to organize the jobs. There is a central pool containing jobs not yet assigned to any machine. With each machine, we associate a stack to hold jobs currently assigned to that machine.

The algorithm A works as follows. Each machine processes the job at the top of its stack. When a new job arrives, the algorithm looks for a machine that is idle or currently processing a job of a higher class than the new job. If it finds one, the new job is pushed into that machine's stack and its processing begins. Otherwise, the job enters the central pool. Finally, if a job is completed on some machine, the algorithm compares the job at the top of the stack of that machine with the minimum class of any job in the pool. If the minimum in the pool is smaller than the class of the job on top of the stack, then any job in the pool of that minimum class is then pushed onto that stack. Using ideas similar to those used in [76], they derive the following result.

Lemma 2.8 *For any input instance I, for any schedule S, and for any time t when all m machines are busy, $A(t) \leq 2S(t) + mO(\log P)$.*

Again, this leads to the result that the algorithm is $O(\log P)$-competitive. With more work, this algorithm can be shown to be $O(\log n)$-competitive, slightly worse than $O(\log(n/m))$-competitive.

New algorithms proposed in [33] achieve the same bounds as SRPT within constant factors for minimizing total flow time by modifying the class definition from [7]. A job is now assigned to class k if its *original* processing time is in the range $[2^k, 2^{k+1})$. Thus, the class of a job does not change as it executes. This simplifies the analysis of their algorithm, particularly when considering total stretch. Furthermore, their simpler analysis allows the optimization of the constant used to define classes (the definitions above use constant 2).

A new algorithm that dispenses with the central pool of unassigned jobs was proposed in [5]. That is, each job is immediately assigned to a machine, and there is no migration of jobs. They show that this algorithm is $O(\min(\log P, \log n))$-competitive for minimizing total flow time on parallel machines. This algorithm uses the class definition of [33]. When a job j of class k arrives, it is assigned to the machine that has been assigned the minimum total processing time of jobs of class k so far. That is, Graham's List Scheduling rule is used to assign jobs to machines within each class of jobs. Note that this assignment rule ignores information such as what is the current load on each machine or which jobs in the specified class have actually been processed or completed at the current time. Each machine then implements the SRPT algorithm which is optimal for scheduling jobs on a single machine to minimize flow time. However, to simplify the analysis, they analyze a modified version of this algorithm that uses SJF on each machine instead.

Here are a few of the key observations in the analysis of this algorithm [5]. The first fact is that the difference in total volume of jobs of any class k assigned to any two machines by any time t is at most 2^{k+1}, the size of the largest job in class k, since they use greedy List Scheduling. This implies that difference in the total volume of work processed by any time t of jobs in class at most k on any two machines is at most 2^{k+2}. Combining these two observations implies that the difference in unfinished work from jobs of class at most k at any time t on any two machines is at most 2^{k+3}. With these facts, [5] are able to apply many of the arguments used in the analysis of other algorithms without migration to prove the flow time bound for their algorithm.

Resource Augmentation Results. With sufficiently faster processors, [83] showed that SRPT will never have extra jobs. Specifically, they extended Graham's analysis of List Scheduling [62] to show that any s-speed algorithm where $s \geq 2 - 1/m$ that never idles a machine when jobs are available always completes as much work by any time t as any 1-speed algorithm on the same input instance. Adding to this the greedy nature of SRPT, their analysis shows that speed-$(2 - 1/m)$ SRPT is locally 1-competitive.

This result has recently been improved to show that SRPT is an s-speed $1/s$-competitive algorithm for $s \geq 2 - 1/m$ [78]. The analysis in [78] uses some new ideas to prove a competitiveness bound smaller than 1. First is the idea that s-speed processors can be approximated by multiplying release dates by a factor of s. In [83, 78], the resulting input instance is called a stretched input instance. The key observation is that an algorithm on a stretched input instance will incur a flow time exactly s times larger than the same algorithm using s-speed processors on the original input instance. Thus, they need only show that SRPT on an s-stretched input instance does as well as the optimal algorithm does on the original input instance to prove the $1/s$ bound for s-speed SRPT. This introduces a complication as they need to compare SRPT on a stretched input instance to the optimal algorithm on the original input instance, and thus jobs are released at different times for the two algorithms. They overcome this difficulty by introducing a proxy algorithm for the original input instance. This proxy algorithm will in some cases produce schedules that are not legal. This is acceptable since the proxy algorithm is used for analysis purposes only. However, they do need to introduce a charging scheme to handle cases when the schedule is not legal. They then show that the proxy algorithm is locally 1-competitive and that the proxy algorithm incurs a flow time on the original input instance that is at least as large as the flow time incurred by SRPT on the s-stretched input instance. This argument does not use local competitiveness but rather a structural relationship between the the proxy schedule and the SRPT schedule on the stretched input instance.

Chekuri, Khanna, and Kumar have shown that the immediate dispatch algorithm of Avrahami and Azar is almost fully scalable for total flow time and total stretch [32]. This result also applies when the algorithm is given extra machines instead of faster machines, and the result extends to show that the algorithm is almost fully scalable for l_p norms of flow and stretch for all $p \geq 1$. Their analysis builds upon Bansal and Pruhs' analysis of SJF and SRPT for minimizing l_p norms of flow and stretch on a single machine [14]. These results are discussed in more detail in Section 2.5.

A few open questions remain regarding resource augmentation and minimizing total flow time. While we now know that there is an almost fully scalable algorithm for minimizing total flow time on parallel machines, no such analysis is known for SRPT.

Open Problem 2.9 *For the problem $P|\text{online-time}, pmtm, r_j|\sum F_j$, is SRPT almost fully scalable?*

Furthermore, from [83], we know that SRPT is at least as good as optimal when given speed-$(2-1/m)$ machines and that no speed-$(22/21-\epsilon)$ 1-competitive algorithm exists for minimizing total flow time on parallel machines for $m \geq 2$ and $\epsilon > 0$.

Open Problem 2.10 *For the problem $P|\text{online-time}, pmtm, r_j|\sum F_j$, what is the minimum speed s such that there exists an s-speed 1-competitive algorithm, and what is the corresponding algorithm?*

Both of these questions can be extended to all l_p norms of flow and stretch.

The difference between total stretch and total flow time. At first glance, it may seem surprising that there exist algorithms with constant approximation factors for total stretch on parallel machines but not total flow time. This discrepancy is explained by considering the structure of extra jobs for SRPT and the fact that the total stretch objective function weights jobs by the inverse of their original processing times. For example, while $SRPT(t) - S(t)$ can be unbounded, there can only be a relatively few extra jobs with small remaining processing times in SRPT's schedule at any time. In particular, the large jobs add a negligible amount to the total weight of jobs at any given moment. This property is exploited more explicitly in the algorithms that use job classifications.

For example, consider the algorithm of [33] and consider the jobs on the stack of any machine of their algorithm. Suppose the job that is currently executing is from class k. The original processing times of the remaining jobs on the stack for that machine are at least $2^{k+1}, 2^{k+2}, 2^{k+3}, \ldots$, and their weights sum to at most $1/2^k$. This is at most twice the weight of the job currently executing and thus the increase in total stretch can be charged to this currently executing job. Handling jobs in the central pool is more complicated and we ignore these details.

On the other hand, we observe that no online algorithm can be optimal for minimizing total stretch on a single machine while there does exist an optimal online algorithm, namely SRPT, for minimizing total flow time on a single machine. The lower bound example below shows that we can create a situation where it is optimal to prioritize one job j_1 over a second job j_2 in some cases while in other cases, it is optimal to prioritize job j_2 over job j_1.

Lemma 2.11 *No online algorithm can be better than 1.036-competitive for the problem of minimizing total stretch on a single machine [81].*

Proof Sketch. Consider an adversary strategy using at most 3 jobs of sizes q, m, and s where $q > m > s$. Under the first scenario, the job of size q is released at time 0, and the job of size m is released at time $q-k$ for some $k \leq m$, and the third job is never released. Under the second scenario, the third job of size s is released at time q. The adversary makes its decision on which scenario to implement based on the online algorithm's decisions up to time q.

The optimal strategy for the first scenario is to run the second job as soon as it arrives. The optimal strategy for the second scenario is to finish the first job first, run the third job as soon as it arrives, and then finish the second job. Clearly no online algorithm can do both. Using a proper choice of q, m, s, and k, the bound of 1.036 follows. ■

More detailed analysis of SRPT for total stretch [81]. The analysis of SRPT for total stretch on a single machine utilizes a matching property between the jobs waiting in SRPT's queue at any time t and the jobs in any other schedule S's queue at time t.

Lemma 2.12 *For any input instance I, for any schedule S, and any time t, and any $k > 1$, $SRPT_k(t) \geq S_{k-1}(t)$.*

Proof Sketch. Suppose this is not true at some time t. Let $k > 1$ be the smallest integer such that the relationship does not hold. Let $b = SRPT_k(t)$. It follows that the number of jobs in $SRPT(t)$ of size at most b is at least k, while the number of jobs in $S(t)$ of size at most b is at most $k-2$. Furthermore, given the definition of k, we have that $SRPT_j(t) \geq S_{j-1}(t)$ for $1 < j < k$. Thus, $vol(SRPT, t, b) - vol(S, t, b) \geq b + SRPT_1(t)$ which means that $V'(S, t, b) > b$. This is a contradiction since Lemma 2.4 implies that $V'(S, t, b) \leq b$, and the result follows. ∎

With this matching property, [81] bound the amount that SRPT's waiting jobs contribute to SRPT's total stretch by the total stretch incurred by any other algorithm. They then observe that the total stretch incurred by SRPT's active job over time is exactly n which is a lower bound on the optimal total stretch, and the factor of 2 result follows.

In the parallel machine case, there is the extra complication that SRPT has extra jobs. However, given the structural property observed earlier, [81] are able to derive a similar mapping of some of SRPT's waiting jobs to at least as small unfinished jobs for schedule S. The unmapped jobs for SRPT then obey the structure observed earlier and their total contribution to total stretch can be bounded by a constant times the optimal total stretch.

2.4.2 Maximum Flow Time and Maximum Stretch

While SRPT and the related algorithms perform well for client jobs on average, these algorithms do have the undesirable property of starving some jobs in order to service most jobs well. For example, consider an input instance on a single machine where a job with processing time 2 is released at time 0 and jobs with processing time 1 are released at unit intervals starting at time 0 and ending at time x. SRPT will always process the jobs with processing time 1 delaying the job with processing time 2 until the end of the long stream of jobs, so its flow time will be $x+3$. An alternative algorithm would schedule the job with processing time 2 first and then schedule the jobs with processing time 1 in order of their arrival. The flow time of the job with processing time 2 will be 2 while the flow time of all jobs with processing time 1 will be 3. As we can make x as large as we desire, this shows that for the $F_{\max}$ or $S_{\max}$ objective functions on a single machine, i.e., $1|\textit{online-time}, \textit{pmtn}, r_j|F_{\max}$ and $1|\textit{online-time}, \textit{pmtn}, r_j|S_{\max}$, SRPT is $\Omega(n)$-competitive.

Different algorithms are needed to provide good guarantees for maximum flow and maximum stretch. The best results known for these objective functions come from Bender, Chakrabarti, and Muthukrishnan [26] and Bender, Muthukrishnan, and Rajaraman [27]. For maximum flow, [26] show that FIFO is $(3 - 2/m)$-competitive for $P|\textit{online-time}, \textit{pmtn}, r_j|F_{\max}$ and provide a lower bound of $4/3$ for any non-preemptive algorithm for $m \geq 2$. (The paper claims a lower bound of $3/2$, but the proof seems to work only for a $4/3$ lower bound.) The maximum stretch objective function turns out to be harder to minimize than maximum flow. This stands as an interesting contrast to the case of total stretch and total flow time where, in the parallel machine environment, there exist constant competitive algorithms for total stretch but no constant competitive algorithms for total flow time, even with preemption. For maximum stretch on a single machine, $1|\textit{online-time}, \textit{pmtn}, r_j|S_{\max}$, [26] provides an algorithm that is $O(P^{1/2})$-competitive that is based on the earliest deadline first (EDF) real-time scheduling algorithm, and they provide a lower bound of $\Omega(P^{1/3})$ on the competitive ratio of any online algorithm for maximum stretch. A simpler and more efficient algorithm to achieve the $O(P^{1/2})$ bound is given in [27]. No results are known for maximum stretch on parallel machines, $P|\textit{online-time}, \textit{pmtn}, r_j|S_{\max}$. These results are summarized in Table 2.

Maximum flow. The fact that FIFO, a non-preemptive algorithm, is constant competitive for minimizing maximum flow shows how this problem is quite different than that of minimizing total flow time. We provide below a proof that FIFO is optimal for the single machine environment.

Theorem 2.13 *FIFO is an optimal algorithm for minimizing maximum flow time on a single machine, $1|\textit{online-time}, \textit{pmtn}, r_j|F_{\max}$ [26].*

Proof Sketch. Without loss of generality, we consider only input instances I such that there is no idle time in $FIFO(I)$. Consider any such input instance I and a job j such that F_j is maximized in $FIFO(I)$.

Algorithm	Max Flow Time		Max Stretch	
	Uniprocessor	Parallel Machines	Uniprocessor	Parallel Machines
Best Upper Bound	1	$3 - 2/m$	$O(P^{1/2})$	?
Non-preemptive Lower Bound	1	4/3	$\Omega(P)$	$\Omega(P)$
Preemptive Lower Bound	1	?	$\Omega(P^{1/3})$	?

Table 2: Summary of results for minimizing max flow time and max stretch for single processor and parallel machines.

This means that from time r_j to time C_j, FIFO is working only on jobs that had release times at most r_j. Since FIFO is not idle prior to r_j, it is not possible to finish all the jobs released prior to r_j plus job j any earlier than C_j. Thus, in any schedule for input instance I, some job released at time no later than r_j must complete no earlier than C_j, and the result follows. ∎

When we consider parallel machines, it is no longer true that FIFO is not idle prior to the release time of the job with maximum flow time in $FIFO(I)$. Thus, FIFO is not optimal for the parallel machine environment, but it is still constant competitive.

The only lower bound known for this problem is 4/3 for $m = 2$ [26] for non-preemptive algorithms. At time 0, two jobs with processing time 3 are released. If the algorithm starts both jobs by time 1, a job with processing time 6 is released at time 1; otherwise, no more jobs are released. In the first case, the optimal $F_{\max}$ is 6 while the online algorithm's $F_{\max}$ is at least 8. In the second case, the optimal $F_{\max}$ is 3 while the online algorithm's $F_{\max}$ is at least 4.

Maximum stretch.

Theorem 2.14 *On a single machine, no preemptive online algorithm is $P^{1/3}/2$-competitive for minimizing maximum stretch,* $1|\text{online-time}, pmtn, r_j|S_{\max}$ *[26].*

Proof Sketch. Consider the following input instance. Two jobs with length P are released at time 0. Meanwhile, jobs of size $k = P^{2/3} - 1$ are released at times $2P - k, 2P, \ldots, P^{4/3} - k$. To simplify the proof, we assume that $y = P^{4/3} - 2P$ is an integral multiple of k.

An optimal schedule for minimizing maximum stretch for this input is FIFO which results in a maximum stretch of 2. Thus, for an online algorithm to be $P^{1/3}/2$-competitive, the first two jobs must be completed by time $P^{4/3}$ giving them a stretch of $P^{1/3}$. This means one of the length k jobs cannot complete before $P^{4/3} + k$.

Now suppose that jobs of length 1 arrive every unit of time starting at time $P^{4/3}$ and ending at time $2P^{4/3} - k - 1$. Either one of the length 1 jobs finishes at essentially $2P^{4/3}$ or one of the length k jobs finishes then. In the first case, the maximum stretch will then be at least $k + 1 = P^{2/3}/2$ while in the second case, the maximum stretch will be essentially $P^{4/3}/k > P^{2/3}/2$. Meanwhile, the optimal algorithm schedules the jobs of size 1 and size k as they arrive and finishes one of the jobs of size P at time $2P^{4/3}$. The result then follows. ∎

One algorithm for minimizing maximum stretch uses ideas from real-time scheduling. Suppose the online algorithm somehow knew in advance what the maximum stretch S^* for the input instance would be. It could then treat each job j as if it had a deadline of $r_j + S^* p_j$ and use algorithms from real-time scheduling to attempt to meet all deadlines. One such online algorithm is Earliest Deadline First (EDF) that prioritizes available jobs by their deadlines breaking ties arbitrarily. EDF is known to legally complete all jobs by their deadlines on a single machine if it is possible to do so. By the definition of maximum stretch, it clearly is possible to schedule all jobs such that they end by $r_j + S^* p_j$. Thus, EDF armed with knowledge of the maximum stretch of the input instance is an optimal online algorithm.

Unfortunately, the online algorithm cannot possibly know the maximum stretch ahead of time. Instead, the best that any online algorithm can do is compute what the maximum stretch of an input instance would be if no more jobs arrive. This algorithm, stretch-so-far [26] has a further refinement of overestimating the maximum stretch computed so far by setting a job's deadline to be $r_j + \alpha S^* p_j$ where $\alpha \geq 1$. Choosing an appropriate value of α is critical to minimizing maximum stretch. Also note that the deadlines will change as S^* is refined. Stretch-so-far with $\alpha = 1$ is P-competitive for this problem [26]. If α is instead chosen to be $O(P^{1/2})$, the algorithm is then $O(P^{1/2})$-competitive. Note that the $P^{1/2}$ used here is based on the jobs seen so far, so this is an online algorithm. Constant competitive algorithms exist if there are only two distinct job lengths [26].

While stretch-so-far achieves a competitive ratio of $O(P^{1/2})$, it has the disadvantage of requiring $\Omega(n^2)$ processing per job arrival. In particular, the optimal maximum stretch must be recalculated on each job arrival requiring the algorithm to remember all jobs seen so far at any point in time during its execution. A simpler greedy strategy that achieves the same competitive ratio is proposed in [27]. Suppose that $P_{min} = 1$ and $P = P_{max}$ is known to the online algorithm. Their algorithm computes a pseudostretch for each available job at any time t that is $(t - r_j)/P^{1/2}$ if $1 \leq p_j \leq P^{1/2}$ and is $(t - r_j)/P$ if $P^{1/2} < p_j \leq P$. That is, they replace the p_j in the denominator by $P^{1/2}$ if the job is small and P if the job is larger. This algorithm is $O(P^{1/2})$-competitive. However, it assumes a priori knowledge of P_{min} and P_{max}. To make this more online, they assume that the algorithm knows the minimum job size 1 in advance, and they use the largest job seen so far as their estimate for P, recalculating as needed when new jobs arrive.

2.5 l_p Norms of Flow and Stretch

In this section, we consider the problems of minimizing the l_p norms of flow times and stretch. We discuss both clairvoyant and nonclairvoyant algorithms. Recall that the motivation for considering the l_p, $1 < p < \infty$, norms of flow and stretch was that they represent some compromise between optimizing for the worst case QoS and the average QoS. Although most of the results we give below also hold when $p = 1$ and when $p = \infty$. The these results generalize both the average and maximal flow and stretch, since the total flow time or stretch is then the l_1 norm while maximum flow time or stretch is the l_∞ norm.

Let us initially focus on one machine. The study of l_p norms of flow and stretch was initiated by Bansal and Pruhs [14]. Bansal and Pruhs [14] show that are no $n^{o(1)}$-competitive online clairvoyant scheduling algorithms for any l_p norm, $1 < p < \infty$ of either flow or stretch. This is a bit surprising, at least for flow time, as there are optimal online algorithms, SRPT and FIFO, for the l_1 and l_∞ norms of flow time.

Theorem 2.15 *For the problems $1|\text{online-time}, \text{pmtn}, r_j|(\sum F_j^p)^{1/p}$ and $1|\text{online-time}, \text{pmtn}, r_j|(\sum S_j^p)^{1/p}$, $1 < p < \infty$, the competitive ratio of any randomized algorithm A against an oblivious adversary is $n^{\Omega(1)}$.*

Proof Sketch. We only give lower bound proofs for flow norms, and only for deterministic algorithms. It is easy to extend the lower bound to randomized algorithms using Yao's technique. The input is parameterized by integers L, $\alpha = (p+1)/(p-1)$, and $\beta = 2$. A long job of size L arrives at time 0. From 0 to time until time $L^\alpha - 1$ a job of size 1 arrives every unit of time.

In the case that A does not finish the long job by time L^α then this is the whole input. Then $F^p(A)$ is at least the flow of the long job, which is at least $L^{\alpha p}$. In this case the adversary could first process the long job and then process the unit jobs. Hence, $F^p(Opt) = O(L^p + L^\alpha \cdot L^p) = O(L^{\alpha+p})$. The competitive ratio is then $\Omega(L^{\alpha p - \alpha - p})$, which is $\Omega(L)$ by our choice of α.

Now consider the case that A finishes the long job by time L^α. In this case $L^{\alpha+\beta}$ short jobs of length $1/L^\beta$ arrive every $1/L^\beta$ time units from time L^α until $2L^\alpha - 1/L^\beta$. One strategy for the adversary is to finish all jobs, except for the long job, when they are released. Then $F^p(Opt) = O(L^\alpha \cdot 1^p + L^{\alpha+\beta} \cdot (1/L^\beta)^p + L^{\alpha p})$. It is obvious that the dominant term is $L^{\alpha p}$, and hence, $F^p(Opt) = O(L^{\alpha p})$. Now consider the subcase that A has at least $L/2$ unit jobs unfinished by time $3L^\alpha/2$. Since these unfinished unit jobs must have been delayed by at least $L^\alpha/2$, $F^p(A) = \Omega(L \cdot L^{\alpha p})$. Clearly in this subcase the competitive ratio is $\Omega(L)$. Alternatively, consider the subcase that A has finished at least $L/2$ unit jobs by time $3L^\alpha/2$. Then A has at least $L^{\alpha+\beta}/2$ released, and unfinished, small jobs at time $3L^\alpha/2$. By the convexity of F^p, the optimal strategy for A from time $3L^\alpha/2$ onwards is to delay each small job by the same amount. Thus A delays $L^{\alpha+\beta}/2$ short jobs by at least $L/2$. Hence in this case, $F^p(A) = \Omega(L^{\alpha+\beta} \cdot L^p)$. This gives a competitive ratio of $\Omega(L^{\alpha+\beta+p-\alpha p})$, which by the choice of β is $\Omega(L)$. ∎

This negative result motivated Bansal and Pruhs [14] to fall back to resource augmentation analysis. They showed that the standard clairvoyant algorithms SJF and SRPT are almost fully scalable for l_p norms of flow and stretch. They showed that SETF and MLF are almost fully scalable for flow objective functions, but not for stretch objective functions. In contrast, RR is not almost fully scalable even for flow objective functions. This is a bit surprising as starvation avoidance is an often cited reason for adopting RR.

While the analysis of SJF used a local competitive argument, the analysis of SRPT was not strictly a local competitiveness argument as a newly released job J_i is not counted until time $r_i + \Theta(p_i)$. The analysis of SETF and MLF used the same method that Bansal *et al.* [13] used to analyze nonclairvoyant average stretch. We shall sketch the analysis of SJF.

Theorem 2.16 *For the problems* $1|\text{online-time}, \text{pmtn}, r_j|(\sum F_j^p)^{1/p}$*, and* $1|\text{online-time}, \text{pmtn}, r_j|(\sum S_j^p)^{1/p}$*, SJF is* $(1+\epsilon)$*-speed* $O(1/\epsilon)$*-competitive.*

Proof Sketch. The proof is by local competitiveness on the objective function $\sum F_j^p$. Let $U(SJF, t)$ and $U(Opt, t)$ denote the unfinished jobs at time t in SJF and Opt respectively, and $\mathcal{D} = U(SJF, t) - U(Opt, t)$. Let $Age^p(X, t)$ denote the sum over all jobs $J_i \in X$ of $(t - r_i)^{p-1}$. Note that $A(t)$, the rate of increase of the objective function for algorithm A, is the sum over all $J_i \in U(A, t)$ of $Age^p(U(A, t), t)$. That is, $F^p(A) = p \int_t Age^p(U(A, t), t)\, dt$.

Thus to have a local competitiveness argument, it is sufficient to establish that

$$Age^p(\mathcal{D}, t) \le O(1/\epsilon^p) Age^p(U(Opt, t), t)$$

This is established in the following manner. Let $V(t, \alpha)$ denote the aggregate unfinished work at time t among those jobs J_j that satisfy the conditions in the list α. Let $1, \ldots, k$ denote the indices of jobs in $\mathcal{D}$ such that $p_1 \le p_2 \ldots \le p_k$. Consider the jobs in $\mathcal{D}$ in the order in which they are indexed. Assume that we are considering job J_i. One can allocate to J_i an $\epsilon p_i/4(1+\epsilon)$ amount of work from $V(t, \{J_j \in U(Opt, t), r_j \le t - \epsilon(t - r_i)/(4(1+\epsilon)), p_j \le p_i\})$ that was previously not allocated to a lower indexed job in $\mathcal{D}$. This establishes $O(1/\epsilon^p)$ local competitiveness for F^p for the following reasons. The total unfinished work in each $J_j \in U(Opt, t)$ is associated with $O(1/\epsilon)$ longer jobs in $\mathcal{D}$. Since the jobs J_j are $\Omega(\epsilon)$ as old as J_i, the contribution to $Age^p(U(Opt, t), t)$ for J_j is $\Omega(\epsilon^{p-1})$ as large as the contribution of J_i to $Age^p(U(SJF, t), t)$. Using the same reasoning, and the fact that $p_j \le p_i$, one establishes local competitiveness for S^p. ∎

Chekuri, Khanna, and Kumar [32] show how to combine immediate dispatching algorithm of Avrahami and Azar [5] with a scheduling policy such as SJF to obtain an almost fully scalable algorithm for l_p norms of flow and stretch on multiple machines. The analysis is essentially a local competitive argument similar to Bansal and Pruhs' [14] analysis of SJF and SRPT.

2.6 Weighted Flow Time

In the online weighted flow time problem, each job J_i has an associated positive weight w_i that is revealed to the clairvoyant scheduler at the release time of J_i. The objective function is $\sum w_i F_i$. If all $w_i = 1$ then the objective function is total flow time, and if all $w_i = 1/p_i$ then the objective function is total stretch. Some systems, such as the Unix operating system, allows different processes to have different priorities. In Unix, users can use the `nice` command to set the priority of their jobs. Weights provide a way that a system might implement priorities. For the moment let us focus on one machine.

Becchetti, Leonardi, Marchetti-Spachemella, and Pruhs [22] show that besides being a sufficient condition, local c-competitiveness is a necessary condition for an algorithm to be c-competitive.

Theorem 2.17 *Every c-competitive deterministic algorithm A for* $1|\text{online-time}, r_j, \text{pmtn}|\sum w_j F_j$ *must be locally c-competitive.*

Proof Sketch. Suppose there is a time t where $A(t) > cOPT(t)$. The adversary can punish the online algorithm by bringing in a stream of dense short jobs with the following properties. The density of the jobs in the stream is large enough so that the optimal strategy for all algorithms is to run each of these jobs immediately. At the same time, the weight of the jobs in the stream can be made small enough so the contribution of the stream jobs to the total weighted flow time is arbitrarily close to 0 when the stream jobs are run immediately. The stream is made long enough so that the ratio of A's total flow time when compared to the adversary's total flow time is arbitrarily close to $A(t)/OPT(t)$.

At first glance, it may seem impossible for the stream of jobs to be dense enough to warrant running immediately yet have low enough weight that they contribute almost nothing to the total weighted flow time. This phenomenon becomes clearer when we consider the following example. Consider a job with weight x, processing time 1, and thus density x released at some time t and compare this to a stream of x jobs of weight 1, processing time $1/x$, and thus density also x released at times $t, t+1/x, \ldots, t+(x-1)/x$. If both are processed immediately, the job of weight x will contribute x to the total weighted flow time while the stream of weight 1 jobs will contribute only 1 to the total weighted flow time. On the other hand, if both are delayed by 1 time unit and then processed at time $t+1$, then the increase in total weighted flow time due to the delay for both cases will be exactly x. What we see is that the stream of density x jobs incurs the same delay cost as the one job with weight x, but the actual processing costs are vastly different. Finally observe that we can push this to the extreme where the stream of density x jobs have arbitrarily small weight $\epsilon > 0$ and processing times ϵ/x. If the stream jobs are processed immediately, they add only ϵ to the total weighted flow time. If the stream jobs are delayed by 1 time unit and then processed at time $t+1$, the increase in weighted flow time will still be exactly x. ■

The following instance shows that the obvious greedy algorithms have high competitive ratios. Consider the following set of jobs, released at time 0: One job of weight k, length k^2, and hence density $1/k$, and k^3 jobs of weight 1, length 1, and hence density 1. Two natural schedules are: (1) First run the low density job followed by the k^3 high density jobs, and (2) First run the k^3 high density jobs followed by the low density job. It is easy to see that the first algorithm is not constant locally competitive at time k^3, and that the second algorithm is not constant locally competitive at time k^3+k^2-1. In fact, what the online algorithm should do in this instance is to first run k^3-k of the high density jobs, then run the low density job, and then finish with the remaining k high density jobs. This instance demonstrates that the scheduler has to balance between delaying low density jobs, and delaying low weight jobs.

Using this intuition, Chekuri, Khanna, and Zhu gave an $O(\log^2 P)$-competitive algorithm for a single machine [33]. This algorithm is semi-online; it needs a priori knowledge of P. The algorithm partitions jobs based on approximate weights and on approximate densities. It then considers the weight classes from largest to smallest. Assume it is considering weight class w. It runs the densest job J_i from weight class w if and only if the total weight of jobs, with weight $< w$ and density greater than the density of J_i, is less than w. Otherwise it is safe for the algorithm to proceed to lower weight and higher density jobs. The analysis is a rather complicated local competitiveness argument.

If all jobs have the same weight, or if all jobs have the same processing time, or if all jobs have the same density, then $O(1)$-competitiveness is easy. This leads to 3 obvious algorithms. The algorithm partitions the jobs based on approximate weight or length, or density. Some job in the partition with maximum total weight is run. All jobs within a partition are run using the $O(1)$-competitive algorithm for same weight/length/density jobs. Intuitively, all of these algorithms should have competitive ratios that are linear in the number of partitions, or equivalently, logarithmic in the range of possible weights/lengths/densities. Bansal and Dhamdhere [12] proved this for the version of the algorithm where you partition based on the weight. The analysis is a local competitiveness argument that is a variation on the local competitiveness argument for SRPT.

Perhaps the most intellectually intriguing open question in online scheduling in the *online-time* model, with a flow or stretch objective function, is:

Open Problem 2.18 *For the problem $1|\mathit{online\text{-}time}, \mathit{pmtn}, r_j|\sum w_j F_j$, is there an $O(1)$-competitive clairvoyant algorithm?*

Several positive results have been developed for weighted flow time problems using resource augmentation. Phillips, Stein, Torng, and Wein [83] showed that an algorithm they named Preemptively-Schedule-By-Halves is 2-speed 1-competitive algorithm for minimizing total weighted flow time on a single machine. Becchetti, Leonardi, Marchetti-Spachemella, and Pruhs [22] observed that the analysis of Phillips *et al.* [83] also applied to HDF. Furthermore, using a more direct local competitiveness argument, Becchetti *et al.* [22] showed that HDF is $(1+\epsilon)$-speed $(1+1/\epsilon)$-competitive on a single machine.

Bansal and Pruhs then consider the problem of minimizing the weighted l_p norms of flow time [15]. They show that HDF is almost fully scalable for the problem $1|\mathit{online\text{-}time}, \mathit{pmtn}, r_j|(\sum w_j F_j^p)^{1/p}$. They then consider the obvious generalization, Weighted SETF (WSETF), of the nonclairvoyant algorithm SETF.

WSETF operates as follows. For a job J_i, let $x_i(t)$ denote the amount of work done on that job by time t. Amongst jobs with the smallest $x_i(t)/w_i$, WSETF splits the processor proportionally to weights of the jobs. They show that WSETF is almost fully scalable for the problem $1|$*online-time-nclv*, *pmtn*, $r_j|(\sum w_j F_j^p)^{1/p}$. The analysis of HDF and WSETF are similar to the analysis of SJF and SETF in [14].

For the parallel machine setting, only a few results are known. Chekuri, Khanna, and Zhu [33] give a lower bound on the competitive ratio of any algorithm of $\Omega(\min(\sqrt{P}, \sqrt{W}, (n/m)^{1/4})$ for the problem $P|$*online-time*, *pmtn*, $r_j|\sum w_j F_j$ where W is the largest weight. Becchetti *et al.* [22] show that HDF is a $(2+\epsilon)$-speed $O(1)$-competitive algorithm for the same problem.

2.7 Semi-Clairvoyant Scheduling for Average Flow/Stretch

The concept of semi-clairvoyant scheduling was introduced by Bender, Muthukrishnan, and Rajaraman [27]. A semi-clairvoyant algorithm only has approximate knowledge about processing times. A *strong semi-clairvoyant* algorithm knows a constant approximation of the remaining processing time of a job, and a *weak semi-clairvoyant* algorithm knows only a constant approximation of the original processing time of a job. While there may be some practical application for these results, for example a web server serving dynamic documents may only be able to estimate the size of resulting document as it dynamically constructs the document, the main motivation seems to be that such results may then be used as subroutines in other algorithms that round the processing times of jobs. Rounding processing times often seems to make the development of an algorithm or analysis simpler.

For the parallel machine setting, both the strong and weak semi-clairvoyant models are not significantly different than the clairvoyant setting when considering the total flow time and total stretch objective functions because of the classification nature of the clairvoyant non-migratory algorithms developed earlier. For example, the algorithm of Awerbuch *et al.* [7] that classifies jobs according to their remaining processing times can be adapted to be a strong semi-clairvoyant algorithm for minimizing total flow time and total stretch while the algorithms of Chekuri *et al.* [33] and Avrahami and Azar [5] that classify jobs according to their initial processing times can be adapted to be weak semi-clairvoyant algorithms for minimizing total flow time and/or stretch. Thus, we focus on the uniprocessor setting for the remainder of this subsection.

Let us first consider strong semi-clairvoyant algorithms on a single machine. The most obvious algorithm is to run the job that appears to have the least remaining processing time. Bender *et al.* [27] show that this algorithm is $O(1)$-competitive with respect to average stretch, but is only $\Theta(\log P)$-competitive with respect to average flow time. They then give modified algorithm that is $O(1)$-competitive with respect to average flow time. The main idea behind this algorithm is that if there is a choice between two jobs with similar remaining processing times, then the algorithm should favor the job whose initial processing time was less.

We now consider weak semi-clairvoyant algorithms. Bender *et al.* [27] show that the obvious generalization of SJF is $O(1)$-competitive with respect to average stretch. Bender, Muthukrishnan, and Rajaraman [27] also proposed an algorithm for average flow time. The basic idea of this algorithm is to run the apparent shortest job first, except in one special case. This special case is that if the job J_i with the apparent least original processing time has not been run at all, and the job J_j with the second least apparent original processing time has been partially run, and there are no other jobs with comparable apparent original processing times, then J_j is run instead of J_i.

Becchetti, Leonardi, Marchetti-Spachemella, and Pruhs [23] showed that this algorithm is in fact $O(1)$-competitive with respect to average flow time. A simpler analysis was developed by Nikhil Bansal. Bansal's analysis was a variation of the local competitiveness analysis of SRPT. At any time, order both the online algorithms jobs and the adversary's jobs by increasing remaining processing time. Then Bansal shows that the following invariant always holds: The total work contained in the online algorithm's jobs up to the kth unexecuted job, is at least the total work in the adversary's first k jobs. It is straightforward to observe that this invariant implies $O(1)$-competitiveness.

Becchetti *et al.* [23] show that there is no weak semi-clairvoyant algorithm that can be simultaneously $O(1)$-competitive with respect to average flow time and average stretch. This is in contrast to the clairvoyant setting where SRPT is $O(1)$-competitive with respect to both objective functions.

2.8 Nonclairvoyant Scheduling to Minimize Average/Maximum Flow/Stretch

In the *online-time-nclv* model, the nonclairvoyant scheduler is given no information about the processing time of a job. For example, the process scheduling component of an operating system is best viewed as being nonclairvoyant in that the operating system in general does not know the execution times of the various client processes.

2.8.1 Maximum and Average Flow Time on One Machine

If the objective function is minimizing maximum flow, then a nonclairvoyant scheduler can still be optimal since FIFO does not require knowledge of the processing times of the jobs. The situation for average flow looks more bleak. The optimal algorithm is SRPT. However, from the nonclairvoyant scheduler's point of view, any job might conceivably be the one with shortest remaining processing time. In the absence of any information about remaining processing times, the most obvious nonclairvoyant algorithm is probably RR. Motwani, Philips and Torng [80] show that RR is 2-competitive in the case that all jobs are released at time 0. However, Matsumoto [77], and independently Motwani, Philips and Torng [80] showed that the competitive ratio for RR is $\Omega(n/\log n)$ in the case of release dates. Kalyanasundaram and Pruhs [67] noted that a variation of this lower bound instance shows that modest resource augmentation is not enough to allow RR to be $O(1)$-competitive.

Theorem 2.19 *For the problem* $1|\text{online-time-nclv}, pmtn, r_j|\sum F_j$*, the competitive ratio of RR is at least* $\Omega(n/\log n)$*, and the competitive ratio of RR with speed* s*,* $1 < s < 2$ *processor is at least* $\Omega(n^{2-s})$*.*

Proof Sketch. Let $s = 1+\epsilon$. We divide time into stages. Let the ith stage, $i \geq 0$ start at time t_i. We let $t_0 = 0$, and $t_1 = 1+\epsilon$. There are two jobs of length $(1+\epsilon)$ released at time t_0, and one job is released at each time t_i, $i \geq 1$, with length $p(i)$ that is exactly the same length as RR has left on each of the previous jobs. In order to guarantee that the adversary can finish the job released at time t_{i-1} by time t_i, $i \geq 2$, we let $t_i = t_{i-1} + p(i-1)$. Observe that during the interval $[t_{i-1}, t_i]$, RR executes each of the $i+1$ jobs for $p(i-1)/(i+1)$ units of time. Since RR also uses a $1+\epsilon$ speed processor, the work done on a job during that interval is $(1+\epsilon)p(i-1)/(i+1)$. Therefore, we get the recurrence

$$p(i) = p(i-1) - \frac{(1+\epsilon)p(i-1)}{i+1} = \left(\frac{i-\epsilon}{i+1}\right) p(i-1)$$

The total flow time for the adversary is then $\Theta(\sum_{i=1}^{n} 1/(i-\epsilon)^{1+\epsilon})$, which is a convergent sum. The total flow time for RR is then $\Theta(\sum_{i=1}^{n} i/(i-\epsilon)^{1+\epsilon})$, which is $\Theta(n^{1-\epsilon})$. The result then follows. ■

More generally, Motwani, Phillips and Torng [80] showed that the competitive ratio of every deterministic nonclairvoyant algorithm for average flow is $\Omega(n^{1/3})$. Thus one can not get a strong positive result for deterministic nonclairvoyant algorithms using standard competitive analysis. This construction is the basis for most general lower bound proofs on average flow time.

Theorem 2.20 *For the problem* $1|\text{online-time-nclv}, pmtn, r_j|\sum F_j$*, the competitive ratio of every deterministic algorithm is* $\Omega(n^{1/3})$*.*

Proof Sketch. We present an adversary strategy which works in two stages. In the first stage, the adversary releases k jobs at time 0 and lets the algorithm A schedule them for k time units. The adversary ensures that the remaining processing time of each job at time k for A is $1/(k-1)$ while OPT has 1 unfinished job at time k. The second stage starts at time k, when the adversary releases a job of length $1/(k-1)$ every $1/(k-1)$ time units apart, until time k^2. No matter what A does after time k, A has a total flow time of $\Omega(k^3)$ while OPT has total flow time $O(k^2)$. ■

This strong lower bound motivated Kalyanasundaram and Pruhs [67] to propose resource augmentation analysis as a standard method of analysis. Notice that in this lower bound example the load after time k is 1. Furthermore, if the nonclairvoyant scheduler had a slightly faster processor, then it would not be behind at time k. Kalyanasundaram and Pruhs [67] showed that SETF is almost fully scalable. In [67] the algorithm SETF was called Balance.

Theorem 2.21 *For the problem* $1|$*online-time-nclv, pmtn,* $r_j|\sum F_j$*, SETF is* $(1+\epsilon)$*-speed* $(1+1/\epsilon)$*-competitive.*

Proof Sketch. We give the intuition here using the borrow technique introduced in [67]. The proof is a local competitiveness argument. That is, at any particular time t, SETF does not have too many more unfinished jobs than the adversary. Let J_i and J_j be jobs such that SETF is running J_j during the time interval $[a,b] \subseteq [r_i, C_i]$. The adversary may then do $(b-a)$ units of work on J_i during time $[a,b]$. We think of this as the adversary borrowing $(b-a)$ units of work from J_j to give to J_i. Borrowing can also be transitive; J_i can borrow from J_j which can borrow from J_k, etc. This borrowing might be advantageous to the adversary if J_i is almost finished by SETF. Let $w_i(t)$ be the remaining unfinished work on job J_i for SETF at time t. If the adversary is going to finish a job J_i, then it must arrange for J_i to borrow at least $\epsilon w_t(i)$ units of work since the adversary's processor is ϵ slower than the processor used by SETF. However, the description of SETF ensures that this time can only come from jobs that SETF ran for less time than SETF ran J_i. Hence, we would expect that each job that the adversary borrows time from can only be used to finish $1/\epsilon$ jobs that SETF has not finished. ■

Berman and Coulston [29] improved this result for larger speeds. They showed that SETF is s-speed $2/s$-competitive for total flow time for $s \geq 2$. They showed inductively that for each job that is added to a schedule, the increased cost that $SETF_s$ pays is at most $2/s$ the cost that the adversary pays.

Turning to randomized algorithms, Motwani, Philips and Torng [80] showed that the competitive ratio for total flow time of every randomized algorithm against an oblivious adversary is $\Omega(\log n)$. Kalyanasundaram and Pruhs [68] noted that this argument can be modified to give a lower bound of $\Omega(P)$. Recall that an oblivious adversary must fix the input a priori.

Theorem 2.22 *For the problem* $1|$*online-time-nclv, pmtn,* $r_j|\sum F_j$*, The competitive ratio of every randomized algorithm against an oblivious adversary is* $\Omega(\log n)$*.*

Proof Sketch. We use Yao's technique and prove a lower bound on the competitive ratio of deterministic algorithms on a particular input distribution.

The jobs are released in two phases. In the first phase, at time 0, k jobs are released whose sizes are independently drawn from the exponential distribution with mean 1. The scheduling algorithm is then allowed to run until time $k-2k^{3/4}$. Because the expected remaining work of an unfinished job is independent of how long it has been executed, the state of the unfinished jobs for the nonclairvoyant algorithm at time $k-2k^{3/4}$ is the same for all nonclairvoyant algorithms. Hence, at the end of the first stage, the nonclairvoyant algorithm has remaining $\Omega(k^{3/4})$ jobs with remaining work at least 1. With high probability, the adversary scheduler can set aside $k^{3/4}/\log k$ jobs of size at least $(\log k)/4$, and finish all other jobs by the end of the first phase.

The second phase consists of releasing a job of size 1 at each time unit, for a total of k^2 time units. Clearly the nonclairvoyant algorithm should execute these jobs before the large jobs it has remaining from the first phase, for a expected total flow time of $\Omega(k^{2.75})$. The adversary executes the second phase jobs as they arrive, and lastly schedules the set-aside jobs. The expected total waiting time of the adversary algorithm is $O(k^{2.75}/\log k)$. ■

It is not at all obvious what strategy a randomized algorithm should adopt in order to obtain a logarithmic competitive ratio. One benefit of resource augmentation analysis of deterministic algorithms is that the analysis can suggest a randomized strategy. This problem is an example of this phenomenon. Let us reflect on Kalyanasundaram and Pruhs' resource augmentation analysis of SETF [67] for a moment. One sees that to argue that SETF is locally $O(c \cdot d)$-competitive, it is sufficient to argue the following property holds:

> For an at least $1/c$ fraction of SETF's unfinished jobs, it is the case that they have at least $1/d$ of their original processing time left unfinished.

This suggests finding a randomized algorithm that favors newly released jobs (like SETF does) and that guarantees the above property holds with high probability. This line of reasoning led Kalyanasundaram and Pruhs [68] to propose the algorithm RMLF. RMLF is identical to MLF except that the target of each job in queue Q_i is 2^{i+1} minus an exponentially distributed independent random variable.

We now give some intuition why this approach should give poly-logarithmic competitiveness. Assume that at time 0 the adversary releases a collection of n jobs of length $2^i + x$. The adversary is hoping that at the first time that all remaining unfinished jobs for RMLF are all in Q_{i+1} that the following holds: it is the case that RMLF will have $\omega(1)$ jobs in Q_{i+1} and that these jobs are almost all almost finished. For example, if RMLF uniformly at random selected the target between 2^i and 2^{i+1}, then by picking $x = 2^i/\sqrt{n}$, the adversary could expect that RMLF has $\sqrt{n}$ jobs with at most a $1/\sqrt{n}$ of their initial processing time left. By bringing in a stream of small jobs, the adversary could then push the competitive ratio up to $\Omega(n^{\epsilon})$. So RMLF wants that the number of targets set to $X - 2^i/\log n$ should be a constant fraction of the number of targets set to X. By setting the targets randomly in this way, you expect that a constant fraction of the jobs have $1/\log n$ of their original processing time left. Two more points need to be made. First, this argument is not valid if x is very small, that is if the jobs have very little processing time left on the job when it reaches Q_i. However, in this case, each job is finished in Q_i and does not reach Q_{i+1} with very high probability. Second, in order to turn this into a formal proof, you need to have a high probability argument, which adds another factor of $\log n$ to the calculated competitive ratio. This argument can be formalized to show that RMLF is $O(\log^2 n)$-competitive for the problem $1|\textit{online-time-nclv}, \textit{pmtn}, r_j|\sum F_j$.

This $O(\log^2 n)$ analysis can be improved. Kalyanasundaram and Pruhs [68] showed that RMLF is $\Theta(\log n \log\log n)$ against an adversary that at all times knows the outcome of all of the random events internal to RMLF up until that time. This accounts for the possibility of inputs where future jobs may depend on the past schedule. Becchetti and Leonardi [21] improved upon this analysis to obtain a tight analysis of RMLF.

Theorem 2.23 *For the problem $1|\textit{online-time-nclv}, \textit{pmtn}, r_j|\sum F_j$, RMLF is $O(\log n)$-competitive against an oblivious adversary.*

Note that if the target for jobs in queue Q_i is c^i, then MLF is c-speed $O(1)$-competitive. In particular, if $c = 1 + \epsilon$, MLF devolves into SETF and is also almost fully scalable. These facts (SETF/MLF is almost fully scalable, and RMLF is optimally competitive amongst randomized algorithms) provide strong support for the adoption of MLF for process scheduling within an operating system.

Open Problem 2.24 *Obtain a tight bound on the competitive ratio of deterministic algorithms for the problem $1|\textit{online-time-nclv}, \textit{pmtn}, r_j|\sum F_j$. On one hand, given that there is a high, $\Omega(n^{1/3})$, lower bound on the competitive ratio, this may seem to be only of academic interest. On the other hand, this is arguably the most basic problem in nonclairvoyant scheduling, and it is quite unsatisfactory that a tighter bound is not known.*

2.8.2 Maximum and Average Stretch on One Machine

Kalyanasundaram and Pruhs [66] observed that the competitive ratio for maximum stretch is $\Omega(n)$ for nonclairvoyant algorithms and also that resource augmentation is of minimal help. For average stretch, it is easy to see that the competitive ratio for nonclairvoyant algorithms is $\Omega(n)$ and $\Omega(P)$. However, Bansal, Dhamdhere, Konemann, and Sinha [13] show that a moderately positive result can be obtained using resource augmentation.

Theorem 2.25 *For the problem $1|\textit{online-time-nclv}, \textit{pmtn}, r_j|\sum S_j$, MLF is an $O(1)$-speed $O(\log^2 P)$-competitive algorithm.*

Proof Sketch. It is easy to see that one cannot prove $MLF(J) = O(OPT(J))$ using local competitiveness (even if MLF has O(1) faster processor). To see this consider the case of a single unit length job and a small number of long jobs released at time 0. One can verify that every nonclairvoyant algorithm will be $\Omega(P)$ locally competitive at say time 2.

To show that $MLF(J) = O(\log^2 P) \cdot OPT(J)$, there are two main ideas in the proof. The first main idea was to show that $MLF(J) = O(SJF(L))$, where J is the original input, and L is some other input derived from J. In this modified instance L, each job J_i in J is replaced by a collection of jobs with geometrically increasing work, with aggregate work p_i, and with release date r_i. The idea is that at any particular time, MLF has the original job J_i in the jth queue if and only if SJF finished the $j-1$ shortest jobs in L corresponding to J_i.

To show $MLF(J) = O(SJF(L))$, Bansal *et al.* [13] introduce an auxiliary objective function, called inverse work, that can be used to show local competitiveness. Let $w_i(t)$ be the amount of work done on job i by time t. Then the inverse work for a job is $\int_{r_j}^{C_j} 1/w_j(t)\,dt$. Clearly the inverse work of a job is greater than its stretch $\int_{r_j}^{C_j} 1/p_j\,dt$. Hence, $MLF(J) \leq MLF'(J)$, where $MLF'(J)$ is total inverse work. Then the authors show that by local competitiveness that $MLF'(J) = O(SJF(L))$. Applying Becchetti, Leonardi, Marchetti-Spachemella, and Pruhs' [22] analysis of HDF, they conclude that $SJF(L)$ with a slightly faster processor has total stretch $O(OPT(L))$.

To finish the proof, one needs to upper bound $OPT(L)$ by $O(\log^2 P)\cdot OPT(J)$. The second main idea was the method used to relate $OPT(L)$ and $OPT(J)$. Given the schedule $OPT(J)$, one can construct a schedule for L (which is a union of geometrically scaled copies of J) in the following way. Take the schedule $OPT(J)$ and consider suitably scaled down copies of this schedule. These schedules are thought of running a scaled down copy of J with some carefully chosen processor speed. Executing all these schedules simultaneously can be thought of as a schedule for L. Bansal *et al.* [13] show that this scaling can be done in such a way that the total additional speed required is $O(1)$ and the total stretch for L is $O(\log^2 P)\cdot OPT(J)$. ■

For the problem $1|$*online-time-nclv, pmtn,* $r_j|\sum S_j$, Bansal *et al.* [13] also shows that every $O(1)$-speed algorithm has a competitive ratio of $\Omega(\log P)$. If all release dates are zero, Bansal *et al.* [13] gives an $O(\log P)$-competitive algorithm, and prove a general $\Omega(\log n)$ lower bound on the competitive ratio.

2.8.3 Average Flow Time on Parallel Machines

An immediate question that one has to ask when formalizing a scheduling problem on parallel machines is whether a single job can simultaneously run on multiple machines. In some settings this may not be possible; in other settings this may be possible but the speed-up that one obtains may vary. Thus one can get myriad different scheduling problems on parallel machines depending on what one assumes. A very general model is to assume that each job has a speed-up function that specifies how much the job is sped up when assigned to multiple machines. More formally, a *speed-up function* $\Gamma(s)$ measures the rate at which work is finished on the job if s processing resources (say s processors) are given to the job.

The simplest speed-up model is the *fully parallelizable* where $\Gamma(s) = s$. Fully parallelizable work has the property that if you devote twice as many resources to the work, it completes at twice the rate. The normal assumption in the uniprocessor scheduling literature is that all work is fully parallelizable. In the uniprocessor setting, this means that if you devote a fraction f of a single processor to a job, you will complete work at rate f instead of rate 1. To simplify notation, we will often use the word parallel instead of fully parallelizable when there is no possibility of ambiguity.

The normal multiprocessor setting can be modeled by the speed-up function $\Gamma(s) = s$ for $s \leq 1$ and $\Gamma(s) = 1$ for $s > 1$. That is, a job is fully parallelizable on one processor, but assigning the job to multiple processors does not help.

In any real application, speed-up functions will be sublinear and non-decreasing. A speed-up function is sublinear if doubling the number of processors at most doubles the rate at which work is completed on the job. A speed-up function is non-decreasing if increasing the number of processors does not decrease the rate at which work is completed on the job. One can also generalize this so that jobs are made of phases, each with their own speed-up function. We will use the notation sc_j in the job field of the three-field scheduling notation to denote parallel machines with job phases that have speed-up curves that are sublinear and non-decreasing.

We typically assume that a nonclairvoyant scheduling algorithm does not know the speed-up function of any job. Given how little knowledge a nonclairvoyant scheduler has in this setting, there are few natural algorithms to consider. The obvious ones to analyze are SETF and RR. Edmonds showed that SETF is not a good algorithm when jobs are not fully parallelizable [42].

Theorem 2.26 *The deterministic and randomized versions of SETF are not s-speed $O(1)$-competitive if the speed-up curves of jobs are not fully parallelizable no matter how large s is [42].*

Furthermore, in a remarkable analysis, Edmonds showed that RR is $(2+\epsilon)$-speed $O(1+1/\epsilon)$-competitive for jobs with phases that have speed-up functions that are sublinear and non-decreasing [42]. His result extends with slightly weaker bounds to the case where RR is given extra machines instead of faster machines.

Theorem 2.27 *For the problem $P|online\text{-}time\text{-}nclv, pmtn, r_j, sc_j|\sum F_j$, RR is $(2+\epsilon)$-speed $O(1+1/\epsilon)$-competitive.*

One obvious difficulty in constructing an $O(1)$-speed $O(1)$-competitiveness analysis for RR is that, as the example lower bound instance in Theorem 2.19 shows, one can not use a local competitiveness argument. One of Edmonds' insights was the identification of an appropriate potential function so that one could prove local competitiveness in an amortized sense. Arguably another insight was that analysis of RR for $1|online\text{-}time\text{-}nclv, pmtn, r_j|\sum F_j$, seems to require the introduction of speed-up curves. It is at least of academic interest whether there is an analysis of RR for $1|online\text{-}time\text{-}nclv, pmtn, r_j|\sum F_j$ that does not require the generalization to speed-up curves. Edmonds' analysis is too involved to give in its entirety here. We shall instead focus on the intuition that the proof gives about why RR performs reasonably well.

A key step in the proof is the introduction of the *constant* speed-up curve where $\Gamma(s) = c$ for all $s \geq 0$ and some constant $c > 0$. Devoting additional processing resources to constant jobs does not result in any faster processing of these jobs. In fact constant jobs complete at the same rate even if they are not run. The motivation for defining the constant speed-up curve is its utility for analytic purposes, not as a model of real job behavior. Note, Edmonds uses the term *sequential* instead of constant in [42].

With this definition of constant jobs, Edmonds transforms each possible input into a canonical input that is streamlined. An input is *streamlined* if: (1) every phase is either fully parallelizable or constant, and (2) the adversary is able to execute each job at its maximum possible speed. This implies that at any one time, the adversary has only one parallel job phase to which it is allocating all of its resources. The idea of this transformation is that if RR is devoting more resources to some work than the adversary, it is to the adversary's advantage to make this work be constant work that completes at the rate that the adversary was originally processing that work. In contrast, if the adversary is devoting more resources to a job than is RR, and the adversary has no other unfinished jobs, then it is to the adversary's advantage to make this work to be fully parallelizable. If the adversary has several unfinished jobs, then the transformation is only slightly more involved; each bit of work is replaced by a constant phase, followed by a parallel phase, followed by a constant phase. As a consequence of this transformation, you get that the adversary is never behind RR on any job. Given that the input is streamlined, we can for simplicity assume that RR has one processor of speed $s = 2 + \epsilon$ and OPT has one processor of speed 1.

We now turn to the potential function Φ, which is defined to be the work that has not been completed by RR but that has been completed by the adversary. Then $\Phi(t)$ is the rate of change of Φ at time t. Edmonds then proves local competitiveness using this potential function. That is, he shows that at all times t, $RR_s(t) \leq O(1 + 1/\epsilon)OPT(t) + \Phi(t)$.

We now give the intuition behind the Edmonds' proof from [42]. Let l_t be the number of constant jobs at time t. Note that l_t is the same for all schedules. RR devotes at most $s/(l_t + 1)$ of its speed to the unique fully parallelizable job that the adversary is working on at time t. To ensure that RR falls further behind on this job, l_t must be at least s or else RR may complete as much work on the parallel job as OPT does at time t. On the other hand, the adversary does not want l_t to be too large as the adversary must also pay this cost.

The key observation is that as the fully parallelizable work on which RR is behind builds up, RR self-adjusts by devoting more resources to this parallel work. Let m_t be the number jobs with parallel work for RR at time t. Note that m_t can be larger than 1 since RR is behind the adversary in some jobs. RR devotes $s/(l_t + m_t)$ of its s speed to each of the $l_t + m_t$ jobs it sees. Hence, RR completes fully parallelizable work at a rate of $s \cdot m_t/(l_t + m_t)$. Since the adversary works at unit rate on the fully parallelizable work, RR falls behind on this work at a rate of at most $1 - s \cdot m_t/(l_t + m_t)$. The steady state is when this rate is 0, that is, when $m_t = l_t/(s-1)$.

In this steady state, the competitive ratio is then at most $(l_t + l_t/(s-1))/(l_t + 1) \leq s/(s-1)$. Intuitively, RR tries to move to this steady state. To see this consider that RR is either above or below this steady state. If $m_t < l_t/(s-1)$ then more fully parallelizable work is being released than RR is completing, and hence RR is falling further behind and the potential function increases. The potential function increase is compensated by the fact that RR's flow time is increasing at a slower rate than it is at steady state. On the other hand, if $m_t > l_t/(s-1)$ then RR must be catching up to the adversary in terms of uncompleted parallel work. In this case, the decrease in the potential function must pay the additional increase in flow time that RR has to pay for being behind.

Note that the speed s has to be at least $2+\epsilon$ in order for the potential function to decrease quickly enough. A simple instance that shows that speed 2 is required is n jobs, with equal processing time, that all arrive at time 0. In this case RR needs speed at least 2 so that it is always $O(1)$-competitive in terms of the number of unfinished jobs.

The obvious and interesting open question is then:

Open Problem 2.28 *Is there an almost fully scalable algorithm in the case of sublinear and non-decreasing speed-up functions when the objective function is total flow time?*

Edmonds, Datta, and Dymond [43] extend Edmonds' analysis of RR to Internet TCP protocol. Becchetti and Leonardi [21] extend their analysis of RMLF to show that it is $O(\log n \log(n/m))$-competitive on m machines under the assumption that jobs may not be simultaneously run on multiple machines.

2.9 Multicast Pull Scheduling for Average Flow

In a multicast/broadcast system, when the server sends a requested page/item, all outstanding client requests to this page are satisfied by this multicast. The system may use broadcast because the underlying physical network provides broadcast as the basic form of communication (e.g. if the network is wireless or the whole system is on a LAN). Multicast may also arise in a wired network as a method to provide scalable data dissemination. One commercial example of a multicast-pull client-server system is Hughes' DirecPC system. In the DirecPC system the clients request documents via a low bandwidth dial-up connection, and the documents are broadcast via high bandwidth satellite to all clients. In this section we will restrict our attention to the case that the objective function is total flow time. We use the notation B in the machine field of the 3-field scheduling notation to denote broadcast, or more precisely, multicast pull.

While this problem is interesting in its own right, it is also interesting because of its connection to weighted flow time and its surprising connection to scheduling jobs with speed-up functions. We first explain why this problem generalizes weighted flow time. If one restricts the instances in multicast pull scheduling such that for each page, all requests for that page arrive at the same time, then the multicast pull scheduling problem and the weighted flow scheduling problem are identical. Here the number of requests that arrive for the page is the weight.

At first glance, it seems that the only difficulty the scheduler faces is how to favor both shorter pages as well as more popular pages. However, the situation is more complicated than this. Consider the case where all pages have the same size. The obvious algorithm to consider is Most Requests First (MRF) that broadcasts the page with the most outstanding requests thus generalizing the HDF weighted scheduling algorithm. At first, one might even be tempted to think that MRF is optimal. Kalyanasundaram, Pruhs, and Velauthapillai [69] showed that MRF is not $O(1)$-speed $O(1)$-competitive.

Lemma 2.29 *For the problem $B|$online-time, pmtn, $r_j, p_j = 1|\sum F_j$, the algorithm MRF is not $O(1)$-speed $O(1)$-competitive.*

Proof Sketch. Assume that MRF has an $s = O(1)$ speed processor. Let $k = n^2$. At time 0, the adversary requests pages $P_1, \cdots, P_{n-s}$ once each, and requests pages $P_{n-s+1}, \cdots, P_n$ twice each. At each time t, $1 \leq t \leq k$, the adversary requests pages $P_{n-s+1}, \cdots, P_n$ twice each.

For all times $t \in [1, k]$, MRF will broadcast pages $P_{n-s+1}, \cdots, P_n$. Only after time k will MRF finally broadcast pages $P_1, \cdots, P_{n-s}$. Since the initial requests to pages $P_1, \ldots, P_{n-s}$ are not satisfied by MRF during the first k time units, the total flow time for MRF is $\Omega(nk)$, which is $\Omega(n^3)$ since $k = n^2$.

On the other hand, for time $1 \leq i \leq n-s$, the adversary broadcasts page P_i. ¿From this time on, the adversary broadcasts pages $P_{n-s+1}, \cdots, P_n$ in a round robin fashion from time $(n-s)+1$ to time k. Each of the $O(ns)$ requests made before time $n-s$ is satisfied within n time units, and each of the $O(ks)$ requests made after time $n-s$ is satisfied within s time units. Hence, the total flow time for the adversary is $O(sn^2 + ks^2)$, which is $O(n^2)$ since $k = n^2$ and $s = O(1)$. Therefore, the competitive ratio for MRF is $\Omega(n)$. ■

The lower bound instance in Lemma 2.29 shows that the online scheduler has to be concerned with how to best aggregate jobs. Without knowledge of the future or resource augmentation, this turns out to be impossible. Kalyanasundaram, Pruhs, and Velauthapillai [69] show that no $O(1)$-competitive algorithm exists even in the case of unit pages if preemption is not allowed. Edmonds and Pruhs [44] extend the lower bound to the case that preemption is allowed.

Lemma 2.30 *For the problem $B|$online-time, pmtn, $r_j, p_j = 1|\sum F_j$, the competitive ratio of every randomized online algorithm A against an oblivious adversary is $\Omega(n)$ where n is the number of different pages.*

Proof Sketch. We give only the deterministic lower bound proof. This can be generalized to a lower bound for randomized algorithms using Yao's technique. At time 0, every page is requested once. Then no pages are requested until time $n/2$. From time 1 until time $n/2$, the adversary broadcasts the $n/2$ pages not broadcasted by A by time $n/2$. At time $n/2$, the adversary requests all of the pages previously broadcasted by A. Note that there are at most $n/2$ such pages and they were not previously broadcasted by the adversary. No more pages are requested until time n. After the broadcast at time n, the adversary has satisfied all of the requests to date, while A has at least $n/2$ unsatisfied requests. At each time t, for t from n to $k = n^2$, the adversary requests the page broadcasted by A at time $t-1$. Hence, at each time in $[n, k]$, A has $n/2+1$ unsatisfied requests. At each time $t \in [n+1, k+1]$, the adversary can satisfy the request at time $t-1$. Hence, the adversary has at most 1 unsatisfied request at each time $t \in [n+1, k]$. Hence, the total flow time for the adversary is $O(n^2 + k)$, and the total flow time for A is $\Omega(nk)$. ■

Before considering upper bounds, we need to note that several reasonable models are possible depending on what one assumes about the capabilities of the server and the clients to send and receive segments of the pages out of order. For example, it is not clear whether a client that requests a large page, in the middle of the broadcast, will need the whole page rebroadcast, or only the first half. For example, in a protocol, like the http protocol, where the content is identified only in the header, rebroadcast would be required. Pruhs and Uthaisombut [84] compare the optimal schedules, under various objective functions, in the different models. They show that allowing the server to send segments out of order is of no real benefit. On the other hand, they show that the ability of the clients to receive data out of order can drastically improve the average flow time, but not the maximum flow time. Further they show that a speed 2 server can compensate for clients not be able to receive pages out of order.

The general lower bound in Lemma 2.30 actually contains the key insight that ties multicast pull scheduling to scheduling with speed-up curves and thus suggests a possible algorithm. After the online algorithm has performed a significant amount of work on a page that was requested by a single client, the adversary can again direct another client to request that page. The online algorithm must service this second request as well. In contrast, the optimal schedule knows not to initially give any resources to the first request because the broadcast for the second request simultaneously services the first. Thus, even though the online algorithm devotes a lot of resources to the first request and the optimal algorithm devotes no resources to the first request, it completes under both at about the same time. In this regard, the work associated with the first request can be thought of as "constant". This suggests that the real difficulty of broadcast scheduling is that the adversary can force some of the work to have a constant speed-up curve.

Formalizing this intuition, Edmonds and Pruhs [44] give a method to convert any nonclairvoyant unicast scheduling algorithm A to a multicast scheduling algorithm B under the assumption that the clients must receive all pages in order. A unicast algorithm can only answer one request at a time. All the standard algorithms listed in section 2.1 are unicast algorithms. Edmonds and Pruhs [44] show that if A works well when jobs can have parallel and constant phases, then B works well if it is given twice the resources. The basic idea is that B simulates A, creating a separate job for each request, and then the amount of time that B broadcasts a page is equal to the amount of time that A runs the corresponding jobs. More formally, if A is an s-speed c-competitive unicast algorithm, then its counterpart, algorithm B, is a $2s$-speed c-competitive multicast algorithm. In the reduction, each request in the multicast pull problem is replaced by a job whose work is constant up until the time that either the adversary starts working on the job or the online algorithm finishes the job. After that time, the work of the replacement job is parallel. The amount of parallel work is such that A will complete a request exactly when B completes the corresponding job.

Using RR for algorithm A, one obtains an algorithm, called BEQUI in [44], that broadcasts each page at a rate proportional to the number of outstanding requests. Using Edmonds' analysis of RR for jobs with speed-up functions, one gets the following result.

Theorem 2.31 *For the problem $B|$online-time-nclv, pmtn$, r_j|\sum F_j$, under the assumption that all pages must be received in order, BEQUI is $(4+\epsilon)$-speed $O(1+1/\epsilon)$-competitive.*

Note that BEQUI preempts even unit sized jobs. Edmonds and Pruhs also give a $(4+\epsilon)$-speed $O(1+1/\epsilon)$-competitive algorithm BEQUI-EDF for the problem $B|$*online-time*$, r_j, p_j = 1|\sum F_j$. The idea of the algorithm is to simulate BEQUI to give a deadline for each request of the release time of that job plus some constant times the flow time of the job in BEQUI's schedule. BEQUI-EDF then runs the job with the Earliest Deadline First.

For the problem $B|$*online-time*$, r_j, p_j = 1|\sum F_j$, the most popular algorithm in the computer systems literature is Longest Wait First (LWF). LWF always services the page for which the aggregate waiting times of the outstanding requests for that page is maximized. In the natural setting where for each page, the request arrival times have a Poisson distribution, LWF broadcasts each page with frequency roughly proportional to the square root of the page's arrival rate, which is essentially optimal. Edmonds and Pruhs [45] provide an analysis of LWF. They show that LWF is 6-speed $O(1)$-competitive, but is not almost fully scalable. It is not too difficult to see that there is no possibility of proving such a result using local competitiveness. The rather complicated analysis given by Edmonds and Pruhs [45] compares the total cost of LWF to the total cost of the adversary.

The obvious interesting open question is then:

Open Problem 2.32 *For the problems $B|$online-time, pmtn$, r_j|\sum F_j$, $B|$online-time-nclv, pmtn$, r_j|\sum F_j$, $B|$online-time$, r_j, p_j = 1|\sum F_j$, is there an almost fully scalable algorithm? For the problems $B|$online-time, pmtn$, r_j|\sum F_j$, $B|$online-time-nclv, pmtn$, r_j|\sum F_j$ one should consider both the version where the client has to receive the page in order, and the version where the client can receive the page out of order.*

Bartal and Muthukrishnan [20] stated that FIFO is 2-competitive when the objective is minimizing maximum flow time under the assumption clients may receive documents out of order.

2.10 Nonclairvoyant Scheduling to Minimize Makespan

A general reduction theorem from [98] shows that in any variant of scheduling in *online-time-nclv* environment with makespan objective, any batch-style σ-competitive algorithm can be converted into a 2σ-competitive algorithm in a corresponding variant which in addition allows release times. In [54] it is proved that for a certain class of algorithms the competitive ratio is increased only by additive 1, instead of the factor of 2 in the previous reduction; 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 online 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. These reductions are 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 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. For the basic problem $P|$*online-time-nclv*$|C_{\max}$, no deterministic algorithm is better than $2 - 1/m$, i.e., better than List Scheduling, and randomization does not help much, as the lower bound is $(2 - O(1/\sqrt{m}))$ [98]. In Section 1.4 we mentioned that List Scheduling is $(2 - 1/m)$-competitive even for $P|$*online-time-nclv, prec*$, r_j|C_{\max}$. (Hence we do not lose anything in the competitive ratio for allowing release times, unlike in the general reductions above.)

2.10.1 Different Speeds

Here we consider both 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, and we assume that the speed for each job on each machine is known when the job is released. Only the running time is not known (i.e., for each job we know the relative speeds of machines).

If no preemptions are allowed, even for uniformly related machines, $Qm|\mathit{online\text{-}time\text{-}nclv}|C_{\max}$, a simple example shows that no algorithm is better than $\Omega(\sqrt{m})$-competitive [39]. A matching, $O(\sqrt{m})$-competitive, algorithm is known even for unrelated machines, $Rm|\mathit{online\text{-}time\text{-}nclv}|C_{\max}$, see [39]. This is not very satisfactory, as a trivial greedy algorithm is m-competitive even for $Rm|\mathit{online\text{-}time\text{-}nclv}, \mathit{prec}, r_j|C_{\max}$, see [46].

However, for related machines, allowing preemptions or even only restarts helps significantly. In this case we can use a variant of a doubling method to convert an arbitrary offline algorithm into an online algorithm. Since we do not know the running times, 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 its estimate, and repeat the procedure for all such jobs. This method, together with additional improvements, yields an $O(\log m)$-competitive algorithm for uniformly related machines with restarts, $Qm|\mathit{online\text{-}time\text{-}nclv}, \mathit{pmtn\text{-}restart}, r_j|C_{\max}$ [98]. A matching lower bound shows that this is optimal even for $Qm|\mathit{online\text{-}time\text{-}nclv}, \mathit{pmtn}|C_{\max}$ [98].

2.10.2 Parallel Jobs

In this variant, each job is characterized by its running time and the number of identical machines (processors) it requests. This is denoted by size_j in the middle field of the three-field notation. While the running times are unknown, the number of machines a job requests is known as soon as the job becomes available. We consider two variants according to how strict the request is. 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. Using the terminology of speed-up curves, such jobs are fully parallelizable up to the requested number of machines. That is, scheduling on $q' < q$ machines takes time pq/q' instead of the original processing time p.

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 $(2-1/m)$-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), even with release times, i.e., for $P|\mathit{online\text{-}time\text{-}nclv}, \mathit{size}_j, r_j|C_{\max}$ [82]. This is optimal, 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, $P|\mathit{online\text{-}time\text{-}nclv}, \mathit{size}_j, \mathit{prec}|C_{\max}$, no reasonable online algorithm exists for non-malleable parallel jobs. For deterministic algorithms there is a lower bound of m on the competitive ratio (a trivial greedy algorithm matches this) [53], and for randomized algorithms there is a lower bound of $m/2$ [94].

In contrast, with ideally malleable jobs, $P|\mathit{online\text{-}time\text{-}nclv}, \mathit{size}_j, \mathit{prec}|C_{\max}$ allows a constant competitive ratio. The optimal competitive ratio for deterministic algorithms for $P|\mathit{online\text{-}time\text{-}nclv}, \mathit{size}_j, \mathit{prec}|C_{\max}$ is $1+\phi \approx 2.6180$ [53]. The optimal strategy is again greedy, with the following rules for using malleability of the jobs: (i) If there is an available job requesting q machines and q machines are idle, schedule this job on q machines. (ii) Otherwise, if less than m/ϕ machines are busy and some job is available (requesting more machines), schedule it on all available machines. Note that this algorithm uses malleability 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 [53].

In practice, it is perhaps not realistic to assume that any subset of machines can be used for a parallel job. A model which takes into account a particular network topology of the parallel machine was considered in [55, 53, 95] (without precedence constraints, with precedence constraints, and randomized algorithms, respectively). In this model, if the underlying network is, for example, a mesh (two-dimensional array), each job requests a rectangular subset of processors with given dimensions. Perhaps the most interesting results in this area concern the power of randomization. For $Pm|\mathit{online\text{-}time\text{-}nclv}, \mathit{size}_j|C_{\max}$ with the mesh restriction, no deterministic algorithm has a constant competitive ratio, the tight bound is $\Theta(\sqrt{\log\log m})$; on the other hand, there exists a randomized $O(1)$-competitive algorithm. This randomized algorithm is based on random sampling, and it is one of the first results showing the power of randomization in the area of online algorithms. In contrast, if we allow precedence constraints, there are lower bounds showing that randomization does not change the competitive ratio significantly. For a more complete survey of results for various topologies (linear array, hypercube, mesh), see [97].

3 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 online load balancing as the case where the jobs are permanent and the load is their only parameter corresponding to our processing time. Consequently, there are many results on load balancing that extend the basic results on online scheduling in a different direction. As a reference in this area, we recommend the survey [8].

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. An exception is the area of preemptive scheduling on related machines where introducing idle times seems to be very useful.

We first give the results considering minimizing the makespan; only in Sections 3.8 and 3.9 do we briefly mention results for other objective functions, namely minimizing the l_p norm and the total completion time.

3.1 The Basic Model

We start by studying the basic parallel machine scheduling problem $P|\mathit{online\text{-}list}|C_{\max}$. This is probably the most extensively studied online scheduling problem, yet many questions remain open. In this section, we are interested in deterministic algorithms.

We have m machines and a sequence of jobs characterized by their processing times. The jobs are presented one by one, and we have to schedule each job to a single machine before we see the next one. There are no additional constraints, preemption is not allowed, all the machines have the same speed, and the objective function is the makespan.

The greedy List Scheduling algorithm schedules each arriving job on a least loaded machine. From Graham's analysis [62], it follows that the competitive ratio of List Scheduling is $2 - 1/m$. This is provably the best possible for $m = 2$ and $m = 3$ [52], but for larger m it is possible to develop better algorithms.

From the analysis of List Scheduling, 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 processing time is presented, we may 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 in preparation for large jobs that have not yet arrived.

To design a good algorithm, current results use two different approaches. One is to schedule each job on one of the two currently least loaded machines [60, 36]. 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. The difficulty is that this approach only ensures that there is one lightly loaded machine. Thus, after many small jobs and two long jobs, we get a long schedule and the competitive ratio is at least $2 - 2/m$.

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 [17]. Later better algorithms based on this idea were designed in [72, 1, 58] to give the currently best upper bounds for large m. The analysis of all these algorithms is relatively complicated. However, at a basic level, all of these algorithms use the following three lower bounds on the optimal makespan: (i) the total processing time of all jobs divided by m, (ii) the largest processing time of any job, and (iii) the sum of the mth largest and $m+1$st largest processing times. It has recently been shown that we cannot prove any deterministic algorithm has a better competitive ratio than 1.919 using only these lower bounds on the optimal makespan [2]. It has been conjectured that an improved algorithm and an improved analysis might by using additional lower bounds on the optimal makespan that include the size of the $2m+1$st largest processing time, or in general the

$km + 1$st processing time for $k \geq 2$. However, we still cannot prove any deterministic algorithm has a better competitive ratio than 1.917 using these additional lower bounds on the optimal makespan [2].

The lower bounds for this problem are typically proven by explicitly giving a hard sequence of jobs. The observation that List Scheduling is optimal for $m = 2, 3$ is due to [52]. For $m = 4$, the lower bound is $\sqrt{3} \approx 1.7321$, see [85, 86]. The other lower bounds for small m are from [36]. The lower bounds for large m were gradually improved in [18, 1, 85].

The current state of our knowledge is summarized in Table 3. For comparison we include also the competitive ratio of List Scheduling. (See Section 3.2 for a discussion of results for randomized algorithms and Section 3.3 for preemptive scheduling.)

	deterministic			randomized		preemptive
m	LS	upper bound	lower bound	upper bound	lower bound	upper and lower bound
2	1.5000	1.5000	1.5000	1.3333	1.3333	1.3333
3	1.6666	1.6666	1.6666	1.5373	> 1.4210	1.4210
4	1.7500	1.7333	1.7321	1.6567	1.4628	1.4628
5	1.8000	1.7708	1.7462	1.7338	1.4873	1.4873
6	1.8333	1.8000	1.7730	1.7829	1.5035	1.5035
7	1.8571	1.8229	1.7910	1.8168	1.5149	1.5149
∞	2.0000	1.9230	1.8800	1.9160	1.5819	1.5819

Table 3: Current bounds for $Pm|\textit{online-list}|C_{\max}$ and $Pm|\textit{online-list, pmtn}|C_{\max}$.

3.2 Randomized Algorithms

Much less is known about randomized algorithms for the basic model $P|\textit{online-list}|C_{\max}$ studied in Section 3.1. We only have a known optimal randomized algorithm for the case $m = 2$. A 4/3-competitive randomized algorithm and a matching lower bound for two machines, $P2|\textit{online-list}|C_{\max}$, was presented in [17].

First we show that this is best possible. Consider a sequence of three jobs with processing times 1, 1, and 2. After the first two jobs, the optimal makespan is 1, so the expected makespan of the online algorithm has to be at most 4/3. This means that after the first two jobs, the expected load of the less loaded machine is at least 2/3, and after the third job, even if it is always scheduled on the smaller machine, the expected makespan is at least $2/3 + 2 = 8/3$. Since the optimum is 2, the algorithm cannot be better that 4/3-competitive.

In the proof, we can replace the first two jobs by an arbitrary sequence of jobs with total processing 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 processing 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. It turns out this works, with some additional considerations for large jobs [17, 97].

The idea of the lower bound for two machines can be extended to an arbitrary number of machines [35, 96]. 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. This lower bound shows 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 $4 : 6 : 9$.) An algorithm based on this invariant would be a natural generalization of the optimal algorithm for two machines from [17]; it would also follow the suggestion from [37] (see Section 3.3). However, it is impossible to always maintain this ratio of expected loads. For three machines, $P3|\textit{online-list}|C_{\max}$, we know that this lower bound of 27/19 is not tight [100]. More precisely, we know that for some $\varepsilon > 0$, there is no $(27/19 + \varepsilon)$-competitive algorithm, but the value of ε is very small and not explicit in [100].

New randomized algorithms for small m were developed in [92, 88]. They are provably better than any deterministic algorithm for $m = 3, 4, 5$ and better than the currently best deterministic algorithm for $m = 6, 7$. They always assign the new job on one of the two least loaded machines, similar to the deterministic algorithms for small m from [60, 36]. Consequently, the competitive ratio approaches two as m grows.

Another observation is that any randomized algorithm that never assigns jobs to the most loaded machine is at best 1.5-competitive. Consider a sequence of 2 jobs with processing time 1 and $m-1$ jobs with processing time 2. The first two jobs are assigned to two distinct machines due to the restriction of the algorithm. After the remaining jobs, the makespan is at least 3, while the optimum is 2.

Recently a new 1.916-competitive randomized algorithm for any number of machines, $P|online\text{-}list|C_{\max}$, was given in [2]. It is interesting that this algorithm as well as the algorithms for small m from [88] are *barely random*, i.e., need only a finite number of random bits (or different schedules) independent of the number of jobs. In contrast to this, the optimal algorithm for $m = 2$ from [17] needs to maintain an increasing collection of schedules; their number is linear in the number of jobs. Note also that 1.916 is better than the best possible deterministic competitive ratio provable using "standard" lower bounds on the optimal makespan.

To summarize, we have the optimal randomized algorithm for $m = 2$, a significant improvement over the deterministic algorithms for small m and a tiny improvement over the deterministic algorithms for large m. See Table 3.

3.3 Preemptive Scheduling

Next we consider the preemptive version of the problem, $P|online\text{-}list, pmtn|C_{\max}$. 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. In this model the offline case is easily solved, and the optimal makespan is the maximum of the maximal processing time and the sum of the processing times divided by m (i.e., the average load of a machine), see Chapter 3.

It is easy to see that the lower bounds from Section 3.2 hold in this model, too, as they only use the arguments about expected load (with the exception of the improved bound for 3 machines). 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 [37]. As it turns out, there exists a deterministic algorithm matching this lower bound. It essentially tries to preserve the invariant that the expected loads are in geometric sequence with the ratio $m : (m-1)$ with some special considerations for large jobs [37].

Thus, in this model, both deterministic and randomized cases are completely solved, giving the same bounds as the randomized lower bounds in Table 3. 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 maintain the ideal configuration by spreading the loads as we wish among the m machines. Thus, preemption is more powerful than randomization.

3.4 Semi-Online Algorithms with Known Optimum and Doubling Strategies

Assuming that the algorithm knows the optimum value of the objective function is perhaps not realistic from a practical viewpoint. However, as the following theorem shows, such a semi-online algorithm can be used as a building block for an online algorithm for the same problem. Instead of a known optimal makespan, we use an estimate and double it whenever it turns out that the estimate was too small.

Theorem 3.1 *Suppose that for some scheduling problem in the* online-list *environment with the objective to minimize makespan there exists an R-competitive semi-online algorithm if the optimum is known. Then for the same problem there exists both a deterministic online algorithm with competitive ratio $4R$ and a randomized online algorithm with competitive ratio $eR < 2.7183 \cdot R$.*

Proof Sketch. Let G_0 be the value of the optimal schedule considering only the first job of the sequence and let OPT be the optimal makespan on the whole instance. Let A_G denote the semi-online algorithm provided with the information that G is the optimal makespan. First note that if $G \geq OPT$, then A_G always produces a schedule with makespan at most RG: the sequence can be appended with jobs that increase the makespan to exactly G and on this appended sequence the algorithm guarantees not to schedule any job after time RG.

The deterministic online algorithm computes G_0 and sets $G := G_0$ upon the arrival of the first job. Then it runs the algorithm A_G modified so that the jobs are scheduled in time interval $[RG, 2RG)$ instead of $[0, RG)$. If A_G fails to schedule the next job, the online algorithm sets $G := 2G$ and starts the algorithm A_G with the new value of G. The intervals in which the algorithm A_G schedules for different values of G are disjoint and thus the algorithm is well defined. The value of G can be increased only when $G < OPT$, by the property of the semi-online algorithm mentioned above. Thus at the end of the algorithm we have $G \leq 2 \cdot OPT$ and the final makespan is at most $2RG \leq 4R \cdot OPT$ and the algorithm is $4R$-competitive.

The randomized online algorithm computes G_0 and sets $G := G_0 \cdot e^z$, where z is a random variable uniformly distributed in $[0, 1)$ and e is the base of natural logarithms. Then it runs the algorithm A_G modified so that the jobs are scheduled in the time interval $[RG \cdot 1/(e-1), RG \cdot e/(e-1))$ instead of $[0, RG)$. If A_G fails to schedule the next job, the online algorithm sets $G := eG$ and starts the algorithm A_G with the new value of G. Again, the intervals in which the algorithm A_G schedules for different values of G are disjoint and the value of G can be increased only when $G < OPT$. Thus at the end of the algorithm G is at most $G' = G_0 \cdot e^{k+z}$ where k is the smallest integer such that this value is at least OPT. This is equivalent to saying that $x = \ln(G'/OPT)$ is the fractional part of $y = \ln(G_0/OPT) + k + z$. Since z is uniform in $[0, 1)$, k is an integer and $\ln(G_0/OPT)$ is a constant, x is also uniformly distributed in $[0, 1)$. The expected value of the final makespan is at most $\mathbf{Exp}[RG' \cdot e/(e-1)] = \mathbf{Exp}[Re^x \cdot OPT \cdot e/(e-1)] = \mathbf{Exp}[e^x] \cdot R \cdot OPT \cdot e/(e-1) = eR \cdot OPT$ and the algorithm is eR-competitive. ■

A doubling strategy similar to this theorem is a very common tool in computer science. In the area of online algorithms, it leads to optimal algorithms for search on a line (also known as cow-path problem) and its generalizations, both for deterministic and randomized algorithms, see [11, 71, 70]. In the context of online scheduling, it was used the first time in [98, 3], see Sections 2.10.1 and 3.5. In some cases, to get currently best results, this method may need some refinements, however, the basic idea of multiplying the estimate by a fixed constant as well as type of distribution used for the initial guess of a randomized algorithm is always the same.

If the optimum is known, the problem $P|online\text{-}list|C_{\max}$ is also studied as so-called online bin-stretching. 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 online. For two machines, there exists a 4/3-competitive algorithm and this is tight. For more machines a 1.625-competitive algorithm is presented in [10]. Of course, in this case, doubling algorithms are not useful as other algorithms perform better.

For uniformly related machines non-preemptive scheduling, $Q|online\text{-}list|C_{\max}$, scheduling a job on the slowest machine that completes the job by the time equal to twice the optimal makespan is a 2-competitive semi-online algorithm [3]. For preemptive scheduling on related machines, $Q|online\text{-}list, pmtn|C_{\max}$, we can even produce an optimal schedule if the optimal makespan is known; a 1-competitive semi-online algorithm is given in [47] for two machines and in [41] for any number of machines.

3.5 Different Speeds

For uniformly related machines, most results are based on the doubling strategy from Section 3.4 or its variants. For non-preemptive scheduling, $Q|online\text{-}list|C_{\max}$, a simple doubling strategy leads to a constant competitive ratio [3]. The competitive ratio can be improved by using more sophisticated analysis of doubling strategies. The current best algorithms are $3 + \sqrt{8} \approx 5.828$-competitive deterministic and 4.311-competitive randomized [28]. For an alternative very nice presentation see [16]. The lower bounds are 2.438 for deterministic algorithms [28] and 2 for randomized algorithms [51].

For uniformly related machines preemptive scheduling, $Q|online\text{-}list, pmtn|C_{\max}$, we already mentioned that it is possible to design an optimal (1-competitive) semi-online algorithm if the optimal makespan is known in advance. Thus, by Theorem 3.1, this yields 4-competitive deterministic and 2.7183-competitive randomized algorithms [41]. The lower bound is 2 both for deterministic and randomized algorithms [51].

For unrelated machines, $R|online\text{-}list|C_{\max}$, it is possible to obtain $O(\log m)$-competitive deterministic algorithm [3, 75]. A matching lower bound of $\Omega(\log m)$ holds both for deterministic and randomized algorithms

		deterministic			randomized	
preemption	m	LS	upper bound	lower bound	upper bound	lower bound
non-preemptive	2	1.618	1.618	1.618	1.528	1.500
	∞	$\Theta(\log m)$	5.828	2.438	4.311	2.000
preemptive	2	1.500	1.333	1.333	1.333	1.333
	∞	$\Theta(\log m)$	4.000	2.000	2.718	2.000

Table 4: Current bounds for $Qm|\mathit{online\text{-}list}|C_{\max}$ and $Qm|\mathit{online\text{-}list}, \mathit{pmtn}|C_{\max}$.

even in the special case of the so-called restricted assignment, where each job specifies a set of machines on which it may be processed (it is processed infinitely slowly on the others) and besides this restriction all the machines have the same speed [9]. The lower bound also works for $R|\mathit{online\text{-}list}, \mathit{pmtn}|C_{\max}$.

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 so that it will finish as early as possible (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, $R|\mathit{online\text{-}list}|C_{\max}$, the competitive ratio of List Scheduling is exactly m [3]. For related machines, $Q|\mathit{online\text{-}list}|C_{\max}$ and $Q|\mathit{online\text{-}list}, \mathit{pmtn}|C_{\max}$, the competitive ratio of List Scheduling is asymptotically $\Theta(\log m)$ [38, 3, 41] (the lower bound, the upper bound, and the preemptive case, respectively). The exact competitive ratio for $m = 2$ is ϕ and for $3 \leq m \leq 6$ it is equal to $1 + \sqrt{(m-1)/2}$ [38]; moreover for $m = 2, 3$ it can be checked easily that there is no better deterministic algorithm.

For two machines, $Q2|\mathit{online\text{-}list}|C_{\max}$ and $Q2|\mathit{online\text{-}list}, \mathit{pmtn}|C_{\max}$, we are able to analyze the situation further, depending on the speeds [50]. We first consider the non-preemptive problem. 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 online 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 the 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, and the overall upper bound is 1.5278. The competitive ratio of the optimal deterministic preemptive algorithm is better than the competitive ratio of the optimal non-preemptive randomized algorithm for any $s \geq 1$. Furthermore, the worst-case is the identical machine case when $s = 1$. In contrast, without preemption, the worst competitive ratio (both deterministic and randomized) is achieved for some $s > 1$ [50, 101].

The current bounds for scheduling on uniformly related machines are summarized in Table 4.

3.6 Semi-Online Algorithms

In addition to algorithms that know the optimum which we discussed in Section 3.4, the most commonly studied semi-online variant is the one where the jobs arrive sorted according to their processing times. In case of the makespan objective, the jobs are sorted largest first, i.e., by non-increasing processing time, to improve the performance.

When the jobs are sorted, the greedy online algorithm List Scheduling becomes the so-called LPT (Largest Processing Time first) semi-online algorithm. We already mentioned that for $P|\mathit{online\text{-}list}|C_{\max}$, the competitive ratio of LPT is $4/3 - 1/(3m)$, see [63]. For related machines the competitive ratio of LPT is a small constant, unlike List Scheduling which is only $\Theta(\log m)$-competitive. For $Q|\mathit{online\text{-}list}|C_{\max}$, the competitive ratio of LPT is between 1.52 and 1.66 [59]; a better upper bound of 1.58 is claimed in [40], but the proof appears to be incomplete. For $Q2|\mathit{online\text{-}list}|C_{\max}$, the complete analysis of the dependence of the competitive ratio on the speed ratio was given in [79]. For $Q|\mathit{online\text{-}list}, \mathit{pmtn}|C_{\max}$, the competitive ratio of LPT is 2, see [41].

The semi-online case of $P|\mathit{online\text{-}list}|C_{\max}$ and $P|\mathit{online\text{-}list}, \mathit{pmtn}|C_{\max}$ was further studied in [89]. It turns out that for $P2|\mathit{online\text{-}list}|C_{\max}$, LPT is an optimal deterministic algorithm. For randomized algorithms a better competitive ratio of $8/7$ is possible and optimal. For $P|\mathit{online\text{-}list}, \mathit{pmtn}|C_{\max}$, the optimal competitive ratio is $(1+\sqrt{3})/2 \approx 1.336$; this is surprisingly higher than the performance of LPT in the non-preemptive case. The semi-online case of $Q2|\mathit{online\text{-}list}|C_{\max}$ and $Q2|\mathit{online\text{-}list}, \mathit{pmtn}|C_{\max}$ was completely analyzed in [48, 49].

3.7 Scheduling with Rejections

In this version, jobs may be rejected at a certain penalty. Each job is characterized by the processing 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 processing 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 online 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 processing 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.

We first look at deterministic algorithms in the case when preemption is not allowed [19]. 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. Thus, 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 $1+\phi$, respectively. 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 lower bounds for small m from [19] 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 [93]. It achieves competitive ratio 2.3875 for all m. An interesting question is whether a better than 2-competitive algorithm can be found for $m = 3$: we know several different 2-competitive algorithms even without preemption, but the lower bound does not match this barrier.

Randomized algorithms for this problem, both with and without preemption, were designed in [91, 90, 93]. No algorithms better than the deterministic ones are known for large m. The lower bounds for randomized scheduling without rejection (Table 3) clearly apply here (set the penalties infinitely large), and no better lower bounds are known.

The results are summarized in Table 5. 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.

	deterministic	deterministic upper bounds		randomized upper bounds	
m	lower 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 5: Current bounds for algorithms scheduling jobs one by one with possible rejection.

3.8 Minimizing the l_p Norm

Here we minimize the l_p norm of the vector of the loads of machines, 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 [8, 6]. For identical machines, a convexity argument implies that if all the machine loads are equal, the schedule is optimal. Thus, similar to measuring makespan, this performance measure quantifies how well we can approximate this ideal schedule; however note that a single overloaded machine has a much smaller effect on the l_p objective.

Minimizing the l_2 norm on identical machines was studied in [4]. 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 $\delta > 0$. Since the lower bound of $\sqrt{4/3}$ holds for $m = 3$, this means that the optimal competitive ratio is also not monotone in m. This is perhaps a most interesting feature of these results: for the basic problem $Pm||C_{\max}$ we often expect, based on the current results, that the competitive ratio will increase with the number of machines m; this intuition thus fails at least for a slightly different objective function. For a general p, the same approach leads also to an algorithm better than List Scheduling for large m.

For unrelated machines, [6] gives a simple greedy algorithm with a competitive ratio $1 + \sqrt{2}$ for the l_2 norm and $O(p)$ for a general l_p norm. In contrast to makespan, the competitive ratio is a constant that does not depend on the number of machines or jobs.

3.9 Minimizing the Total Completion Time

In this variant it is necessary to use idle times, as we have to finish the jobs with short processing times first to minimize the total completion time. Even on a single machine, $1|$*online-list*$|\sum C_j$, 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 [57].

3.10 Open problems

Randomized algorithms. We still understand very little about the power of randomization in this online paradigm, despite some recent progress. In particular, for the basic problem $P|$*online-list*$|C_{\max}$, the lower bound is 1.581 while the best algorithm is 1.916-competitive; this gap is quite large compared to the case of deterministic algorithms. It is reasonable to expect that improvements of the algorithm are more likely, but the lower bound of [100] for $P3|$*online-list*$|C_{\max}$ indicates some possibility of improving the lower bound as well.

Preemptive scheduling on related machines. In the offline case, $Q|$*pmtn*$|C_{\max}$ we understand preemptive scheduling very well. The optimum is easy to calculate and the structure of optimal schedules is well understood [65, 61]. In the online identical machine case, $P|$*online-list, pmtn*$|C_{\max}$ we have a similar complete understanding of the optimal online algorithm. Despite some effort, the case of online scheduling on uniformly related machines, $Q|$*online-list, pmtn*$|C_{\max}$ remains open. Our intuition is that, similarly as for $P|$*online-list, pmtn*$|C_{\max}$, randomization should not help and thus the deterministic 4-competitive algorithm can be improved.

Acknowledgments

We are grateful to many colleagues for useful comments, pointers to the literature, and manuscripts. Without them this survey could not possibly cover as many results as it does.

References

[1] S. Albers. Better bounds for online scheduling. *SIAM Journal on Computing*, 29:459–473, 1999.

[2] S. Albers. On randomized online scheduling. In *Proc. 34th Symp. Theory of Computing (STOC)*, pages 134–143. ACM, 2002.

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

[4] A. Avidor, Y. Azar, and J. Sgall. Ancient and new algorithms for load balancing in the l_p norm. In *Proc. 9th Symp. on Discrete Algorithms (SODA)*, pages 426–435. ACM/SIAM, 1998.

[5] N. Avrahami and Y. Azar. Minimizing total flow time and total completion time with immediate dispatching. In *Proc. 15th Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 11–18. ACM, 2003.

[6] 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. 36th Symp. Foundations of Computer Science (FOCS)*, pages 383–391. IEEE, 1995.

[7] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. *SIAM Journal on Computing*, 31:1370–1382, 2001.

[8] Y. Azar. On-line load balancing. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms: The State of the Art*, pages 178–195. Springer, 1998.

[9] Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignments. *Journal of Algorithms*, 18:221–237, 1995.

[10] Y. Azar and O. Regev. On-line bin stretching. *Theoretical Computer Science*, 268:17–41, 2001.

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

[12] N. Bansal and K. Dhamdhere. Minimizing weighted flow time. In *Proc. 14th Symp. on Discrete Algorithms (SODA)*, pages 508–516. ACM/SIAM, 2003.

[13] N. Bansal, K. Dhamdhere, J. Konemann, and A. Sinha. Non-clairvoyant scheduling for mean slowdown. In *Proc. 20th Symp. on Theoretical Aspects of Computer Science (STACS)*, volume 2607 of *Lecture Notes in Computer Science*, pages 260–270. Springer, 2003.

[14] N. Bansal and K. Pruhs. Server scheduling in the L_p norm: A rising tide lifts all boats. In *Proc. 35th Symp. Theory of Computing (STOC)*, pages 242–250. ACM, 2003.

[15] N. Bansal and K. Pruhs. Server scheduling in the weighted l_p norm. Manuscript, 2003.

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

[17] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. *Journal Computer Systems Science*, 51:359–366, 1995.

[18] Y. Bartal, H. Karloff, and Y. Rabani. A better lower bound for on-line scheduling. *Information Processing Letters*, 50:113–116, 1994.

[19] Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, and L. Stougie. Multiprocessor scheduling with rejection. *SIAM Journal on Discrete Mathematics*, 13:64–78, 2000.

[20] Y. Bartal and S. Muthukrishnan. Minimizing maximum response time in scheduling broadcasts. In *Proc. 11th Symp. on Discrete Algorithms (SODA)*, pages 558–559. ACM/SIAM, 2000.

[21] L. Becchetti and S. Leonardi. Non-clairvoyant scheduling to minimize the average flow time on single and parallel machines. In *Proc. 33rd Symp. Theory of Computing (STOC)*, pages 94–103. ACM, 2001. To appear in JACM.

[22] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, and K. Pruhs. Online weighted flow time and deadline scheduling. In *RANDOM-APPROX*, volume 2129 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2001.

[23] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, and K. Pruhs. Semi-clairvoyant scheduling. In *Proc. 11th European Symp. on Algorithms (ESA)*, volume 2832 of *Lecture Notes in Computer Science*. Springer, 2003.

[24] L. Becchetti, S. Leonardi, and S. Muthukrishnan. Scheduling to minimize average stretch without migration. In *Proc. 11th Symp. on Discrete Algorithms (SODA)*, pages 548–557. ACM/SIAM, 2000.

[25] S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Widgerson. On the power of randomization in on-line algorithms. In *Proc. 22nd Symp. Theory of Computing (STOC)*, pages 379–386. ACM, 1990.

[26] M. A. Bender, S. Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *Proc. 9th Symp. on Discrete Algorithms (SODA)*, pages 270–279. ACM/SIAM, 1998.

[27] M. A. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *Proc. 13th Symp. on Discrete Algorithms (SODA)*, pages 762–771. ACM/SIAM, 2002.

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

[29] P. Berman and C. Coulston. Speed is more powerful than clairvoyance. *Nordic Journal of Computing*, 6(2):181–193, 1999.

[30] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[31] R. Càceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: The devil is in the details. In *Proceedings of the ACM SIGMETRICS Workshop on Internet Server Performance*, 1998.

[32] C. Chekuri, S. Khanna, and A. Kumar. Multi-processor scheduling to minimize l_p norms of flow and stretch. Manuscript, 2003.

[33] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for weighted flow time. In *Proc. 33rd Symp. Theory of Computing (STOC)*, pages 84–93. ACM, 2001.

[34] B. Chen, C. N. Potts, and G. J. Woeginger. A review of machine scheduling: Complexity, algorithms and approximability. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 3, pages 21–169. Kluewer, 1998.

[35] B. Chen, A. van Vliet, and G. J. Woeginger. Lower bounds for randomized online scheduling. *Information Processing Letters*, 51:219–222, 1994.

[36] B. Chen, A. van Vliet, and G. J. Woeginger. New lower and upper bounds for on-line scheduling. *Operations Research Letters*, 16:221–230, 1994.

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

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

[39] E. Davis and J. M. Jaffe. Algorithms for scheduling tasks on unrelated processors. *Journal of the ACM*, 28:721–736, 1981.

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

[41] T. Ebenlendr and J. Sgall. Optimal and online preemptive scheduling on uniformly related machines. Manuscript.

[42] J. Edmonds. Scheduling in the dark. *Theoretical Computer Science*, 235:109–141, 2000.

[43] J. Edmonds, S. Datta, and P. W. Dymond. Tcp is competitive against a limited adversary. In *Proc. 15th Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 174–183. ACM, 2003.

[44] J. Edmonds and K. Pruhs. Multicast pull scheduling: when fairness is fine. *Algorithmica*, 36:315–330, 2003.

[45] J. Edmonds and K. Pruhs. A maiden analysis of longest wait first. In *Proc. 15th Symp. on Discrete Algorithms (SODA)*. ACM/SIAM, 2004.

[46] L. Epstein. A note on on-line scheduling with precedence constraints on identical machines. *Information Processing Letters*, 76:149–153, 2000.

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

[48] L. Epstein and L. M. Favrholdt. Optimal non-preemptive semi-online scheduling on two related machines. In *Proc. 27th Symp. on Mathematical Foundations of Computer Science (MFCS)*, volume 2420 of *Lecture Notes in Computer Science*, pages 245–256. Springer, 2002.

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

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

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

[52] U. Faigle, W. Kern, and G. Turán. On the performane of online algorithms for partition problems. *Acta Cybernetica*, 9:107–119, 1989.

[53] A. Feldmann, M.-Y. Kao, J. Sgall, and S.-H. Teng. Optimal online scheduling of parallel jobs with dependencies. *Journal of Combinatorial Optimization*, 1:393–411, 1998.

[54] A. Feldmann, B. Maggs, J. Sgall, D. Sleator, and A. Tomkins. Competitive analysis of call admission algorithms that allow delay. Technical Report CMU-CS-95-102, Carnegie-Mellon University, 1995.

[55] A. Feldmann, J. Sgall, and S.-H. Teng. Dynamic scheduling on parallel machines. *Theoretical Computer Science*, 130:49–72, 1994.

[56] A. Fiat and G. J. Woeginger, editors. *Online Algorithms: The State of the Art.* Springer, 1998.

[57] A. Fiat and G. J. Woeginger. On-line scheduling on a single machine: Minimizing the total completion time. *Acta Informatica*, 36:287–293, 1999.

[58] R. Fleischer and M. Wahl. On-line scheduling revisited. *Journal of Scheduling*, 3:343–353, 2000.

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

[60] G. Galambos and G. J. Woeginger. An on-line scheduling heuristic with better worst case ratio than Graham's list scheduling. *SIAM Journal on Computing*, 22:349–355, 1993.

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

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

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

[64] K. S. Hong and J. Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computing*, 41:1326–1331, 1992.

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

[66] B. Kalyanasundaram and K. Pruhs. Fault-tolerant scheduling. In *Proc. 26th Symp. Theory of Computing (STOC)*, pages 115–124. ACM, 1994.

[67] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47:214–221, 2000.

[68] B. Kalyanasundaram and K. Pruhs. Minimizing flow time nonclairvoyantly. *Journal of the ACM*, 50:551–567, 2003.

[69] B. Kalyanasundaram, K. R. Pruhs, and M. Velauthapillai. Scheduling broadcasts in wireless networks. *Journal of Scheduling*, 4:339–354, 2001.

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

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

[72] D. R. Karger, S. J. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. *Journal of Algorithms*, 20:400–430, 1996.

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

[74] S. Leonardi. A simpler proof of preemptive flow-time approximation. In *Approximation and On-line Algorithms*, Lecture Notes in Computer Science. Springer, 2003.

[75] S. Leonardi and A. Marchetti-Spaccamela. On-line resource management with applications to routing and scheduling. *Algorithmica*, 24:29–49, 1999.

[76] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proc. 29th Symp. Theory of Computing (STOC)*, pages 110–119. ACM, 1997.

[77] T. Matsumoto. Competitive analysis of the Round Robin algorithm. In *Proc. 3rd International Symp. on Algorithms and Computation (ISAAC)*, volume 650 of *Lecture Notes in Computer Science*, pages 71–77. Springer, 1992.

[78] J. McCullough and E. Torng. SRPT optimally uses faster machines to minimize flow time. In *Proc. 15th Symp. on Discrete Algorithms (SODA)*. ACM/SIAM, 2004.

[79] P. Mireault, J. B. Orlin, and R. V. Vohra. A parametric worst case analysis of the LPT heuristic for two uniform machines. *Operations Research*, 45:116–125, 1997.

[80] R. Motwani, S. Phillips, and E. Torng. Non-clairvoyant scheduling. *Theoretical Computer Science*, 130:17–47, 1994.

[81] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize avarage strech. In *Proc. 40th Symp. Foundations of Computer Science (FOCS)*, pages 433–443. IEEE, 1999.

[82] E. Naroska and U. Schwiegelshohn. On an on-line scheduling problem for parallel jobs. *Information Processing Letters*, 81:297–304, 2002.

[83] C. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, pages 163–200, 2002.

[84] K. Pruhs and P. Uthaisombut. A comparison of multicast pull models. In *Proc. 10th European Symp. on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 808–819. Springer, 2002.

[85] J. F. Rudin III. *Improved Bound for the Online Scheduling Problem.* PhD thesis, The University of Texas at Dallas, 2001.

[86] J. F. Rudin III and R. Chandrasekaran. Improved bound for the online scheduling problem. *SIAM Journal on Computing*, 32:717–735, 2003.

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

[88] S. Seiden. Barely random algorithms for multiprocessor scheduling. *Journal of Scheduling*, 6:309–334, 2003.

[89] S. Seiden, J. Sgall, and G. J. Woeginger. Semi-online scheduling with decreasing job sizes. *Operations Research Letters*, 27:215–221, 2000.

[90] S. S. Seiden. More multiprocessor scheduling with rejection. Technical Report Woe-16, TU-Graz, 1997.

[91] S. S. Seiden. *Randomization in On-line Computation.* PhD thesis, University of California, Irvine, 1997.

[92] S. S. Seiden. Randomized online multiprocessor scheduling. *Algorithmica*, 28:173–216, 2000.

[93] S. S. Seiden. Preemptive multiprocessor scheduling with rejection. *Theoretical Computer Science*, 262:437–458, 2001.

[94] 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.

[95] J. Sgall. Randomized on-line scheduling of parallel jobs. *Journal of Algorithms*, 21:149–175, 1996.

[96] J. Sgall. A lower bound for randomized on-line multiprocessor scheduling. *Information Processing Letters*, 63:51–55, 1997.

[97] J. Sgall. On-line scheduling. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms: The State of the Art*, pages 196–231. Springer, 1998.

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

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

[100] T. Tichý. Randomized on-line scheduling on 3 processors. *Operations Research Letters*, 2003. To appear.

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