Alcom-FT Technical Report Series
ALCOMFT-TR-03-109

Scheduling to minimize flow time metrics *

Luca Becchetti
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
becchett@dis.uniroma1.it

Stefano Leonardi
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
leon@dis.uniroma1.it

Alberto Marchetti-Spaccamela
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
marchetti@dis.uniroma1.it

Guido Schäfer
Max-Planck Institut für Informatik
Saarbrücken, Germany
schaefer@mpi-sb.mpg.de

Tjark Vredeveld
Konrad-Zuse-Zentrum für Informationstechnik
Berlin, Germany
vredeveld@zib.de

Abstract

Scheduling on multiple machines is a classical problem in scheduling and dates back to the 60's. In this survey we review work on scheduling to minimize average flow time or related metrics, on single and parallel machines. We consider an abstract model, in which a set of jobs is presented on line to a set of identical machines. Each job has a processing time and has to be processed, possibly over a non-continuous interval, for an overall amount of time equal to its processing time. All techniques we shall present have been initially applied to average flow time, while some of them have also been used to prove competitiveness results for average stretch and weighted flow time. For this reason, our focus will mainly be on average flow time, while we will only provide an overview of results and main open issues for average stretch and weighted flow time.

1. Introduction

Scheduling on multiple machines is a classical problem in scheduling and dates back to the 60's (see for example the seminal work of R. Graham [11, 12]), at the time being motivated by the design of efficient CPU scheduling algorithms for multi-tasking operating systems. More recent motivating applications include Data service to mobile users over multiple wireless channels [13] and service request scheduling in replicated server systems [14]. In this survey, we review work on scheduling to minimize average flow time or related metrics, on single and parallel machines. We consider an abstract model, in which a set of jobs is presented on line to a set of identical machines. Each job has a processing time and has to be processed, possibly over a non-continuous interval, for an overall amount of time equal to its processing time. A high-level goal of the scheduling algorithm, for instance in the motivating applications sketched above, is minimizing average end-user latency. This is typically abstracted as the problem of scheduling jobs on the multiprocessor system to optimize some metric.

The metric to be optimized depends on the notion of responsiveness. Traditionally, the time a job spends in the system, namely, the *flow time* (or response time) of a job, defined as the difference between its completion time and its arrival time, has been considered a suitable measure. More recently, *stretch*, namely, the factor by which a request is slowed down relative to the time it takes on an unloaded system, has emerged as a more suitable measure. Formally, the stretch of a job is the ratio between its response time and its processing time. Intuitively, it relates the users' waiting time to their demands and measures the quality of service provided to a job in proportion to its requirements. The performance of a system is measured as the average performance per user. Average stretch (equivalently, total stretch) may be a better indicator of the system performance than

*Partially supported by the EU project AMORE grant HPRN-CT-1999-00104, IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT), IST-2000-14084 (APPOL), IST-2001-33555 (COSIN) and by the MIUR Project "Resource Allocation in Real Networks".

average response time, because it reflects per user expectation better, and it is a good indicator of the overall system load — a system with a large average stretch is necessarily overloaded while one with a large average response time need not be overloaded, since a single large job would result in large response times. A generalization of the above metrics is *weighted flow time*, In this case, each job comes with a positive weight and the objective function to minimize is the weighted sum of the flows of the jobs.

Considerable attention has been devoted to this problems in the past, but only in the second half of the 90's were major results achieved by researchers. The amount of work done since then is considerable and a taxonomy may proceed along different axes. The main ones are the metric considered, the power of the algorithm, for instance in terms of its knowledge of the input, last but not least, the analysis techniques used. As to this last point, some of them are general, while others are more specifically tailored for the abstract model we consider.

Our choice has been to present existing work according to the metric considered, while the general aspects related to the power of the algorithm and the general techniques available are presented in the first section.

Our focus is on average flow time, while we will only provide an overview of results and main open issues for average stretch and weighted flow time. The reason of this choice is that all techniques we shall present have been initially applied to the analysis of average flow time, while some of them have also been used to prove competitiveness results for other metrics. We briefly review these in Section 4. The paper is organized as follows. Section 2 describes theoretical framework, Section 3 reviews the work concerning average flow time, while Section 4 briefly summarizes the main results concerning other flow related metrics.

2. Model

In this section, we describe the abstract model we consider and provide the theoretical framework we'll refer to in the next sections. We are given a set $\mathcal{J}$ of n jobs and a set of m identical machines. Each job is described by a pair (r_j, p_j), where r_j is the *release time* of job j and p_j its *processing time* (in the sequel we may sometimes use the words *duration* or *size*). Each machine can process at most one job at any time and a job cannot start being processed before its release time. If $x_j(t)$ denotes the overall amount of time job j has been processed up to time t, the *remaining processing time* of j at time t is $p_j - x_j(t)$. Let C_j be the *completion time* of job j, $F_j = C_j - r_j$ is then the *flow time* (also called *response time*) of job j, while the *stretch* of job j upon completion is $\frac{F_j}{p_j}$. The *average flow time* of a schedule is defined as $\frac{1}{n}\sum_{j\in\mathcal{J}}\frac{F_j}{p_j}$, while the *average stretch* is $\frac{1}{n}\sum_{j\in\mathcal{J}}\frac{F_j}{p_j}$. A generalization of both these metrics is *weighted flow time*. In this case, each job j comes with a positive weight w_j and the objective function is $\sum_{j\in\mathcal{J}} w_j F_j$. In general, considered any of the metrics defined above, the goal is to minimize it for every instance of the problem. In the off-line version of the problem all jobs (for each job, its release time, processing time and possibly its weight) are known in advance, while in the on-line setting jobs arrive over time and at any instant, the algorithm has to take its decisions without knowledge of jobs that will be released in the future.

2.1. Further assumptions on the scheduling algorithm

The general setting described above is assumed in all the work we review in this survey. Differences concern the capabilities and the knowledge of the algorithm. In particular the following aspects may play a crucial role:

Knowledge of the algorithm. A *claivoyant* algorithm learns the size of a job at release time, while a *nonclaivoyant* one learns the size of a job only at completion time. Recall that all algorithms considered here are on line and thus learn the existence of a job only at release time.

Preemption. When *preemption* is allowed, a job that is being processed may be interrupted and resumed later after processing other jobs in the interim. As shown further, preemption is necessary to design efficient algorithms in the framework we consider.

Migration. When preemption is allowed, a job that has been preempted may resume execution on the same or on a different machine. When this is possible, we say that the model allows *migration*, otherwise we speak of nonmigratory scheduling algorithms. Migration may improve performance of the system in the model we consider. Still, migration may be expensive in real systems. For example, moving a job from one machine to another in a distributed server system calls for communication and data transfer between machines, which is time consuming. As a consequence, attention has been devoted to algorithms that do not have this feature.

Randomization. The use of randomization may improve average performance of the algorithm, reducing the impact of adverse input sequences that may cause poor worst case performance. Randomization may also be a way to provide an indirect proof of the good average behaviour of an algorithm.

2.2. Tools of analysis

The general technique used to analyze on line algorithms is competitive analysis [23, 8]. Extensions to the basic technique have been proposed, some of which apply to the framework we describe. The basic definitions, necessary in order to make this survey self-contained, are provided here, while the reader is referred to the bibliography for an in-depth description.

Competitiveness. Competitive analysis measures the quality of an on-line algorithm by comparing its performance to that achieved by an optimal off-line algorithm that has full knowledge of the future. More formally, an on line algorithm A is c-competitive with respect to the objective function $\mathcal{F}$ if, for any input $\mathcal{J}$,

$$\mathcal{F}^A(\mathcal{J}) \leq c\mathcal{F}^{OPT}(\mathcal{J}),$$

where $\mathcal{F}^A(\mathcal{J})$ and $\mathcal{F}^{OPT}(\mathcal{J})$ respectively denote the values of $\mathcal{F}$ achieved by A and by the off-line optimum on input instance $\mathcal{J}$. The instance $\mathcal{J}$ will be omitted in the following when clear from the context.

If A is a randomized algorithm, the performance of A is compared to that achieved by an *adversary*, that both decides the input instance and executes optimally on it. Since A takes random decisions, there is some freedom as to the power of the adversary. To the purpose of this survey, it is enough to define the weakest notion of adversary. In this case, the performance of the randomized algorithm is compared to that of an *oblivious adversary*, that decides the input sequence in advance, having knowledge of the probability distribution used by the algorithm, but without any knowledge of the random choices of the algorithm. In our specific case, the oblivious adversary has to decide, before the randomized algorithm begins execution, the number of jobs n of the sequence together with the release time and the processing time of each job. The adversary is charged with the optimal value of the objective function for the sequence.

More formally, a randomized on line algorithm A is c-competitive against an oblivious adversary if for any input J,

$$\mathbf{E}_\sigma[\mathcal{F}_\sigma^A(J)] \leq c\mathcal{F}^{OPT}(J),$$

where the expectation is taken over any possible outcome of the random choices σ of the algorithm for the input instance J.

More powerful, *adaptive* adversaries, are possible. It is enoguh to say that, in general, these do not have to decide the whole input sequence in advance. Rather, they release the input instance piece by piece over time, fixing every piece according to the decisions taken so far by the randomized algorithm. This qualitative explanation is sufficient to the purpose of this survey. In-depth discussion about adversaries and their power is provided in [8]. In this survey, we will explicitely point out when a randomized on-line algorithm is compared against an adaptive adversary. If this is not specified, it is understood that we assume an oblivious one.

Competitive analysis may provide pessimistic estimation of the performance of an algorithm, due to the presence of pathological bad instances that rarely occur in practice. For this reason, extensions to the basic technique have been proposed, in order to cope with such discrepancies between theory and praxis. Here, we describe two of them that have found application in the analysis of on line algorithms for multiprocessor scheduling, namely, *resource augmentation* and *smoothed competitive analysis*.

Resource augmentation. One way to cope with the presence of pathological instances may be to assume an (moderate) extra amount of resources for the on-line algorithm, with respect to the off-line optimum. In this case, the performance of the augmented on-line algorithm is compared to that of the off-line optimum that has no extra resources. In the setting we are interested in, extra resources means faster and/or more machines.

Smoothed competitiveness. The basic idea of smoothed competitive analysis is to randomly perturb the input instances and to compare the average performance of the algorithm and the off-line optimum with respect to the perturbed instances. Intuitively, if pathological instances are sparse in the instance space, this should translate into a good smoothed competitiveness. Since the input instance is smoothened according to some probability distribution f, we define the smoothed competitive ratio as the supremum over all input instances of the expected ratio between the algorithm and the optimal cost. More formally, we define

$$c = \sup_{\bar{\mathcal{J}}} \mathbf{E}_{\mathcal{J} \in N_f(\bar{\mathcal{J}})} \left[\frac{\mathcal{A}_{\mathcal{J}}}{\mathcal{OPT}_{\mathcal{J}}} \right],$$

where the supremum is taken over all input instances $\bar{\mathcal{J}}$ and the expectation is taken with respect to all instances $\mathcal{J}$ chosen according to f from the neighborhood $N_f(\bar{\mathcal{J}})$ of $\bar{\mathcal{J}}$, i.e., all instances I that are obtainable by smoothening the input instance $\bar{\mathcal{J}}$ according to f.

3. Minimizing average flow time

It is a well-known result [2] that the Shortest Remaining Processing Time heuristic (SRPT in the sequel) is optimal to minimize the average flow time on a single machine. SRPT is a an on-line, preemptive, migratory algorithm that, at each time, processes the m jobs with smallest remaining processing times. SRPT is a clairvoyant algorithm, since it needs to know the processing times of jobs. Only in the second half of the 90's did researchers achieve new results in this area of research. The first result was an analysis of the essential role played by preemption for this problem, even if the off line case. In particular, in [17] it is proved that even for one machine no non-preemptive algorithm can achieve an approximation factor $O(n^{1/2-\epsilon})$ unless $NP = P$. This also holds in the multiple machines case. In this case, no non-preemptive algorithm can achieve an approximation factor $O(n^{1/3-\epsilon})$ for $m > 1$ identical machines [20]. The first breakthrough was a complete analysis of SRPT for m machines, due to Leonardi and Raz [20]. The authors proved that SRPT achieves a logarithmic competitive ratio for the multiple machine case, showing a tight bound $\Theta(\log(\min\{n/m, P\}))$ on $m > 1$ machines with n jobs, where P denotes the ratio between the processing time of the longest and the shortest jobs. Whether better approximation ratios may be achieved in the off-line setting remains an open question.

A simplified analysis of SRPT on multiple machines is given in [19].

It is useful to give an overview of the analysis provided in [20], since the basic tools introduced there have also been used in most of subsequent work. Here, we shall just present the main concepts and ingredients, so as to provide the reader with an idea of the general approach used to prove competitiveness results in this framework. In order to do this, we need some definitions. We say that job j has *class* k if its remaining processing time falls in $[2^k, 2^{k+1})$. In the following, $\delta^A_{=k}(t)$ ($\delta^A_{\leq k}(t)$) denotes the number of jobs of class k (at most k) in SRPT's schedule at time t, while $V^A_{=k}(t)$ ($V^A_{\leq k}(t)$) denotes their volume, i.e their overall remaining processing time at time t. $\delta^{OPT}_{=k}(t)$, $\delta^{OPT}_{\leq k}(t)$, $V^{OPT}_{=k}(t)$ and $V^{OPT}_{\leq k}(t)$ denote the analogous values for OPT at t. We also set $\Delta_{=k}V(t)$ ($\Delta_{\leq k}V(t)$) to denote the difference $V^A_{=k}(t) - V^{OPT}_{=k}(t)$ ($V^A_{\leq k}(t) - V^{OPT}_{\leq k}(t)$). The absence of subscripts refers to the contribution of all jobs in the system. The first, basic result that is used in the analysis is the following (see for instance [20])

Fact 1. *For any schedule S,*

$$F^S = \int_t \delta^S(t)dt,$$

Fact 1 shows that the crucial aspect to consider is the relationship existing between $\delta^{OPT}(t)$ and $\delta^A(t)$, for every t. This is done by first bounding $\Delta V_{\leq k}(t)$, i.e. the volume difference between SRPT and OPT, restricted to the subset of jobs of class at most k. It is possible to prove that

Lemma 1. *For any t such that all machines are busy at time t,*

$$\Delta V_{\leq k}(t) \leq m2^{k+1}.$$

The proof of the Lemma uses the fact that SRPT always gives precedence to the jobs that have smallest remaining processing time. Lemma 1 allows to relate the number of pending jobs in SRPT's schedule to that in OPT's schedule. The following lemma holds:

Lemma 2. *For t, such that all machines are busy at t, for* $k_{min} \leq k_1 \leq k_2 \leq k_{max}$, $\delta^A_{\geq k_1, \leq k_2}(t) \leq m(k_2 - k_1 + 2) + 2\delta^{OPT}_{\leq k_2}(t)$.

Lemma 2 is a consequence of Lemma 1, from which it is derived using a volume argument. It is easy to prove that the overall contribution to the total flow time due to the time instants in which less than m machines are busy, is bounded by $\sum_j p_j$, which in turn is a trivial lower bound to the optimal total flow time. From this fact and from Lemma 2 it is not hard to derive the first bound for SRPT, namely:

Theorem 1. $F^A = O((\log P)F^{OPT})$.

This is the core of the analysis provided in [20]. In order to derive the complete bound given in the paper, it is necessary to evaluate the contribution of the term $m(k_2 - k_1 + 2)$. In particular, the authors study the contribution of this term over time instants in which all machines are busy and they are able to prove that this contribution is $O(\log(n/m)F^{OPT})$. This second part of the analysis is beyond the scope of this survey.

Migration. The role of migration in minimizing total flow time has been investigated in [1]. Here, the authors propose a non-migratory algorithm that has slightly worse performance than SRPT. This algorithm is named AALR after its inventors. AALR holds a pool of jobs that have not yet been assigned to a machine and a stack for each machine, comprising jobs that have already begun processing on that machine and will be processed up to completion on that machine. The jobs in a machine's stack are ordered by increasing remaining processing time. The job on the top of the stack is scheduled for processing. Roughly speaking, a job is moved from the pool to a stack if its processing time is smaller than the remaining processing time of every other job in the stack by at least a factor 2, or if the stack is empty.

The authors of [1] prove that this algorithm is $O(\min\{\log n, \log P\})$-competitive, which is asymptotically tight on a constant number of machines. The analysis uses the same tools as the one in [20], but it is more involved, the most difficult part being bounding the contribution to the total flow time due to jobs in the algorithm's pool.

Non-clairvoyance. Recently, many efforts have been devoted to the non-claivoyant model. This case is of both theoretical and practical interest, since assuming knowledge of jobs' sizes may be unrealistic in many applications. This case is more difficult to handle. In particular, Motwani et al. [21] prove that no deterministic non-clairvoyant algorithm can be competitive at all, even with preemption. They also prove that every non-clairvoyant, preemptive, randomized algorithm is $\Omega(logn)$-competitive, even in the case of a single machine and against the oblivious adversary. This problem has been addressed along two lines of attack.

Some contributions propose randomized algorithms that are asymptotically optimal or sub-optimal, while others provide a theoretical explanation of why certain deterministc heuristics behave well in practice.

To the first group belong the works of [15] and [3]. In [15], the authors consider the Multi-Level Feedback algorithm (MLF). It is at the very basis of the most successful algorithms used in practice for processor scheduling in time sharing multitasking operating systems. The basic idea of MLF is to organize jobs into a set of queues. Each job is processed for 2^i time units if in queue Q_i, before being promoted to queue Q_{i+1} if not completed. At any time, MLF processes the job at the front of the lowest queue. Being deterministic, the worst-case performance of MLF is poor. It may be useful to analyze the structure of a typical worst case sequence, in order to better understand the ideas behind the randomized strategies that have been proposed. One typical worst case sequence is the following: n jobs $0, \ldots, n-1$ are released. All jobs have processing time $1+\epsilon$, where $\epsilon > 0$ is suitably small. For $j = 0, \ldots, n-1$, $r_j = j$. It is easy to prove that at time $n/2$, MLF has $\Omega(n)$ jobs left, while the algorithm that processes jobs in FIFO order always has $O(1)$ jobs in the system. Since the overall schedule's length is $\Theta(n)$, $F^{FIFO} = O(n)$, while $F^{MLF} = \Omega(n^2)$. This example shows that the poor performance of MLF (and, in general, of deterministic algorithms) is due to the fact that, at some point in the schedule, we have a large amount of jobs that are almost finished, but that are preempted to give precedence to other, possibly larger, jobs entering the system. Randomization is used to ensure that these pathological events occur with small probability. The algorithm proposed in [15] is the natural extension of MLF, in which the time a job is processed in any queue Q_i is chosen at random and independently of other jobs, roughly within interval $[2^i, 2^{i+1})$ (for the exact algorithm see [15]). For this reason, it is called RMLF. We point out that the obvious uniform distribution may lead to poor performance in this case (see [3] for a counterexample). Instead, the authors use an exponential distribution, that also depends on the number of jobs released. They prove that RMLF achieves a competitive ratio $O(\log\log n \log n)$ against the stronger, adaptive adversary. In [3], the authors extend RMLF to the multiple machines case and prove that this algorithm has competitive ratio $O(\log n \min\{\log P, \log(n/m)\})$ against the oblivious adversary. For the specific case of 1 machine, they prove that RMLF is actually $O(\log n)$-competitive. Recalling [21], this last result is tight. It is interesting to observe that the competitive ratio is basically the product of two contributions: the first is the price of non-clairvoyance, while the second is only present in the multiple machine case and is due to the fact that the algorithm might assign jobs in an unbalanced way, thus having machines work less than in the optimal schedule. This problem is also present in the clairvoyant case and is the reason for the lower bound of [20].

In both [15] and [3], the main problem is to give an estimation of the number of jobs that are almost finished in the system at any instant, with respect to the total. In order to do this, both [15] and [3] use a concept of *unlucky job*. Roughly, job j is unlucky if $p_j = 2^i + \epsilon$, but j is completed in queue Q_i, otherweise j is *lucky*. Intuitively, the event of a large fraction of unlucky jobs in the system at some time t corresponds to a pathological instance as the one described above. Of course, the fraction of unlucky jobs in the system at time t is a random variable. Furthermore, although the processing times of jobs in the queues are chosen independently of each other, the event of a job being unlucky at some time t may depend on the other jobs, essentially because we are considering a specific instant. This makes the analysis complicated. In order to tackle this problem, the authors of [15] and [3] choose different proof strategies. In the first case, the authors roughly prove that i) with high probability, it is never the case that, at any point of RMLF's schedule, $\Omega(\log\log n \log n)$ unlucky jobs are in the system and ii) if i) holds, than $\delta^{DMLF}(t) = O(\log\log n \log n)\delta^{OPT}(t)$, where DMLF denotes a deterministic execution of RMLF, for a specific set of choices done by the algorithm.

In [3], the authors do not use any high probability argument. They prove that i) the expected number of unlucky jobs in the system over the entire schedule of RMLF is $O(\log n)$, ii) at any time t, in the average, a constant fraction of the lucky jobs in the system has a sufficiently large fraction (roughly logarithmic) of the initial processing time and iii) considered the generic, deterministic execution of RMLF, $\delta^{RMLF}(t) = O(\log n(m\ k(t))\delta^{OPT}(t)$ for any time t, where $k(t)$ is related to the number of non-empty queues at time t in RMLF. Proving ii) requires a non trivial analysis of the probability space, while iii) uses volume arguments, refining some tools of analysis presented in [20].

We mention the work presented in [16] and [5], where the authors provide a theoretical explanation of the effectiveness of some heuristics in practice.

Resource augmentation. In [16], the authors use resource augmentation to prove that a simple heuristic behaves well in practice. They provide evidence of this fact indirectly, by proving that it is enough to equip the algorithm with slightly more resources than OPT to have an almost optimal behaviour. More in detail, they consider the Shortest Elapsed Time First heuristic (denoted by A for brevity), which always runs the job that has been processed the least to date. In particular, they prove that $F^A \leq (1+\epsilon)F^{OPT}$ on a single machine for any $\epsilon > 0$, if A uses a $(1+\epsilon)$-fast machine, while OPT is equipped with a unit speed one. They also prove that this is equivalent to providing the algorithm with two processors, a unit speed one and an ϵ-fast one.

Smoothed competitive analysis. In [5] the authors introduce smoothed competitiveness and apply it to the analysis of the Multi-Level Feedback (MLF) algorithm described above.

The basic idea is to perturb an initial instance by a sharp random distribution centered around the initial instance, proving that the expected total flow time achieved by MLF in this neighbourhood is related to the optimum by a factor that decreases as the variance of the perturbation increases. When this holds, it is proof of the fact that worst case instances are scattered in the instance space. More in detail, the initial instance is in our case given by n pairs $(r_1, p_1), \ldots, (r_n, p_n)$. Actually, the authors show that the results obtained do not depend on the choice of $r_1, \ldots, r_n$, hence, in the sequel, we are interested in the n values $p_1, \ldots, p_n$. These are assumed to be integers in the range $[1, 2^K]$, if K bits are used to represent processing times of the jobs. The basic idea is to randomly change the least significant k bits and consider the average value of F^{MLF}/F^{OPT} on the set of instances thus obtained. The authors show that this ratio is $O(2^{K-k}/\alpha)$, where $\alpha = (\sigma/2^k)^2$, with σ the standard deviation of the distribution. They also prove an $\Omega(2^{K-k})$-smoothed lower bound for any deterministic algorithm.

A direct consequence of this results is also the first average case analysis of MLF. For instance, these results show a constant expected ratio of the total flow time of MLF to the optimum under several distributions, including the uniform one.

4. Other metrics

In this section, we briefly review work done on the minimization of average stretch and weighted flow time.

Average stretch. The problem of minimizing average stretch has only been considered in the clairvoyant case. In [22], the authors prove that SRPT is 2-competitive to minimize average stretch on a single machine, while it is $O(1)$-competitive on multiple machines. They also prove constant lower bounds (1.036 and 1.186 respectively) on the competitive ratio of any on line algorithm.

In [6], the authors prove that the AALR algorithm proposed in [1] is also $O(1)$-competitive to minimize average stretch on multiple machines. This means that AALR performs well for both metrics simultaneously. When migration is allowed, a similar phenomenon occurs, as illustrated by the good performance of SRPT for both average flow time and average stretch. The competitive analysis of AALR proposed in [6] does not make any assumptions on the behaviour of the optimal off line schedule, which is therefore allowed to migrate jobs. This shows that migration is not necessary to be $O(1)$-competitive in minimizing average stretch. On the other hand, the authors show that preemption is essential. More in detail, they prove an $\Omega(\Delta)$ lower bound on the competitive ratio of any non-preemptive, randomized on-line algorithms, where Δ is the ratio between the maximum and the minimum processing time of a job.

More recently, the first Polynomial Time Approximation Scheme for average stretch was proposed [7].

Weighted flow time. Weighted flow time generalizes both average flow time and average stretch. In general, it well describes situations in which one has to schedule activities, each of which has an associated penalty that is proportional to the time elapsed between start and completion. It is known that the problem is strongly NP-hard even in the single machine case [18]. Only recently were the first approximation results for this metric proposed. In particular, in [10], the authors present results for both the off-line and the on-line problem. As to the former, they propose a $(2+\epsilon)$-approximation algorithm that works in quasi-polynomial time, for any $\epsilon > 0$. This result was then improved in [9], but the challenge of devising a PTAS for this problem is still open.

As to on-line algorithms, in [10] the authors propose an $O(\log^2 P)$-competitive algorithm, P being the ratio between the maximum and the minimum processing time. They also prove that no deterministic (respectively randomized), on-line algorithm, can achieve a competitive ratio better than 1.618 (respectively $4/3$). They also prove a very interesting, negative result, for parallel machines. Namely, they prove that no on-line, randomized algorithm can be $o(\min\{\sqrt{P}, \sqrt{W}, (n/m)^{1/4}\})$-competitive, P (W) being the ratio between the maximum and the minimum processing time (weight) of jobs.

A very natural heuristic for minimizing weighted flow time is Highest Density First: at any time t, schedule the job j such that $w_j/(p_j - x_j(t))$, i.e. the ratio between its weight and remaining processing time, is minimum. Unfortunately, it is not hard to show that this heuristic can have extremily poor performance. In [4], the authors prove that HDF is almost optimal on one machine when the algorithm runs on a slightly faster machine than the optimum. More precisely, they prove that HDF is $(1 + 1/\epsilon)$-competitive if it is equipped with a $(1 + \epsilon)$-fast machine. They also prove that the same algorithm is $(2 + 2\epsilon)$-competitive on parallel machines, if these are $(1 + \epsilon)$-fast.

5 Open problems

Our knowledge of average flow time is now rather well settled. One open question concerns the approximability of the problem in the off-line case.

The non-clairvoyant case is still open for average stretch. It would be interesting to explore the role of randomization in this setting, in particular to understand whether or not it may be beneficial to achieve competitiveness results. It might also be of interest to investigate the effectiveness of heuristics such as MLF, at least in the sense of smoothed competitive analysis. Observe that the results obtained in [4] using resource augmentation also extend to average stretch.

Minimizing weighted flow time remains the problem with the biggest open issues, the most important being whether or not it admits a constant competitive (hence approximation) algorithm. Also interesting is whether or not it admits a PTAS in the off-line setting.

References

[1] B. Awerbuch, Y. Azar, S. Leonardi, and O. Regev. Minimizing the flow time without migration. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 198–205, 1999.

[2] K. Baker. *Introduction to Sequencing and Scheduling*. Wiley, 1974.

[3] L. Becchetti and S. Leonardi. Non-clairvoyant scheduling to minimize the average flow time on single and parallel machines. In *Proceedings of the Thirty-Third Annual ACM Symposium on the Theory of Computing*, pages 94–103, 2001.

[4] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, and K. Pruhs. Online weighted flow time and deadline scheduling. In *APPROX: International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 36–47, 2001.

[5] L. Becchetti, S. Leonardi, A. Marchetti-Spaccamela, and G. Schäfer. Smoothed competitive analysis for the multilevel feedback algorithm. unpublished manuscript, 2002.

[6] L. Becchetti, S. Leonardi, and S. Muthukrishnan. Scheduling to minimize average stretch without migration. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 548–557, 2000.

[7] M. Bender, S. Muthukrishnan, and R. Rajaraman. Improved algorithms for stretch scheduling. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)*, pages 762–771, 2002.

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

[9] C. Chekuri and S. Khanna. Approximation schemes for preemptive weighted flow time. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC-02)*, pages 297–305, 2002.

[10] C. Chekuri, S. Khanna, and A. Zhu. Algorithms for minimizing weighted flow time. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing: Hersonissos, Crete, Greece, July 6–8, 2001*, pages 84–93, 2001.

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

[12] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[13] S. Hameed and N. Vaidya. Log-time algorithms for scheduling single and multiple channel data broadcast. In *International Conference on Mobile Computing and Networking (MOBICOM)*, pages 90–99, 1997.

[14] M. Harchol-Balter, M. Crovella, and C. Murta. On choosing a task assignment policy for a distributed server system. *Lecture Notes in Computer Science*, 1469:231–242, 1998.

[15] B. Kalyanasundaram and K. Pruhs. Minimizing flow time nonclairvoyantly. In *In Proceedings of the Thirty-Eight IEEE Symposium on Foundations of Computer Science*, pages 345–352, 1997. To appear in Journal of the ACM.

[16] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47(4):617–643, 2000.

[17] H. Kellerer, T. Tautenhahn, and G. J. Woeginger. Approximability and nonapproximability results for minimizing total flow time on a single machine. In *Proceedings of The Twenty-Eighth Annual ACM Symposium On The Theory Of Computing (STOC '96)*, pages 418–426, 1996.

[18] J. K. Lenstra, A. H. G. R. Kan, and P. Brucker. Complexity of machine schedulibg problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[19] S. Leonardi. A simpler proof of preemptive total flow time approximation on parallel machines. unpublished manuscript, 2002.

[20] S. Leonardi and D. Raz. Approximating total flow time on parallel machines. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, pages 110–119, 1997.

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

[22] S. Muthukrishnan, R. Rajaraman, A. Shaheen, and J. E. Gehrke. Online scheduling to minimize average stretch. In *40th Annual Symposium on Foundations of Computer Science: October 17–19, 1999, New York City, New York,*, pages 433–443, 1999.

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