On the Definition of "On-Line" in Job Scheduling Problems

Dror G. Feitelson and Ahuva W. Mu'alem
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Keywords: analysis of algorithms, performance evaluation, system dynamics

1 Introduction

The problem of scheduling jobs on resources (e.g. computations to processors) has been a fertile ground for theoretical research for many years [6]. Likewise, many practical scheduling systems have been designed, implemented, and used in production situations [3]. However, there seems to be little interaction between these two approaches, at least in the area of scheduling computational jobs submitted to a computer system on-line by multiple users.

One of the reasons may be that practitioners may feel that the system models used by theoreticians are too far removed from reality. We identify a certain aspect of system models in which this may indeed be the case, and show how to incorporate more realistic constraints into the common theoretical framework. These constraints turn out to yield significant improvements in the provable bounds on the performance of scheduling algorithms.

Specifically, we concentrate on the fact that real systems typically operate in an on-line setting. Theoreticians have long since identified two salient aspects of on-line systems, and have included them into their models: the fact that the future is not known, and the fact that there is a future at all (that is, not everything is available at the outset) [11]. However, they do not impose additional constraints. We make the observation that in real systems, and also in queueing models, a very important constraint does indeed exist: that the system not become saturated. In the theoretical framework, this constraint can be translated into a relationship between arrival times (a.k.a. release dates) and processing times. The rest of this paper investigates this relationship and its consequences.

2 What is "On-Line"

2.1 The Common Scheduling Model

It is common to classify scheduling problems using a notation of three fields $\alpha|\beta|\gamma$, where α specifies the machine environment, β the job characteristics, and γ the optimality criteria [6]. The job characteristics that are relevant for this discussion are the following (the machine model and optimality criteria are discussed below in Section 3):

p_j: The processing time of job j.
omitting this from β means that there are no restrictions.

r_j: A release date when job j becomes available for processing.
omitting this from β means that all jobs are available from the outset.

$size_j$: The number of processors needed by job j.
This is only relevant for scheduling of parallel jobs.
omitting it from β means that jobs require a single processor.

$pmtn$: preemption is allowed.
omitting it from β means that jobs must run to completion.

The r_j attribute is designed to capture part of the notion of being on-line: it reflects the fact that the jobs arrive over a certain span of time, and are not all available for scheduling at the outset.

The term "on-line" itself is used in the literature with a different meaning: it implies that the algorithm is non-clairvoyant about the future. Thus an on-line algorithm has to contend with each job as it arrives, without the priviledge of looking ahead into the future and taking future arrivals into consideration. An off-line algorithm, on the other hand, has this capability. Naturally, on-line algorithms are harder to design and generally yield poorer results.

It should be noted that including release dates and being on-line are orthogonal issues, and all four combinations are possible (Table 1). For example, bin packing algorithms can operate in a setting where bins represent time slots, and items have release dates which indicate the first bin in which they can be placed. At the same time we can have an off-line version that is allowed to look ahead into the list of items and reserve space for them, or an on-line version that cannot do so.

2.2 Real System Dynamics

It seems that the combination of having release dates and an on-line algorithm captures the notion of being "on-line" — jobs arrive unpredictably over time, and the algorithm has to contend with them with no prior information. However, we argue that this combination does not capture the dynamics of real systems correctly. We base this argument on a comparison with queueing systems, which are also used to analyze the performance of on-line systems.

	off-line	on-line
no release dates	• all jobs available at time 0 • all information known in advance *the most permissive model*	• all jobs available at time 0 • information obtained one job at a time
with release dates	• jobs become available at different times • all information known in advance	• jobs become available at different times • information obtained one job at a time *the most realistic model*

Table 1: *Possible combinations of model attributes.*

The problem stems from the fact that the theoretical framework assumes a finite set of jobs $j \in \{1..N\}$. Within this set, there are no restrictions on the release dates. They can be spread out evenly over time, they can be bunched together in several bursts, or they can all come in a single burst. The case of trivial release dates, where all jobs are available at time 0, is just a special case that also has to be handled. In particular, the algorithm is evaluated by how well it performs for the worst case.

The problem with this formulation is that the release dates are not related to the processing times in any way. Real systems, on the other hand, are subject to a stability constraint: on average, the requirements of arriving jobs must be less than the system's capacity. In queueing theory, this is expressed by the requirement that $\lambda/\mu < 1$, where λ is the arrival rate (in jobs per unit time) and μ is the service rate (in the same units) [7]. Moreover, the system is *open* — an infinite number of jobs may arrive, provided we observe the system for a long enough period, and the stability has to be maintained continuously. It is true that there is a certain probability for a high momentary load, but over time, it should even out.

Queueing theory is about predicting the *average* time jobs spend in the system, provided the system does not become saturated. If the system does become saturated, the queueing time may rise without bound, and this is taken to represent a situation that is not tolerable in a working system [7]. In the common theoretical model of scheduling, there is an illusion of stability because all jobs are indeed scheduled. However, this is not because of the constraints on arriving jobs, but because of the finite input (Fig. 1). Thus a common occurence is that initially the load increases and jobs wait, and later the system drains. In a real system this is not allowed (except for limited fluctuations).

As an informal example, consider the loading of trucks at a depot. The trucks arrive at unpredictable random times, and each takes a certain time to load. This matches the attributes of having release dates and being on-line. However, a real depot also has a limited

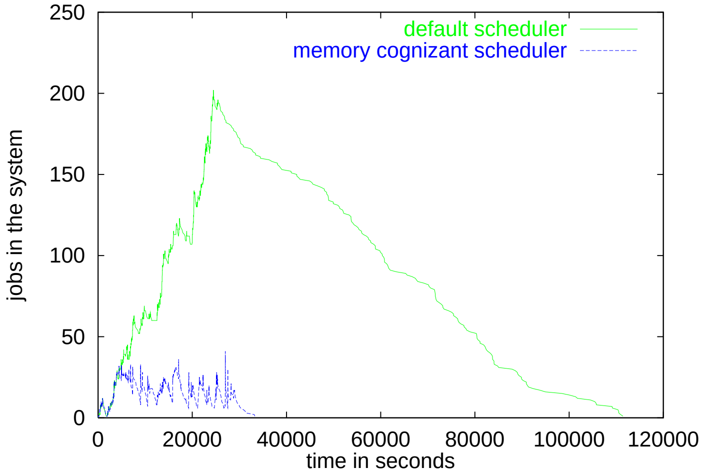

Figure 1: *Behavior of a system scheduling 1000 parallel jobs with two different schedulers. The offered load is 0.7 of the system's capacity. With the default scheduler the system saturates, and only completes the work because of the finite number of jobs involved. In effect, the system's capacity with this scheduler is less than 0.7 of the theoretical maximum, but considering the makespan or average completion time of these 1000 jobs may lead to the erroneous conclusion that the system is within its operating range. With the memory-cognizant scheduler, the system indeed remains stable. Figure courtesy of Anat Batat [1].*

parking lot, and can only serve a certain number of trucks at once (both being loaded and waiting their turn). Thus it can only operate if the truck arrivals are spread throughout the day. If they all come at once, the system will break down.

Note that this whole discussion does not involve the notion of being on-line. Being on-line only restricts information acquisition, and is not time-dependent. It is taken as self-evident that real systems are also on-line.

2.3 Expressing the Stability Constraint

The essence of the stability constraint is that we require the average resource consumption to be bounded by the average resource availability, and this should apply at all times. In particular, we must keep in mind the open system model, where any number of jobs may arrive.

In queueing theory, the stability constraint is indeed expressed by a requirement on job arrival and processing (service) times, using the expression $\lambda/\mu < 1$. As this reflects a constraint on average interarrival and processing times, it applies to any number of jobs, without any explicit time dependencies. However, porting this formulation directly to the framework commonly used in scheduling theory entails a sharp departure from that framework. We therefore prefer another formulation.

Our formulation is based on the conventional model with release dates. The twist is to place restrictions on the arrival rate by requiring the release dates to be correlated with the cummulative processing requirements of previous jobs. Specifically, if r_j is the release date for job j, and p_j is its processing time, then require that for all j the accumulated processing time of previous jobs "fits in" up to a constant:

$$\sum_{r_i \le r_j, i \ne j} p_i \le r_j + c.$$

The positive constant c allows for load fluctuations, especially at the beginning of the list, where the offered load may be higher than the system capacity (in the context of the trucks example, it reflects the number of trucks that can wait for loading in the parking lot).

To see the connection between this and the $\lambda/\mu < 1$ requirement from queueing theory, note that λ is the average arrival rate. If we index jobs according to their arrivals, that is $r_i \le r_j$ iff $i < j$, and look at a large enough number of jobs, then we get $\lambda \approx j/r_j$. On the other hand μ is the average service rate, i.e. the reciprocal of the average service time, and is therefore approximated by $\mu \approx j/\sum_{i \le j} p_i$. Putting these two expressions together we get $1 > \lambda/\mu \approx \frac{j/r_j}{j/\sum_{i \le j} p_i} = \sum_{i \le j} p_i/r_j$.

The same expression is readily extended to parallel jobs. Using the notation $size_j$ for job j's processor demand, and m for the total number of (homogeneous) processors in the system, the expression for stability becomes

$$\sum_{r_i \le r_j, i \ne j} \frac{p_i \cdot size_i}{m} \le r_j + c.$$

In this case the constant c is also used to allow a certain degree of overlap between jobs that do not use all the processors (that is, jobs for which $size_j < m$).

It should be noted that while this formula expresses the fact that the offered load on the system is bounded (on average) by its capacity, it does not guarantee that the system indeed stays stable. The system may saturate due to some other rigid constraints on the jobs (e.g. if for all jobs $size_j = \lceil m/2 \rceil + 1$, half of the processing power will be wasted), or due to inefficiency in the scheduling algorithm. Nevertheless, we will call it "the stability constraint".

3 Results with the New Constraint

In this section we show how the new stability constraints allows for simple analysis of scheduling problems and produces relatively tight bounds (Table 2). The constraint is denoted by the clause *stbl* in the β part of the problem classification. *stbl* obviously implies r_j, so we exclude it from the notation.

3.1 preliminaries

The models: We are given m identical processors and n independent parallel jobs to be scheduled. The jobs are independent in the sense that no precedence constraints are imposed.

model	competitive ratio for $\sum w_jC_j$	for C_{max}	for $\sum F_j$	section
$1 \mid stbl \mid \sum w_jC_j, C_{max}, \sum F_j$	2	2	1	3.2
$P \mid stbl \mid \sum w_jC_j, C_{max}, \sum F_j$	2	2	1	3.2
$P \mid stbl, size_j, pmtn \mid \sum w_jC_j, C_{max}$	3	3		3.3

Table 2: *Summary of the new results (the additive factors are omitted).*

Each job $j \in N = \{1, \ldots, n\}$ has the following three parameters: a positive processing time p_j, a number of processors needed for execution $size_j$, $1 \leq size_j \leq m$, and a release date $r_j \geq 0$ before which it cannot be processed. The release dates are subject to our new stability constraint: $\sum_{r_i \leq r_j, i \neq j} p_i \cdot size_i/m \leq r_j + c$. A machine can process at most one job at a time. All the $size_j$ processors are required to start the job j at the same starting time; any set of $size_j$ processors is proper for execution of job j. If we allow preemptions, the job may be preempted from time to time and later continued to run on a possibly different set of $size_j$ processors, without affecting its total processing time p_j. In addition, each job j may have a fourth parameter: a positive weight w_j. For simplicity we use the following notations: Let $r_j^{max} = \max_{k=1,\ldots,j} r_k$ and $p_j^{max} = \max_{k=1,\ldots,j} p_k$ denote the largest release date and the largest processing time among the jobs $1, \ldots, j$, respectively.

The case $m = 1$ represents the single machine model, and the case wherein $size_j = 1$ for each $j = 1, \ldots, n$ and $m \geq 1$ represents the identical parallel machine model. If we allow $size_j \geq 1$ this represents the multi-processor system model. Note that these assignments preserve the proper stability constraint that is needed in each model.

Let C_j denote the completion time of job j in a feasible schedule. Let F_j denote the flow time of job j in a feasible schedule, i.e., $F_j = C_j - r_j$. One possible goal for a scheduling algorithm may be to minimize $\sum_j C_j$ or $\sum_j w_jC_j$, or equivalently minimizing $(\sum_j C_j)/n$ or $(\sum_j w_jC_j)/n$. Another possible goal may be to minimize $\sum_j F_j$ or equivalently to minimize $(\sum_j F_j)/n$. A third objective is minimizing the makespan, the largest C_j, denoted by C_{max}. It seems that there are many more results in the literature for approximating total sum of completion times than for total sum of flow times, although at optimality these two objectives are equivalent.

An on-line scheduler is f-competitive if for any finite input sequence N of jobs the value obtained by the on-line scheduler on N divided by the best off-line scheduler on N is at most f (up to a constant additive factor). If a scheduler is f-competitive we sometimes say that the scheduler achieves a competitive ratio of f or that it has performance guarantee of f.

Consider an optimal schedule that minimizes $\sum_j (w_j)C_j$, $j \in N$, and let C_j^* denote the completion time of job j in it; thus $\sum_j (w_j)C_j^*$ denotes the optimal (weighted) sum of completion times value. We shall use the notation $\bar{C}_j$ to denote the value assigned to the job j by a relaxation. Therefore $\sum_j (w_j)\bar{C}_j$ is a lower bound on $\sum_j (w_j)C_j^*$. We use the notation $\tilde{C}_j$ to denote the completion time of job j in the schedule that our algorithm computes. (Respectively, $\sum_j F_j^*$ denotes the optimal sum of flow times value, and $\tilde{F}_j$ denotes the flow time of job j in the schedule that our algorithm computes.) In addition, we use C^*_{max} to denote the optimal makespan value.

Since the schedules computed in all our algorithms are based upon a list algorithm rule, we conclude that our algorithms have a constant performance guarantee for makespan [5].

3.2 Identical Parallel Machine Model ($P \mid stbl \mid \sum w_j C_j,\ C_{max},\ \sum F_j$)

In this case a very simple list-scheduling algorithm based on release dates suffices. This is simply the first-come-first-serve (FCFS) algorithm which we call here List-Schedule-by-r_j. Note that the following results apply as well for the single machine model, i.e., $m = 1$.

Assume w.l.o.g. that the jobs are indexed so that $r_1 \leq \cdots \leq r_n$, namely in the order of their of release dates. Let $\tilde{C}_j$ ($\tilde{F}_j$) denote the completion time (flow time) of job j in the schedule found by List-Schedule-by-r_j. Since List-Schedule-by-r_j is list based algorithm, we have:

Lemma 3.1 $\tilde{C}_j \leq r_j + \sum_{k=1}^{j-1} \frac{p_k}{m} + p_j$, *for each* $j = 1, \ldots, n$.

Thus we conclude immediately that algorithm List-Schedule-by-r_j is a 2-competitive algorithm with respect to makespan.

Lemma 3.2 *Algorithm* List-Schedule-by-r_j *is 2-competitive with respect to (weighted) sum of completions times.*

Proof: Applying the new stability constraint we have $\sum_{k=1}^{j-1} \frac{p_k}{m} \leq r_j + c$. Recall that the following trivial lower bound for sum of completion times holds: $\sum w_j(r_j + p_j) \leq \sum w_j C^*_j$.

Using the above Lemma we have for the unweighted case:

$$\frac{\sum \tilde{C}_j}{n} \leq \frac{\sum (r_j + p_j)}{n} + \frac{\sum r_j}{n} + c \ \leq\ 2 \cdot \frac{\sum C^*_j}{n} + c$$

similarly, for the weighted case we have:

$$\frac{\sum w_j \tilde{C}_j}{n} \leq \frac{\sum w_j (r_j + p_j)}{n} + \frac{\sum w_j r_j}{n} + wc \ \leq\ 2 \cdot \frac{\sum w_j C^*_j}{n} + wc$$

where $w = \sum w_j$. □

Note the interesting fact that since we list schedule the jobs in order of release dates we do not have to know the processing time of job j upon release; in fact, we do not need to know the processing times at all.

This non-preemptive and weighted result is pretty good compared to previous on-line algorithms for this model without imposing the new stability constraint. For example, Chekuri et al. achieved a 3-competitive algorithm for sum of completion times for the model $P|r_j|\sum C_j$ (that is, with no weights) [2]. However, their work requires p_j to be known.

Lemma 3.3 *Algorithm* List-Schedule-by-r_j *is 1-competitive with respect to sum of flow times.*

Proof: Recall that the following trivial lower bound for sum of flow times holds: $\sum p_j \leq \sum F_j^*$. From the stability constraint, upon release of the job j we have to wait at most c units of time to ensure that there is at least one idle machine, because there is at least one machine whose load (the sum of processing times of jobs that where assign to be scheduled on that machine by that time) is not more than the average load. Hence $\tilde{F}_j \leq p_j + c$, and

$$\frac{\sum \tilde{F}_j}{n} \leq \frac{\sum p_j}{n} + c \leq \frac{\sum F_j^*}{n} + c$$

□

3.3 Multi-Processor System Model ($P|stbl, size_j, pmtn|\sum w_j C_j, C_{max}$)

Let $\bar{C}_j = r_j + p_j$. Note that $\bar{C}_j$ is a trivial lower bound for the completion time of job j. Hence, $\sum_j (w_j)\bar{C}_j \leq \sum_j (w_j) C_j^*$. Algorithm Parallel-List-Schedule-by-$\bar{C}_j$ works as follows. Compute $\bar{C}_j$ for each $j = 1, \ldots, n$. Sort the jobs according to non-decreasing order of $\bar{C}_j$. Then use this permutation as an input to algorithm List-GG-PR.

List-GG-PR is a version of Garey and Graham's list scheduling with preemptions, release dates, and parallel jobs [9]. Its main idea is that jobs are packed onto selves in the order defined by the input list. If the next job in the list requires too many processors, it is skipped, and jobs further down in the list are considered. Thus the shelf is considered full only when no additional job from the list can fit in. However, the height of each shelf is only until the first job in the shelf terminates, or a new job that precedes some scheduled job is released, whichever comes first. When this happens, all jobs are preempted and a new shelf is packed in the same way. Note that the number of preemptions is at most $2n$, and that the completion time of each job depends only on jobs that precede it in the list.

Assume w.l.o.g. that the jobs are indexed so that $\bar{C}_1 \leq \cdots \leq \bar{C}_n$. Let $\tilde{C}_j$ denote the completion time in the schedule found by Parallel-List-Schedule-by-$\bar{C}_j$ for job j. Since we use List-GG-PR in algorithm Parallel-List-Schedule-by-$\bar{C}_j$ we have:

Lemma 3.4 $\tilde{C}_j \leq r_j^{max} + 2 \cdot \max\left\{\sum_{k=1}^{j} \frac{size_k p_k}{m},\ p_j^{max}\right\}$.

We refer the reader to [9] for the derivation leading to this result; it is not complicated, but somewhat long.

Lemma 3.5 $\bar{C}_j \geq r_j^{max}$ *and* $\bar{C}_j \geq p_j^{max}$ *for each* $j = 1, \ldots, n$.

Proof: As Parallel-List-Schedule-by-$\bar{C}_j$ produces a feasible schedule, we get that $\bar{C}_k \geq r_k$ and $\bar{C}_k \geq p_k$ for each $k = 1, \ldots, n$. In addition, recall that jobs are indexed so that $\bar{C}_1 \leq \bar{C}_2 \leq \cdots \leq \bar{C}_n$. Hence we conclude that $\bar{C}_j \geq r_k$ and $\bar{C}_j \geq p_k$ for each $k = 1, \ldots, j$. □

Lemma 3.6 $\sum_{k=1}^{j} \frac{size_k p_k}{m} \leq \bar{C}_j + c$.

Proof: The jobs are indexed so that $\bar{C}_k \leq \bar{C}_j$ for each $k = 1, \ldots, j$. Let us examine the following set $A_j = \{k \mid r_k \leq r_j^{max}\}$, the set of all jobs that are released by the time r_j^{max}. Obviously, the set of jobs $k = 1, \ldots, j$ is a subset of A_j. Hence,

$$\sum_{k \leq j} \frac{size_k p_k}{m} \leq \sum_{r_k \leq r_j^{max}} \frac{size_k p_k}{m}$$

and from the new stability constraint and Lemma 3.5 we have:

$$\leq r_j^{max} + c \leq \bar{C}_j + c. \quad \square$$

Lemma 3.7 *Algorithm* Parallel-List-Schedule-by-$\bar{C}_j$ *is 3-competitive with respect to average weighted completion times (or sum of completion times).*

Proof: Using the above lemmas, it suffices to show that $\frac{\sum w_j \tilde{C}_j}{n} \leq \frac{3 \cdot \sum w_j C_j^*}{n} + 2wc$, where $w = \sum w_j$.

$$\tilde{C}_j \leq r_j^{max} + 2 \cdot \max\left\{\sum_{k=1}^{j} \frac{size_k p_k}{m},\ p_j^{max}\right\} \leq \bar{C}_j + 2 \cdot \max\left\{\bar{C}_j + c,\ \bar{C}_j\right\} \leq 3 \cdot \bar{C}_j + 2 \cdot c.$$

Hence we conclude that

$$\frac{\sum w_j \tilde{C}_j}{n} \leq \frac{3 \cdot \sum w_j \bar{C}_j}{n} + \frac{2nwc}{n} = \frac{3 \cdot \sum w_j \bar{C}_j}{n} + 2wc \leq \frac{3 \cdot \sum w_j C_j^*}{n} + 2wc.$$

Note that $2wc$ is an additive constant factor of the worst case performance analysis. □

Note that the algorithm is actually oblivious of the weights, and they only appear in the analysis. Taking them into account in the algorithm may lead to better bounds.

Lemma 3.8 *Algorithm* Parallel-List-Schedule-by-$\bar{C}_j$ *is 3-competitive with respect to makespan.*

Proof: Recall the following trivial lower bounds on the optimum value of the makespan:

$$\max\left\{\sum_{k=1}^{n} \frac{size_k p_k}{m},\ p_n^{max}, r_n^{max}\right\} \leq C_{max}^*$$

Hence,

$$\max_{j=1,\ldots,n} \tilde{C}_j \leq r_n^{max} + 2 \cdot \max\left\{\sum_{k=1}^{n} \frac{size_k p_k}{m},\ p_n^{max}\right\} \leq 3C_{max}^*. \quad \square$$

In summary, algorithm Parallel-List-Schedule-by-$\bar{C}_j$ is 3-competitive both for the weighted sum of completion times and for the makespan. This should be compared with algorithm

Simulate-wSRA, which achieves a competitive ratio of 18 for weighted sum of completion times, and 4 for makespan [9], for the same problem (except for having release dates with no stability constraint). Both algorithms require p_j to be known once the jobs are released.

4 Conclusions and Discussion

We observe that the dynamics of real systems arise from a combination of three things:

1. Jobs arrive at random times throughout the life of the system. This is commonly known as having non trivial release dates.

2. Job arrivals and runtimes are not known in advance. This is commonly referred to as being on-line.

3. Jobs are spread out so that the system does not saturate. This is our new constraint on the job system.

Including the new constraint leads to improved bounds for a variety of scheduling problems. These improved bounds are not a huge surprise because we impose severe constraints on the input. The point is that this is justified, because these constraints reflect reality (much like the experimentally-motivated model for clock synchronization in [10]). Therefore the new bounds are more relevant than the worst-case bounds usually derived.

While these results seem promising, it should be noted that there is much more work to be done in order to fully understand the implications of such constraints. For example, the proposed constraint restricts the values of p_j and creates a negative correlation between the average of p_j and the number of jobs arriving in a certain time span. Could it be that such a restriction already guarantees some performance bounds, regardless of the algorithm used? And to what extent does it limit the degree to which the scheduling algorithm can affect the performance?

Another issue is the formula itself. The one we used here is equivalent to the $\lambda/\mu < 1$ formula from queueing theory. However, we also know from queueing theory that performance degrades sharply as the load approaches saturation. Therefore it may be advisable to study the system under conditions that bound the load away from 100%. This can be achieved by adding a multiplicative factor $\alpha < 1$ into the expression thus:

$$\sum_{r_i \leq r_j, i \neq j} p_i \leq \alpha r_j + c$$

(and similarly for parallel systems). In addition, the formula is susceptible to the creation of pathological workloads in which arbitrarily large gaps in the input stream allow for arbitrarily large bursts of jobs later, undermining the concept of stability. This can be solved by bounding r_j from above in additional to the original bound from below, as in

$$\sum_{r_i \leq r_j, i \neq j} p_i - c_1 \leq \alpha r_j \leq \sum_{r_i \leq r_j, i \neq j} p_i + c_2$$

where c_1 and c_2 are positive constants.

Finally, this constraint focuses attention on the possible inadequacy of completion times as a performance metric for such on-line problems. The point is that for large numbers of jobs spread out over time, the sum of completion times becomes dominated by the sum of release dates, which is constant. A better metric is the sum of flow times, which are indeed influenced by the scheduling algorithm. Under the conventional models, approximating optimal flow times is very hard, and very little work has been done on such problems (e.g. [8]). We showed one case in which it becomes trivial with the new constraint. Finally, we mention that practitioners often use the slowdown metric, which is a normalized form of the flow time: it is the ratio of the flow time to the processing time, and measures how much slower jobs execute due to contention for resources [4].

References

[1] A. Batat, *Gang Scheduling with Memory Considerations.* Master's thesis, Hebrew University, Sep 1999.

[2] C. Chekuri, R. Motwani, B. Natarajan, and C. Stein, "*Approximation techniques for average completion time scheduling*". In 8th *ACM-SIAM Symp. Discrete Algorithms*, pp. 609–618, Jan 1997.

[3] D. G. Feitelson, *A Survey of Scheduling in Multiprogrammed Parallel Systems.* Research Report RC 19790 (87657), IBM T. J. Watson Research Center, Oct 1994.

[4] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong, "*Theory and practice in parallel job scheduling*". In *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), pp. 1–34, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

[5] M. Garey and R. Graham, "*Bounds for multiprocessor scheduling with resource constraints*". *SIAM J. Comput.* **4(2)**, pp. 187–200, Jun 1975.

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

[7] L. Kleinrock, *Queueing Systems.* Vol. I: Theory, John Wiley & Sons, 1975.

[8] S. Leonardi and D. Raz, "*Approximating total flow time on parallel machines*". In 29th *Ann. Symp. Theory of Computing*, pp. 110–119, 1997.

[9] A. W. Mu'alem and D. G. Feitelson, "*Preemptive bicriteria scheduling for parallel jobs*". URL http://www.cs.huji.ac.il/~ahumu/bicriteria.ps, Sep 1999.

[10] D. L. Palumbo, "*The derivation and experimental verification of clock synchronization theory*". *IEEE Trans. Comput.* **43(6)**, pp. 676–686, Jun 1994.

[11] J. Sgall, "*On-line scheduling — a survey*". In *Online Algorithms: The State of the Art*, A. Fiat and G. J. Woeginger (eds.), pp. 196–231, Springer-Verlag, 1998. Lect. Notes Comput. Sci. Vol. 1442.