On the Definition of “On-Line” in Job Scheduling Problems

Dror G. Feitelson and Ahuva W. Mu’alem
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

The conventional model of on-line scheduling postulates that jobs have non-trivial release dates, and are not known in advance. However, it fails to impose any stability constraints, leading to algorithms and analyses that must deal with unrealistic load conditions arising from trivial release dates as a special case. In an effort to make the model more realistic, we show how stability can be expressed as a simple constraint on release times and processing times. We then give empirical and theoretical justifications that such a constraint can close the gap between the theory and practice. As it turns out, this constraint seems to trivialize the scheduling problem.

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 [7]. Likewise, many practical scheduling systems have been designed, implemented, and used in production situations [4]. 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. For example, greedy algorithms (such as FCFS with simple optimizations) are widely used and perform well in practice [10], but provide poor competitive bounds with respect to many criteria.

One of the reasons might be that practitioners may feel that the system models used by theoreticians are too far removed from reality. 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) [13]. 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. However, the proposed constraint may be too strong in the sense that it yields significant improvements in the provable bounds on the performance of trivial scheduling algorithms such as FCFS. This evokes two directions of further research: searching for other less potent constraints, and understanding the relevance of scheduling theory to realistic on-line scenarios.

2 The Meaning of “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 [7]. 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 privilege 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.

Given a scheduling problem as described above, the task of the algorithm designer is to find a scheduling algorithm that performs well for arbitrary inputs. “Performs well” is typically quantified using competitive analysis, that is by comparison to the best possible (even if this is not achievable in practice). Having to deal with arbitrary inputs can be visualized as working opposite an adversary that knows about your algorithm, and constructs the worst possible input based on this knowledge. The important thing is that the adversary is not constraint in any way, except by the model being used.

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.

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. This is part of the freedom afforded to the adversary when constructing the input. 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. Significantly, the algorithm is evaluated by how well it performs for the worst case.

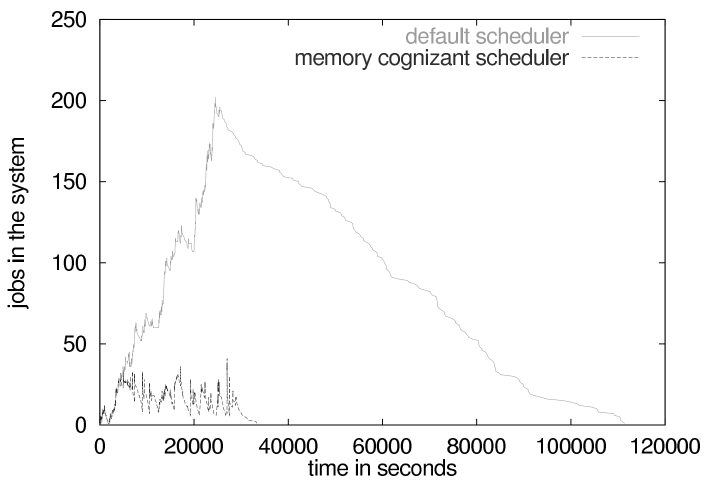

Figure 1: *Behavior of a system scheduling 1000 parallel jobs with two different schedulers. The offered load is 0.7 of the system's capacity. With the default scheduler the system saturates, and only completes the work because of the finite number of jobs involved. In effect, the system's capacity with 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].*

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) [8]. Analogous formulations have been devised for various system types, e.g. computer communication networks [2].

Queueing theory deals with 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 [8]. 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 (see Fig. 1). Thus a common ocurrence 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).

Note that this requirement for stability is *in addition* to the understanding that jobs have non-trivial release dates and the system is on-line (in the conventional meaning of not knowing the future). 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 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.

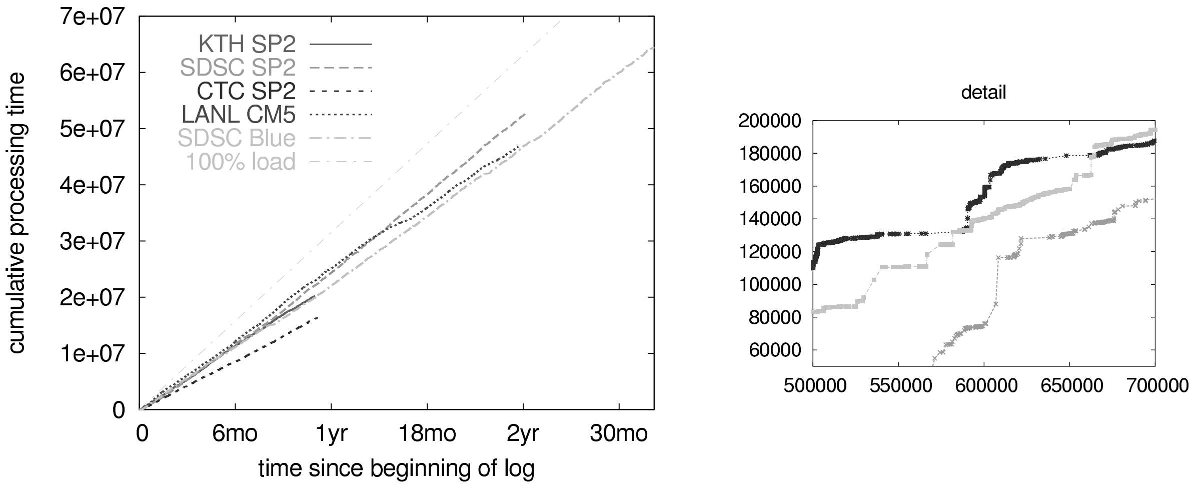

Figure 2: *Rendition of* $\sum_{r_k \leq r_j} \frac{size_k p_k}{m}$ *as a function of* r_j *for five parallel supercomputers. In all systems, arriving work is approximately linear with time, with the slope expressing the average load or utilization, which is smaller than 1. Data from the parallel workloads archive (www.cs.huji.ac.il/labs/parallel/workload/).*

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 cumulative processing requirements of previous jobs. Thus this is a restriction on the adversary that creates the input. 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:

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

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). It is analogous to the time window used by Borodin et al. to restrict the load on communication channels [2].

Fig. 2 shows how this formula works for the arrival process on several large scale supercomputers installed at the San-Diego Supercomputer Center (SDSC), Los Alamos National Lab (LANL), Cornell Theory Center (CTC), and the Swedish Royal Technical University (KTH). The workload data for these systems spans between one and nearly three years. The work arriving during this period is spread out quite evenly,

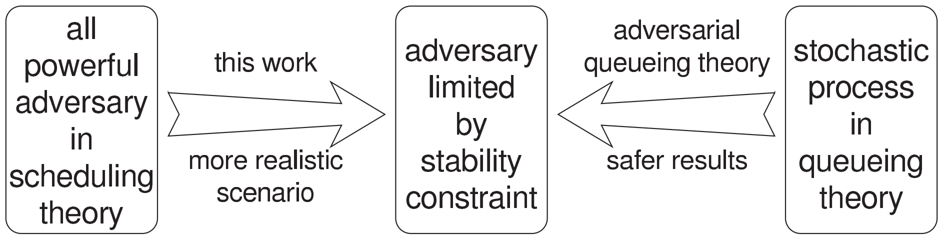
all powerful adversary in scheduling theory
this work
more realistic scenario
adversary limited by stability constraint
adversarial queueing theory
safer results
stochastic process in queueing theory

Figure 3: *Construction of the arrival process in different models.*

so rendering the cumulative requested resources[1] as a function of time leads to relatively straight lines. Of course, if we look at the finer details we see horizontal gaps between arrivals, and vertical jumps when a large request arrives (detail in Fig. 2), but these are small relative to the general trend. Therefore a linear bound is appropriate. Such a bound actually has two parameters: the slope and the intercept. We define the slope to be 1 (as in the queueing theory requirement that $\lambda/\mu < 1$), and use the constant c for the intercept. This implies that the bound is actually quite loose when compared with real data that has a load smaller than 1, and that c is only meaningful at the very beginning of the time period spanned. In fact, for long-term real data c can be taken as zero.

To see the connection between our formula 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 $r_i \leq r_j$ 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 $\mu \approx j/\sum_{i\leq j} p_i$. Putting these two expressions together we get $1 > \lambda/\mu \approx \frac{j/r_j}{j/\sum_{i\leq j} p_i} = \sum_{i\leq j} p_i/r_j$.

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, or due to inefficiency in the scheduling algorithm. Nevertheless, we will call it "the stability constraint".

It is interesting to note that our formulation is very similar to that in the "adversarial queueing theory" of Borodin et al. [2], but the motivation is exactly the opposite (Fig. 3). They add adversaries to queueing theory, allowing the adversary to control the arrival process (under suitable stability constraints), rather than using the conventional stochastic process. We restrict the adversary commonly used in scheduling theory by imposing a stability constraint on it.

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 1). In particular we focus on FCFS. It is well known that in general (without the stability constraint) the competitive ratio of FCFS can be arbitrarily bad (w.r.t common metrics such as sum of completion times or sum of flow times). Imposing the stability constraint we get that FCFS performs exceptionally well, and so the gap between the theory and practice is reduced.

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. Other notation is summarized in Table 2

[1]As these are parallel machines, the requested resources of each job k are the product of the processing time p_k and the number of processors used $size_k$. To plot different machines together, we normalize this by the number of processors in the machine m.

model	competitive ratio for $\sum C_j$	competitive ratio for C_{max}
$1 \mid stbl \mid \sum C_j, C_{max}$	2	1
$P \mid stbl \mid \sum C_j, C_{max}$	2	2
$P \mid stbl, size_j, pmtn \mid \sum C_j, C_{max}$	3	3

Table 1: *Summary of the new results (the additive factors are omitted).*

n	number of jobs
m	number of processors (default 1)
p_j	processing time of job j (positive)
p_j^{max}	$\max_{k=1,\ldots,j} p_k$
r_j	release date of job j (non-negative)
r_j^{max}	$\max_{k=1,\ldots,j} r_k$
$size_j$	number of processors needed by job j (default 1)
C_j	completion time of job j in a feasible schedule
C_j^*	completion time of job j in an optimal schedule
C_{max}	completion time of the last job (makespan)

Table 2: *Summary of notation.*

3.1 Single Machine and Identical Parallel Machine Models

For a single machine model $1|stbl|\sum C_j, C_{max}$ the greedy first-come-first-serve (FCFS) algorithm produces very good results. 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 $\tilde{C}_j$ denote the completion time of job j in the schedule found by FCFS. Since FCFS is a list based algorithm, we have [6]:

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

Lemma 3.2 *FCFS is 2-competitive with respect to sum of completions times.*

Proof: Using the previous lemma we have

$$\tilde{C}_j \leq r_j + \sum_{k=1}^{j-1} p_k + p_j.$$

Applying the new stability constraint $\sum_{k=1}^{j-1} p_k \leq r_j + c$ we re-write this as

$$\tilde{C}_j \leq 2r_j + p_j + c.$$

In addition, recall the following trivial lower bound for the optimal completion time of each job: $r_j + p_j \leq C_j^*$. Using this and summing over all jobs we get

$$\frac{1}{n}\sum \tilde{C}_j \leq \frac{2}{n}\sum C_j^* + c.$$

□

In addition, with a single machine the processor is always busy when work is available, so C_{max} is optimal.

The above derivation is trivially extended to the identical parallel machine model, in which m processors are available. In this case it is 2-competitive with respect to makespan because C^*_{max} is trivially bounded from below by $\max\{r^{max}_n, \sum_{k=1}^n p_k\}$. 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 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 deterministic algorithm for sum of completion times for the model $P \mid r_j \mid \sum C_j$ [3]. However, their work requires p_j to be known.

3.2 Multi-Processor System Model

In the multi-processor system model jobs require the simultaneous use of multiple processors. Any set of the required size can be used, and the set used can change when the job is preempted and subsequently continued. The stability constraint is expressed in this setting as

$$\sum_{r_i \leq r_j, i \neq j} \frac{size_i \, p_i}{m} \leq r_j + c$$

where $\frac{size_i}{m}$ is actually the fraction of processors used.

List-GG-PR is a version of Garey and Graham's list scheduling with preemptions and release dates that is suitable for this model [11]. 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.

Let $\bar{C}_j = r_j + p_j$. We use this quantity to define the on-line algorithm List-$\bar{C}_j$ as follows:

1. Compute $\bar{C}_j$ for each $j = 1, \ldots, n$.
2. Sort the jobs according to non-decreasing order of $\bar{C}_j$.
3. Use this permutation as an input to algorithm List-GG-PR.

Note that $\bar{C}_j$ is a trivial lower bound for the completion time of job j. Hence, $\sum_j \bar{C}_j \leq \sum_j C^*_j$. 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 List-$\bar{C}_j$ for job j. Since we use List-GG-PR to do the actual scheduling, we have

Lemma 3.3 $\tilde{C}_j \leq r^{max}_j + 2 \cdot \max\left\{\sum_{k=1}^{j} \frac{size_k \, p_k}{m},\ p^{max}_j\right\}$.

We refer the reader to [11] for the derivation leading to this result; it is not complicated, but somewhat long.

Lemma 3.4 *$\bar{C}_j \geq r^{max}_j$ and $\bar{C}_j \geq p^{max}_j$ for each $j = 1, \ldots, n$.*

Proof: By definition $\bar{C}_k \geq r_k$ 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 3.5 $\sum_{k=1}^{j} \frac{size_k\, p_k}{m} \leq \bar{C}_j + c.$

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 3.4 we have:

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

□

Lemma 3.6 *Algorithm* List-$\bar{C}_j$ *is 3-competitive with respect to average completion times.*

Proof: We will show that $\frac{1}{n} \sum \tilde{C}_j \leq \frac{3}{n} \sum C_j^* + 2c$. From the above lemmas we get

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

Hence we conclude that

$$\frac{1}{n} \sum \tilde{C}_j \;\leq\; \frac{1}{n} \sum (3\bar{C}_j + 2c) = \frac{3}{n} \sum \bar{C}_j + 2c \leq \frac{3}{n} \sum C_j^* + 2c.$$

□

Lemma 3.7 *Algorithm* List-$\bar{C}_j$ *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,

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

□

In summary, algorithm List-$\bar{C}_j$ is 3-competitive both for the sum of completion times and for the makespan. This should be compared with algorithm Simulate-SRA, which achieves a competitive ratio of 6 for sum of completion times, and 3 for makespan [11], 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 arbitrary 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 commonly referred to as the stability requirement, and has so far been absent from theoretical scheduling work.

Including the stability constraint (in the way we formulated it) 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 [12]). Therefore the new bounds are more relevant than the worst-case bounds usually derived. They may also explain why greedy algorithms seem to work well in practice.

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. But this is still 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

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

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. [9]). It might be easier to derive results with 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 [5].

References

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

[2] A. Borodin, J. Kleinberg, P. Raghavan, M. Sudan, and D. P. Williamson, "*Adversarial queuing theory*". *J. ACM* **48(1)**, pp. 13–38, Jan 2001.

[3] 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.

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

[5] 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*, pp. 1–34, Springer Verlag, 1997. Lect. Notes Comput. Sci. vol. 1291.

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

[7] 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.

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

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

[10] D. Lifka, "*The ANL/IBM SP scheduling system*". In *Job Scheduling Strategies for Parallel Processing*, pp. 295–303, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949.

[11] A. Mu'alem and D. G. Feitelson, "*Bicriteria scheduling for parallel jobs*". In *Multidisciplinary Int'l Conf. Scheduling: Theory & Apps. (MISTA)*, pp. 606–619, Aug 2003.

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

[13] 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.