Analysis of Scheduling Algorithms with Reservations

Lionel Eyraud-Dubois[2], Grégory Mounié[1] and Denis Trystram[1]

[1]LIG , Grenoble Universités,
ENSIMAG-montbonnot, 51 avenue Kuntzmann
F-38330 Montbonnot St. Martin, France
{mounie, trystram}@imag.fr

[2]LIP, ÉNS Lyon,
46 allée d'Italie,
69364 Lyon Cedex 07, France
Lionel.Eyraud-Dubois@ens-lyon.fr

Abstract

In this work, we analyze the problem of scheduling a set of independent jobs on a homogeneous parallel computer. This problem has been widely studied from both a theoretical perspective (complexity analysis, and predictability of scheduling algorithms) and practical side (schedulers in production systems). It is common for some processors of a cluster to become unavailable for a certain period of time corresponding to reservations. These reservations represent blocks of time and quantities of resources set asigned in advance for specific applications.

We propose here to investigate the scheduling problem where there are restricted resource availabilities. Our main result is to provide a deep analysis for this problem (complexity, lower bounds and upper bounds) for several variants of list scheduling algorithms. More precisely, we show that the problem of scheduling with any reservations can not be approximated. This leads to the study of restricted versions of this problem where the amount of reservation is limited.

Our analysis is based on an old bound of Graham for resource constraint list scheduling for which we propose a new simpler proof by considering the continuous version of this problem.

Keywords. Scheduling, list scheduling, cluster computing, Parallel Tasks, reservations.

LIG is supported by CNRS, INPG, INRIA, UJF. This work has been partially realized within the "action de recherche concertée" Otaphe of INRIA. A part of this work has also been conducted under the frame of the Network of Excellence CoreGrid of the European Community (W.P. 6).

1-4244-0910-1/07/$20.00 ©2007 IEEE.

1. Introduction and Motivation

1.1. Scheduling on new computing platforms.

Today, many high performance applications are implemented in clusters or computational grids. Clusters are collections of homogeneous standard processors interconnected by a fast communication network [4]. More than 70 percent of parallel and distributed systems of the top-500 are clusters. Although huge progress has been done for implementing specific applications on such systems, most researchers of the field agree that they lack of high level software tools for running any application without too much effort for the programmer. In such tools, scheduling is a crucial issue. The *jobs* (corresponding to applications) are submitted to one particular node of the cluster. The scheduling problem consists in determining the processors that will perform each job and the time when it will start its execution. The objective is to minimize the execution time (makespan).

In this work, we consider the parallel tasks model [5, 7]. In this model, the jobs can be executed on a number of processors which is fixed for each job. A more detailed description is given in section 2.

This general scheduling problem has been widely studied by theoretical approaches, with several variations taking into account several characteristics of the target platform. The scheduling algorithms that are used on actual clusters are usually very simple, based on a First Come First Serve policy (FCFS in short). It is a very popular technique which has been implemented in many actual platforms. The principle is to put the jobs into queues and execute them in a FIFO order. Since this policy may be very bad for the small jobs, it is usually implemented with a back-filling mechanism that allows to put small jobs if it remains enough room. Such a technique can be more or less aggressive as it is discussed in section 2.

Another popular policy is the list scheduling. Here, the principle is to build a list of ready jobs and to allocate them to the available processors according to some priority rules. Like FCFS, the list schedulers have a low complexity, but the makespan can be guaranteed in the worst case. That make them very good candidates for practical tools. In this work, we concentrate on list scheduling since it is the only policy that exhibits performance guarantees.

1.2. Reservations.

The main focus of this paper is the presence of restricted availability periods that constraints the scheduling algorithm. There is an increasing demand for advanced reservation mechanisms in software tools for batch scheduling: the users have the possibility to reserve some resources in advance, and have a guarantee that these resources will be available at the requested time. This feature is useful in at least two situations: first, for Grid Computing, when users want to run their application at several different remote sites, reservation is a way to make sure that the application starts at the same time on all sites; secondly, for demonstration purposes, when a user wants to show the operation of an application on a scheduled meeting. To the best of our knowledge, there has been no study on the impact of this feature on the performances of the standard scheduling algorithms in the parallel tasks model.

1.3. Results on related models with reservations.

Several analysis of the impact of restricted availability of some processors on scheduling algorithms have been conducted under various approaches (probabilistic analysis [3], average analysis [16], etc..). Most theoretical studies which consider unavailability constraints are performed in the context of sequential (single processor) tasks. Most well known negative results on independent task scheduling can trivially be applied to this problem and lead to straightforward lower bounds for approximation ratios of standard heuristics. A recent survey about scheduling under availability constraints may be found in [14].

The scheduling under availability constraints has been studied in the context of the open/flow shop problems, but also in the case of sequential task scheduling on multiprocessor machine. The classification of availability into various kinds of constraints [15, 17] is perfectly relevant, but in our opinion it is not sufficient. We need additional hypothesis as shown in section 4.2 in order to get an approximation ratio better than the ratio of m proved by [13] for list scheduling algorithms.

According to our knowledge, most of the existing work, like [2], considers models where preemption is allowed.

1.4. Contributions.

The first contribution of this paper is to analyze the scheduling problem in the presence of reservations. We give a preliminary inapproximability result which states that the problem of scheduling a set of independent jobs with the objective of minimizing the makespan by a list policy is arbitrary far from the optimal if no restriction is put on the reservations. This leads to define a new problem of scheduling with restricted reservation windows where at least some proportion α of the total amount of resources is available. This is a more realistic situation, because most software tools impose a limit on the reservation feature to ensure a good behavior of the system.

We focus on list algorithms, which are low cost algorithms with relatively good performance. We provide a lower bound of the problem of scheduling with restricted reservations by exhibiting instances whose resulting list schedule is $\frac{2}{\alpha} - 1$ longer than the optimal. Then, we analyze the general list algorithm, and derive an approximation ratio of $\frac{2}{\alpha}$, close to the lower bound.

2. Preliminaries about the basic problem without reservations

2.1. Basic scheduling model

In this work, we consider a classical computational model which has been considered in many related problems of scheduling parallel applications on clusters. In the following, we will denote it as the RIGIDSCHEDULING problem.

Let us consider a set of n independent applications (that will be called *jobs* in the following) to be processed on a cluster of m identical processors.

Each job j requires a given number of processors (denoted by q_j); job j can be scheduled on any subset of processors on the cluster. The execution time of job j is p_j. We recall below the formal definition of this problem (according to the well-known 3 field notation, it is denoted as $P|p_j, \text{size}_j|C_{\max}$ [10]):

An instance of RIGIDSCHEDULING is represented by an integer m (the number of machines) and n jobs (characterized by a duration $p_j > 0$ and a number of required processors $q_j \in [1..m]$, for $1 \leq j \leq n$).

The question is to determine a feasible schedule which minimizes the makespan.

A *solution* of such an instance is a set of n starting times, $(\sigma_i)_{i=1..n}$, such that the resulting schedule is feasible:

$$\forall t \geq 0, \quad \sum_{i \in I_t} q_i \leq m$$

where $I_t = \{i \in [1..n] \mid \sigma_i \leq t < \sigma_i + p_i\}$.

The objective is to minimize the makespan of the schedule, defined as the largest completion time of the tasks: $C_{\max} = \max_{i \in [1..n]}(\sigma_i + p_i)$.

This problem is NP-Hard, since the standard problem of scheduling sequential tasks on two processors is already weakly NP-Hard (it is exactly the same as PARTITION[1]). The RIGIDSCHEDULING problem is NP-Hard in the strong sense, even when m is fixed to a value greater or equal to 5 [6].

The execution of jobs is usually represented as a packing in the Gantt chart. But it is interesting to remark that this model does not consider contiguity. This is a reasonable assumption because in most recent cluster architectures, all processors are identical and fully connected, so applications can be executed on any subset of processors of the cluster.

Another restriction is that it is an off-line model: all jobs are assumed to be present in the system at the beginning of the scheduling step. However, in an actual system, jobs are submitted over time, and the algorithm has to react online to these unpredictable events. Nevertheless, the study of off-line algorithms is important, because it gives insights about the intrinsic performance of different strategies. Furthermore, any off-line algorithm may be used in an on-line fashion, with a doubling factor for the performance ratio on the makespan criterion [18]. The idea is to schedule jobs in successive *batches* so that all new jobs arriving during the execution of a batch will only be considered after the whole current batch is finished.

2.2. Classical algorithms

One of the simplest and probably the most popular algorithm is First Come First Served (FCFS). Such an algorithm considers tasks in the order of their arrival in the system, and greedily schedules each task until there is not enough resources available to schedule a task. The algorithm then waits for enough resources to be freed by the completion of the previously scheduled tasks. The main reason for its popularity is that its behavior is perfectly understood by the users and administrator of the cluster. But its major drawback is that it leads to a very poor utilization of the machine, since a task requiring a large number of processors may cause a large part of the resource to be left idle.

A common optimization is the use of back-filling techniques, which exist in several variants. For example, conservative back-filling considers all tasks, and greedily schedules each task at the earliest possible date, without delaying any previously scheduled task. Unlike pure FCFS, it might happen that a given task x gets to run before another task y that was submitted before it, but in that case the task y could not have been scheduled earlier, even if x was not present.

[1]and thus optimally solvable in pseudo-polynomial time.

This kind of algorithms is common in the batch scheduling literature [1], which contains several more aggressive variants that allow a task to delay an earlier task if it can be scheduled right now. Aggressivity improves the utilization of the machine, but it may make it possible for a job to *starve*, being constantly delayed by other smaller jobs arriving in a continuous stream.

From a theoretical point of view, FCFS, even with conservative backfilling, has no constant performance guarantee for the makespan criterion. Indeed, on a machine with m nodes, it is possible to build an instance with optimal makespan 1, and whose resulting FCFS schedule has makespan m.

The most aggressive variant of back-filling is more focused on improving the utilization of the resources than respecting the order of arrival of the tasks: it allows any task to delay a previously scheduled task, if this task is able to start earlier than the delayed task. The resulting algorithm is exactly the same as the initial definition of *List Scheduling* as introduced by Garey and Graham [8] in the context of scheduling with resource constraints. For independent tasks, this algorithm has a performance guarantee of $s + 1$, where s is the number of resources shared among tasks. In this problem, the only resource is processors, so List Scheduling has a performance guarantee of 2 (which can be tightened to $2 - \frac{1}{m}$, see a simpler proof in the appendix).

Let us emphasize that this result is quite different from the well-known $2 - \frac{1}{m}$ result about list scheduling with sequential tasks [11, 12]. In the sequential model with independent tasks, FCFS is a list scheduling algorithm. But with parallel tasks, FCFS does not behave like a list algorithm, since it may keep some resources idle even if there are tasks ready to be scheduled. To distinguish the classical list scheduling algorithm from the sequential tasks model, usually denoted as LS, we will denote the list scheduling algorithm with resource constraints as LSRC.

2.3. Performance guarantees

We recall briefly below the standard definition of performance ratio $\rho_{\mathcal{A}}$ for an approximation algorithm $\mathcal{A}$.

$$\rho_{\mathcal{A}} = \inf \{r \geq 1 | \rho_{\mathcal{A}}(I) \leq r \text{ for all problem instances } I\}$$

where $\rho_{\mathcal{A}}(I)$ is the ratio between the criterion value of the solution produced by $\mathcal{A}$ on instance I and the best solution for I.

3. Modelization

In this section, we present the general problem of scheduling in the presence of reservations, which we will denote as RESASCHEDULING, and analyze it.

3.1. Formal problem

An instance of the RESASCHEDULING problem can be formally described by an integer m (the number of machines), a set of n independent jobs $(T_i)_{i=1..n}$ (characterized by a duration $p_i > 0$ together with a number of required processors $q_i \in [1..m]$) and n' reservations $(R_j)_{j=n+1..n+n'}$ (characterized by a duration $p_j > 0$, a number of processors $q_j \in [1..m]$ and a starting time $r_j > 0$).

The problem is to provide a feasible schedule which minimizes the makespan.

We will only consider feasible instances, i.e. those whose reservations can be scheduled on the m machines:

$$\forall t \geq 0, \quad \sum_{j \in J_t} q_j \leq m$$

where $\quad J_t = \{j \in [n+1..n+n'] \,|\, r_j \leq t < r_j + p_j\}$.

We can thus equivalently consider that the given reservations yield an *unavailability* function U, defined at every time by $U(t) = \sum_{j \in J_t} q_j$. $U(t)$ is the number of unavailable machines at time t; U is piecewise constant, and an instance is feasible if and only if $\forall t, U(t) \leq m$.

Similarly to the previous section, a *solution* is a set of n starting times, $(\sigma_i)_{i=1..n}$, such that the resulting schedule is feasible: $\forall t \geq 0, \quad \sum_{i \in I_t} q_i \leq m - U(t)$, where I_t is defined as in the previous section.

The objective is here to minimize the makespan of the schedule.

3.2. Analysis

First, it is straightforward to remark that this problem is NP-hard since it contains the problem of scheduling independent parallel rigid jobs [5] ($P_m \,|\, \text{size}_j, p_j \,|\, C_{\max}$) without reservations ($n' = 0$). Finding a schedule with minimal makespan is a difficult problem, even without reservations.

With the way the RESASCHEDULING problem has been defined, it is impossible to design a polynomial-time approximation algorithm to solve it. Informally, this comes from the fact that it is possible to insert a very large and very long reservation that starts just at the optimal value of $C_{\max}$. This reservation will not disturb any optimal schedule, but it will lead to an arbitrarily large makespan for any non optimal schedule.

Theorem 1 *If $P \neq NP$, there is no polynomial algorithm for the* RESASCHEDULING *problem with a finite performance ratio, even in the restricted case $m = 1$ (only one machine) or $n' = 1$ (only one reservation).*

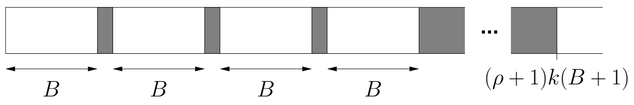

Figure 1. Transformation from 3PARTITION

Proof: We prove here the theorem in the case $m = 1$ using a reduction from 3PARTITION. The $n' = 1$ case can be easily obtained by the same technique, with a reduction from RIGIDSCHEDULING.

Let us assume, by contradiction, that $\mathcal{A}$ is an algorithm for solving the RESASCHEDULING problem, with a performance guarantee of ρ. Let I_P be an instance of 3PARTITION ($3k$ integers x_i and an integer B such that $\sum x_i = kB$) [9]. We build an instance I of the RESASCHEDULING problem, with one machine, such that the time between two reservations is exactly B (see figure 1):

- $m = 1$;
- $n = 3k$ jobs with $\forall i, q_i = 1$ and $p_i = x_i$;
- k reservations $(R_j)_{j=n+1..n+k}$ defined by $q_j = 1, r_{n+1} = B$, and $r_j = r_{j-1} + B + 1$ for $n + 1 < j \leq n + k$ [2]. The lengths of the reservations are $p_j = 1$ for $j \neq n + k$. The length of the last reservation is $p_{n+k} = \rho k(B + 1) + 1$ (and thus ends at time $(\rho + 1)k(B + 1)$).

If there is a solution to 3PARTITION for the instance I_P (i.e. it is possible to partition $[1..n]$ into k groups G_l of three elements such that $\forall l, \sum_{i \in G_l} x_i = B$), then it is possible to realize a schedule of makespan $C^*_{\max} = k(B + 1) - 1$ by scheduling the tasks of group G_l between the $(l - 1)$th and the lth reservations[3]. Since $\mathcal{A}$ is a ρ-approximation algorithm, it must yield a schedule with makespan $C^{\mathcal{A}}_{\max} \leq \rho(k(B + 1) - 1) < \rho k(B + 1)$. Since $\mathcal{A}$ can schedule no task between times $k(B + 1) - 1$ and $(\rho + 1)k(B + 1)$, we must have $C^{\mathcal{A}}_{\max} = C^*_{\max}$. Hence, the schedule of $\mathcal{A}$ yields a solution to the instance I_P by assigning the tasks between two reservations to the same group.

The converse is straightforward. □

4. Restricted problems

We are going to study two restrictions of the general problem, that will allow us to yield performance bounds for the LSRC algorithm.

[2]i.e. $r_j = (j - n)(B + 1) - 1$ for $n + 1 \leq j \leq n + k$

[3]This schedule is optimal because the machine is used to perform a task each time it is available

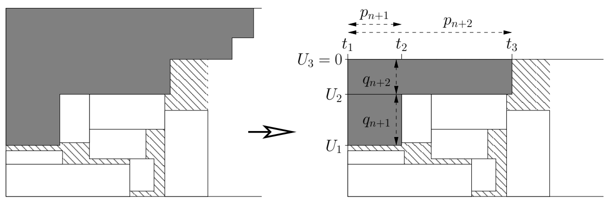

Figure 2. An example of non-increasing reservations and the transformation used in the proof

4.1. Non-increasing reservations

In this section, we are going to study a subset of all possible instances for RESASCHEDULING, containing all instances with non-increasing reservations (or equivalently, non-decreasing availabilities; see figure 2). Though it may not seem very relevant in our setting, this restriction on availabilities is quite common in the literature [15], and usually it allows to derive better algorithms.

This additional constraint adds a new hypothesis on the unavailability function U, which is now supposed to be non-increasing. For simplicity, we will note in the following the available resources at time t as $m(t) = m - U(t)$. The following proposition shows a performance guarantee for LSRC for these instances.

Proposition 1 *For every instance I with non-increasing reservations, we have:*

$$C_{\max}^{LSRC} \leq \left(2 - \frac{1}{m(C_{\max}^*)}\right) C_{\max}^* \leq \left(2 - \frac{1}{m}\right) C_{\max}^*$$

Proof: Consider a transformation of the instance I into an instance I' defined as follows: set $m^{I'} = m^I(C_{\max}^*)$, and $m^{I'}(t) = m^I(t)$ for all $t \leq C_{\max}^*$. It is clear that both instances have the same optimal value $C_{\max}^*$, and that a feasible solution for I' is also a feasible solution for I.

Assume that $U^{I'}$ takes k different values $U_1, \cdots, U_k = 0$, with $U^{I'}(t) = U_j$ for $t_j \leq t < t_{j+1}$ (we have thus $t_1 = 0$ and $t_{k+1} = \infty$). Then we can build an instance I'' of RIGIDSCHEDULING by replacing the reservations by $k-1$ tasks $T_{n+1}, \cdots, T_{n+k-1}$, defined by $q_{n+j} = U_j - U_{j+1}$ and $p_{n+j} = t_{j+1}$ (see figure 2). It is clear that to every feasible schedule of I' corresponds a feasible schedule of I'' (the opposite being not necessarily true).

However, if we place the additional tasks of I'' at the head of the list, the LSRC algorithm will yield the same schedule for instance I'' and for instance I'. From theorem 2, we have $C_{\max}^{\text{LSRC}}(I'') \leq \left(2 - \frac{1}{m^{I''}}\right) C_{\max}^*(I'')$. Since the optimal schedule for I' is feasible for I'', we have $C_{\max}^*(I'') \leq C_{\max}^*(I')$.

Considering now that $C_{\max}^{\text{LSRC}}(I) \leq C_{\max}^{\text{LSRC}}(I') = C_{\max}^{\text{LSRC}}(I'')$, and that $C_{\max}^*(I) = C_{\max}^*(I')$, we have the final result :

$$C_{\max}^{\text{LSRC}}(I) \leq \left(2 - \frac{1}{m^I(C_{\max}^*(I))}\right) C_{\max}^*(I)$$

□

4.2. Restricted reservations

In actual scheduling systems that feature advance reservations, there is a limit imposed on users, in order to avoid that the cluster be totally blocked by the reservations. For example, it is common to disallow reservations that require more than half of the machines of the cluster. In this section, we extend the model to deal with this kind of constraints, and derive results about the LSRC algorithm.

Keeping this goal in mind, we define another, more realist constraint to restrict the possible instances to the problem RESASCHEDULING. Given a parameter $\alpha \in]0;1]$, we define the (sub)problem α-RESASCHEDULING by restricting all reservations at a given time to require no more than $(1-\alpha)m$ machines, and tasks to require no more than αm machines. More formally:

$$\forall t \geq 0, \quad U(t) = \sum_{j \in J_t} q_j \leq (1-\alpha)m$$
$$\forall i \leq n, \quad q_i \leq \alpha m$$

These constraints define instances in which always at least αm machines are available; and since no task can require more machines, it is always possible to schedule at least one task. This will rule out the pathological instances of the previous section, and will allow to derive performance guarantees.

Of course, this problem remains strongly NP-Hard, so we are interested in the performance of list scheduling in this context.

Lower bound. We are going to give a lower bound for the performance of LSRC, which shows that it is not possible to prove a performance guarantee $\rho \leq \frac{2}{\alpha} - 1$ for general LSRC. To show it, we build an instance for the case $\alpha = \frac{2}{k}$, where k is an integer, in which the optimal schedule uses m machines almost all the time, but there is an order of the list for which LSRC uses only αm machines almost all the time.

Proposition 2 *If $\frac{2}{\alpha}$ is an integer, the performance guarantee of LSRC is at least $\frac{2}{\alpha} - 1 + \frac{\alpha}{2}$.*

Proof: Assume that $\alpha = 2/k$, with $k \in \mathbb{N}$. We define an instance I with $m = k^2(k-1)$ machines, that contains two different kinds of tasks (see figure 3):

- k tasks, from T_1 to T_k, with $p_i = 1/k$ and $q_i = (k-1)^2$;
- $k-1$ tasks, from T_{k+1} to T_{2k-1}, with $p_i = 1$ and $q_i = k(k-1)+1$;

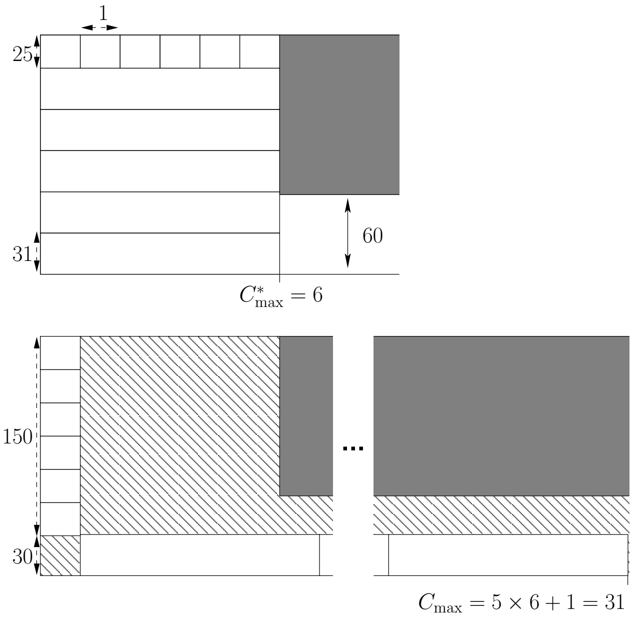

Figure 3. An optimal schedule and the corresponding LSRC schedule, for $\alpha = \frac{1}{3}$ ($m = 180$).

Additionally, I contains one reservation that starts at time 1, and occupies $m(1-\alpha) = m - 2m/k = k(k-1)(k-2)$ processors during $2k$ time units.

Since $(k-1)\times(k(k-1)+1)+(k-1)^2 = (k-1)(k(k-1)+k) = (k-1)k^2 = m$, it is possible to schedule all tasks before time 1. The optimal makespan for this instance is thus $C^*_{\max} = 1$.

On the other hand, LSRC, when the list ordered by increasing i, schedules all the tasks from the first set to begin at time 0 (this is possible since $k \times (k-1)^2 \leq m$). But then, no task from the second set can start its execution before these tasks have finished (i.e. time $1/k$) because $k(k-1)^2 + k(k-1) + 1 = m + 1$. But it is impossible that two tasks from the second set run concurrently if they start later than time 0. Indeed, after time 1, only $2m/k = 2k(k-1)$ processors are available, and 2 tasks from the second set occupy $2k(k-1)+2$ processors. These tasks must then be scheduled sequentially, and the makespan of the resulting schedule is $C_{\max} = \frac{1}{k} + k - 1 = \frac{2}{\alpha} - 1 + \frac{\alpha}{2}$. $\square$

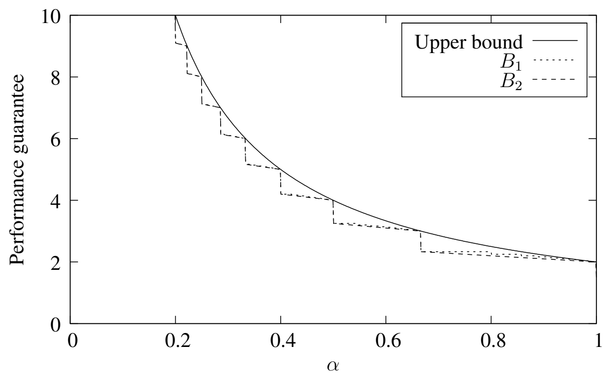

Figure 4. Upper and lower bounds for the performance of LSRC on the α-RESASCHEDULING problem, as a function of α.

For a more general α, a similar but more tedious instance proves that

$$\rho \geq \left\lceil \frac{2}{\alpha} \right\rceil - 1 + \frac{1}{\left\lfloor \frac{1-(\alpha/2)}{1-(\alpha/2)(\lceil 2/\alpha \rceil - 1)} \right\rfloor + 1} \equiv B_1$$

$$\geq \left\lceil \frac{2}{\alpha} \right\rceil - \frac{\lceil 2/\alpha \rceil - 1}{2/\alpha} \equiv B_2$$

The bound B_2 is a bit less precise than B_1, but easier to express. Figure 4 plots these bounds together with the upper bound proven below, and shows that the upper and lower bounds can be arbitrarily close to each other for some values of the parameter α.

Upper bound. As we have stated before, the standard list scheduling algorithm without reservation in the parallel task model has a performance guarantee equal to $2 - \frac{1}{m}$ (see appendix). If we restrict to αm processors, we can easily obtain a first bound by simply applying any list scheduling on the available processors. Thus, we will get a performance guarantee of $\frac{2}{\alpha}$. For $\alpha = \frac{1}{2}$, we obtain a bound of 4.

Proposition 3 *For the problem α-RESASCHEDULING, LSRC has a performance guarantee ρ which is at most $\frac{2}{\alpha}$.*

Proof:

The proof is a direct adaptation of theorem 2 (see appendix), with t' set to $t + \frac{1}{\alpha}C^*_{\max}$. $\square$

5. Conclusion

In this paper, we have analyzed the problem of scheduling a set of n independent jobs in the presence of reservations. We focused on list scheduling algorithms because of their simplicity and solid theoretical foundations. We defined the problem of scheduling with restricted reservations in order to avoid stupid effects that lead to algorithms whose makespan are arbitrary far from the optimal ones. Then, we provided a lower bound ($\frac{2}{\alpha} - 1 + \frac{\alpha}{2}$) and derived a performance guarantee for any list scheduling algorithms ($\frac{2}{\alpha}$) which is close to the lower bound.

An immediate but not trivial perspective is to study some variants of list scheduling that can improve the upper bound (for instance adding a priority based on sorting the jobs by decreasing durations).

Another further direction is to investigate different kind of heuristics like those based on packing (partition on shelves) algorithms.

References

[1] M. Baker, G. Fox, and H. Yau. Cluster computing review, 1995.

[2] J. Blazewicz, P. Dell'Olmo, M. Drozdowski, and P. Maczka. Scheduling multiprocessor tasks on parallel processors with limited availability. *European Journal of Operational Research*, 149:377–389, 2003.

[3] E.G. Coffman Jr., P. R. Jelenkovic, and B. Poonen. Reservation probabilities. *Advances in Performance Analysis*, 1999.

[4] D. E Culler and J. P. Singh. *Parallel Computer Architecture*. Pitman/MIT Press, 1989.

[5] M. Drozdowski. *Handbook of Scheduling — Algorithms, Models, and Performance Analysis*, chapter 25 - Scheduling parallel tasks — Algorithms and complexity, pages 25–25. Computer and Information Science Series. Chapman & Hall/CRC, Boca Raton-London-New York-Washington, D.C., 2004.

[6] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM J. Discrete Math.*, 2(4):473–487, 1989.

[7] D. G. Feitelson. Scheduling parallel jobs on clusters. In Rajkumar Buyya, editor, *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 519–533. Prentice Hall PTR, Upper Saddle River, NJ, 1999. Chap. 21.

[8] Garey and Graham. Bounds for multiprocessor scheduling with resource constraints. *SICOMP: SIAM Journal on Computing*, 4, 1975.

[9] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-complete ness*. W.H. Freeman, New York, 1979.

[10] 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. *Ann. Discrete Mathematics*, 5:287–326, 1979.

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

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

[13] Chung-Yee Lee. Machine scheduling with an availability constraint. *Journal of Global Optimization*, 9(Issue 3 - 4):395 – 416, Dec 1996.

[14] Chung-Yee Lee. *Handbook of Scheduling — Algorithms, Models, and Performance Analysis*, chapter 22 - Machine scheduling with availability constraints. Computer and Information Science Series. Chapman & Hall/CRC, Boca Raton-London-New York-Washington, D.C., 2004.

[15] Zhen Liu and Eric Sanlaville. Preemptive scheduling with variable profile, precedence constraints and due dates. *Discrete Applied Mathematics*, 58(3):253–280, 1995.

[16] Zhen Liu and Eric Sanlaville. Stochastic scheduling with variable profile and precedence constraints. *SIAM Journal on Computing*, 26(1):173–187, 1997.

[17] G. Schmidt. Scheduling on semi-identical processors. *J. of Operational Research*, A28:153–162, 1984.

[18] D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on-line. *SIAM J. Comput.*, 24(6):1313–1331, 1995.

6. Appendix

6.1. Revisiting Graham's bound

In this section, we propose a new and simpler proof of the well-know result of Graham on list scheduling with resource constraints [8], in the case of independent jobs and a single resource ($s = 1$).

Notations. Let I be an instance of n independent parallel tasks to be sheduled on m machines. Each task i uses q_i machines and must be executed in an exclusive way for a time p_i, without preemption. We will note p_{max} the maximum execution time of the tasks $max_{1\leq i \leq n}(p_i)$, and $W(I)$ the total work of the instance, defined as $W(I) = \sum_{1\leq i\leq n} p_i q_i$.

Given a list scheduling algorithm $\mathcal{A}$, we will note $\mathcal{A}(\mathcal{I})$ the schedule produced by $\mathcal{A}$ for the instance I. This schedule is represented by a function σ that gives the starting time σ_i of every task T_i. For a given time t, we will note I_t the set of tasks running at time t: $I_t = \{i \in [1..n] \,|\, \sigma_i \leq t < \sigma_i + p_i\}$, and $r(t)$ the number of machines used at time t by $\mathcal{A}(\mathcal{I})$: $r(t) = \sum_{i\in I_t} q_i$.

Lemma 1

$$\forall t, t' \in [0, C_{\max}^{\mathcal{A}(\mathcal{I})}[, \quad t' \geq t + p_{\max} \;\Rightarrow\; r(t) + r(t') > m$$

Proof: If $t' \geq t + p_{\max}$, then necessarily $I_{t'} \cap I_t = \emptyset$. On the other hand, since $t' \leq C_{\max}^{\mathcal{A}(\mathcal{I})}$, there is at least one task T_i running at time t'. The algorithm $\mathcal{A}$ has chosen not to start this task at time t. By definition of a list algorithm, this means that the task T_i cannot be executed together with the tasks from I_t. Since this can only be because of a lack of resources, we have $r(t) + q_i > m$.

The result follows immediately. □

Remark. Since both $r(t)$ and $r(t')$ are integers, we can write more precisely: $r(t) + r(t') \geq m + 1$.

We can now establish the main result:

Theorem 2 *If $\mathcal{A}$ is a list algorithm, then for every instance I with m machines,*

$$C_{\max}^{\mathcal{A}}(I) \leq \left(2 - \frac{1}{m}\right) C_{\max}^{*}(I)$$

Proof: Let us consider an instance I with m processors. We are going to prove that if there exists a real number x such that $C_{\max}^{\mathcal{A}} \geq (2-x)C_{\max}^*$, then $x \geq \frac{1}{m}$.

Since $C_{\max}^* \geq p_{\max}$, we have:

$$\forall t \in [0, (1-x)C_{\max}^*[, \quad r(t) + r(t + C_{\max}^*) \geq m + 1$$

After integrating this relation, we obtain:

$$X \equiv \int_0^{(1-x)C_{\max}^*} r(t) + r(t + C_{\max}^*)\mathrm{d}t \geq (m+1)(1-x)C_{\max}^*$$

With some rearrangement of this integral, we can bound it by the total work of the instance:

$$\begin{aligned} X &= \int_0^{(1-x)C_{\max}^*} r(t)\mathrm{d}t + \int_{C_{\max}^*}^{(2-x)C_{\max}^*} r(t)\mathrm{d}t \\ &= \int_0^{(2-x)C_{\max}^*} r(t)\mathrm{d}t - \int_{(1-x)C_{\max}^*}^{C_{\max}^*} r(t)\mathrm{d}t \end{aligned}$$

and since $r(t) \geq 1$ for all t,

$$\leq \int_0^{C_{\max}^{\mathcal{A}}} r(t)\mathrm{d}t - xC_{\max}^* = W(I) - xC_{\max}^*$$

Obviously, since $mC_{\max}^*$ is the total area available to the optimal schedule, we have $W(I) \leq mC_{\max}^*$. We deduce:

$$(m - x)C_{\max}^* \geq X \geq (m+1)(1-x)C_{\max}^*$$

We deduce the final result: $x \geq \frac{1}{m}$. □