HAL
open science

A Proactive Approach for Coping with Uncertain Resource Availabilities on Desktop Grids

Louis-Claude Canon, Adel Essafi, Denis Trystram

To cite this version:

Louis-Claude Canon, Adel Essafi, Denis Trystram. A Proactive Approach for Coping with Uncertain Resource Availabilities on Desktop Grids. HiPC'14, IEEE Int. Conf. on High Performance Computing, 2014, Goa, India. pp.1-9. hal-01230767

HAL Id: hal-01230767

https://hal.science/hal-01230767

Submitted on 19 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT FEMTO-ST

UMR CNRS 6174

A Proactive Approach for Coping with Uncertain Resource Availabilities on Desktop Grids

Louis-Claude CANON — Adel ESSAFI — Denis TRYSTRAM

Rapport de Recherche n° RRDISC2014-1

DÉPARTEMENT DISC – March 3, 2014

A Proactive Approach for Coping with Uncertain Resource Availabilities on Desktop Grids

Louis-Claude CANON , Adel ESSAFI , Denis TRYSTRAM

Département DISC

March 3, 2014 (28 pages)

Abstract: Uncertainties stemming from multiple sources affect distributed systems and jeopardize their efficient utilization. Desktop grids are especially concerned by this issue as volunteers lending their resources may have irregular and unpredictable behaviors. Efficiently exploiting the power of such systems raises theoretical issues that received little attention in the literature. In this paper, we assume that there exist predictions on the intervals during which machines are available. When these predictions have a limited error, it is possible to schedule a set of jobs such that the effective total execution time will not be higher than the predicted one. We formally prove it is the case when scheduling jobs only in large intervals and when provisioning sufficient slacks to absorb uncertainties. We present multiple heuristics with various efficiencies and costs that are empirically assessed through simulations.

Key-words: Scheduling; Desktop grids; Uncertainties; Availabilities.

FEMTO-ST Institute, DISC research department
UFR Sciences - Route de Gray - F-25030 BESANÇON cedex FRANCE
Tél : (33 3) 81 66 64 00 – Fax : (33 3) 81 66 64 23 – e-mail : brigitte.bataillard@univ-fcomte.fr

1 Introduction

Harnessing the power of modern parallel platforms is compromised by many uncertainties coming from their high scales and resources volatility. In this work, we focus on desktop grids, which gather idle computing resources of common desktops distributed over the Internet for running massively parallel computations. Such systems provide a very large computing power for many applications issued from a wide range of scientific domains [14, 19]. Most of these parallel applications are composed of workflows of sequential jobs that are submitted by successive batches to a particular user interface machine. Then, the corresponding jobs are transferred and executed on the distributed available resources according to some scheduling policy.

Usually the resources are not continuously available over time since users give their idle CPU time only for some periods when they are not using their desktops. Moreover, even if the dates of unavailability periods can easily be estimated in advance, they are subject to uncertainties. This may drastically impact the global performances by delaying the completion of the whole application.

In this paper, we study how to schedule efficiently a set of jobs under the constraints of unavailability periods. At the same time, we are interested in reducing the effect of disturbances on the unavailabilities by maximizing the *stability* that measures the ratio between the maximum completion time (makespan) of the disturbed instance over a predicted completion time [2].

To the best of our knowledge, there is no related work studying scheduling with unavailability under uncertainties except our previous contribution [3] which provided a preliminary analysis for restricted disturbance patterns. Our proposed approach relies on a slack-based proactive approach (with temporal protection). The uncertainties are taken into account before job executions such that the execution of the generated schedule is robust to variations in the environment characteristics. As the execution of some jobs may be interrupted, we introduce slacks just before each unavailability period. The problem consists then in assigning each job to an availability (like in a classical bin packing) while preserving enough idle time in the slack whose lengths depends on the allocated jobs.

The first contribution is to investigate the problem of scheduling with unavailabilities from the view point of studying the effect of uncertainties on the availability periods. We characterize general conditions that any schedule must verify to obtain an optimal stability (i.e., with a disturbed completion time not worse than the predicted one). The processors are assumed to be uniform whereas they were identical in our previous work [3]. Moreover, we

consider that unavailabilities can start not only earlier than expected but also later. Our main contribution is the design of a complete set of algorithms and their analysis. The proposed bounds on the stability outperform our previous one. Additionally, the algorithmic scheme relies on global procedures that use either greedy or dual approximation paradigms. Each of these procedure uses a local procedure, including a novel dynamic programming, for assigning jobs to a given availability. Then, the good behaviour of the algorithms is assessed by running simulations derived from actual workflows of BOINC [1].

The paper is organized as follows. We start by briefly recalling the most significant studies on scheduling under availability constraints in Section 2. Section 3 is devoted to the description of the computation model and the formal presentation of the problem. Section 4 provides a structural analysis of the proposed slack-based mechanism in terms of stability. Then, Section 5 describes the various algorithms and their combinations. Before concluding, Section 6 presents some experiments based on simulations on actual workflows and availability constraints.

2 Related Work

In this section, we recall briefly the most significant works related to the problem of scheduling under unavailability constraints.

Most of the existing approaches that addressed the problem of scheduling with unavailabilities are based on the well-known LPT rule (Largest Processing Times) for which many assumptions have been explored. For instance, Lee [15, 16] introduced the problem of scheduling independent jobs with several availability patterns. He showed that the problem is not polynomially approximable if no restrictions are done on the availabilities. This is a commonly accepted result which shows that without specific assumptions the problem may lead to an non-feasible solution. Hwang and Chang [9] have analyzed the problem when no more than half of the machines are unavailable simultaneously. Liao et al. [18] studied the restriction of the problem on two machines where each machine has one fixed unavailability period. A variant of this particular problem was studied in [20] where the first machine is always available whereas periodic unavailabilities are scheduled on the second. The problem where one machine is always available and with an arbitrary number of unavailabilities on the other processors was analyzed in [5]. Although this problem does not admit Fully Polynomial Time Approximation Scheme (FPTAS), a costly PTAS based on the multiple knapsack was designed [8]. While all the previous approaches are related to sequential jobs, Eyraud et al. [6] studied the problem of scheduling with unavailabilities for parallel rigid jobs.

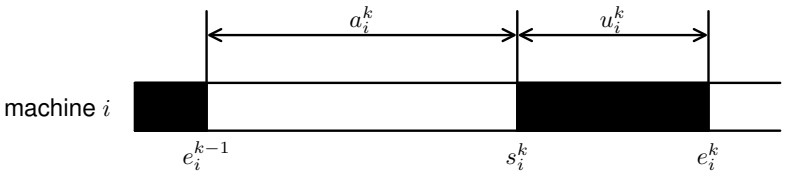

Figure 1: Representation of interval k on machine i, which contains an availability period followed by an unavailability period.

They proved that there is no approximation algorithm in the general case, and they proposed an approximation algorithm for non-increasing unavailability patterns.

We developed a preliminary study of scheduling under unavailability constraints with uncertainties in [3]. Each interrupted job was restarted from the beginning, without migration, but the unavailability constraints were only allowed to advance forward.

The solution proposed here is much more general. It is based on the introduction of a slack mechanism before the unavailability intervals. Several studies used the same idea and proposed proactive heuristics based on slacks in different contexts. In [7], the authors investigate the use of preemption. In [12, 13], the authors explored stochastic resource breakdown.

3 Description of the Problem

3.0.1 Execution Model

The target parallel platform is composed of m uniform machines (also called processors) that are indexed by i in the rest of the paper. The cycle time of processor i is denoted by b_i, which is the inverse of its speed. The workload consists of n independent jobs that are indexed by j. The processing amount of job j is denoted by p_j. Hence, the duration of job j executed on processor i is $b_i p_j$. Moreover, each machine is subject to a set of unavailability constraints. An interval is defined as an availability followed by an unavailability. The intervals are indexed by k and the start date of unavailability period k on processor i is denoted by $s_i(k)$. As depicted in Figure 1, the availability (resp. unavailability) has a duration of $a_i(k)$ (resp. $u_i(k)$). Hence, this period ends at $e_i(k) \triangleq s_i(k) + u_i(k)$ and we also have $s_i(k) \triangleq a_i(k) + e_i(k-1)$. To be consistent, the first availability on processor i starts at $e_i(0)$ (by default, $s_i(0) = -\infty$ and no job is scheduled on the 0th interval) and there are K_i intervals on machine i.

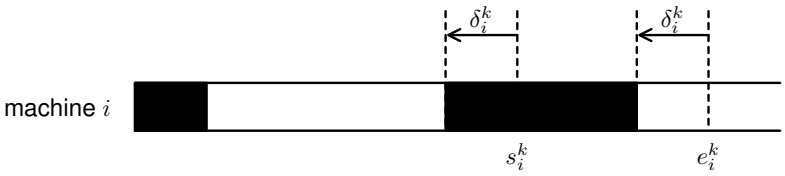

Figure 2: Unavailability k may start and end earlier or later due to the disturbance $\delta_i(k)$.

The job durations are assumed to be bounded such that $2p_{\max} \leq a_{\min}$ where $p_{\max}$ is the maximum job duration (*i.e.*, $\max_{1\leq i\leq m, 1\leq j\leq n} b_i p_j$) and $a_{\min}$ is the shortest availability (*i.e.*, $\min_{1\leq i\leq m, 1\leq k\leq K_i} a_i(k)$).

3.0.2 Disturbance Model

Solving scheduling problems with uncertain data has received recently a great attention. There exist several possible approaches depending on the target problem and the desired objectives. The survey of Billaut et al. [2] discusses several complementary approaches from pure proactive methods (sensitivity analysis), pure online strategies and semi online methods (flexibility). We focus here on this last approach which consists in building an efficient solution on estimated data followed by correction mechanisms at runtime.

Let $\delta_i(k)$ be the disturbance that impacts the kth unavailability period on processor i. The disturbed start date of this unavailability is $s_i(k) + \delta_i(k)$ and its disturbed end date is $e_i(k) + \delta_i(k)$.

As shown in Figure 2, the start and end dates are disturbed, but not the unavailability durations (*i.e.*, $u_i(k)$ remains constant). Moreover, the disturbances are limited to a restricted interval to limit prediction errors, *i.e.*, $-a_i(k) \leq \delta_i(k) \leq a_i(k+1)$. This assumption ensures that there are at most two interruptions per interval.

3.0.3 Problem Definition

The objective is to generate a schedule given a set of jobs and a set of machines with their unavailability constraints. A schedule is specified by an allocation function $\pi_i(k)$ that provides the set of jobs to be executed during each kth interval on each ith processor. We restrict this study to feasible schedules, *i.e.*, schedules for which the availability periods are long enough for any job execution. Assessing the quality of a schedule is done through two objectives: the efficiency and the ability to cope with uncertainties.

The first objective is the horizon λ, which measures the performance and is specified while building the schedule. Since jobs allocated to any interval must fit in the availability, the horizon is then greater than or equal to the makespan without uncertainties ($C_{\max} \triangleq \max_{1 \leq j \leq n} C_j$ where C_j is the completion time of job j in a given schedule [17]). In [3], we have considered the makespan instead of an horizon. As a consequence, we assumed that at least one processor did not have any unavailability which is unrealistic and not required anymore. Using an horizon provides thus more flexibility and is more general as it may be equal to the makespan as a special case.

The second objective is the *stability*. It is defined as the ratio between the highest disturbed makespan $\widetilde{C}_{\max}$ among all possible disturbed scenarios and the specified horizon λ, *i.e.*, $\sigma \triangleq \frac{\widetilde{C}_{\max}}{\lambda}$. It represents the insensitivity of a schedule to disturbances. A schedule is *stable* if $\sigma \leq 1$ (values strictly lower than one are not necessary and may impair the efficiency). The problem consists in determining a schedule with minimum values of horizon and stability.

3.0.4 Discussion

The proposed models rely on some assumptions that are briefly discussed below.

The $2p_{\max} \leq a_{\min}$ assumption has been confronted to actual data from the Failure Trace Archive [11]. Among a sample of five millions availabilities, 95% of the computing time is distributed in availabilities longer than two hours, which is much greater than the duration of common desktop grid jobs. Moreover, the assumption could be relaxed by considering each processor separately (*i.e.*, by assuming that for each processor, twice its longest allocated job must be lower than its shortest availability). Theoretically, this assumption avoids interruptions to occur in cascade indefinitely (the same job being interrupted by every unavailability).

The $-a_i(k) \leq \delta_i(k) \leq a_i(k+1)$ assumption reflects the precision of the prediction. Performing a static allocation strategy would be irrelevant if the prediction error was too high. The disturbance model also supposes that each unavailability can interrupt only one job and that the unavailability durations remain constant, which implies that the ratio between availabilities and unavailabilities remains constant.

4 Dealing with Uncertainties

In this section, we present a mechanism based on the insertion of a slack inside each availability. The size of each slack that is required to obtain stable schedules should be carefully determined. Moreover, we assume that at runtime jobs scheduled in interval k are not started before jobs scheduled in interval $k-1$ has not been completed for any $1 < k \leq K_i$ (i.e., the order between the allocations of each interval is preserved).

Definition 1 (Slack)**.** *The* slack $d_i(k)$ *is the amount of idle time reserved in availability* k *on processor* i *in a given schedule.*

Let $M_i(k) \triangleq \max_{j \in \pi_i(k)} b_i p_j$ be the duration of the longest job assigned to the kth interval on processor i ($M_i(k) \triangleq 0$ if $\pi_i(k) = \emptyset$ or if $k \geq K_i$).

Definition 2 (Slack rule)**.** *A schedule respects the* slack rule *if and only if* $d_i(k) \geq \max(M_i(k), M_i(k+1), M_i(k+2))$ *for each interval* k *and for each machine* i*.*

Proposition 1. *When* $2p_{\max} \leq a_{\min}$ *and when the jobs are executed without delay, any schedule respecting the slack rule is stable.*

Proof. The proof is by induction on the interval indexes using a single machine as it is illustrated in Figure 3. The induction hypothesis $H(k)$ is that all jobs scheduled in the first k intervals have been executed before the following date (three cases):

1. $e_i(k)$ if unavailability k is early (i.e., $\delta_i(k) \leq 0$);
2. $s_i(k) - d_i(k)$ if unavailability k is late and unavailability $k-1$ finishes before $s_i(k) - d_i(k)$;
3. $s_i(k) - d_i(k) - u_i(k-1) - d_i(k-1)$ otherwise (i.e., if unavailability k is late and unavailability $k-1$ finishes after $s_i(k) - d_i(k)$).

Induction basis for $k = 2$. Previous cases are enumerated (see Figure 4). In case 1), unavailability 2 is early. When unavailability 1 starts before $s_i(1) - d_i(1)$ (case a), the worst case occurs when this unavailability interrupts the longest job of this interval, which size is $M_i(1)$. As the slack rule is respected, $d_i(1) \geq M_i(1)$ and the interrupted job can be executed again before $e_i(1)$ with all uninterrupted jobs scheduled in the first interval. Similarly, unavailability 2 interrupts the job of size $M_i(2)$ and the slack is large enough for re-executing it. When unavailability 1 starts after $s_i(1) - d_i(1)$ (case b), all jobs scheduled

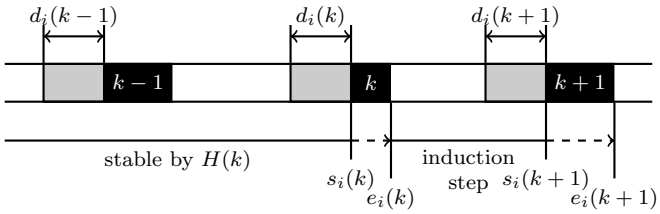

Figure 3: Induction on the intervals for the proof of Proposition 1.

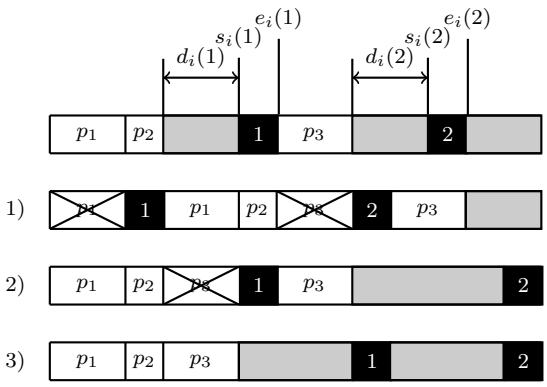

Figure 4: Induction basis for the proof of Proposition 1: initial schedule on the first line and the three cases on the following lines.

in interval 1 are executed before $s_i(1) - d_i(1)$. Then, unavailabilities 1 and 2 interrupt the longest job scheduled in interval 2. As both slacks are greater than or equal to $M_i(2)$, then this job may be restarted and completed before $e_i(2)$.

In case 2), unavailability 2 is late and unavailability 1 finishes before $s_i(2) - d_i(2)$. In the worst case, unavailability 1 will interrupt the longest job scheduled in the first two intervals. As the slack rule is respected, $d_i(1) \geq \max(M_i(1), M_i(2))$. Then, all jobs scheduled in these intervals terminate before $s_i(2) - d_i(2)$.

Finally, case 3) is when unavailability 2 is late and unavailability 1 finishes after $s_i(2) - d_i(2)$. In this case, all jobs scheduled in the first two intervals are executed without interruption. Thus, they finish their executions at $s_i(2) - d_i(2) - u_i(1) - d_i(1)$. Therefore, $H(2)$ is true.

Induction step. $H(k)$ is assumed to be true for a given $k \geq 2$. We consider again three cases depending on the date of unavailability k. In case 1), unavailability k is early and all jobs scheduled in the first k intervals are executed before $e_i(k)$ by induction hypothesis. Given the assumptions on the online strategy, the jobs scheduled in interval $k+1$ may start in interval k

(i.e., before $e_i(k)$). Although these jobs may be interrupted by previous unavailabilities, they are assumed to be restarted in interval $k+1$ (i.e., after time $e_i(k)$) during which they can only be interrupted by unavailability $k+1$. We show that $H(k+1)$ is true when unavailability k is early by considering that unavailability $k+1$ is either early or late analogously to the induction base case. In the earliness case (case a), unavailability $k+1$ interrupts the longest job scheduled in interval $k+1$ and the slack can absorb it before $e_i(k+1)$. Otherwise (case b), jobs scheduled in interval $k+1$ are not interrupted and finish thus before $s_i(k+1) - d_i(k+1)$.

In case 2), unavailability k is late, unavailability $k-1$ finishes before $s_i(k-1) - d_i(k-1)$ and all jobs scheduled in the first k intervals are executed before $s_i(k) - d_i(k)$ by induction hypothesis. We prove below that if unavailability $k+1$ is early (case a), then jobs scheduled in interval $k+1$ can be executed between $s_i(k) - d_i(k)$ and $e_i(k+1)$. This duration comprises enough slack to tolerate the longest job to be interrupted twice, by unavailability k, which is late, and by unavailability $k+1$, which is early: $d_i(k) \geq M_i(k+1)$ and $d_i(k+1) \geq M_i(k+1)$. In the case when unavailability $k+1$ is late (case b), unavailability k finishes either before or after $s_i(k+1) - d_i(k+1)$. In the first case (case b.1), unavailability k can interrupt the longest job scheduled in interval $k+1$, which can be absorbed by slack $d_i(k) \geq M_i(k+1)$. Thus, jobs scheduled in the first $k+1$ intervals are completed before $s_i(k+1) - d_i(k+1)$. In the second case (case b.2), no job is interrupted in interval $k+1$ and these jobs are completed before $s_i(k+1) - d_i(k+1) - u_i(k) - d_i(k)$.

In case 3), unavailability k is late, unavailability $k-1$ finishes after $s_i(k-1) - d_i(k-1)$ and all jobs scheduled in the first k intervals are executed before $s_i(k) - d_i(k) - u_i(k-1) - d_i(k-1)$ by induction hypothesis. Thus, jobs scheduled in interval $k+1$ are executed after this date. Since $2p_{\max} \leq a_{\min}$ and $\delta_i(k-2) \leq a_i(k-1)$, unavailability $k-2$ (for $k \geq 3$) cannot interrupt any of these jobs. Remark that this is true even when shifting the first unavailability that finishes after the horizon makes the last availability shorter than $a_{\min}$ because $2p_{\max}$ needs to be lower than or equal to the availability duration of an interval k that is never the last one. If unavailability $k+1$ is early (case a), then the longest job among the jobs scheduled in interval $k+1$ can be interrupted by unavailabilities $k-1$ to $k+1$. As $d_i(k-1)$, $d_i(k)$ and $d_i(k+1)$ are all greater than or equal to $M_i(k+1)$, there is enough duration between $s_i(k) - d_i(k) - u_i(k-1) - d_i(k-1)$ and $e_i(k+1)$ to absorb these three interruptions. The last two cases (b.1 and b.2) can be derived analogously.

In every cases, the induction hypothesis $H(k+1)$ is true when $H(k)$ is supposed to be true, which concludes the proof in the general case. □

The previous proof does not require any information on the platform characteristics because each machine is considered independently. Thus, Proposition 1 holds for any kind of parallel platforms (identical, uniform and unrelated). Additionally, the problem of generating a schedule that respects the slack rule with an optimal makespan can be proven to be NP-Hard using the same proof as in [3].

5 Algorithms

5.1 Preliminary

We propose several heuristics and present them using the following notation: $S_i(k) = \sum_{j \in \pi_i(k)} b_i p_j$ (i.e., the sum of the duration of the jobs scheduled on the kth interval of machine j). Moreover, we assume that $d_i(k) = \infty$ for each $k \leq 0$ (i.e., there is an infinite amount of idle time on non-existing intervals).

The proposed heuristics are based on a global method that prepares the intervals by shifting and sorting them. A local scheduling policy then assigns the jobs to the intervals. Each local method relies on a common mechanism for identifying the longest job that can be scheduled on a given interval. Before covering the different local methods, this mechanism is described below. This section ends with the description of the two global methods that can both use any local methods.

Algorithm 1 computes the duration of the longest job that can be scheduled on one interval. The current slack must account for the longest jobs scheduled on the three next intervals (this corresponds to the base case of Proposition 1). Moreover, any candidate job must also fit in the previous slacks due to the slack rule when applied to the previous intervals.

Algorithm 1 Procedure `longestJob`(i, k)

Input: the slack of intervals $k-2$ to k ($d_i(k-2)$, $d_i(k-1)$ and $d_i(k)$) and the length of the longest jobs scheduled on intervals k to $k+2$ ($M_i(k)$, $M_i(k+1)$ and $M_i(k+2)$)

Output: the length of the longest job that can be allocated on interval k of machine i

1: prev $\leftarrow \min(d_i(k-2), d_i(k-1))$ {The candidate job must fit in previous slacks}
2: curr $\leftarrow \frac{d_i(k)}{2}$ {The candidate job must fit in the current slack}
3: next $\leftarrow d_i(k) - \max(M_i(k), M_i(k+1), M_i(k+2))$ {Slack rule base case}
4: **return** $\min(\text{prev}, \text{curr}, \text{next})$

5.2 Local Scheduling Policies

Local scheduling policies consider intervals as bins that are successively considered for allocating jobs. These policies are then used with higher-level heuristics that select the intervals and the order in which they are visited. Local policies assume therefore that they are given an ordered set of intervals to which a set of jobs have to be allocated. The output is an allocation function π that provides the set of jobs to be executed during each kth interval on each processor.

The first scheduling policies are the classical LPT (Longest Processing Time) and SPT (Shortest Processing Time). These strategies ignore the order between the intervals and schedules jobs according to their finishing times. This finishing times include the slack to ensure that any disturbed makespan is never larger than the horizon. The only difference between LPT [4, Algorithm 2] and SPT is that the first algorithm considers jobs in non-increasing order of their processing costs, while it is non-decreasing for the second.

Algorithm 2 Longest Processing Time (LPT)

Input: a set of n jobs, a set of m machines with unavailabilities
Output: an allocation function π

$J \leftarrow \{j_1, j_2, \ldots\}$ such that $p_{j_l} \geq p_{j_{l+1}}$ {Sort jobs by non-increasing processing cost}
for all $j \in J$ **do** {Consider each job in given order}
  $E_{\min} \leftarrow \infty$
  for all $1 \leq i \leq m$ **do** {Consider each machine i candidate for allocation}
    $k \leftarrow 1$
    while $b_i p_j >$ `longestJob`(i, k) **do** {Search for the first unavailability}
      $k \leftarrow k + 1$
    end while
    $E \leftarrow e_i(k-1) + S_i(k) + b_i p_j + \max(b_i p_j, M_i(k), M_i(k+1), M_i(k+2))$
    if $E < E_{\min}$ **then** {Earliest end date of job j on machine i, slack included}
      $E_{\min} \leftarrow E$
      $(i', k') \leftarrow (i, k)$
    end if
  end for
  $\pi_{i'}(k') \leftarrow \pi_{i'}(k') \cup \{j\}$ {Schedule job j in interval k' on machine i'}
end for
return π

For clarity, we ignore the time taken to sort the jobs, $\mathcal{O}(n\log(n))$, when characterizing the complexity of local scheduling policies. The complexity of LPT is $\mathcal{O}(nK)$, which is the number of jobs times the total number of reservations ($K = \sum_{i=1}^{m} K_i$). For SPT, it is $\mathcal{O}(nm + K)$ because when a job does not fit into an interval, any subsequent jobs will not fit either.

We also use two well-known bin-packing heuristics, namely First Fit Decreasing (FFD) and First Fit Increasing (FFI). With FFD [4, Algorithm 3], jobs are considered in non-increasing order and each one is allocated to the first interval for which the slack rule is respected. While FFD shares the same complexity with LPT, FFI has time complexity of $\mathcal{O}(n + K)$ (plus the time for sorting the jobs).

Algorithm 3 First Fit Decreasing (FFD)

Input: a set of n jobs, a sequence of intervals $\mathcal{I} = \{(i_1, k_1), (i_2, k_2), \ldots\}$
Output: an allocation function π

```
J ← {j_1, j_2, ...} such that p_{j_l} ≥ p_{j_{l+1}}   {Sort jobs by non-increasing processing cost}
for all (i, k) ∈ I do {Consider each interval k on machine i in given order}
  for all j ∈ J do                                    {Consider each job in given order}
    if b_i p_j ≤ longestJob(i, j) then
      π_i(k) ← π_i(k) ∪ {j}                           {Schedule job j in interval k on machine i}
      J ← J \ {j}
    end if
  end for
end for
return π
```

We also propose a new algorithm based on the dynamic programming paradigm, *DPslack* [4, Algorithm 4], that is derived from the classical knapsack dynamic programming solution. First, jobs are sorted by non-decreasing processing cost. Then, for each interval (in the given order), we compute recursively the value $P_s(m, a_i(k))$, which is the maximum duration during which a subset of the first m jobs can be executed in the first $a_i(k)$ time units of this interval (which is distinct from $a_i(k)$ due to the slack rule and the fact that jobs may not fit perfectly in the interval). The recurrence relation is

$$P_s(j,l) = \max(P_s(j-1,l), P(j-1, l - b_i p_j - \max(b_i p_j, M_i(k+1), M_i(k+2))) + b_i p_j)$$
$$P(j,l) = \max(P(j-1,l), P(j-1, l - b_i p_j) + b_i p_j)$$

with the initialization $P_s(j,l) = P(j,l) = 0$ for $0 \le j \le m, 1 \le l \le a_i(k)$ and with $P(j,l)$ corresponding to the related knapsack problem without the slack constraint. As the jobs are ordered, the removed quantity due to the slack if job j is selected is done only for the last added job (i.e., $b_i p_j$ instead of $M_i(k)$),

which is indeed the longest. The complexity of DPslack is $\mathcal{O}(na_{\max}K)$ where $a_{\max} = \max_{1\leq i\leq m, 1\leq k\leq K_i} a_i(k)$ is the maximum availability duration.

Algorithm 4 DPslack (Dynamic Programming)

Input: a set of n jobs, a sequence of intervals $\mathcal{I} = \{(i_1, k_1), (i_2, k_2), \ldots\}$
Output: the allocated set of jobs $\pi_i(k)$

```
J ← {j_1, j_2, ...} such that p_{j_l} ≤ p_{j_{l+1}}   {Sort jobs by non-decreasing processing cost}
for all (i, k) ∈ I do {Consider each interval k on machine i in given order}
  Initialize each element of matrices P and P_s to 0
  for all j ∈ J do                                    {Consider each job in given order}
    if b_i p_j ≤ longestJob(i, k) then
      for all 1 ≤ l ≤ a_i(k) do                       {Consider a sub-interval}
        P[j][l] ← P[j − 1][l]
        P_s[j][l] ← P_s[j − 1][l]
        if b_i p_j ≤ l then                           {Check if the job fits}
          P[j][l] ← max(P[j][l], P[j − 1][l − b_i p_j] + b_i p_j)
          d ← max(b_i p_j, M_i(k + 1), M_i(k + 2))
          if b_i p_j + d ≤ l then                     {Check if the job and its slack fits}
            P_s[j][l] ← max(P_s[j][l], P[j − 1][l − b_i p_j − d] + b_i p_j)
          end if
        end if
      end for
    end if
  end for
  l ← a_i(k)                                          {Initialize the backtracking of the solution}
  s ← true                                            {Start with the slack memoization matrix}
  for all j ∈ J do                                    {Consider each job in reverse order}
    if (P_s[j][l] ≠ P_s[j − 1][l] or not s) and (P[j][l] ≠ P[j − 1][l] or s) then
      π_i(k) ← π_i(k) ∪ {j}
      l ← l − b_i p_j
      if s then
        l ← l − max(b_i p_j, M_i(k + 1), M_i(k + 2))
        s ← false                                     {Switch to the other matrix}
      end if
    end if
  end for
end for
return π
```

5.3 Global Scheduling Policies

The global scheduling policies prepare the instance by shifting and sorting the intervals before calling local ones for scheduling each job. Any scheduling policy can then be used (FFI, FFD, SPT, LPT, DPslack) with both following strategies.

First, we present a greedy strategy, *GreedySlack* [4, Algorithm 5], in which availabilities are sorted by non-decreasing end date of unavailability. Intervals are shifted by advancing their unavailabilities as much as possible, which corresponds to the worst case. The horizon is the end date of the last allocated job plus the associated slack. The complexity of this strategy is $\mathcal{O}(K \log(K))$ and it calls a local policy only once. While LPT and SPT are only affected by the shifting of the intervals and not their order, it is the contrary for bin-based heuristics (FFI, FFD and DPslack).

Algorithm 5 GreedySlack

Input: a set of n jobs, a set of m machines with unavailabilities
Output: an allocation function π

1: **for all** $1 \leq i \leq m$ **do** {For all machines}
2: **for all** $0 \leq k < K_i$ **do** {For all unavailabilities}
3: $s_i(k) \leftarrow e_i(k)$ {Move unavailabilities at the start of the intervals}
4: $e_i(k) \leftarrow e_i(k) + u_i(k+1)$
5: **end for**
6: $K_i \leftarrow K_i - 1$
7: **end for**
8: $\mathcal{I} \leftarrow \{(i_1, k_1), (i_2, k_2), \ldots\}$ such that $s_{i_l(k_l)} \leq s_{i_{l+1}}(k_{l+1})$ {Sort intervals by non-decreasing availability end date}
9: Call a scheduling policy with $\mathcal{I}$ and the jobs to obtain π
10: **return** π

The second solution, *DAslack* ([4, Algorithm 6]), uses the dual approximation paradigm that consists in scheduling all jobs within various horizons until the minimum one is reached (by binary search) [8]. For each horizon, the availabilities are sorted by non-decreasing duration because we assume it is easier to fill small availabilities first. The complexity of this strategy is $\mathcal{O}(K \log(K))$ for sorting the intervals. It calls a local strategy $\mathcal{O}(\log(M))$ times where M is an upper bound on the horizon and it takes $\mathcal{O}(\log(K))$ steps to adapt the intervals each time. For instance, the complete complexity of the combination DPslack and DAslack is $\mathcal{O}(K \log(K) + n \log(n) + \log(M)(\log(K) + n a_{\max} K))$ (sorting the jobs has to be done only once).

Algorithm 6 DAslack (Dual Approximation)

Input: a set of n jobs, a set of m machines with unavailabilities
Output: an allocation function π

```
u ← upper bound on λ (e.g., max_{1≤i≤m,1≤k≤K_i} e_i(k))
l ← lower bound on λ (e.g., 0)
while u − l > 1 do
    λ ← (u+l)/2
    for all 1 ≤ i ≤ m do                                  {All machines}
        for all 0 < k ≤ K_i do                            {All unavailabilities}
            if e_i(k) > λ then                            {Move last interval}
                s_i(k) ← max(λ − u_i(k), e_i(k − 1))
                e_i(k) ← λ
            end if
            if e_i(k) = λ then                            {Discard the next intervals}
                K_i ← k
            end if
        end for
    end for
    I ← {(i_1, k_1), (i_2, k_2), ...} such that a_{i_l(k_l)} ≤ a_{i_{l+1}}(k_{l+1})  {Sort intervals
    by non-decreasing availability duration}
    Call a scheduling policy with I and the jobs to obtain π
    if π is invalid then
        u ← λ
    else
        l ← λ
    end if
end while
return last valid schedule
```

6 Experiments

We evaluated the proposed algorithms by simulating the execution of a set of jobs on a set of availabilities issued from an actual trace gathered from BOINC using the scheduling algorithms proposed in Section 5 in order to observe the impact of the slack on their stability.

6.1 Settings

6.1.1 Availabilities

The traces of the hosts were gathered by BOINC clients that were installed on volunteer hosts between April 1, 2007 and January 1, 2009 for the SETI@home project [10]. In total, about 230,000 hosts were involved and provided over 57,800 years of CPU availability spread on 102,416,434 intervals continuous intervals. We selected 10 arbitrarily subsets of 500 processors in the middle of the trace and only processors for which the speed is known were selected.

6.1.2 Workload

The traces of the jobs were collected from Docking@Home project[1] and provided over 150,000 jobs. We selected 10 subsets of 10,000 jobs. Each execution duration is rounded to the closest integer in second.

6.1.3 Disturbed Instance Generation

In order to generate disturbed instances, we modified the start and end times of the availabilities following a uniform law. Unavailabilities that overlap following this process were merged.

6.2 Methodology

We performed three sets of simulations to study the main characteristics of the proposed algorithms:

Performance In order to compare the performance of the proposed algorithms in terms of schedule length, we run all the algorithms and measured the horizon. For each instance, we computed the ratios between the horizons obtained with all heuristics and the minimum horizon among this set.

[1]The traces were provided by Michella Taufer.

Stability Following the adopted proactive scheme, the execution scenarios are run in two phases: each schedule algorithm is first run using the predicted non-disturbed dates before being run on 30 disturbed instances.

Execution time We measured the execution time to build any schedule.

6.3 Analysis

As a baseline, we included a version of each algorithm that ignores the slack rule (denoted *without slack rule* in Figures 5 to 7). These figures depict values using boxplots in which the bold line is the median, the box shows the quartiles, the bars show the whiskers (1.5 times the interquartile range from the box) and additional points are outliers.

6.3.1 Performance Comparison

Figure 5 show the horizon ratios for each heuristic. For instance, the horizon obtained with LPT are about 1.5 times the minimum horizon that can be obtained with any other heuristic.

We first observe that any heuristic that ignores the slack rule is significantly better than its counterpart (respecting the slack rule incurs at least a 50% degradation). Additionally, those that use DAslack are among the best ones.

We can also see that DAslack leads to better results than GreedySlack except for SPT and LPT, which do not benefit from the binary search. Moreover, with GreedySlack, LPT outperforms bin-based heuristics (FFI, FFD and DPslack) due to the last bins that are filled even though the jobs could finish earlier on other machines. Using DAslack limits this problem as the last bins are cut.

Overall, the best heuristic is LPT (with both general policies), followed by FFD combined with DAslack.

6.3.2 Impact of the Slack on the Stability

The boxplots in Figure 6 represent the ratios between the disturbed values of the makespan and the predicted horizons. Schedules are always stable (stability below one) with heuristics that respect the slack rule, which is consistent with Proposition 1. The least stable strategies (those that do not respect the slack rule) lead also to the most compact schedule as suggested by Figure 5. Inversely, the most stable ones (FFI, SPT and DPslack combined with GreedySlack) are also the least efficient ones. This emphasizes, as expected, the antagonism between the efficiency and the stability.

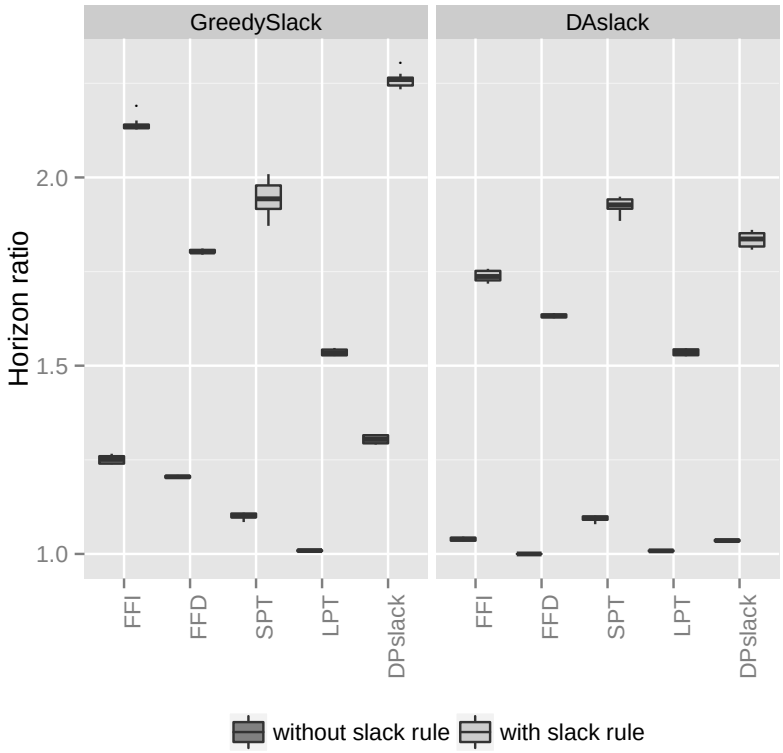

Figure 5: Horizon ratios of schedules generated with 10 sets of 10,000 jobs and 10 sets of 500 machines (100 instances). Each combination between a global and a local policy has an additional version that ignores the slack rule (left).

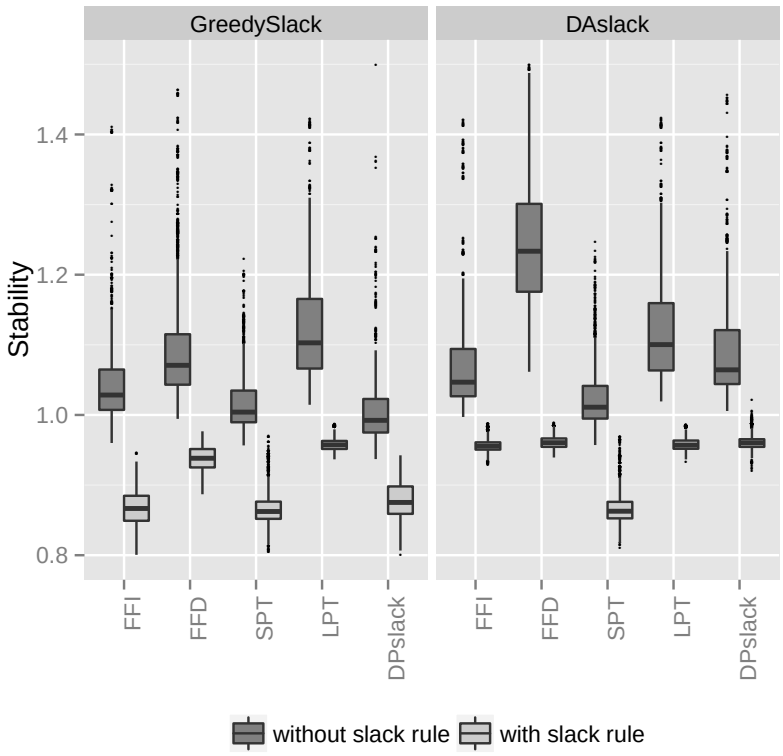

Figure 6: Stabilities of schedules generated with 10 sets of 10,000 jobs and 10 sets of 500 machines (100 instances). Each combination between a global and a local policy has an additional version that ignores the slack rule (left).

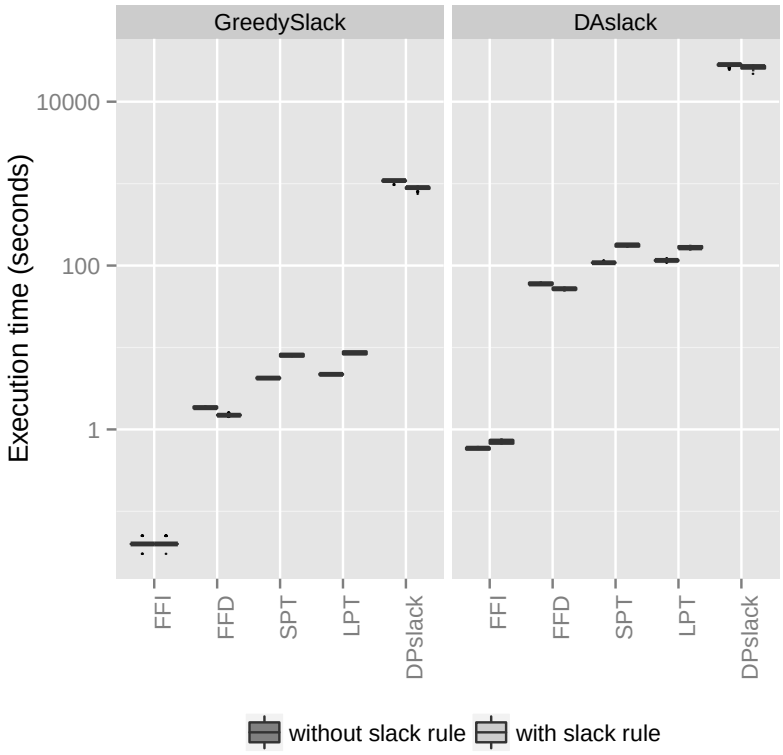

Figure 7: Execution times for generating the same schedules as in Figures 5 and 6.

6.3.3 Execution Time

Figure 7 depicts the execution times of the heuristics, which are sorted by their execution times. The ratio between the execution times of GreedySlack strategies and DAslack is constant as this last strategy repeats each local policies the same number of times. Overall, the fastest local policy is FFI while the slowest is DPslack and all other policies have similar costs.

To conclude, the combination of LPT with GreedySlack shows the best performance both in terms of efficiency and stability for a reasonable cost. When the cost is an issue, the next fastest strategy with decent performance is FFI when combined with DAslack (or even GreedySlack if the cost is particularly critical).

7 Conclusion

This paper focuses on the problem of scheduling jobs under unavailability constraints that may occur earlier of later than expected. We derived a rule that a scheduling algorithm must respect to produce stable solutions (i.e., schedules for which the effective duration is no longer than the expected one). A collection of algorithms was proposed and assessed on simulations with actual traces. Among the studies heuristics, which all achieve stability, LPT minimizes efficiency degradation the most.

As future works, we could study other complementary mechanisms such as checkpointing and migration. Also, the stretch is an interesting metric in the context of desktop grids (where a continuous flow of jobs must be computed) and has not been considered in this work. Finally, we plan to extend our empirical investigation to new settings such as testing multiple levels of disturbance.

8 Acknowledgement

The authors would like to thank Grégory Mounié with whom this work was initiated.

References

[1] D. P. Anderson. Boinc: A system for public-resource computing and storage. In *5th International Workshop on Grid Computing (GRID)*, pages 4–10, Nov. 2004.

[2] J.-C. Billaut, A. Moukrim, and E. Sanlaville. *Flexibility and Robustness in Scheduling*. Wiley Online Library, 2008.

[3] L.-C. Canon, A. Essafi, G. Mounié, and D. Trystram. A Bi-Objective Scheduling Algorithm for Desktop Grids with Uncertain Resource Availabilities. In *Euro-Par*, Bordeaux, France, Sept. 2011.

[4] L.-C. Canon, A. Essafi, and D. Trystram. A Proactive Approach for Coping with Uncertain Resource Availabilities on Desktop Grids. Technical Report RRDISC2014-1, FEMTO-ST, Mar. 2014. `http://lifc.univ-fcomte.fr/~lccanon/report2014.html`.

[5] F. Diedrich, K. Jansen, F. Pascual, and D. Trystram. Approximation Algorithms for Scheduling with Reservations. *Algorithmica*, 58(2):391–404, 2010.

[6] L. Eyraud, G. Mounié, and D. Trystram. Analysis of Scheduling Algorithms with Reservations. In *21st IEEE International Parallel & Distributed Processing Symposium*, 2007.

[7] M. Fallah, M. Aryanezhad, and B. Ashtiani. Preemptive resource constrained project scheduling problem with uncertain resource availabilities: Investigate worth of proactive strategies. In *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 646 –650, Dec. 2010.

[8] D. S. Hochbaum and D. B. Smoys. Using dual approximation algorithms for scheduling problems: theoretical and practical results. *Journal of ACM*, 34(1):144–162, 1987.

[9] H.-C. Hwang and S. Y. Chang. Parallel Machines Scheduling with Machine Shutdowns. *Computers & Mathematics with Applications*, 36(11):21–31, Aug. 1998.

[10] B. Javadi, D. Kondo, J.-M. Vincent, and D. P. Anderson. Discovering statistical models of availability in large distributed systems: An empirical study of seti@home. *Parallel and Distributed Systems, IEEE Transactions on*, 22(11):1896–1903, 2011.

[11] D. Kondo, B. Javadi, A. Iosup, and D. H. J. Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. In *CCGrid*, pages 398–407. IEEE, 2010.

[12] O. Lambrechts, E. Demeulemeester, and W. Herroelen. Proactive and reactive strategies for resource-constrained project scheduling with uncertain resource availabilities. *Journal of Scheduling*, 11(2):121–136, 2008.

[13] O. Lambrechts, E. Demeulemeester, and W. Herroelen. Time slack-based techniques for robust project scheduling subject to resource uncertainty. *Annals of Operations Research*, pages 1–22, 2010.

[14] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. *ArXiv e-prints*, Jan. 2009.

[15] C.-Y. Lee. Parallel machines scheduling with nonsimultaneous machine available time. *Discrete Applied Mathematics*, 30(1):53–61, Jan. 1991.

[16] C.-Y. Lee. Machine scheduling with an availability constraint. *Journal of Global Optimization*, 9(3):363–382, 1996.

[17] J. Y.-T. Leung, editor. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman & Hall/CCR, 2004.

[18] C.-J. Liao, D.-L. Shyur, and C.-H. Lin. Makespan minimization for two parallel machines with an availability constraint. *European Journal of Operational Research*, 160(2):445–456, June 2005.

[19] LIGO Scientific Collaboration. The Einstein@Home search for periodic gravitational waves in LIGO S4 data. *ArXiv e-prints*, Apr. 2008.

[20] D. Xu, Z. Cheng, Y. Yin, and H. Li. Makespan minimization for two parallel machines scheduling with a periodic availability constraint. *Computers & Operation Research*, 36(6):1809–1812, June 2009.

Annex

Symbol	Definition
i	index of the machines
j	index of the jobs
k	index of the intervals
m	number of machines
n	number of jobs
K_i	number of intervals on machine i
K	total number of intervals ($\sum_{i=1}^{m} K_i$)
b_i	cycle time of machine i
p_j	processing cost of job j
$a_i(k)$	duration of availability k on machine i
$u_i(k)$	duration of unavailability k on machine i
$s_i(k)$	start date of unavailability k on machine i
$e_i(k)$	end date of unavailability k on machine i
$p_{\max}$	maximum execution duration ($\max_{1\leq i\leq m, 1\leq j\leq n} b_i p_j$)
$a_{\min}$	minimum availability duration ($\min_{1\leq i\leq m, 1\leq k\leq K_i} a_i(k)$)
$a_{\max}$	maximum availability duration ($\max_{1\leq i\leq m, 1\leq k\leq K_i} a_i(k)$)
$\delta_i(k)$	disturbance on unavailability start date $s_i(k)$
$\pi_i(k)$	set of jobs allocated to availability k on machine i
C_j	completion time of job j in a given schedule
$C_{\max}$	makespan ($\max_{1\leq j\leq n} C_j$)
$\widetilde{C}_{\max}$	highest disturbed makespan
λ	horizon
σ	stability ($\frac{\widetilde{C}_{\max}}{\lambda}$)
$M_i(k)$	maximum size of the jobs assigned to availability k on machine i ($M_i(k) = \max_{j\in\pi_i(k)} b_i p_j$)
$S_i(k)$	sum of the sizes of the jobs assigned to availability k on machine i ($S_i(k) = \sum_{j\in\pi_i(k)} b_i p_j$)
$d_i(k)$	slack preceding unavailability k on machine i ($a_i(k) - S_i(k)$)

Table 1: Notation summary

FEMTO-ST INSTITUTE, headquarters
32 avenue de l'Observatoire - F-25044 BESANÇON Cedex FRANCE
Tél : (33 3) 81 85 39 99 – Fax : (33 3) 81 85 39 68 – e-mail : contact@femto-st.fr

FEMTO-ST - AS2M : TEMIS, 24 rue Alain Savary, F-25000 Besançon
FEMTO-ST - DISC : UFR Sciences - Route de Gray - F-25030 Besançon cedex France
FEMTO-ST - ENERGIE : Parc Technologique, 2 Av. Jean Moulin, Rue des entrepreneurs, F-90000 Belfort France
FEMTO-ST - MEC'APPLI : 24, chemin de l'épitaphe - F-25000 Besançon France
FEMTO-ST - MN2S : 32, rue de l'Observatoire - F-25044 Besançon cedex France
FEMTO-ST - OPTIQUE : UFR Sciences - Route de Gray - F-25030 Besançon cedex France
FEMTO-ST - TEMPS-FREQUENCE : 26, Chemin de l'Epitaphe - F-25030 Besançon cedex France

http://femto-st.fr