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

Louis-Claude Canon
Université de Franche-Comté, DISC, FEMTO-ST
16, route de Gray
25000 Besançon, France

Adel Essafi
LaTICE research Laboratory, University of Tunis,
Bab Mnara, Tunis, Tunisia

Denis Trystram
Univ. Grenoble Alpes, 655 avenue de l’Europe,
38334 St Ismier, France
Institut Universitaire de France

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 estimation, 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 that 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 based on actual traces.

Keywords—Scheduling; desktop grids; uncertainties; availabilities.

I. INTRODUCTION

Harnessing the power of modern parallel platforms is compromised by many uncertainties coming from their high scales and resources volatility. 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 [1], [2]. 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 performance 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 [3].

To the best of our knowledge, there is few related work studying scheduling with unavailability under uncertainties except our previous contribution [4] 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 depend 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 [4]. 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 [5]. These results are also available in the companion research report [6].

The paper is organized as follows. We start by briefly recalling the most significant studies on scheduling under availability constraints in Section II. Section III is devoted to the description of the computation model and the formal presentation of the problem. Section IV provides a structural analysis of the proposed slack-based mechanism in terms of stability. Then, Section V describes the various algorithms and their combinations. Before concluding, Section VI presents some experiments based on simulations on actual workflows and availability constraints.
II. 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 (Longest Processing Times) for which many assumptions have been explored. For instance, Lee [7], [8] 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 availability patterns. 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. [10] 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 [11] 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 [12]. Although this problem does not admit Fully Polynomial Time Approximation Scheme (FPTAS), a costly PTAS based on the multiple knapsack was designed [13]. While all the previous approaches are related to sequential jobs, Eyraud et al. [14] studied the problem of scheduling with unavailabilities for parallel rigid jobs. 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 [4]. 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 [15], the authors investigate the use of preemption and in [16], [17], the authors explored stochastic resource breakdown. Casanova et al. [18] study a similar problem in which each job requires to receive data from the master. They also consider that jobs may be temporarily suspended by the user and they propose heuristics for the online case. Finally, Wingstrom et al. [19] relies on statistical characteristics of the availabilities for scheduling the jobs.

III. Description of the Problem

All notations are summarized in Table I.

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 execution cost of any task is thus proportional to the processor cycle time. 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 period followed by an unavailability period. The intervals are indexed by \( k \) and the start date of availability period \( k \) is denoted by \( s_k \). As depicted in Figure 1, the availability (resp. unavailability) starts at \( s_k \) (resp. \( u_k \)). Hence, this period ends at \( e_k \) and we also have \( s_k = a_i(k) \). To be consistent, the first availability on processor \( i \) starts at \( e_i(0) \) (by default, \( e_i(0) = -\infty \) and no job is scheduled on the 0th interval) and there are \( K_i \) intervals on machine \( i \).

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., \( \max_{1 \leq l \leq m, 1 \leq k \leq n} b_i p_j \)) and \( a_{\min} \) is the shortest availability (i.e., \( \min_{1 \leq l \leq m, 1 \leq i \leq K} a_i(k) \)). This assumption is discussed below in Section III-4.

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. [3] 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 \( k \)th 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) 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 \( k \)th interval on each 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 \( \lambda \), which measures the performance and is specified while building the schedule. Since jobs allocated to any interval must fit into the availability, the horizon is then greater than or equal to the makespan without uncertainties \( C_{\text{max}} \triangleq \max_{1 \leq j \leq n} C_j \) where \( C_j \) is the completion time of job \( j \) in a given schedule [20]). In [4], we have considered the makespan instead of a horizon. As a consequence, we assumed that at least one processor did not have any unavailability which is unrealistic and not required anymore. Using a 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 \( \sigma \). It is defined as the ratio between the highest disturbed makespan \( C_{\text{max}} \) among all possible disturbed scenarios and the specified horizon \( \lambda \), i.e., \( \sigma \triangleq \frac{C_{\text{max}}}{\lambda} \). 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.

4) Discussion: The proposed models rely on some assumptions that are briefly discussed below.

The \( 2p_{\text{max}} \leq a_{\text{min}} \) assumption has been verified by analyzing actual data from the Failure Trace Archive [21]. 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). This assumption prevents cascading interruptions (the same job being interrupted by every unavailability) and is used in the proof of Proposition 1.

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.

IV. 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, at runtime, jobs scheduled in interval \( k \) are not started before jobs scheduled in interval \( k-1 \) has 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 \( M_i(k) \triangleq \max_{j \in \pi_i(k)} b_jp_j \) be the duration of the longest job assigned to the \( k \)th interval on processor \( i \) (\( M_i(k) \triangleq 0 \) if no job is assigned, i.e., 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 \( 2p_{\text{max}} \leq a_{\text{min}} \) 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 given as follows. 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
We show that when unavailability \( u_k \) finishes before \( s_k \) in interval \( k \), \( s_k \) is an infinite amount of idle time on non-existing intervals). In the worst case, unavailability \( u_k \) is late and unavailability \( u_{k-1} \) are all greater than or equal to \( M_i(k+1) \). Thus, jobs scheduled in the first \( k+1 \) intervals are completed before \( s_k(k+1) - d_i(k+1) \). In the second case (case b.2), no job is interrupted in interval \( k+1 \) and these jobs are completed before \( s_k(k+1) - d_i(k+1) - u_i(k) - d_i(k) \).

In case 3), unavailability \( u_k \) is late, unavailability \( u_{k-1} \) finishes after \( s_k(k-1) - d_i(k-1) \) and all jobs scheduled in the first \( k \) intervals are executed before \( s_k(k) - d_i(k) \) 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 \( s_k(k) - d_i(k) \) 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 \( s_k(k+1) - d_i(k+1) \). 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 \( s_k(k+1) - d_i(k+1) \). In the second case (case b.2), no job is interrupted in interval \( k+1 \) and these jobs are completed before \( s_k(k+1) - d_i(k+1) - u_i(k) - d_i(k) \).

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 is NP-Hard using the same proof as in [4].

V. ALGORITHMS

A. 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 \( k \)th interval of machine \( i \)). 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 (Algorithm 1). This section ends with the description of the two global methods that can both use any local method.

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 into 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 ← min(d_i, (k − 2), d_i, (k − 1)) {The candidate job must fit into previous slacks}
2: curr ← d_i, (k) {The candidate job must fit into the current slack}
3: next ← d_i, (k) − max(M_i, (k), M_i, (k + 1), M_i, (k + 2)) {Slack rule base case}
4: return min(prev, curr, next)

Algorithm 2 Longest Processing Time (LPT)
Input: a set of n jobs, a set of m machines with unavailabilities
Output: an allocation function \( \pi \)
1: \( J \leftarrow \{j_1, j_2, \ldots\} \) such that \( p_{j_1} \geq p_{j_{i+1}} \) {Sort jobs by non-increasing processing cost}
2: for all \( j \in J \) do {Consider each job in given order}
3: \( E_{\min} \leftarrow \infty \)
4: for all \( 1 \leq i \leq m \) do {Consider each machine i candidate for allocation}
5: \( k \leftarrow 1 \)
6: while \( b_i p_j > \text{longestJob}(i, k) \) do {Search for the first unavailability}
7: \( k \leftarrow k + 1 \)
8: end while
9: \( E \leftarrow e_i(k - 1) + S_i(k) + b_i p_j + \max(b_i p_j, M_i, (k), M_i, (k + 1), M_i, (k + 2)) \)
10: if \( E < E_{\min} \) then {Earliest end date of job j on machine i, slack included}
11: \( E_{\min} \leftarrow E \)
12: \( (i', k') \leftarrow (i, k) \)
13: end if
14: end for
15: \( \pi_{i'}(k') \leftarrow \pi_{i'}(k') \cup \{j\} \) {Schedule job j in interval k' on machine i'}
16: end for
17: return \( \pi \)

Algorithm 3 First Fit Decreasing (FFD)
Input: a set of n jobs, a sequence of intervals \( I = \{(i_1, k_1), (i_2, k_2), \ldots\} \)
Output: an allocation function \( \pi \)
1: \( J \leftarrow \{j_1, j_2, \ldots\} \) such that \( p_{j_1} \geq p_{j_{i+1}} \) {Sort jobs by non-increasing processing cost}
2: for all \( (i, k) \in I \) do {Consider each interval k on machine i in given order}
3: for all \( j \in J \) do {Consider each job in given order}
4: if \( b_i p_j \leq \text{longestJob}(i, j) \) then
5: \( \pi_{i}(k) \leftarrow \pi_{i}(k) \cup \{j\} \) {Schedule job j in interval k on machine i}
6: \( J \leftarrow J \setminus \{j\} \)
7: end if
8: end for
9: end for
10: return \( \pi \)

B. 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 \( \pi \) 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 (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.

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

We also propose a new algorithm based on the dynamic programming paradigm, DPslack (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 into the interval). The recurrence relation is

\[
P_s(j, l) = \max(P_s(j - 1, l), P(j - 1, l - b_j p_j) - \max(b_j p_j, M_i, (k + 1), M_i, (k + 2))) + b_j p_j)
P(j, l) = \max(P(j - 1, l), P(j - 1, l - b_j p_j) + b_j p_j)
\]

with the initialization \( P_s(j, l) = P(j, l) = 0 \) for \( 0 \leq j \leq m, 1 \leq l \leq a_i(k) \) 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., \( b_j p_j \) instead of \( M_i(k) \)), which is indeed the longest. The complexity of \( DP_{slack} \) is \( O(n a_{\text{max}} K) \) where \( a_{\text{max}} = \max_{1 \leq i \leq m, 1 \leq k \leq K} a_i(k) \) is the maximum availability duration.

**Algorithm 4 DPslack (Dynamic Programming)**

**Input:** a set of \( n \) jobs, a sequence of intervals \( I = \{(i_1, k_1), (i_2, k_2), \ldots\} \)

**Output:** the allocated set of jobs \( \pi_i(k) \)

1. \( J \leftarrow \{j_1, j_2, \ldots\} \) such that \( p_{j_1} \leq p_{j_{i+1}} \) \{Sort by non-decreasing processing cost\}
2. for all \( (i,k) \in I \) do \{Consider each interval \( k \) on machine \( i \) in given order\}
3. Initialize each element of matrices \( P \) and \( P_s \) to 0
4. for all \( j \in J \) do \{Consider each job in given order\}
5. if \( b_j p_j \leq 1 \) then \{Consider a sub-interval\}
6. \( P[j][l] \leftarrow P[j − 1][l] \)
7. \( P_s[j][l] \leftarrow P_s[j − 1][l] \)
8. if \( b_j p_j \leq l \) then \{Check if the job fits\}
9. \( P[j][l] \leftarrow \max(P[j][l], P[j − 1][l − b_j p_j] + b_j p_j) \)
10. \( d \leftarrow \max(b_j p_j, M_i(k + 1), M_i(k)) + b_j p_j \)
11. if \( b_j p_j + d \leq l \) then \{Check if the job and its slack fits\}
12. \( P_s[j][l] \leftarrow \max(P_s[j][l], P[j − 1][l − b_j p_j] + b_j p_j) \)
13. end if
14. end if
15. end if
16. end for
17. end if
18. end for
19. \( l \leftarrow a_i(k) \) \{Initialize the backtracking of the solution\}
20. \( s \leftarrow \text{true} \) \{Start with the slack memoization matrix\}
21. for all \( j \in J \) do \{Consider each job in reverse order\}
22. if \( (P[j][l] \neq P[j − 1][l] \text{ or not } s) \text{ and } (P[j][l] \neq P[j − 1][l] \text{ or } s) \) then
23. \( \pi_i(k) \leftarrow \pi_i(k) \cup \{j\} \)
24. \( l \leftarrow l − b_j p_j \)
25. if \( s \) then
26. \( s \leftarrow \text{false} \) \{Switch to the other matrix\}
27. end if
28. end if
29. end for
30. end for
31. return \( \pi \)

**C. 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 (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 \( O(K \log(K)) \) 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 \( \pi \)

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. \( I \leftarrow \{(i_1, k_1), (i_2, k_2), \ldots\} \) such that \( s_{i_1}(k_1) \leq s_{i_{t+1}}(k_{t+1}) \) \{Sort intervals by non-decreasing availability end date\}
9. Call a scheduling policy with \( I \) and the jobs to obtain \( \pi \)
10. return \( \pi \)

The second solution, \( DA_{slack} \) (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) [13]. 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 \( O(K \log(K)) \) for sorting the intervals. It calls a local strategy \( O(\log(M)) \) times where \( M \) is an upper bound on the horizon and it takes \( O(\log(K)) \) steps to adapt the intervals each time. For instance, the complete complexity of the combination \( DP_{slack} \) and \( DA_{slack} \) is \( O(K \log(K) + n \log(n) + \log(M)(\log(K) + n a_{\text{max}} K)) \) (sorting the jobs has to be done only once).

VI. 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 various combinations of the scheduling algorithms proposed in Section V in order to observe the impact of the slack on their stability. The code used in this section is available in [22].

A. Settings

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 [23]. In total, about 230,000 hosts were involved and provided over 57,800 years of CPU availability spread on 102,416,434 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.

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

---

\(^1\)Trilce Estrada et al. [24] modeled the in-progress delay, i.e., the computation time required by jobs in several desktop grid projects. Additionally, she provided us these workload traces.
Algorithm 6 DAslack (Dual Approximation)

Input: a set of \(n\) jobs, a set of \(m\) machines with unavailabilities

Output: an allocation function \(\pi\)

1. \(u \leftarrow\) upper bound on \(\lambda\) (e.g., \(\max_{1 \leq i \leq m, 1 \leq k \leq K_i} e_i(k)\))
2. \(l \leftarrow\) lower bound on \(\lambda\) (e.g., 0)
3. while \(u - l > 1\) do
4. \(\lambda \leftarrow \frac{u + l}{2}\)
5. for all \(1 \leq i \leq m\) do \{All machines\}
6. \(\lambda \leftarrow \frac{u + l}{2}\)
7. for all \(0 < k \leq K_i\) do \{All unavailabilities\}
8. if \(e_i(k) > \lambda\) then \{Move last interval\}
9. \(s_i(k) \leftarrow \max(\lambda - u_i(k), e_i(k-1))\)
10. \(e_i(k) \leftarrow \lambda\)
11. end if
12. if \(e_i(k) = \lambda\) then \{Discard the next intervals\}
13. \(K_i \leftarrow k\)
14. end if
15. end for
16. \(\mathcal{I} \leftarrow \{(i_1, k_1), (i_2, k_2), \ldots\}\) such that \(a_{i_t}(k_t) \leq a_{i_{t+1}}(k_{t+1})\) \{Sort intervals by non-decreasing availability duration\}
17. Call a scheduling policy with \(\mathcal{I}\) and the jobs to obtain \(\pi\)
18. if \(\pi\) is invalid then
19. \(u \leftarrow \lambda\)
20. else
21. \(l \leftarrow \lambda\)
22. end if
23. end while
24. return last valid schedule

rounded to the closest integer in second. Half the jobs have between \(2.789 \times 10^{13}\) and \(3.177 \times 10^{13}\) FLOP (about three hours on a classical core).

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.

B. 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.

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. The implementation was compiled with the `-O2` optimisation flag with `gcc` version 4.4.7 and was run on Intel Xeons CPU E5-2660 at 2.20GHz.

C. 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.

1) Performance Comparison: Figure 5 shows the horizon ratios for each heuristic. For instance, horizons 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.

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.

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 no significant variations.

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).

**VII. Conclusion**

This paper focuses on the problem of scheduling jobs under unavailability constraints that may occur earlier or 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 performance metric in the context of deskop 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.

**VIII. Acknowledgement**

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

Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté.

**References**


