HAL
archives-ouvertes.fr

# Scheduling Tightly-Coupled Applications on Heterogeneous Desktop Grids

Henri Casanova, Fanny Dufossé, Yves Robert, Frédéric Vivien

## To cite this version:

Henri Casanova, Fanny Dufossé, Yves Robert, Frédéric Vivien. Scheduling Tightly-Coupled Applications on Heterogeneous Desktop Grids. HCW 2013 - 22nd International Heterogeneity in Computing Workshop, May 2013, Boston, United States. 2013. <hal-00788606>

**HAL Id: hal-00788606**

**https://hal.inria.fr/hal-00788606**

Submitted on 18 Feb 2013

**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.

# Scheduling Tightly-Coupled Applications on Heterogeneous Desktop Grids

Henri Casanova[1], Fanny Dufossé[2], Yves Robert[3,4] and Frédéric Vivien[3]
1. Univ. of Hawai'i at Manoa, Honolulu, USA, henric@hawaii.edu
2. Laboratoire I3S, Polytech'Nice, France, dufosse@i3s.unice.fr
3. ENS Lyon & INRIA, France, {Yves.Robert|Frederic.Vivien}@ens-lyon.fr
4. University of Tennessee Knoxville, USA

***Abstract*—Platforms that comprise volatile processors, such as desktop grids, have been traditionally used for executing independent-task applications. In this work we study the scheduling of tightly-coupled iterative master-worker applications onto volatile processors. The main challenge is that workers must be simultaneously available for the application to make progress. We consider three additional complications: (i) one should take into account that workers can become temporarily reclaimed and, for data-intensive applications; (ii) one should account for the limited bandwidth between the master and the workers; and (iii) workers are strongly heterogeneous, with different computing speeds and availability probability distributions.**

**In this context, our first contribution is a theoretical study of the scheduling problem in its off-line version, i.e., when processor availability is known in advance. Even in this case the problem is NP-hard. Our second contribution is an analytical approximation of the expectation of the time needed by a set of workers to complete a set of tasks and of the probability of success of this computation. This approximation relies on a Markovian assumption for the temporal availability of processors. Our third contribution is a set of heuristics, some of which use the above approximation to favor reliable processors in a sensible manner. We evaluate these heuristics in simulation. We identify some heuristics that significantly outperform their competitors and derive heuristic design guidelines.**

## I. Introduction

In this paper we study the problem of scheduling parallel applications onto volatile processors. We target typical scientific iterative applications in which a master process parallelizes the execution of each iteration across worker processes. Each iteration requires the execution of a fixed number of tasks, with a global synchronization at the end of each iteration. In [1] we have studied this problem when these tasks are independent. In this work instead we consider tightly-coupled tasks that exchange data throughout each iteration, thus requiring that workers be simultaneously available. This work and that in [1] cover the two extremes of the parallelization spectrum, and are together representative of a large class of scientific applications. However, the new requirement that workers be all simultaneously available for tightly-coupled applications introduces another level of difficulty in all resource selection and scheduling decisions.

We consider a platform that consists of processors that alternate between periods of availability and periods of unavailability. When available each processor runs a worker process, and a master process can choose to enroll a subset of these workers to participate in the application execution. Worker unavailability can be due to software faults, in which case unavailability may last only the time of a reboot. A hardware failure can lead to a longer unavailability period, until a repair is completed and followed by a reboot. We consider a third source of processor unavailability, which comes from cycle-stealing scenarios: when a processor is contributed to the platform by an individual owner, this owner can reclaim it at any time without notice for some unknown length of time. A difference here is that the processor is merely preempted (as opposed to being terminated) until the processor is no longer reclaimed. A worker process on this processor can later resume its computation. Accordingly, we use a 3-state availability model: $UP$ (available), $DOWN$ (crashed, computation is lost) and $RECLAIMED$ (preempted, but computation can resume later). Our platform model also accounts for the fact that, due to bandwidth limitation, the master is only able to communicate simultaneously with a limited number of workers (to send them the application program as well as task data). This limitation corresponds to the bounded multi-port model [2]. It turns out that limiting the communication capacity of the master dramatically complicates the design of scheduling strategies. But without this limitation, it would be in principle possible to enroll thousands of new processors at each iteration, which is simply not feasible in practice even if this many processors are available.

Given the above application and platform models, and given a deadline (typically expressed in hours or days), the scheduling problem under study is that of maxi-

An extended abstract of this paper appears in Euro-PDP'2013.This work has been supported in part by the French ANR *Rescue* project.

mizing the expected number of application iterations successfully completed before the deadline. Informally, during each iteration, one must use the "best" processors among those that are simultaneously $UP$; these could be the fastest ones, or those expected to remain $UP$ for the longest time. In addition, with processors failing, becoming reclaimed, and becoming $UP$ again later, one has to decide when and how to change the set of currently enrolled processors. Each such change comes at a price: first, the application program needs to be sent to newly enrolled processors, thereby consuming some of the master's bandwidth; second, and more importantly, iteration computation that was only partially completed is lost due to the tight coupling of tasks.

Our contribution in this work is threefold. First, we determine the complexity of the off-line scheduling problem, i.e., when processor availability is known in advance. Even with such knowledge the problem is NP-hard. Second, we compute approximations of the expectation of the time needed by a set of processors to complete a set of tasks and of the probability that this computation succeeds. These approximations provide a sound basis for making sensible scheduling decisions. Third, we design several on-line heuristics that we evaluate in simulation. Some of these contributions assume Markovian processor availability, which is not representative of real-world platforms but provides a tractable framework for obtaining theoretical and experimental results in laboratory conditions.

This paper is organized as follows. In Section II, we discuss related work. We give a formal definition of the application and platform models in Section III. In Section IV we define the scheduling problem and establish off-line complexity results. In Section V, we introduce a 3-state Markovian model of processor availability, and use this model to compute approximations of relevant probabilistic quantities. In Section VI, we describe several heuristics for solving the on-line scheduling problem, which are evaluated in simulation in Section VII. Finally, we summarize our results and provide perspectives on future work in Section VII-B.

## II. Related work

Iterative applications that can be implemented in master-worker fashion are widely used in computational linear algebra for sparse linear systems [3], [4], [5] or eigenvalue problems [6]), image processing [7], [8], signal processing [9], [10], [11], etc. While iterative applications can be asynchronous [4], [12], [13], in this work we only consider the synchronous case.

Several authors have proposed scheduling approaches for such applications [14], [15], [16], [17]. In this work we consider volatile compute resources, such as those found in desktop grids, whose volatility has been studied in [18], [19], [20], [21]. Several authors have studied the "bag-of-tasks" scheduling problem on these platforms, either at an Internet-wide scale or within an Enterprise [22], [23], [24], [25], [26], [27], [28], [29]. Most of these works propose simple greedy scheduling algorithms that rely on mechanisms to select processors based on static criteria (e.g., processor clock-rates, benchmark results, time zone), on simple statistics of past availability [22], [24], [27], [23], and on predictive statistical models of availability [28], [29], [25], [26]. These criteria are used to rank processors but also to exclude them from consideration [22], [24]. The work in [26] is particularly related to our own in that it uses a Markov model of processor availability (but without accounting for temporary preemption). Most of these works also advocate for task replication as a way to cope with volatile resources. Expectedly, injecting task replicas is sensible toward the end of application execution. Given the wealth of scheduling approaches, in [27] the authors propose to automatically instantiate the parameters that together define the behavior of a generic scheduling algorithm. Most works published in this area are of a pragmatic nature and few theoretical results have been sought or obtained (one exception is the work in [30]). Note that while in this work we focus on scheduling a single application, other authors have studied the scheduling problem for multiple simultaneous applications [31], [32].

A key aspect of our work is that we seek to develop scheduling algorithms that explicitly manage the master's bandwidth. Limited master bandwidth is a known issue for desktop grid computing [33], [34], [35] and must therefore be addressed even though it complicates the scheduling problem. To the best of our knowledge, except our previous study for independent tasks [1], no previous work has made such an attempt.

## III. Models and assumptions

We assume that time is discretized into a sequence of time-slots of arbitrarily chosen duration. For simplicity when we say "at time $t$" we imply "at discrete time-slot $t$". Our approach is agnostic to the time-slot duration. The duration that makes sense in practice depends on the application and/or platform, ranging from seconds to minutes or possibly hours.

### A. Application model

We consider an application that performs a sequence of iterations. Each iteration consists of executing $m$ tasks and ends with a global synchronization. All $m$ tasks are identical (in terms of computational cost) and communicate throughout the iteration execution. Therefore, all tasks must make progress at the same rate. If a task is terminated prematurely (due to a

worker failure), all computation performed so far for the current iteration is lost, and the entire iteration has to be restarted. If a task is suspended (due to a worker becoming temporarily reclaimed), then the entire execution of the iteration is also suspended. Due to the global synchronization, there is no overlap between communication and computations. We thus consider that an iteration proceeds in two phases: a communication phase and a computation phase. Finally, before being able to compute, a worker must acquire the application code once (e.g., binary executable, byte code), of constant size $V_{\text{prog}}$ in bytes, and the input data for each task and iteration, of constant size $V_{\text{data}}$ in bytes.

### B. Platform model

The platform comprises $p$ processors. Since each processor executes a worker process, we use the terms processor and worker interchangeably. Worker $P_q$, $q = 1, \ldots, p$, can be in one of three states ($UP$, $RECLAIMED$ or $DOWN$), and transitions between these states occur for each processor at each time-slot independently of the other processors. More precisely:

- Any $UP$ processor can become $DOWN$ or $RECLAIMED$.
- Any $UP$ or $RECLAIMED$ processor can become $DOWN$. It then loses the application program and all the data for its current tasks. If it was computing some of these tasks, these computations are lost.
- Any $UP$ processor can become $RECLAIMED$. The processor does not lose any state. If it was receiving the application program or data for a task, the communication is temporarily suspended. If it was computing a task, the computation on *all* processors is temporarily suspended.

We denote by $\mathcal{S}_q$ the vector that gives the state of $P_q$ at each time-slot starting with time-slot 0.

$P_q$ computes a task in $w_q$ time-slots if it remains $UP$. If $w_q = w$ for each processor $P_q$, then the processors are homogeneous. The master has network bandwidth $BW$ and communicates with a worker with bandwidth $bw$, meaning that we assume same capacity links from the master to each worker. Here we equate bandwidth with data transfer rate, acknowledging that in practice the data transfer rate is a fraction of the physical bandwidth. Let $n_{\text{prog}}$ be the number of workers receiving the program at time $t$, and let $n_{\text{data}}$ be the number of workers receiving the input data of a task at time $t$. The constraint on the master's bandwidth writes $n_{\text{prog}} + n_{\text{data}} \leq n_{\text{com}} = \lfloor BW/bw \rfloor$. Indeed, consider a worker on processor $P_q$ that is communicating at time $t$. Either $P_q$ is receiving the program, or it is receiving data for a task. In both cases, it does this at data transfer rate $bw$. Overall, the master can execute only a limited number $n_{\text{com}}$ of such communications simultaneously. The time for a worker to receive the program is $T_{\text{prog}} = V_{\text{prog}}/bw$, and the time to receive the data is $T_{\text{data}} = V_{\text{data}}/bw$. For simplicity we assume that $T_{\text{prog}}$ and $T_{\text{data}}$ consist of integral numbers of time-slots. We also assume that the master is always $UP$, which can be enforced by using, for instance, two dedicated servers with a primary backup mechanism.

### C. Application execution model

Let $\mathit{config}(t)$ denote the set of workers enrolled by the master, or *configuration*, at time $t$. The configuration is determined by an *application scheduler*, and in this work we propose algorithms to be used by this scheduler. To complete an iteration, enrolled workers must progress concurrently throughout the computations. One worker may be assigned several tasks and execute them concurrently if it has enough memory to do so. Formally, we define for each worker $P_q$ a bound $\mu_q$ on the maximum number of tasks that it can execute concurrently. We assume that $\sum_{q=1}^{p} \mu_q \geq m$, otherwise the configuration cannot execute the application. The $m$ tasks are mapped onto $k \leq m$ workers. Each enrolled worker $P_q$ is assigned $x_q$ tasks, where $\sum_{q=1}^{k} x_q = m$. To be able to compute their tasks, the $k$ enrolled workers must have received the application program and all necessary data. More precisely:

- Each enrolled worker $P_q$ must receive the program, unless it has received it at some previous time and has not be $DOWN$ since then.
- In addition, each worker $P_q$ must receive $x_q$ data messages (one per task) from the master. Suppose that since the begin of the current iteration $P_q$ has received $x'_q$ data messages. At least $(x_q - x'_q)T_{\text{data}}$ time-slots are needed for this communication, and likely more since the master can be engaged in at most $n_{\text{com}}$ concurrent communications.

Overall, the computation can start at a time $t$ only if each of the $k$ enrolled workers is in the $UP$ state, has the program, has the data of all its allocated tasks, and has never been in the $DOWN$ state since receiving these messages. Because tasks must proceed in locked steps, the execution goes at the pace of the slowest worker. Hence the computation of an iteration requires $\max_q(x_q w_q)$ time-slots of concurrent computations (not necessarily consecutive, due to workers possibly being reclaimed). Consider the interval of time between time $t_1$ and time $t_2 = t_1 + \max_q(x_q w_q) + t' - 1$ for some $t'$. For the iteration to be successfully completed by time $t_2$, between $t_1$ and $t_2$ there must be $\max_q(x_q w_q)$ time-slots for which all enrolled workers are simultaneously $UP$, and there may be $t'$ time-slots during which one or more workers are $RECLAIMED$.

The scheduler may choose a new configuration at each time $t$. If at least one worker in $\mathit{config}(t)$ becomes

$DOWN$, the scheduler must select another configuration and restart the iteration from scratch. Even if all workers in $config(t)$ are $UP$, the scheduler may decide to change the configuration because more desirable (i.e., faster, more reliable) workers have become available. Let $P_q$ be a newly enrolled worker at that point, i.e., $P_q \in config(t+1) \setminus config(t)$. $P_q$ needs to receive the program unless it already has a copy of it and has not been $DOWN$ since receiving it. In all cases, $P_q$ needs to receive task data, i.e., $x_q$ messages of $V_{\text{data}}$ bytes. This holds true even if $P_q$ had been enrolled at time $t' < t$ but was un-enrolled since then. In other words, any interrupted communication must be resumed from scratch if the worker became $DOWN$ or was removed from the configuration.

An example iteration execution with $m = 5$ tasks and $p = 5$ processors is shown in Figure 1. For this example the processors are heterogeneous with $w_i = i$ for $1 \leq i \leq 5$, $n_{\text{com}} = 2$, $T_{prog} = 2$, and $T_{data} = 1$.

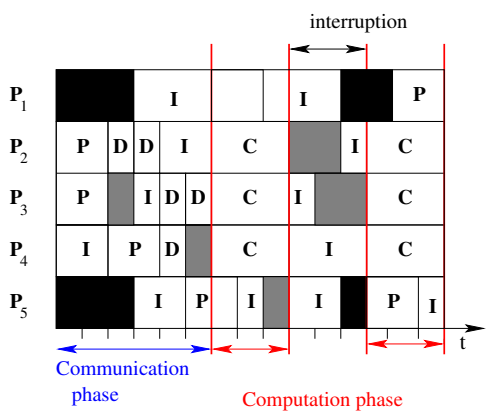


Figure 1. Example iteration execution. White means $UP$, gray means $RECLAIMED$, and black means $DOWN$. $P$ means "receiving the program," $D$ means "receiving the data for a task," $C$ means "computing," and $I$ means "idle."

In the schedule, two tasks are assigned to $P_2$ and $P_3$, and one task is assigned to $P_4$, for a workload of 4 time slots on $P_2$, 6 time slots on $P_3$, and 4 time slots on $P_4$. As a result, the iteration computation in this configuration requires 6 time-slots with all processors in the $UP$ state, and during each of these time-slots, $1/6$ of each task is executed. This configuration is selected at time 1. At this time, $P_1$ and $P_5$ are not $UP$, so they cannot be included in the configuration.

The communication phase of this iteration is executed between time 1 and time 7. At time 1, the 3 selected processors can receive data, but because of the bandwidth constraint, $P_4$ remains idle during the first 3 time-steps. $P_3$ is temporarily reclaimed after it has downloaded the application program. Due to the bandwidth constraint, processors are idle while others download task data (e.g., $P_3$ is idle between time 4 and time 5). At time 7, all 3 processors have downloaded the application program and the data for all tasks assigned to them. They all begin computing. At time 10, $P_2$ is temporarily reclaimed and the computation is suspended with half of the computation of each task completed. When $P_2$ becomes available again, at time 12, $P_3$ has been reclaimed and the computation cannot resume immediately. $P_3$ becomes available again at time 13, $P_2$ and $P_4$ are $UP$, and the computation continues. If a processor had become $DOWN$, say, at time 14, all the computation would have been lost and the communication phase would have been restarted from scratch. At time 16, the processors synchronize and a new iteration can start. At that time, $P_1$ and $P_5$ may be included in the configuration.

## IV. Off-line complexity

The scheduling problem is to maximize the expected number of completed application iterations before time $N$, where $N$ is a specified deadline. In this section, we assess the complexity of the off-line version of this problem, assuming full knowledge of future worker states. In other words, $\mathcal{S}_q[j]$ is known for $1 \leq q \leq p$ and $1 \leq j \leq N$. We show that the simplest off-line and deterministic versions of the problem are NP-hard.

**Fixed number of workers:** Consider the problem with no communication ($T_{\text{prog}} = T_{\text{data}} = 0$), and identical workers with $w_q = w$ and $\mu_q = \mu = 1$. $m$ workers must be enrolled to complete an iteration. The problem reduces to finding $w$ time-slots such that there exist $m$ workers that are simultaneously $UP$ during all these $w$ time-slots. We call this version of the problem Off-Line-Coupled ($\mu = 1$).

**Flexible number of workers:** Consider the problem with no communications ($T_{\text{prog}} = T_{\text{data}} = 0$), and identical processors with $w_q = w$ and $\mu_q = \mu = +\infty$ (in fact $\mu = m$ is sufficient). The problem is less constrained than Off-Line-Coupled ($\mu = 1$). Either one finds $m$ processors that are simultaneously $UP$ during $w$ time-slots, or one finds $\lceil \frac{m}{2} \rceil$ workers that are simultaneously $UP$ during $2w$ time-slots, or one finds $\lceil \frac{m}{3} \rceil$ workers that are simultaneously $UP$ during $3w$ time-slots, and so on. We call this version of the problem Off-Line-Coupled ($\mu = +\infty$).

*Theorem 4.1:* Problems Off-Line-Coupled ($\mu = 1$) and Off-Line-Coupled ($\mu = \infty$) are NP-hard.

*Proof:* We prove both instances with a similar reduction, as detailed below.

**(i) NP-completeness of Off-Line-Coupled ($\mu = 1$) .** The decision problem associated to Off-Line-Coupled ($\mu = 1$) writes: given a value $w$ and $p$ state vectors $\mathcal{S}_q$, can we find $m$ processors that are

simultaneously $UP$ during at least $w$ time-steps? This problem clearly belongs to NP: the $m \times w$ sub-matrix is a certificate of polynomial (and even linear) size.

For the completeness, we use a reduction from ENCD, the Exact Node Cardinality Decision problem [36]. Let $\mathcal{I}_1$ be an instance of ENCD: given a bipartite graph $G = (V \cup W, E)$ and two integers $a$ and $b$ such that $1 \leq a \leq |V|$ and $1 \leq b \leq |W|$, does there exist a bi-clique with exactly $a$ nodes in $V$ and $b$ nodes in $W$? Recall that a bi-clique $C = U_1 \cup U_2$ is a complete induced sub-graph: $U_1 \subset V$, $U_2 \subset W$, and for every $u_1 \in U_1, u_2 \in U_2$, the edge $(u_1, u_2) \in E$.

We construct the following instance $\mathcal{I}_2$ of OFF-LINE-COUPLED ($\mu = 1$): we let $p = |V|$ and $N = |W|$. Resource $R_i$ (which corresponds to vertex $v_i \in V$) is $UP$ at time-step $j$ (which corresponds to vertex $w_j \in W$) if and only if $(v_i, w_j) \in E$. Finally we let $m = a$ and $w = b$. The size of the instance $\mathcal{I}_2$ is linear in the size of the instance $\mathcal{I}_1$. We show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does. Suppose first that $\mathcal{I}_1$ has a solution $C = U_1 \cup U_2$. We select the corresponding $U_1$ processors and the same $U_2$ time-steps. Because we have a clique, each processor is $UP$ at each time-step, hence $\mathcal{I}_2$ has a solution. Suppose now that $\mathcal{I}_2$ has a solution. The corresponding sub-matrix translates into a bi-clique with $a$ nodes in $V$ and $b$ nodes in $W$, hence a solution to $\mathcal{I}_2$.

**(ii) NP-completeness of OFF-LINE-COUPLED ($\mu = \infty$).** We use the same instance $\mathcal{I}_1$ of ENCD as in (i). We construct the following instance $\mathcal{I}_2$ of OFF-LINE-COUPLED ($\mu = +\infty$): we let $p = |V|$ and $N = 2|W| + 1$. Resource $R_i$ (which corresponds to vertex $v_i \in V$) is $UP$ at time-step $j \leq N$ (which corresponds to vertex $w_j \in W$) if and only if $(v_i, w_j) \in E$. All processors are up at each step $j$ such that $|W| + 1 \leq j \leq N$. Finally we let $m = a$ and $w = b + |W| + 1$. Intuitively, this amounts to add $|W| + 1$ new vertices in $W$ which are interconnected to every vertex in $V$. The size of the instance $\mathcal{I}_2$ is linear in the size of the instance $\mathcal{I}_1$. We show that $\mathcal{I}_1$ has a solution if and only if $\mathcal{I}_2$ does. Suppose first that $\mathcal{I}_1$ has a solution $C = U_1 \cup U_2$. We select the corresponding $U_1$ processors and the same $U_2$ time-steps, plus the last $|W| + 1$ time-steps. We have $w = b + |W| + 1$, hence $\mathcal{I}_2$ has a solution. Suppose now that $\mathcal{I}_2$ has a solution. The corresponding sub-matrix translates into a bi-clique with $x$ processors and $y$ time-steps. If $x < m$ then at least one processor executes two tasks per iteration, and we need $2w$ times-steps to perform an iteration. But $2w > N$, what is a contradiction. Hence $x = m$ and $y = K$. At most $|W| + 1$ of the $UP$ time-steps are greater than $|W|$, hence at least $b$ of them are smaller than or equal to $|W|$: this leads to a solution to $\mathcal{I}_2$. ■

## V. ANALYTICAL APPROXIMATIONS

In this section, we compute the expectation of the time needed by a configuration to compute a given workload conditioned on this computation being successful (i.e., with no worker becoming $DOWN$), as well as the probability of success. Intuitively, these quantities seem relevant for developing scheduling heuristics that account for the need for workers to be $UP$ simultaneously, and for workers that can become temporarily $RECLAIMED$. To compute the above expectation and probability, we introduce a Markov model of processor availability. The availability of processor $P_q$ is described by a 3-state recurrent aperiodic Markov chain, defined by 9 probabilities: $P_{i,j}^{(q)}$, with $i, j \in \{u, r, d\}$, is the probability for $P_q$ to move from state $i$ at time $t$ to state $j$ at time $t + 1$, which does not depend on $t$.

We are aware that the Markov (memory-less) assumption for processor availability does not hold in practice. Several authors have observed that the duration of availability intervals in production desktop grids is often far from being exponentially distributed for a 2-state scenario in which processors are either $UP$ or $DOWN$ [18], [37], [19], [20], [28]. Unfortunately, there is no consensus in the literature on a realistic model, even though some of these studies may suggest semi-Markov models with approximately Weibull or Log-Normal holding times. Deriving a realistic 3-state statistical model of processor availability is thus an open research question that is outside the scope of this work. Instead, we opt for a Markov model because it is simple and lends itself to tractable analysis. This model gives a framework in which to design heuristics that trade off worker speed for reliability. Furthermore, it allows us to evaluate such trade-off approaches in "laboratory conditions."

### A. Probability of success and expected duration of a computation

Consider a set $S$ of workers all in the $UP$ state at time 0. This set is assigned a workload that requires $W$ time-slots of simultaneous computation. To complete this workload successfully, all the workers in $S$ must be simultaneously $UP$ during another $W - 1$ time-slots. They can possibly become $RECLAIMED$ (thereby temporarily suspending the execution) but must never become $DOWN$ in between. What is the probability of the workload being completed? And, if it is successfully completed, what is the expectation of the number of time-slots until completion?

*Definition 1:* Knowing that all processors in a set $S$ are $UP$ at time-slot $t_1$, let $\mathbf{P}_+^{(\mathbf{S})}$ be the conditional probability that they will all be $UP$ simultaneously at a later time-slot, without any of them going to the $DOWN$ state in between. Formally, knowing that

$\forall P_q \in S, \mathcal{S}_q[t_1] = u$, $P_+^{(S)}$ is the conditional probability that there exists a time $t_2 > t_1$ such that $\forall P_q \in S$, $\mathcal{S}_q[t_2] = u$ and $\mathcal{S}_q[t] \neq d$ for $t_1 < t < t_2$ .

*Definition 2:* Let $\mathbf{E^{(S)}(W)}$ be the conditional expectation of the number of time-slots required by a set of processors $S$ to complete a workload of size $W$ knowing that all processors in $S$ are $UP$ at the current time-slot $t_1$ and none will become $DOWN$ before completing this workload. Formally, knowing that $\mathcal{S}_q[t_1] = u$, and that there exist $W-1$ time-slots $t_2 < t_3 < \cdots < t_W$, with $t_1 < t_2$, $\mathcal{S}_q[t_i] = u$ for $i \in [2, W]$, and $\mathcal{S}_q[t] \neq d$ for $t \in [t_1, t_W]$, $E^{(S)}(W)$ is the expectation of $t_W - t_1 + 1$ conditioned on success.

*Theorem 5.1:* It is possible to approximate the values of $P_+^{(S)}$ and $E^{(S)}(W)$ numerically up to an arbitrary precision $\varepsilon$ in fully polynomial time.

*Proof:* Consider a set $S$ of processors, all available at time slot 0. Consider the probability $P_+^{(S)}(t)$ that all these processors are simultaneously $UP$ again for the first time at time $t$. This means that for all $0 < t' < t$, there exists at least one processor $RECLAIMED$ at time $t'$. Also, none of the processors in $S$ goes $DOWN$ between 0 and $t$.

Let $P^{(q)}_{u \xrightarrow{t} u}$ be the probability that a processor $P_q$ that was $UP$ at time 0 is $UP$ again at time $t$, without having been $DOWN$ in between, and let $P^{(S)}_{u \xrightarrow{t} u} = \prod_{P_q \in S} P^{(q)}_{u \xrightarrow{t} u}$. For each processor $P_q$, the value $P^{(q)}_{u \xrightarrow{t} u}$ can be computed by considering its transition matrix raised to the power $t$, knowing that the initial state is $UP$. We form the product to compute $P^{(S)}_{u \xrightarrow{t} u}$. We derive that

$$P_+^{(S)}(t) = P^{(S)}_{u \xrightarrow{t} u} - \sum_{0<t'<t} P_+^{(S)}(t') \times P^{(S)}_{u \xrightarrow{t-t'} u}.$$

The probability $P_+^{(S)}$ that all the processors in $S$ will be simultaneously $UP$ again at some point, before the first failure of any of them, is

$$\begin{aligned} P_+^{(S)} &= \textstyle\sum_{t>0} P_+^{(S)}(t) \\ &= \textstyle\sum_{t>0} P^{(S)}_{u \xrightarrow{t} u} - \sum_{0<t'<t} P_+^{(S)}(t') \times P^{(S)}_{u \xrightarrow{t-t'} u} \\ &= \textstyle\sum_{t>0} P^{(S)}_{u \xrightarrow{t} u} - \sum_{t>0} P_+^{(S)}(t) \times \sum_{t'>0} P^{(S)}_{u \xrightarrow{t'} u} \end{aligned}$$

Let $E_u(S) = \sum_{t>0} P^{(S)}_{u \xrightarrow{t} u}$. Suppose that all processors are $UP$ at time slot 0. Let $A_t$ the random variable that is equal to 1 if all processors are $UP$ at time slot $t$ without that ant processor goes $DOWN$ in between. Then $E(A_t) = P^{(S)}_{u \xrightarrow{t} u}$. By linearity of the expectation, we have $E(\sum_{0 \leq t' \leq t} A_{t'}) = \sum_{0 \leq t' \leq t} P^{(S)}_{u \xrightarrow{t'} u}$. Suppose that, in set $S$, at least one processor has a nonzero probability of going $DOWN$. Then, $\lim_{t \to \infty} \sum_{0 \leq t' \leq t} P^{(S)}_{u \xrightarrow{t'} u}$ converges. We can conclude that $E(\sum_{t>0} A_t) = \sum_{t>0} P^{(S)}_{u \xrightarrow{t} u}$. Then, $E_u(S)$ is the expected number of time slots with all processors $UP$, before one of these processors fails. Then, $P_+^{(S)} = E_u(S) - E_u(S) \times P_+^{(S)}$, from which we derive that $P_+^{(S)} = \frac{E_u(S)}{1+E_u(S)}$ if , in set $S$, at least one processor has a nonzero probability of going $DOWN$. Otherwise, $P_+^{(S)} = 1$.

We now consider the expected time $E^{(S)}(W)$ to execute $W$ time slots of computation, conditioned by the fact that no processor in $S$ will fail. The first time slot of computation is done at $t = 0$. Let $E_c^{(S)}$ be the expected time of the next time slot of computation. Then,

$$\begin{aligned} E_c^{(S)} &= \textstyle\sum_{t>0} t \times P_+^{(S)}(t) \\ &= \textstyle\sum_{t>0} t \times P^{(S)}_{u \xrightarrow{t} u} \\ &\quad -t \times \left( \textstyle\sum_{0<t'<t} P_+^{(S)}(t') \times P^{(S)}_{u \xrightarrow{t-t'} u} \right) \\ &= \textstyle\sum_{t>0} t \times P^{(S)}_{u \xrightarrow{t} u} \\ &\quad - \left( \textstyle\sum_{t>0} P_+^{(S)}(t) \right) \times \left( \textstyle\sum_{t'>0} (t+t') P^{(S)}_{u \xrightarrow{t'} u} \right) \end{aligned}$$

Let $A(S) = \sum_{t>0} t \times P^{(S)}_{u \xrightarrow{t} u}$. Then, $E_c^{(S)} = A(S) - E_c^{(S)} \times E_u(S) - P_+^{(S)} \times A(S)$. Then, $E_c^{(S)} = \frac{A(S)\left(1-P_+^{(S)}\right)}{1+E_u(S)}$ and $E^{(S)}(W) = \frac{1+(W-1)E_c^{(S)}}{(P_+^{(S)})^{W-1}}$.

We now explain how we numerically approximate the values of $E_u(S)$ and $A(S)$. Let $\varepsilon$ be the desired precision. Consider for some value $T$ the difference between $E_u(S)$ and $\sum_{0<t<T} P^{(S)}_{u \xrightarrow{t} u}$. We have $P^{(S)}_{u \xrightarrow{t} u} = \prod_{P_q \in S} P^{q}_{u \xrightarrow{t} u}$ and $P^{q}_{u \xrightarrow{t} u}$ the probability that a processor that was $UP$ at time 0 is $UP$ at time $t$ without having been $DOWN$. For a processor $P_q \in S$, let $M_q = \begin{bmatrix} P^{(q)}_{u,u} P^{(q)}_{u,r} \\ P^{(q)}_{r,u} P^{(q)}_{r,r} \end{bmatrix}$. Then, $P^{q}_{u \xrightarrow{t} u} = (M_q^t)[0,0]$. We obtain $P^{q}_{u \xrightarrow{t} u} = \mu(\lambda_1^q)^t + \nu(\lambda_2^q)^t$ with $\mu, \nu \geq 0$, $\mu + \nu = 1$ and $\lambda_1^q > \lambda_2^q$ eigenvalues of $M_q$. Then, $P^{q}_{u \xrightarrow{t} u} \leq (\lambda_1^q)^t$. We obtain $P^{(S)}_{u \xrightarrow{t} u} \leq \left(\prod_{P_q \in S} \lambda_1^q\right)^t$ and $\sum_{t \geq T} P^{(S)}_{u \xrightarrow{t} u} \leq \left(\prod_{P_q \in S} \lambda_1^q\right)^T \times \frac{1}{1-\prod_{P_q \in S} \lambda_1^q}$. Let $\Lambda = \prod_{P_q \in S} \lambda_1^q$. We obtain that $T > \frac{\ln(\varepsilon(1-\Lambda))}{\ln(\Lambda)}$ implies $E_u(S) - \sum_{0<t<T} P^{(S)}_{u \xrightarrow{t} u} \leq \varepsilon$. Thus, we can compute in polynomial time an approximation of $E_u(S)$ at $\varepsilon$ in polynomial time.

Similarly, we obtain $A(S) - \sum_{0<t<T} t \times P^{(S)}_{u \xrightarrow{t} u} \leq \varepsilon$ as soon as $\Lambda^T \left( \frac{T}{1-\Lambda} + \frac{\Lambda}{(1-\Lambda)^2} \right) \leq \varepsilon$. Therefore $A(S)$ can be approximated with precision $\varepsilon$ in polynomial time. ■

### B. Probability of success and expected duration of a communication

The previous section gave approximations for the probability of success and conditional expected duration for computations. Unfortunately, similar approximations cannot be obtained for communications due to complexity added by the $n_{\text{com}}$ constraint. Instead, we resort to a coarser approximation as explained hereafter. Let $S$ be a set of enrolled workers. For worker $P_q \in S$, let $n_q$ be the number of time-slots of communication needed to receive the application program and all the data of its allocated tasks. Suppose first that $|S| \leq n_{\text{com}}$. In this case, the expected communication time on worker $P_q$, $E_q$, can be estimated precisely reusing the result in the previous section: $E_q = E^{(P_q)}(n_q)$. We then estimate the expected communication time of the current configuration as $E_{comm}^{(S)} = \max_{P_q \in S}\{E^{(P_q)}(n_q)\}$ . In the case $|S| \geq n_{\text{com}}$, obtaining an estimate close to the actual expected communication time seems out of reach. Instead, we use a coarser estimation: $E_{comm}^{(S)} = \max\left\{\max_{P_q \in S}\left\{E^{(P_q)}(n_q)\right\}, \frac{\sum_{P_q \in S} n_q}{n_{\text{com}}}\right\}$ .

Let $P_{ND}^{(P_q)}(t)$ denote the probability that worker $P_q$ that was $UP$ at time $t'$ does not become $DOWN$ between time $t'$ and time $t'+t$. The probability of success is then estimated as $P_{comm}^{(S)} = \prod_{P_q \in S} P_{ND}^{(P_q)}(E_{comm}^{(S)})$ . The expression for $P_{comm}^{(S)}$ does not take into account the time needed after the end of all communications for all workers to be $UP$ simultaneously. The probability of success of an iteration is estimated by multiplying the probability of success of the communications and the probability of success of the computations.

## VI. On-line heuristics

We propose heuristics for solving the on-line version of the scheduling problem, i.e., assuming no knowledge of future processor states. Conceptually, we distinguish between two classes of heuristics. *Passive heuristics* conservatively keep current processors active as long as possible. In other words, the current configuration is changed only when one of the enrolled processors becomes $DOWN$. In this case, all previously executed work is lost. However, a worker that has not become $DOWN$ but has already received task data, can reuse that data if the scheduler reassigns tasks to it. *Proactive heuristics* allow for a complete reconfiguration even if no worker fails, possibly aborting ongoing computation if a better configuration is found. This makes it possible for an iteration to never complete. A criterion must thus be derived to decide whether and when such an aggressive reconfiguration is worthwhile. Our proactive heuristics are defined by a pair (criterion, passive heuristic). When a new configuration is computed using the heuristic, it is compared to the current configuration according to the criterion. If the new configuration is better than the current one, then it is launched, leading to new communications and task allocations. Otherwise, the execution continues with the current configuration for an additional time slot.

We also include results for a baseline Random heuristic that allocates tasks to $UP$ processors randomly using a uniform distribution.

### A. Passive heuristics

Passive heuristics assign tasks to workers, which must be in the $UP$ state, one by one until $m$ tasks are assigned. Each task is assigned to a worker according to a criterion that defines the heuristic. As described hereafter, we consider four different criteria: probability of success, expected completion time, estimated yield, and estimated apparent yield. We derive the following heuristics:

- **IP (Incremental: Probability of success) –** This heuristic attempts to find configurations with high probability of success. The next task is assigned to the worker such that the probability of success of all currently assigned tasks (including the new one) is maximized. More precisely, consider the set $S$ of workers with at least one task already assigned. For each worker $P_q$, either in $S$ or not, we compute the probability $P^{(S)}(q)$ of success of the communication and the computation if the additional task is assigned to $P_q$, using the results of Section V: $P^{(S)}(q) = P^{(S\cup\{P_q\})}(W_q) \times P_{comm}^{(S\cup\{P_q\})}$ with $W_q$ the maximal load in $S \cup \{P_q\}$ with an additional task on $P_q$. We assign the next task to worker $P_{q_0}$, with $q_0 = \text{ArgMax}\left\{P^{(S)}(q)\right\}$.
- **IE (Incremental: Expected completion time) –** This heuristic attempts to find fast configurations, without considering reliability. The next task is assigned to the worker that minimizes the expected execution time of the iteration. More precisely, consider the set $S$ of workers with at least one task already assigned. For each worker $P_q$, either in $S$ or not, we compute the expected communication time $E_{comm}^{(S\cup\{P_q\})}$ and the expected computation time $E^{(S\cup\{P_q\})}(W_q)$ with an additional task on $P_q$. We obtain the expected duration of the iteration $E^{(S)}(q) = E_{comm}^{(S\cup\{P_q\})} + E^{(S\cup\{P_q\})}(W_q)$. We assign the next task to worker $P_{q_0}$, with $q_0 = \text{ArgMin}\left\{E^{(S)}(q)\right\}$.
- **IY (Incremental: Expected yield) –** This heuristic assigns the next task to the worker that maximizes the *yield* of the configuration. The yield is the expected value of the inverse of the execution time of the current iteration, which we estimate as follows. For a given configuration with probability

of success $P$ and expected completion time $E$ for an iteration that has already been running for $t$ time slots, the yield is estimated as $Y = \frac{P}{E+t}$. Intuitively, we expect the yield to achieve a trade-off between reliability (probability of success) and execution speed. Consider the set $S$ of workers with at least one task already assigned. For each processor $P_q$, either in $S$ or not, we compute the expected yield with an additional task on $P_q$: let $P^{(S)}(q)$ be the probability computed for heuristic IP, $E^{(S)}(q)$ be the expected completion time computed for heuristic IE, and $t$ be the time spent since the beginning of the current iteration. We assign the next task to worker $P_{q_0}$, with $q_0 = \text{ArgMax}\left\{\frac{P^{(S)}(q)}{t+E^{(S)}(q)}\right\}$.

- **IAY (Incremental: Expected apparent yield) –** The yield takes into account the time already spent in the current iteration. It could be worthwhile to consider only future work, i.e., the remaining time until iteration completion. To this end we define the *apparent yield* as $AY = \frac{P}{E}$. Using the same notations as for heuristic IY, we assign the next task to processor $P_{q_0}$ with $q_0 = \text{ArgMax}\left\{\frac{P^{(S)}(q)}{E^{(S)}(q)}\right\}$.

### B. Proactive heuristics

Our proactive heuristics are designed as follows. Consider an application executing on a platform using a passive heuristic $H$ and criterion $C$ at some time $t$. The configuration $\mathit{config}(t-1)$ was selected by $H$ at time $t' \leq t-1$ because of a configuration change due to a proactive decision, due to a worker becoming $DOWN$, or due to the beginning of a new iteration. Let $\mathit{config}_1 = \mathit{config}(t') = \mathit{config}(t-1)$. At time $t'$, the configuration was measured by criterion $C$ with value $c'$. Suppose that by time $t$ no worker in this configuration has failed. Between $t'$ and $t$, some work may have been done: some communications may be in process or completed, and computations may have started. Consequently, the measure of this configuration given by $C$ should be updated to account for the progress between $t'$ and $t$. Let $c$ be the updated value of criterion $C$ for the current configuration. At step $t$, a new configuration is computed from scratch using heuristic $H$, as if no task were allocated to any worker. Let $\mathit{config}_2$ be this new configuration and $c_2$ its measure by $C$. If $c \geq c_2$, then the current configuration at time $t-1$ is kept for another time-slot: $\mathit{config}(t) = \mathit{config}_1$. Otherwise, the current configuration is interrupted, and the new configuration is $\mathit{config}(t) = \mathit{config}_2$.

For certain criterion choices, a heuristic could diverge and continually change the configuration, even with workers that are reliably $UP$. To avoid this divergence, proactive criteria have to respect the following constraint: a given configuration that has been running for $t+1$ time-slots must be better for the proactive criterion than the same configuration running for $t$ time slots. With this constraint, all possible configurations are ordered by their value for the selected criterion at the beginning of the iteration, and a lower-ranked configuration in this order cannot be chosen to replace the current configuration. As the number of possible configurations is finite, no proactive heuristic can diverge. The four criteria used to define passive heuristics in the previous section meet this constraint. However, AY (Apparent Yield) leads to many (unnecessary) configuration changes before converging, while the other criteria should be stable. Hence, for the proactive criterion $C$, we only retain P (Probability of success), E (Expected completion time) and Y (Expected yield). Any passive heuristic $H$ can be used as the building block for a proactive heuristic. We thus obtain $3 \times 4$ proactive heuristics named $C$-$H$ where $C \in \{\text{P, E, Y}\}$ and $H \in \{\text{IP, IE, IY, IAY}\}$, plus the RANDOM heuristic.

## VII. Experimental Evaluation

### A. Methodology

To evaluate our heuristics we use a discrete-event simulator, which is publicly available at http://graal.ens-lyon.fr/~fdufosse/changing_platforms.tar.gz. The input to the simulator are the values for all the parameters listed in Section III. The simulator implements temporal processor availability as Markov processes as described in Section V. All our experiments are for $p = 20$ processors. The Markov model for processor availability is defined as follows. For each processor $P_q$, we pick a random value uniformly distributed between 0.90 and 0.99 for each $P^{(q)}_{x,x}$ value (for $x = u, r, d$). We then set $P^{(q)}_{x,y}$ to $0.5 \times (1 - P^{(q)}_{x,x})$, for $x \neq y$. An experiment is defined by the Markov model for each processor and by three parameters: $m$, the number of tasks per iteration; $n_{\text{com}}$, the constraint on the master's communication bandwidth; and a third parameter, $w_{min}$, defined as follows. For each processor $P_q$, we pick $w_q$ uniformly between $w_{min}$ and $10 \times w_{min}$. $T_{\text{data}}$ is set to $w_{min}$, meaning that the fastest processor has a computation-communication ratio of 1. $T_{\text{prog}}$ is set to $5 \times w_{min}$, meaning that downloading the program takes 5 times as much time as downloading the data for a task. We define our experimental space as $(m, n_{\text{com}}, w_{min})$ with $m \in \{5, 10\}$, $n_{\text{com}} \in \{5, 10, 20\}$, and $w_{min} \in \{1, 2, \ldots, 10\}$. For each possible instantiation of $(m, n_{\text{com}}, w_{min})$, we create 10 random experimental scenarios so as to obtain different instantiations of the various random parameters. Then, for each experimental scenario, we run 10 trials where each trial uses a different random number generator seed to produce multiple realizations of the Markov chain transitions. The total number of generated

problem instances is thus $2 \times 3 \times 10 \times 10 \times 10 = 6,000$. We emphasize that our goal here is not to instantiate a representative model for a desktop grid and application, but rather to create arbitrary and simple synthetic experimental scenarios that will make it possible to highlight inherent strengths and weaknesses of the proposed heuristics.

In all experiments, rather than fixing $N$, the deadline in number of time-slots, we instead fix the number of iterations to 10. The quality of an application execution is then measured by the time needed to complete 10 iterations, or *makespan*. This problem is equivalent to the problem of maximizing the number of iterations before a deadline. It is also simpler to instantiate since it does not require choosing a meaningful deadline, which would depend on application and platform characteristics. For some experimental scenarios and some heuristics, the time needed to complete 10 iterations successfully can be extremely large, making it impossible to obtain results in a reasonable amount of time. Consequently, we limit the makespan to 1,000,000 seconds and declare that a heuristic fails if it reaches this limit, and succeeds otherwise.

The makespans vary widely between problem instances depending on processor availability patterns, which requires that we define relative metrics for a sound comparison of our heuristics. As expected, with more tasks (larger $m$), the number of failures increases for each heuristic. With $m = 5$, no heuristic fails for more than 5 out of the 3,000 problems instances. With $m = 10$, heuristics fail for up to 6% of the 3,000 problem instances. In both cases, IE is the most robust: It fails only for 1 of the $3,000$ $m = 5$ instances, and for 81 of the $3,000$ $m = 10$ instances. Importantly, whenever heuristic IE fails, all the other heuristics also fail. For this reason we use $IE$ as a reference. For a heuristic $H$, we compute the relative difference (expressed in percentage) between the makespan that it achieves for a given experimental scenario (averaged over all 10 trials) and the one achieved by IE for the same scenario, assuming that heuristic $H$ succeeds. The relative difference is defined as:

$$\frac{\text{makespan}_H - \text{makespan}_{IE}}{\min(\text{makespan}_H, \text{makespan}_{IE})} .$$

The denominator is always the makespan of the best performing heuristic, so as to allow consistent comparisons. We denote this metric by %diff. We also count the fraction of trials where heuristic $H$ obtains a makespan smaller than or equal to IE (denoted by %wins) and the fraction of trials where heuristic $H$ obtains a makespan that does not exceed that of IE by more than 30% (denoted by %wins30). We also report the standard deviation over all scenarios (column %stdv).

Table I
RESULTS WITH $m = 5$ TASKS.

| Heuristic | #fails | %diff | %wins | %wins30 | $stdv$ |
|---|---|---|---|---|---|
| Y-IE | 2 | -11.82 | 72.58 | 92.09 | 0.42 |
| P-IE | 2 | -10.50 | 70.98 | 91.19 | 0.44 |
| E-IAY | 4 | -10.40 | 64.75 | 85.15 | 0.77 |
| E-IY | 4 | -3.40 | 59.91 | 81.64 | 0.80 |
| IE | 1 | 0.00 | 100.00 | 100.00 | 0.00 |
| IAY | 2 | 13.59 | 51.07 | 76.42 | 1.93 |
| E-IP | 4 | 19.35 | 47.73 | 69.69 | 0.98 |
| IY | 2 | 24.22 | 45.26 | 70.85 | 1.96 |
| IP | 2 | 52.03 | 34.79 | 58.54 | 2.11 |
| E-IE | 5 | 53.93 | 39.57 | 64.51 | 2.57 |
| Y-IAY | 3 | 99.75 | 53.89 | 70.77 | 5.55 |
| Y-IY | 3 | 113.01 | 49.22 | 66.80 | 5.73 |
| P-IAY | 3 | 125.27 | 50.28 | 67.33 | 6.08 |
| Y-IP | 2 | 145.05 | 38.56 | 55.54 | 5.90 |
| P-IY | 3 | 145.78 | 42.54 | 59.66 | 6.22 |
| P-IP | 2 | 176.92 | 36.92 | 52.00 | 6.61 |
| RANDOM | 0 | 2124.42 | 0.00 | 0.20 | 22.54 |

Table I shows results for $m = 5$ tasks, with heuristics sorted by decreasing %diff. The number of failures for all heuristics is shown in the first column of the table and is at most 5 (recall that IS, the reference heuristic, only fails for 1 instance). Consequently, although some heuristics fail on some scenarios, these failures do not have a large impact on our results.

These results show the efficiency of all our heuristic, when compared with the RANDOM Heuristic. RANDOM is on average more than 20 times worse than IE while all other heuristics have a %diff less than 200%. As seen in the table, only 4 heuristics lead to a %diff value lower than that obtained by IE, with 3 of these heuristics more than 10 points lower. These 4 heuristics are all proactive. We conclude that the best proactive heuristics are significantly better than the best passive heuristics. Several observations can be made on the results in the table. A first one is that using the yield as a heuristic or a criterion is better than using the probability of success. In other terms, heuristic $C$-IY is better than heuristic $C$-IP, and heuristic Y-$H$ is marginally better than P-$H$ (in this case an inspection of the simulation traces shows that Y-$H$ and P-$H$ lead to mostly identical executions). Everything else being equal, considering the yield is better than considering the probability of success because it accounts for the host computing speeds in addition to their reliability. A second observation is that heuristic $C$-IAY is better than heuristic $C$-IY, thus confirming that the "apparent yield" has merit and is a direct improvement of the yield metric. Anecdotally, while E-IY and E-IAY obtain similar results for %wins and %wins30, E-IAY is significantly better than E-IY on average. A third observation is that although Y-IE and P-IE lead to good results, all other proactive heuristics with the same criteria (i.e., Y-$H$ and P-$H$) rank last, with %diff values reaching 100%. Finally,

the key observation is that the best heuristics (the top 5 heuristics, including the reference IE) all account for expected execution time either as a criterion for selecting a new configuration (E-IAY, E-IY) or as a host selection mechanism (Y-IE, P-IE, IE). A seemingly sensible expectation is thus that E-IE would be very efficient. But instead, E-IE leads to poor results, with on average makespan almost more than 50% longer than that of IE, the fourth lowest %wins value and the fifth lowest %wins30 value. The reason for these poor aggregate results is that E-IE leads to inefficient schedules for problem instances in which the fastest processor is unreliable.

Table II
RESULTS WITH $m = 10$ TASKS FOR THE BEST EIGHT HEURISTICS.

| Heuristic | #fails | %diff | %wins | %wins30 | $stdv$ |
|---|---|---|---|---|---|
| Y-IE | 141 | -10.33 | 71.35 | 88.42 | 0.54 |
| P-IE | 141 | -8.62 | 69.64 | 87.23 | 0.55 |
| E-IAY | 178 | -6.10 | 66.62 | 81.93 | 1.58 |
| E-IY | 176 | 8.04 | 61.90 | 77.87 | 3.07 |
| E-IP | 168 | 29.68 | 55.12 | 71.86 | 3.01 |
| IAY | 152 | 136.65 | 46.98 | 69.31 | 14.76 |
| IY | 152 | 147.77 | 42.06 | 64.47 | 14.76 |

### B. Results for $m = 10$

In this section we discuss results for $m = 10$ tasks, but only for the reference IE and those 7 heuristics that achieve a %diff value below 50% (the largest such value being in fact below 25%): Y-IE, P-IE, E-IAY, E-IY, IAY, E-IP and IY. Results are shown in Table II. Only two of these heuristics do not consider expected completion time as a criterion: IAY and IY. These two heuristics rank reasonably high in terms of %diff with $m = 5$ tasks, but are over 130% with $m = 10$ tasks. The ranking of the heuristics is almost unchanged when compared to the $m = 5$ results, even if %diff values are lower. When for $m = 5$, E-IY leads to a negative %diff value, for $m = 10$ this value becomes positive. For $m = 10$ only three heuristics achieve positive %diff values: Y-IE, P-IE and E-IAY. With $m = 10$, most heuristics fail for more than 5% of the problem instances. Given that IE is the most robust heuristic, it should come to no surprise that those proactive heuristics that use IE lead to the lowest number of failures. One conclusion from these results is that Y-IE is only slightly less robust than IE (failing on 4.7% of instances as opposed to 2.7% for IE) but leads to significantly better performance with a %diff value above 11%, leading to a lower makespan for more than 72% of the instances, and leading to a makespan more than 30% larger in less than 8% of the instances.

Figure 2 shows %diff values versus $w_{min}$ for $m = 10$ tasks. $w_{min}$ is a synthetic parameter defined to instantiate problem instances. Essentially a larger $w_{min}$ value

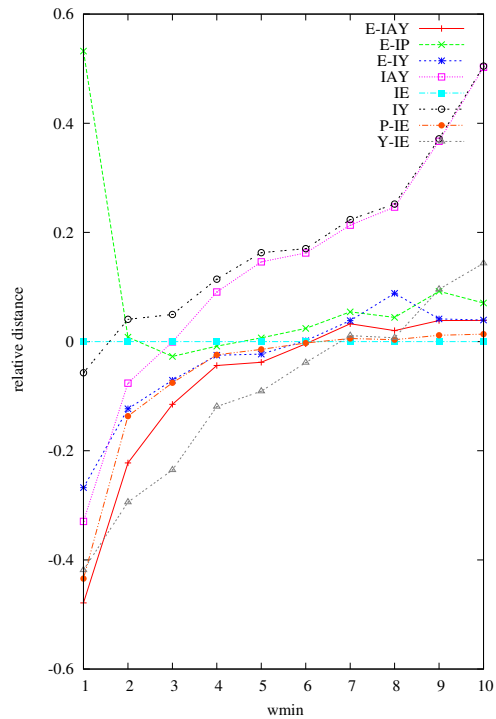


Figure 2. %diff for $m = 10$ tasks vs. $w_{min}$.

means longer tasks and longer data transfers, leading to a more "difficult" instance. The results show that Y-IE is the best or close to the best heuristic up to $w_{min} \approx 8$. For large values of $w_{min}$, it is outperformed by other several other heuristics, such as P-IE, but also by the reference heuristic IE. IE is the best option for large values of $w_{min}$! An intuitive explanation is that when the instance is difficult, meaning that the probability of success is low due to long computations and communications, a good way to obtain a short makespan is to try to find the fastest workers and "hope for the best." When looking at the whole $w_{min}$ range, P-IE appears like a good alternative to Y-IE. For low $w_{min}$ values, it outperforms IE significantly, and for large $w_{min}$ it is outperformed by it only marginally. Recall from Table II that Y-IE and P-IE experience exactly the same number of failures (141 failures out of the 3,000 instances).

Unlike previous work that has considered loosely-coupled master-worker applications, in this work a single processor failure can have a dramatic effect on application execution. The requirement that the application can progress only when all enrolled processors are simultaneously available, dramatically complicates

all scheduling decisions. Furthermore, our problem formulation includes a limit on the available bandwidth from the master to the workers, and the possibility for processors to be temporarily reclaimed. We have proved the problem to be NP-complete in an off-line setting, i.e., with full knowledge of future processor states. By assuming a Markov model of processor availability, we have proposed polynomial time approximation schemes to compute the expected completion time of a computation, and its probability of success. We have then proposed 16 heuristics that are either passive (change the set of enrolled processors only when a processor failure occurs) or proactive (change the set of enrolled processors when a better set is available even if no failure occurs). These heuristics are easily defined as combinations of two among four sensible metrics: probability of success, expected completion time, expected yield and expected apparent yield.

All these heuristics widely outperform a baseline heuristic that allocates tasks to processors randomly. In addition, simulation results show that four of our 16 heuristics lead to significantly better results than the remaining 12. Passive heuristic IE is the most robust, which is why we have used it as a reference, but it does not lead to the best makespans. Heuristic Y-IE, which attempts to optimize expected execution time while proactively deciding to change the set of enrolled processors based on yield, leads to the best average results. Heuristic P-IE, which changes configuration based on probability of success, leads to more stable performance across our set of experimental problem instances as it is never significantly outperformed by IE. The conclusion is that a proactive heuristic that selects processors to maximize expected execution time and changes configuration based on yield or probability of success is very promising.

We have made it plain that the Markov assumption for processor availability is not meant to be representative of real-world platforms. Nevertheless, faced with the lack of an acknowledged and validated model, we have opted for a Markov model. The advantage of this model is that it provides us with a tractable framework in which we can not only develop heuristics but also evaluate the merit of heuristical ideas in "laboratory conditions". If a more realistic model arises, then a next step in this research would be to determine to which extent the principles from our heuristics can be adapted to the new model. Given that the results in Section V rely on the Markov assumption heavily (to provide accurate expressions for the expected completion time, probability of success and yield), it is unlikely that similar results would hold in a non-Markovian model. However, it may be possible to develop coarser estimates of our four criteria in the new model so as to design meaningful and effective heuristics. If no such new model arises, then an interesting next step would be to simply build a flawed Markov model based on real-world processor availability traces, and investigate how "wrong" the Markov heuristics behave in a real-world setting, when compared to heuristics that have been proposed in previous works and that do not rely on sophisticated probabilistic criteria for making scheduling decisions.

*Acknowledgments.:* Y. Robert is with the Institut Universitaire de France. This work was supported in part by the ANR RESCUE project.

## REFERENCES

[1] H. Casanova, F. Dufossé, Y. Robert, and F. Vivien, "Scheduling parallel iterative applications on volatile resources," in *IPDPS'2011, the 25th IEEE Int. Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2011.

[2] B. Hong and V. K. Prasanna, "Adaptive allocation of independent tasks to maximize throughput," *IEEE TPDS*, vol. 18, no. 10, pp. 1420–1435, 2007.

[3] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199 – 222, 1969.

[4] J. C. Strikwerda, "A probabilistic analysis of asynchronous iteration," *Lin. Algebra Applications*, vol. 349, pp. 125–154, 2002.

[5] B. Uçar and C. Aykanat, "Partitioning sparse matrices for parallel preconditioned iterative methods," *SIAM Journal on Scientific Computing*, vol. 29, no. 4, pp. 1683–1709, 2007.

[6] G. L. G. Sleijpen and H. A. V. d. Vorst, "A jacobi-davidson iteration method for linear eigenvalue problems," *SIAM Review*, vol. 42, no. 2, pp. pp. 267–293, 2000.

[7] T. Y. Zhang and C. Y. Suen, "A fast parallel algorithm for thinning digital patterns," *Comm. ACM*, vol. 27, pp. 236–239, 1984.

[8] Y. Censor, D. Gordon, and R. Gordon, "Component averaging: An efficient iterative parallel algorithm for large and sparse unstructured problems," *Parallel Computing*, vol. 27, no. 6, pp. 777 – 808, 2001.

[9] C. Byrne, "A unified treatment of some iterative algorithms in signal processing and image reconstruction," *Inverse Problems*, vol. 20, no. 1, p. 103, 2004. [Online]. Available: http://stacks.iop.org/0266-5611/20/i=1/a=006

[10] M. Crouse, R. Nowak, and R. Baraniuk, "Wavelet-based statistical signal processing using hidden markov models," 1998.

[11] A. K. Katsaggelos, J. Biemond, R. W. Schafer, and R. M. Mersereau, "A regularized iterative image restoration algorithm," *IEEE Trans. Signal Processing*, vol. 39, pp. 914–929, April 1991.

[12] J.-C. Charr, R. Couturier, and D. Laiymani, "Jacep2p-v2: A fully decentralized and fault tolerant environment for executing parallel iterative asynchronous applications on volatile distributed architectures," *Future Generation Computer Systems*, vol. 27, no. 5, pp. 606 – 613, 2011.

[13] J. M. Bahi, R. Couturier, D. Laiymani, and K. Mazouzi, "Java and asynchronous iterative applications: large scale experiments," *Parallel and Distributed Processing Symposium, International*, vol. 0, p. 230, 2007.

[14] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, *Parallel Iterative Algorithms: From Sequential to Grid Computing*. Chapman and Hall/CRC Press, 2007.

[15] H. Jagadish, S. Rao, and T. Kailath, "Array architectures for iterative algorithms," *Proceedings IEEE*, vol. 75, pp. 1304–1321, 1987.

[16] A. Heddaya and K. Park, "Mapping parallel iterative algorithms onto workstation networks," in *HPDC'94*, 1994, pp. 211 –218.

[17] A. Legrand, H. Renard, Y. Robert, and F. Vivien, "Mapping and load-balancing iterative computations on heterogeneous clusters with shared links," *IEEE TPDS*, vol. 15, pp. 546–558, 2004.

[18] D. Nurmi, J. Brevik, and R. Wolski, "Modeling Machine Availability in Enterprise and Wide-area Distributed Computing Environments," in *Proc. of Europar*, 2005.

[19] R. Wolski, D. Nurmi, and J. Brevik, "An Analysis of Availability Distributions in Condor," in *Proc. of the IPDPS Workshop on Next-Generation Software*, 2007.

[20] B. Javadi, D. Kondo, J. Vincent, and D. Anderson, "Mining for Statistical Models of Availability in Large-Scale Distributed Systems: An Empirical Study of SETI@home," in *Proc. of the 17th MASCOTS*, 2009.

[21] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging," in *2003 ACM/IEEE Supercomputing Conf.* ACM Press, 2003.

[22] D. Kondo, A. Chien, and H. Casanova, "Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids," in *Proc. of SC*, 2004.

[23] D. Zhou and V. Lo, "Wave Scheduler: Scheduling for Faster Turnaround Time in Peer-based Desktop Grid Systems," in *Proc. of the 11th JSSPP Workshop*, 2005.

[24] T. Estrada, D. Flores, M. Taufer, P. Teller, A. Kerstens, and D. Anderson, "The Effectiveness of Threshold-Based Scheduling Policies in BOINC Projects," in *Proc. of e-Science'06*, 2006.

[25] C. Anglano, J. Brevik, M. Canonico, D. Nurmi, and R. Wolski, "Fault-aware scheduling for Bag-of-Tasks applications on Desktop Grids," in *Proc. of Grid Computing*, 2006, pp. 56–63.

[26] E. Byun, S. Choi, M. Baik, J. Gil, C. Park, and C. Hwang, "MJSA: Markov job scheduler based on availability in desktop grid computing environment," *FGCS*, vol. 23, pp. 616–622, 2007.

[27] T. Estrada, O. Fuentes, and M. Taufer, "A distributed evolutionary method to design scheduling policies for volunteer computing," *SIGMETRICS Perf. Eval. Rev.*, vol. 36, no. 3, pp. 40–49, 2008.

[28] J. Wingstrom and H. Casanova, "Probabilistic Allocation of Tasks on Desktop Grids," in *Proc. of PCGrid*, 2008.

[29] E. Heien, D. Anderson, and K. Hagihara, "Computing Low Latency Batches with Unreliable Workers in Volunteer Computing Environments," *Journal of Grid Computing*, vol. 7, no. 4, pp. 501–518, 2009.

[30] N. Fujimoto and K. Hagihara, "Near-Optimal Dynamic Task Scheduling of Independent Coarse-Grained Tasks onto a Computational Grid," in *Proc. of ICPP*, 2003.

[31] C. Anglano and M. Canonico, "Scheduling algorithms for multiple Bag-of-Task applications on Desktop Grids: A knowledge-free approach," in *Proc. of IPDPS*, 2008.

[32] J. Celaya and L. Marchal, "A Fair Decentralized Scheduler for Bag-of-Tasks Applications on Desktop Grids," in *Proc. of CCGrid*. IEEE CS Press, 2010, pp. 538–541.

[33] C. Moretti, T. Faltemier, D. Thain, and P. Flynn, "Challenges in Executing Data Intensive Biometric Workloads on a Desktop Grid," in *Proc. of PCGrid*, 2007.

[34] T. Toyoma, Y. Yamada, and K. Konishi, "A Resource Management System for Data-Intensive Applications in Desktop Grid Environments," in *Proc. of PDCS*, 2006.

[35] H. He, G. Fedak, B. Tang, and F. Cappello, "BLAST Application with Data-Aware Desktop Grid Middleware," in *Proc. of CCGrid*, 2009, pp. 284–291.

[36] M. Dawande, P. Keskinocak, J. Swaminathan, and S. Tayur, "On bipartite and multipartitie clique problems," *Journal of Algorithms*, vol. 41, pp. 388–403, 2001.

[37] D. Kondo, G. Fedak, F. Cappello, A. Chien, and H. Casanova, "Characterizing Resource Availability in Enterprise Desktop Grids," *FGCS*, vol. 23, no. 7, pp. 888–903, 2007.