Mapping Applications on Volatile Resources

Henri Casanova, Fanny Dufossé, Yves Robert, Frédéric Vivien

Mapping Applications on Volatile Resources

Henri Casanova[1], Fanny Dufossé[2], Yves Robert[3,4] and Frédéric Vivien[3]

[1] University of Hawai'i at Manoa, USA
[3] Laboratoire LIP, ENS Lyon & INRIA
[4] University of Tennessee, USA
henric@hawaii.edu,Fanny.Dufosse@laas.fr
Yves.Robert@ens-lyon.fr,Frederic.Vivien@inira.fr

November 15, 2013

Abstract

In this paper, we study the execution of iterative applications on volatile processors such as those found on desktop grids. We envision two models, one where all tasks are assumed to be independent, and another where all tasks are tightly coupled and keep exchanging information throughout the iteration. These two models cover the two extreme points of the parallelization spectrum. We develop master-worker scheduling schemes that attempt to achieve good trade-offs between worker speed and worker availability. Any iteration entails the execution of a fixed number of independent tasks or of tightly-coupled tasks. A key feature of our approach is that we consider a communication model where the bandwidth capacity of the master for sending application data to workers is limited. This limitation makes the scheduling problem more difficult both in a theoretical sense and in a practical sense. Furthermore, we consider that a processor can be in one of three states: available, down, or temporarily preempted by its owner. This preempted state also complicates the scheduling problem. In practical settings, e.g., desktop grids, master bandwidth is limited and processors are temporarily reclaimed. Consequently, addressing the aforementioned difficulties is necessary for successfully deploying master-worker applications on volatile platforms.

Our first contribution is to determine the complexity of the scheduling problems in their offline versions, i.e., when processor availability behaviors are known in advance. Even with this knowledge, the problems are NP-hard. Our second contribution is an evaluation of the expectation of the time needed by a worker to complete a set of tasks. We obtain a close formula for independent tasks and an analytical approximation for tightly-coupled tasks. Those evaluations rely on a Markovian assumption for the temporal availability of processors, and are at the heart of some heuristics that aim at favoring "reliable" processors in a sensible manner. Our third contribution is a set of heuristics for both model, which we evaluate in simulation. Our results provide guidance to selecting the best strategy as a function of processor state availability versus average task duration.
Key words: Iterative applications, scheduling, complexity results, heuristics, volatile platforms.

1 Introduction

In this paper we study the problem of efficiently executing parallel applications onto platforms that comprise volatile resources. More specifically we focus on scientific iterative applications implemented using the master-worker paradigm. The master parallelizes the execution of the iterations across available processors. Each iteration corresponds to the execution of a fixed number of tasks, and there is a synchronization

(or checkpoint) at the end of each iteration, before proceeding to the next one. We envision two generic models for the tasks that constitute each iteration: in the INDEPENDENT model, all tasks are assumed to be independent, while in the TIGHTLY-COUPLED model, we consider that all tasks are tightly coupled and keep exchanging information throughout the iteration. These two models cover the two extreme points of the parallelization spectrum, and are representative of a very large class of scientific applications. For example, this scheme applies to a broad spectrum of scientific computations including, but not limited to, mesh based solvers (e.g., elliptic PDE solvers), signal processing applications (e.g., recursive convolution), and image processing algorithms (e.g., stencil algorithms). We study such applications when they are executed on networked processors whose availability evolves over time, meaning that each processor alternates between being available for executing a task and being unavailable.

Solutions for executing master-worker applications, and in particular applications implemented with the Message Passing Interface (MPI), on failure-prone platforms have been developed (e.g., [21, 17, 35, 8]). In these works, the focus is on tolerating *failures* caused by software or hardware faults. For instance, a software fault will cause the processor to stall, but computations may be restarted from scratch or be resumed from a saved state after rebooting. A hardware failure may keep the processor down for a long period of time, until the failed component is repaired or replaced. In both cases, fault-tolerant mechanisms are implemented in the aforementioned solutions to make faults transparent to the application execution.

In addition to failures, processor volatility can also be due to *temporary interruptions*. Such interruptions are common in volunteer computing platforms [6] and desktop grids [15]. In these platforms processors are contributed by resource owners who can reclaim them at any time, without notice, and for arbitrary durations. A task running on a reclaimed processor is simply suspended. At a later date, when the processor is released by its owner, the task can be resumed without any wasted computation. In fact, fault-tolerant MPI solutions were proposed in the specific context of desktop grids [8], to accommodate for such interruptions. While mechanisms for executing master-worker applications on volatile platforms are available, our focus is on scheduling algorithms for deciding which processors should run which tasks and when.

At a given time a (volatile) processor can be in one of three states: UP (available), $DOWN$ (crashed due to a software or hardware fault), or $RECLAIMED$ (temporarily preempted by owner). Accounting for the $RECLAIMED$ state, which arises in desktop grid platforms, complexifies scheduling decisions. More specifically, since before going to the $DOWN$ state a processor may alternate between the UP and $RECLAIMED$ states, the time needed by the processor to compute a given workload to completion is difficult to predict. A way to make such prediction tractable is to assume that state transitions obey a Markov process. The Markov (i.e., memoryless) assumption is popular because it enables analytical derivations. In fact, recent work on desktop grid scheduling has made use of this assumption [10]. Unfortunately, the memoryless assumption is known not to hold in practice. Several authors have reported that the durations of availability intervals in production desktop grids are not sampled from exponential distributions [38, 47, 32]. There is no true consensus regarding what is a "good" model for availability intervals defined by the elapsed time between processor *failures*, let alone regarding a model for the durations of *recoverable interruptions*. While some authors have attempted to model processor availabilities using (non-memoryless) semi-Markov processes [39], we use a Markov model for transitions between the UP, $DOWN$, and $RECLAIMED$ states. The goal of this work is to provide algorithmic foundations for scheduling iterative master-worker applications on processors that can fail or be temporarily reclaimed. A 3-state Markovian model allows us to achieve this goal, and the insight from our results should provide guidance for dealing with more complex, and hopefully more realistic, stochastic models of processor availabilities (or ideally Markov approximation thereof). The transition probabilities for a real-life platform can be determined using traces. For example, for a given processor P_q, knowing that the resource is UP at time t, the probability that it remains UP will be computed as the number of occurrences of two successive UP states, divided by the number of UP states during all time slots described by the trace of this resource.

A unique aspect of this work is that we account for network bandwidth constraints for communica-

tion between the master and the workers. More specifically, we bound the total outgoing communication bandwidth of the master while ensuring that each communication uses a reasonably large fraction of this bandwidth. The master is thus able to communicate simultaneously with only a limited number of workers, sending them either the application program or input data for tasks. This assumption, which corresponds to the bounded multi-port model [29], applies to concurrent data transfers implemented with multi-threading. One alternative is to simply not consider these constraints. In this case, a scheduling strategy could enroll a large (and vastly suboptimal) number of processors to which it would send data concurrently each at very low bandwidth. Another alternative is to disallow concurrent data transfers from the master to the workers. In this case, the bandwidth capacity of the master may not be fully exploited, especially for workers executing on distant processors. We conclude that considering the above bandwidth constraints is necessary for applications that do not have extremely low communication-to-computation ratios. It turns out that the addition of these constraints makes the problem dramatically more difficult at the theoretical level, and thus complicates the design of practical scheduling strategies.

The specific scheduling problem under consideration is to maximize the number of application iterations that are successfully completed before a deadline. Informally, during each iteration, we have to identify the "best" processors among those that are available (e.g., the fastest, the likeliest to remain available, etc.). In addition, since processors can become available again after being unavailable for some time, it may be beneficial to change the set of enrolled processors even if all enrolled processors are available. We thus have to decide whether to release enrolled processors, to decide which ones should be released, and to decide which ones should be enrolled instead. Such changes come at a price: the application program file must be sent to newly enrolled processors, which consumes some (potentially precious) fraction of the master's bandwidth.

Our contributions are the following. First, we assess the complexity of the problems in their offline versions, i.e., when processor availability behaviors are known in advance. Even with this knowledge, the problems are NP-hard. Note that the offline scenario assesses the intrinsic complexity of the problem. Next, relying on the Markov assumption for processor availability, we evaluate the expectation of the time needed by a worker to complete a set of tasks. We provide a closed-form formula for the INDEPENDENT model and an analytical approximation for the TIGHTLY-COUPLED model. Those evaluations are at the heart of several heuristics that aim at giving priority to "reliable" resources rather than to "fast" ones. In a nutshell, when the task size is very small in comparison to the expected duration of an interval between two consecutive processor state changes, "classical" heuristics based upon the estimated completion time of a task perform reasonably well. But when the task size is no longer negligible with respect to the expected duration of such an interval, it is mandatory to account for processor reliability, and only those heuristics building upon such knowledge are shown to achieve good performance. Altogether, we design a set of heuristics, which we thoroughly evaluate in simulation. The results provide insights for selecting the best strategy as a function of processor state availability versus task duration.

This paper is organized as follows. Section 2 discusses related work. Section 3 describes the application and platform models. Complexity results for the offline study are given in Section 4; these results do not rely on any assumption regarding stochastic distribution of resource availability. In Section 5, we describe our 3-state Markovian model of processor availability, and we show how to compute the expected time for a configuration to complete an iteration in both application models. Heuristics for the online problem are described in Section 6, some of which use the result in Section 5 for more informed resource selection. An experimental evaluation of the heuristics is presented in Section 7. Section 8 concludes with a summary of our findings and perspectives on future work.

2 Related work

Iterative applications that can be implemented in master-worker fashion are widely used in computational linear algebra for sparse linear systems [14, 42, 45] or eigenvalue problems [40], image processing [48, 12, 33], signal processing [9, 16], etc. Paper [14] presents a relaxation method with chaotic iterations, extended in [42] to non-deterministic decisions. Paper [45] studies partitioning for iterative preconditioner matrices. Paper [40] introduces an iterative method for computing extremal eigenvalues of symmetric matrices. Iterative applications are useful to a wide spectrum of research topics, including digital pattern thinning [48], image reconstitution [12] or image denoising [33]. Similarly, iterative methods exist for signal processing [9] or signal detection [16]. While iterative applications can be asynchronous [42, 13, 5], in this work we restrict to the synchronous case. Several authors have proposed scheduling approaches for asynchronous applications [4, 31, 27, 36]. Paper [42] presents a Java environment for asynchronous iterative applications, and paper [13, 5] presents the asynchronous deployment of an iterative solution to a Poisson problem.

Scheduling methods for iterative applications have been studied on a wide range of architectures, including processor arrays [31], workstations with congestion control protocol [27], and virtual rings of processors [36]. In this work we consider volatile compute resources, such as those found in desktop grids. Resource volatility in these platforms has been studied in in [38, 47, 32, 7], leading to several interesting scheduling problems. Papers [38, 47, 32, 7] consider many possible statistical distributions to real life traces. They all consider the Weibull distribution as one of the most accurate choices. Paper [38] considers the hyper-exponential distribution as the best choice, paper [47] prefers the log-normal distribution, and paper [32] elects the Gamma distribution. However, none of these papers compare any two of these latter distributions.

Several authors have studied the "bag-of-tasks" scheduling problem, i.e., the scheduling of independent tasks, on volatile desktop grids, either at an Internet-wide scale or within an Enterprise [34, 49, 19, 1, 10, 20, 46, 28, 30]. 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 statistics of past availability [34, 19, 20, 49, 30], and on predictive statistical models of availability (e.g., average availability interval duration, average available CPU power) [46, 28, 1, 10]. These criteria are used to rank processors but also to exclude them from consideration [34, 19]. The impact of uncertainty in long-distance communications (high latency and unpredictable throughput) is discussed in [3]. The work in [10] is related to our work in that it uses a Markov model of processor availability (but without accounting for temporary preemption). Note that, because they target independent tasks, 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, an approach that we use in this work as well.

Most works published in this area are of a pragmatic nature (one exception is the work in [22]). In fact, given the wealth of proposed scheduling approaches and the lack of theoretical results, in [20], the authors propose to automatically instantiate the parameters that together define the behavior of a generic scheduling algorithm. By contrast, a large part of this work focuses on theoretical results. Some of the heuristics that we evaluate in this work directly reuse the same ideas as those pioneered in the works cited in the previous paragraph, while other heuristics rely on novel ideas that come from our theoretical results. Several of the heuristics listed above have been designed for independent tasks, while this work also targets tightly-coupled tasks, which imposes more stringent scheduling constrains.

Furthermore, a distinctive aspect of our work is that we seek to develop scheduling algorithms that explicitly manage the master's bandwidth. To the best of our knowledge, no previous work has made such an attempt. Limited master bandwidth is a known issue for desktop grid computing [37, 44, 26] and must therefore be addressed even though it complicates the scheduling problem. Paper [37] considers a homogeneous 1-port communication model with dynamic storage availability, and paper [26] considers a computational grid whose resources are scattered across several sites, with a limited bandwidth between

Table 1: List of notations.

Notation	Definition
N	number of time-slots
m	number of parallel tasks within each iteration
p	total number or resources
w_q	number of times-slots for P_q to execute a task
x_q	number of tasks allocated to processor P_q by current configuration
u, r, d	processor states: UP, $RECLAIMED$, and $DOWN$
$\mathcal{S}_q[t] \in \{u, r, d\}$	state of P_q at time-slot t
$P_{i,j}^{(q)}$	probability for P_q to move from state $i \in \{u, r, d\}$ to state $j \in \{u, r, d\}$
$\pi_u^{(q)}, \pi_r^{(q)}, \pi_d^{(q)}$	steady-state fraction of time that P_q is UP, $RECLAIMED$ and $DOWN$
T_{prog} (resp. V_{prog})	number of time-slots to send (resp. size of) the program
T_{data} (resp. V_{data})	number of time-slots to send (resp. size of) a task data file
bw (resp. BW)	bandwidth of the workers (resp. of the master)
n_{prog} (resp. n_{data})	number of processors receiving the program (resp. a task data) at current time slot
n_{com}	maximum number of processors that the master can communicate with at any time-slot
$config(t)$	configuration (set of enrolled processors) at time-slot t

sites.

Finally, note that while in this work we focus on scheduling a single application, other authors have studied the scheduling problem for multiple simultaneous applications [2, 11].

3 Problem Definition

In this section, we detail our application and platform models, describe the scheduling model, and provide a precise statement of the scheduling problem. Table 1 summarizes all notations.

3.1 Application Model

We target an iterative application in which iterations entail the execution of a fixed number m of same-size tasks. Each iteration is executed in a master-worker fashion, with a synchronization of all tasks at the end of the iteration. A processor is assigned one or more tasks during an iteration. Each task needs some input data, of constant size V_{data} in bytes. This data depends on the task and the iteration, and is received from the master. Such applications allow for a natural overlap of computation and communication: computing for the current task can occur while data for the next task (of the same iteration) is being received. Before it can start computing, a processor needs to receive the application program from the master, which is of size V_{prog} in bytes. This program is the same for all tasks and iterations.

We consider two variants for the application model:

- In the INDEPENDENT model, tasks are independents.
- In the TIGHTLY-COUPLED model, the m same-size tasks steadily communicate throughout the iteration. 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.

3.2 Platform Model

We consider a platform that consists of p *processors*, $P_1, \ldots, P_p$, encompassing with this term compute nodes that contain multiple physical processor cores. Each processor is volatile, meaning that its availability for computing application tasks varies over time. More precisely, a processor can be in one of three states: UP (available for computation), $RECLAIMED$ (temporarily reclaimed by its owner), or $DOWN$ (crashed and to be rebooted). We assume that the master, which implements the scheduling algorithm, executes on a processor that is always UP (otherwise a simple redundancy mechanism such as primary back-up [24] can be used to ensure reliability of the master). We also assume that the master is aware of the states of the processors, e.g., via a simple heart-beat mechanism [41]. Processor availabilities evolve independently, and all state transitions are allowed, with the following implications:

- When a UP or $RECLAIMED$ processor becomes $DOWN$, it loses the application program, all the data for its assigned tasks, and all partially computed results. When it later becomes UP it has to acquire the program again before executing tasks;
- When a UP processor becomes $RECLAIMED$, its activities are suspended. However, when it becomes UP again it can simply resume task computations and data transfers.

We discretize time so that the execution occurs over a sequence of discrete *time slots*. We assume that task computations and data transfers all require an integer number of time slots, and that processor state changes occur at time-slot boundaries. We leave the time slot duration unspecified. The time slot duration that achieves a good approximation of continuous time varies for different applications and platforms.

The temporal availability of P_q is described by a vector $\mathcal{S}_q$ whose component $\mathcal{S}_q[t] \in \{u, r, d\}$ represents its state at time-slot t. Here u corresponds to the UP state, r to the $RECLAIMED$ state, and d to the $DOWN$ state. Vector $\mathcal{S}_q$ is unknown before executing the application.

Processor P_q requires w_q time-slots of availability (i.e., UP state) to compute a task. If all w_q values are identical, then the platform is homogeneous. We model communications between the master and the workers using the *bounded multi-port* communication model [29]. In this model, the master can initiate multiple concurrent communications, each to a different worker. Each communication is allotted a bandwidth fraction of the master's network card, and the sum of all fractions cannot exceed the total capacity of the card. This model is enabled by popular multi-threaded communication libraries [23]. We consider that the master can communicate up to bandwidth BW (we use the term "bandwidth" loosely to mean maximum data transfer rate). Communication to each worker is performed at some fixed bandwidth bw. This bandwidth can be enforced in software or can correspond to same-capacity communication paths from the master's processor to each other processor. We define $n_{\text{com}} = BW/bw$ as the maximum number of workers to which the master can send data simultaneously (i.e., the maximum number of simultaneous communications). For simplicity, we assume n_{com} to be an integer. Let n_{prog} be the number of processors receiving the application program at time t, and n_{data} be the number of processors receiving the input data of a task at time t. Given that the bandwidth of the master must not be exceeded, we have

$$n_{\text{prog}} + n_{\text{data}} \leq n_{\text{com}} = BW/bw.$$

Let P_q be a processor engaged in communication at time t, for receiving either the program or input data. In both cases, it does this with bandwidth bw. Hence 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 $T_{\text{data}} = V_{\text{data}}/bw$.

3.3 Scheduling Model

Let $config(t)$ denote the set of processors enrolled for computing the m application tasks in an iteration, or *configuration*, at time t.

- In the INDEPENDENT model, enrolled processors work independently, and execute their tasks sequentially. While a processor could conceivably execute two tasks in parallel (provided there is enough available memory), this would only delay the completion time of the first task, thereby increasing the risk of not completing it at all due to volatile availability.

- In the TIGHTLY-COUPLED model, 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 assigns tasks to processors and may choose a new configuration at each time-slot t. Let P_q be a newly enrolled processor at time t, i.e., $P_q \in \mathit{config}(t+1) \setminus \mathit{config}(t)$. P_q needs to receive the program unless it already received a copy of it and has not been in the $DOWN$ state since. In all cases, P_q needs to receive data for a task before computing it. This holds true even if P_q had been enrolled at some previous time-slot $t' < t$ but has been un-enrolled since: we assume any received data is discarded when a processor is un-enrolled. In other words, any input data communication is resumed from scratch, even if it had previously completed. Note that a processor that is un-enrolled keeps the application program until it eventually goes to the $DOWN$ state. In addition, in the TIGHTLY-COUPLED model, if some computations was in progress during time-slot t, all progress is lost by this modification of configuration.

If a processor of the configuration becomes $DOWN$ at time t, the scheduler may simply use the remaining UP processors in $\mathit{config}(t)$ to complete the iteration, or enroll a new processor. In the TIGHTLY-COUPLED model, all computation in progress is lost. Even if all processors in $\mathit{config}(t)$ are in the UP state, the scheduler may decide to change the configuration. This can be useful if a more desirable (e.g., faster, more available) but un-enrolled processor has just returned to the UP state. Removing an UP processor from $\mathit{config}(t)$ has a cost: partial results of task computations, partial task data being received, and previously received task data are all lost. Note, however, that, in the INDEPENDENT model, results obtained for previously completed tasks are not lost because already sent back to the master.

In the INDEPENDENT model, due to the possibility of a processor leaving the configuration (either due to becoming $DOWN$ or due to a decision of the scheduler), the scheduler enforces that task data is received for at most one task beyond the one currently being computed. In other terms, the processor does not accumulate task data beyond that for the next task. This is sensible so as to allow some overlap of computation and communication while avoiding wasting bandwidth for data transfers that would be increasingly likely to be redone from scratch.

Throughout the paper, we do not take the output of results by the workers into account, because these results can be merged with the next incoming communication from the master. This amounts to increasing the value of T_{data} in the analysis.

3.4 Problem Statement

The scheduling problem that we address in this work is to maximize the number of successfully completed application iterations before a deadline. Given the discretization of time, the objective of the scheduling problem is then to maximize the number of successfully completed iterations within some integral number of time slots, N. In the offline case (see Section 4), if an efficient algorithm can be found to solve this problem,

then, using a binary search, an efficient algorithm can be designed to solve the problem of executing a given number of iterations in the minimum amount of time.

4 Off-line complexity

In this section, we study the offline complexity of the INDEPENDENT and TIGHTLY-COUPLED problems. This means that we assume a priori knowledge of all processor states. In other words, the value of $\mathcal{S}_q[j]$ is known in advance, for $1 \leq q \leq p$ and $1 \leq j \leq N$. The problem turns out to be difficult: even minimizing the time to complete the first iteration with same-speed processors is NP-complete for both models. We also identify some polynomial instances. For approximation questions, we take into account incomplete instances: if p tasks have been executed in the current time slot, then for the current iteration we consider that a part $\frac{p}{n}$ is completed. Without this assumption, the problem is inapproximable.

For the offline study, we can simplify the model and have only two processor states, UP (also denoted by u) and $RECLAIMED$ (also denoted by r). Indeed, suppose that processor P_q is $DOWN$ for the first time at time-slot t: $\mathcal{S}_q[t] = d$. We can replace P_q by two 2-state processors $P_{q'}$ and $P_{q''}$ such that: 1) for all $j < t$, $\mathcal{S}_{q'}[j] = \mathcal{S}_q[j]$ and $\mathcal{S}_{q''}[j] = r$, 2) $\mathcal{S}_{q'}[t] = \mathcal{S}_{q''}[t] = r$, and 3) for all $j > t$, $\mathcal{S}_{q'}[j] = r$ and $\mathcal{S}_{q''}[j] = \mathcal{S}_q[j]$. In this way, we remove a $DOWN$ state and add a two-state processor. If we do this modification for each $DOWN$ state, we obtain an instance with only UP or $RECLAIMED$ processors. In the worst case, the total number of processors is multiplied by N, which does not affect the problem's complexity (polynomial versus NP-hard). Let OFFLINE-INDEPENDENT denote the problem of minimizing the time to complete the first iteration, with same-speed processors:

For the sake of brevity, all results in this section are provided without proof. Full details are available in [18].

4.1 Complexity of INDEPENDENT problems

Theorem 1 *Problem* OFFLINE-INDEPENDENT *is NP-hard.*

Proposition 1 *Problem* OFFLINE-INDEPENDENT *cannot be approximated within* $\frac{8}{7} - \epsilon$ *for all* $\epsilon > 0$.

Proof. MAXIMUM 3-SATISFIABILITY cannot be approximated within $\frac{8}{7} - \epsilon$ for all $\epsilon > 0$ [25]. The result is immediate for problem OFFLINE-INDEPENDENT by construction of the proof of Theorem 1. □

Now we show that the difficulty of problem OFFLINE-INDEPENDENT is due to the bound n_{com}: if we relax this bound, the problem becomes polynomial.

Proposition 2 OFFLINE-INDEPENDENT *is polynomial when* $n_{com} = +\infty$*, even with different-speed processors.*

4.2 Complexity of TIGHTLY-COUPLED problems

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 OFFLINE-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 OFFLINE-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 OFFLINE-COUPLED ($\mu = +\infty$).

Theorem 2 *Problems* OFFLINE-COUPLED *($\mu = 1$) and* OFFLINE-COUPLED *($\mu = \infty$) are NP-hard.*

5 Computing the expectation

In this section, we first introduce a Markov model for processor availability. Then we show how to compute in the INDEPENDENT model the *expected time* needed by a processor to complete a given workload. Finally, we show how to compute in the TIGHTLY-COUPLED model the probability of success and the expected execution time of a workload. These quantities will guide the design of some online heuristics in Section 6.

5.1 The INDEPENDENT model

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-slot t to state j at time-slot $t+1$, which does not depend on t. We denote by $\pi_u^{(q)}$, $\pi_r^{(q)}$ and $\pi_d^{(q)}$ the limit distribution of P_q's Markov chain (i.e., steady-state fractions of state occupancy for states UP, $RECLAIMED$, and $DOWN$). This limit distribution is easily computed from the transition probability matrix, and $\pi_u^{(q)} + \pi_r^{(q)} + \pi_d^{(q)} = 1$.

When designing heuristics to assign tasks to processors, it seems important to take into account the expected execution time of a processor until it completes all tasks assigned to it. Indeed, speed is not the only factor, as the target processor may well become $RECLAIMED$ several times before executing all its scheduled computations. We develop an analytical expression for such an expectation as follows.

Consider a processor P_q in the UP state at time t, which is assigned a workload that requires W time-slots in the UP state for completing all communications and/or computations. To complete the workload, P_q must be UP during another $W-1$ time-slots. It can possibly become $RECLAIMED$ but never $DOWN$ in between. What is the probability of the workload being completed? And, if it is completed, what is the expectation of the number of time-slots until completion?

Definition 1 *Knowing that P_q is UP at time-slot t_1, let $\mathbf{P_+^{(q)}}$ be the conditional probability that it will be UP at a later time-slot, without going to the $DOWN$ state in between. Formally, knowing that $\mathcal{S}_q[t_1] = u$, $P_+^{(q)}$ is the conditional probability that there exists a time t_2 such that $\mathcal{S}_q[t_2] = u$ and $\mathcal{S}_q[t] \neq d$ for $t_1 < t < t_2$.*

Definition 2 *Let $\mathbf{E^{(q)}(W)}$ be the conditional expectation of the number of time-slots required by P_q to complete a workload of size W knowing that it is UP at the current time-slot t_1 and will not 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 $t \in [t_1, t_W]$, $E^{(q)}(W)$ is the conditional expectation of $t_W - t_1 + 1$.*

Lemma 1 $P_+^{(q)} = P_{u,u}^{(q)} + \frac{P_{u,r}^{(q)} P_{r,u}^{(q)}}{1 - P_{r,r}^{(q)}}$.

Proof. The probability that P_q will be available again before crashing is the probability that it remains available during the next time-slot, plus the probability that it becomes $RECLAIMED$ and later returns to the UP state before crashing. We obtain that

$$P_+^{(q)} = P_{u,u}^{(q)} + P_{u,r}^{(q)} \left(\sum_{t=0}^{+\infty} (P_{r,r}^{(q)})^t \right) P_{r,u}^{(q)},$$

hence the result. □

Theorem 3 $E^{(q)}(W) = W + (W-1) \times \frac{P_{u,r}^{(q)} P_{r,u}^{(q)}}{1-P_{r,r}^{(q)}} \times \frac{1}{P_{u,u}^{(q)}(1-P_{r,r}^{(q)})+P_{u,r}^{(q)}P_{r,u}^{(q)}}$.

Proof. To execute the whole workload, P_q needs $W-1$ additional time-slots of availability. Consequently, the probability that P_q successfully executes its entire workload before crashing is $(P_+^{(q)})^{W-1}$.

The key idea to prove the result is to consider $E^{(q)}(up)$, the expected value of the number of time-slots before the next UP time-slot of P_q, knowing that it is up at time 0 and will not become $DOWN$ in between:

$$E^{(q)}(up) = \frac{P_{u,u}^{(q)} + \sum_{t\geq 0}(t+2)P_{u,r}^{(q)}(P_{r,r}^{(q)})^t P_{r,u}^{(q)}}{P_+^{(q)}}.$$

To compute $E^{(q)}(up)$, we study the value of

$$A = \sum_{t\geq 0}(t+2)P_{u,r}^{(q)}(P_{r,r}^{(q)})^t P_{r,u}^{(q)} = \frac{P_{u,r}^{(q)}P_{r,u}^{(q)}}{P_{r,r}^{(q)}} \sum_{t\geq 0}(t+2)(P_{r,r}^{(q)})^{t+1} = \frac{P_{u,r}^{(q)}P_{r,u}^{(q)}}{P_{r,r}^{(q)}} g'(P_{r,r}^{(q)})$$

with $g(x) = \sum_{t\geq 0} x^{t+2} = \frac{x^2}{1-x}$. Differentiating, we obtain $g'(x) = \frac{x(2-x)}{(1-x)^2}$ and

$$A = \frac{P_{u,r}^{(q)}P_{r,u}^{(q)}}{P_{r,r}^{(q)}} \times \frac{P_{r,r}^{(q)}(2-P_{r,r}^{(q)})}{(1-P_{r,r}^{(q)})^2}.$$

Letting $z = \frac{P_{u,r}^{(q)}P_{r,u}^{(q)}}{P_{u,u}^{(q)}(1-P_{r,r}^{(q)})}$, we derive

$$E^{(q)}(up) = \frac{1 + z\frac{(2-P_{r,r}^{(q)})}{(1-P_{r,r}^{(q)})}}{1+z} = 1 + \frac{z}{(1-P_{r,r}^{(q)})(1+z)}$$

We then conclude by noting that $E^{(q)}(W) = 1 + (W-1) \times E^{(q)}(up)$. □

5.2 The TIGHTLY-COUPLED model

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 3 *Knowing that all processors in a set S are UP at time-slot t_1, let $\mathbf{P_+^{(S)}}$ 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 4 *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 4 *It is possible to approximate the values of $P_+^{(S)}$ and $E^{(S)}(W)$ numerically up to an arbitrary precision ε 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 $P^{(q)}_{\substack{t\\u\to u}}$ 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 $P^{(S)}_{\substack{t\\u\to u}} = \prod_{P_q\in S} P^{(q)}_{\substack{t\\u\to u}}$. For each processor P_q, the value $P^{(q)}_{\substack{t\\u\to u}}$ 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 $P^{(S)}_{\substack{t\\u\to u}}$. We derive that

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

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

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

Let $E_u(S) = \sum_{t>0} P^{(S)}_{\substack{t\\u\to u}}$. 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 $E(A_t) = P^{(S)}_{\substack{t\\u\to u}}$. By linearity of the expectation, we have $E(\sum_{0\le t'\le t} A_{t'}) = \sum_{0\le t'\le t} P^{(S)}_{\substack{t'\\u\to u}}$. Suppose that, in set S, at least one processor has a nonzero probability of going $DOWN$. Then, $\lim_{t\to\infty} \sum_{0\le t'\le t} P^{(S)}_{\substack{t'\\u\to u}}$ converges. We can conclude that $E(\sum_{t>0} A_t) = \sum_{t>0} P^{(S)}_{\substack{t\\u\to u}}$. 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,

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

Let $A(S) = \sum_{t>0} t \times P^{(S)}_{\substack{t\\u\to u}}$. 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 ε be the desired precision. Consider for some value T the difference between $E_u(S)$ and $\sum_{0<t<T} P^{(S)}_{\substack{t\\u\to u}}$. We have $P^{(S)}_{\substack{t\\u\to u}} = \prod_{P_q\in S} P^{q}_{\substack{t\\u\to u}}$ and $P^{q}_{\substack{t\\u\to u}}$ 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, $P^{q}_{\substack{t\\u\to u}} = (M^t_q)[0,0]$. We obtain $P^{q}_{\substack{t\\u\to u}} = \mu(\lambda^q_1)^t + \nu(\lambda^q_2)^t$ with $\mu,\nu \geq 0$, $\mu+\nu = 1$ and $\lambda^q_1 > \lambda^q_2$ eigenvalues of M_q. Then, $P^{q}_{\substack{t\\u\to u}} \leq (\lambda^q_1)^t$. We obtain $P^{(S)}_{\substack{t\\u\to u}} \leq \left(\prod_{P_q\in S}\lambda^q_1\right)^t$ and $\sum_{t\geq T} P^{(S)}_{\substack{t\\u\to u}} \leq \left(\prod_{P_q\in S}\lambda^q_1\right)^T \times \frac{1}{1-\prod_{P_q\in S}\lambda^q_1}$. Let $\Lambda = \prod_{P_q\in S}\lambda^q_1$. We obtain that $T > \frac{\ln(\varepsilon(1-\Lambda))}{\ln(\Lambda)}$ implies $E_u(S) - \sum_{0<t<T} P^{(S)}_{\substack{t\\u\to u}} \leq \varepsilon$. Thus, we can compute in polynomial time an approximation of $E_u(S)$ at ε in polynomial time.

Similarly, we obtain $A(S) - \sum_{0<t<T} t \times P^{(S)}_{\substack{t\\u\to u}} \leq \varepsilon$ 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 ε in polynomial time. □

We conclude this section with the following remark: when scheduling with heterogeneous processors, the most widely used approach is HEFT [43], a list-schedule heuristic that assigns the next ready task to the processor that will complete its execution first (given already taken decisions). Theorems 3 and 4 provide an estimation of the time needed to complete a given workload on one processor (INDEPENDENT model) and on a set of processors (TIGHTLY-COUPLED model), which is exactly what is needed to design HEFT-like heuristics. Such heuristics give priority to completion times, and will be compared with others giving priority to reliability.

6 Online heuristics

In this section, we propose heuristics to address the online version of the problem. Conceptually, we can distinguish three main classes of heuristics:

Passive heuristics that conservatively keep current processors active as long as possible: the current configuration is changed only when one of the enrolled processors becomes $DOWN$. 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.

Dynamic heuristics that may change configuration on the fly even if no processor fails, while preserving ongoing work. More precisely, if a processor is engaged in a computation, it finishes it; if it is engaged in a communication, it finishes it together with the corresponding computation. But otherwise, tasks can be freely reassigned among processors, whether already enrolled or not. Intuitively, the idea is to benefit from, say, a fast and reliable resource that has just become UP, while not risking losing part of the work already completed for the current iteration.

Proactive heuristics that may change configuration on the fly even if no processor fails and possibly aborting ongoing computation (i.e., 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.

In the INDEPENDENT model, the dynamic strategy is the most sensible. A passive strategy would be unnecessarily restrictive. Assigning all m tasks once and for all without possible reassignment does not make sense because better processors may become available, and assigning one of more independent tasks to these processors is a better choice. A proactive strategy could be useful to handle the well-known problems of "straggler tasks," i.e., last tasks in the schedule that determine iteration time but that are assigned to processors that turn out to be slow. In this case, these tasks should be terminated and reassigned to faster

processors, which could have significant benefit when m is small. However, a simpler and popular solution is to use a dynamic strategy but to *replicate* these last tasks on one or more UP processors, canceling all remaining replicas when one of them completes. When considering communications, a simple approach would be to give priority to one replica by task. Task replication may seem wasteful, but it is a commonly used technique in desktop grid environments in which resources are plentiful and often free of charge. While never detrimental to execution time (provided communication is throttled to one replica per task), task replication is more beneficial when m is small. We use the following replica strategy in the INDEPENDENT model. A task is replicated whenever there are more processors in the UP state than there are remaining tasks to execute. We limit the number of additional replicas of a task to two, which has been used in previous work [34] and works well in our experiments (better performance than with only one additional replica, not significantly worse performance than with more additional replicas). For simplicity, we describe all our heuristics assuming no task replication, but it is to be understood that there are up to $3m$ tasks (instead of m) distributed by the master during each iteration; the m original tasks are given priority over replicas, which are scheduled only when room permits.

In the TIGHTLY-COUPLED model, it is the dynamic strategy that is too restrictive, since it would have to wait until an iteration completes (i.e., all computations and communications terminate) to change to a better configuration. Instead, proactive strategies can forcefully terminate computation/communication during an iteration to take advantage of newly available fast/reliable processor. 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. Note that the heuristics in the works reviewed in Section 2 are for the most part passive because they are designed for independent tasks. The exception is the "time-out" heuristic in [34] in which a task that has not completed by some arbitrary deadline is considered lost. Our proactive heuristics do not use such a time-out, as it is difficult to define its value in practice for a given application and platform, but instead consider a task lost when a better processor is available for running it.

All heuristics assign tasks to processors (that must be in the UP state) one-by-one, until m tasks are assigned. More precisely, at time slot t, there are enrolled processors that are currently active, either receiving some message, or computing a task, or both. Let m' be the number of tasks whose communication or computation has already begun at time t. Since ongoing activities are never terminated, there remain $m-m'$ tasks to assign to processors. The objective of the heuristics is to decide which processors should be used for these tasks.

The dynamic heuristics below fall into two classes, *random* and *greedy*. Most of these heuristics rely on the assumption that processor availability follows a Markov process, as discussed in Section 5. In real-world scenarios, these heuristics can be used provided empirical Markov approximations of the stochatic processes that determines processor availabilities can be derived. Before describing the heuristics, we make a short digression and start with a brief presentation of some *random* heuristics, which are both simple and natural, and will be used as reference for comparisons.

6.1 Random heuristics

The heuristics described in this section use randomness to select which processor, among the ones that are in the UP state, will execute the next task. The simplest heuristic, RANDOM, assigns the next task to a processor picked randomly using a uniform probability distribution. Going beyond RANDOM, it is possible to assign a weight to processor P_q, in a view to giving larger weight to more "reliable" processors. Processors are picked with a probability equal to their normalized weights. We propose four ways of defining these weights:

1. **Long time** UP: the weight of P_q is $P_{u,u}^{(q)}$, the probability that P_q remains UP, hence favoring processors that stay UP for a long time.
2. **Likely to work more**: the weight of P_q is $P_{+}^{(q)}$, the probability that P_q will be UP another time slot before crashing (see Section 5), hence favoring processors with high probability of becoming UP again before crashing.
3. **Often** UP: the weight of P_q is $\pi_u^{(q)}$, the steady-state fraction of time that P_q is UP, hence favoring processors that are UP more often.
4. **Rarely** $DOWN$: the weight of P_q is $(1-\pi_d^{(q)})$, hence favoring processors that are $DOWN$ less often.

We call the corresponding heuristics RANDOM1, RANDOM2, RANDOM3, and RANDOM4. For each of these four heuristics P_q's weight can be divided by w_q, attempting to account for processing speed as well as reliability. We thus obtain four additional variants, designed by the suffix 'W'.

In the TIGHTLY-COUPLED model, due to the large number of heuristics to compare, only one of theses four heuristics is evaluated.

6.2 Dynamic heuristics for the INDEPENDENT model

We propose four general greedy heuristics, each of which can be enhanced to account for network contention.

6.2.1 MCT (Minimum Completion Time)

Assigning a task to the processor that can complete it the soonest is the optimal policy in the offline case without network contention (Proposition 2). We apply MCT here as follows. For each processor P_q we compute Delay(q), the delay before P_q finishes its current activities, and after which it could be enrolled for one of the $m-m'$ remaining tasks to be scheduled. In addition to processors finishing ongoing work, other processors could need to receive all or part of the program. Because of processors becoming $RECLAIMED$, we cannot exactly compute Delay(q). As a first approach, we estimate it assuming that P_q remains in the UP state and that there is no network contention whatsoever. We then greedily assign each of the remaining $m-m'$ tasks to processors, picking each time the processor with the smallest task completion time. More formally, for each processor P_q, let n_q be the number of tasks already assigned to it (out of the $m-m'$ tasks), and let $CT(P_q, n_q)$ be the estimation of its completion time:

$$CT(P_q, n_q) = \text{Delay}(q) + T_{\text{data}} + \max(n_q - 1, 0)\max(T_{\text{data}}, w_q) + w_q \ . \tag{1}$$

MCT assigns the next task to processor P_{q_0}, where $q_0 = \text{ArgMin}\{CT(P_q, n_q + 1)\}$.

MCT with contention – The estimated completion time in Equation 1 does not account for network contention (caused by the master's limited network capacity). Because of the overlap between communications and computations, it is difficult to predict network traffic. Instead, we use a simple correcting factor, and replace T_{data} by $\left\lceil \frac{n_{\text{active}}}{n_{\text{com}}} \right\rceil T_{\text{data}}$, where n_{active} denotes the number of active processors, i.e., those processors that have been assigned one or several of the $m-m'$ tasks. The n_{active} counter is initialized to zero and is incremented when a task is assigned to a newly enrolled processor. The intuition is that this counter measures the average slowdown encountered by a worker when communicating with the master. This estimation is simple but pessimistic since all scheduled communications do not necessarily take place simultaneously. We derive the new estimation:

$$CT(P_q, n_q) = \text{Delay}(q) + \left\lceil \frac{n_{\text{active}}}{n_{\text{com}}} \right\rceil T_{\text{data}} + \max(n_q - 1, 0)\max(\left\lceil \frac{n_{\text{active}}}{n_{\text{com}}} \right\rceil T_{\text{data}}, w_q) + w_q \tag{2}$$

We call MCT* the version of the MCT heuristic that uses the above definition of $CT(P_q, n_q)$.

Expected MCT – Given a workload (i.e., a number of needed time-slots of computation) $CT(P_q, n_q)$, Theorem 3 gives the value of $E^{(q)}(CT(P_q, n_q))$, the expected number of time-slots needed for P_q to be UP during $CT(P_q, n_q)$ times-slots without becoming $DOWN$ in between. Using this expectation as the criterion for selecting processors, and depending on whether the correcting factor on T_{data} is used, we obtain one new version of MCT and one new version of MCT*, which we call EMCT and EMCT*, respectively.

6.2.2 LW (Likely to Work)

We build heuristics that consider the probability that a processor P_q, which is UP, will be UP again at least once before becoming $DOWN$. This probability, $P_+^{(q)}$, is given by Lemma 1. We assign the next task to processor P_{q_0} with the highest probability of being UP for at least the estimated number of needed time-slots to complete its workload, before becoming $DOWN$:

$$q_0 = \text{ArgMax}\left\{(P_+^{(q)})^{CT(P_q, n_q+1)}\right\} .$$

Therefore, we first estimate the size $\mathcal{W}$ of the workload and then the probability that a processor will be in the UP state $\mathcal{W}$ time-slots without becoming $DOWN$ in between. Using Equation 2 instead of Equation 1, one obtains the LW* heuristic.

6.2.3 UD (Unlikely Down)

Here, we estimate the number $\mathcal{N}$ of time-slots needed for a processor to complete its workload, knowing that it can become $RECLAIMED$. Then we compute the probability that it will not become $DOWN$ for $\mathcal{N}$ time-slots. Given that P_q starts in the UP state, the probability that it does not go to the $DOWN$ state during k time-slots is:

$$P_{UD}^{(q)}(k) = \begin{bmatrix} 1 & 1 \end{bmatrix} . \begin{bmatrix} P_{u,u}^{(q)} & P_{u,r}^{(q)} \\ P_{r,u}^{(q)} & P_{r,r}^{(q)} \end{bmatrix}^{k-1} . \begin{bmatrix} 1 \\ 0 \end{bmatrix} .$$

We approximate this expression by forgetting the state of P_q after the first transition:

$$P_{UD}^{(q)}(k) = (1 - P_{u,d}^{(q)}) \left(1 - \frac{P_{u,d}^{(q)}\pi_u^{(q)} + P_{r,d}^{(q)}\pi_r^{(q)}}{\pi_u^{(q)} + \pi_r^{(q)}}\right)^{k-2} .$$

We use this value with $k = E^{(q)}(CT(P_q, n_q + 1))$. UD assigns the next task to the processor P_{q_0} that maximizes the probability of not becoming $DOWN$ before the estimated number of time-slots needed for it to complete its workload, counting the time-slots spent in the $RECLAIMED$ state:

$$q_0 = \text{ArgMax}\{P_{UD}^{(q)}(E^{(q)}(CT(P_q, n_q + 1)))\} .$$

Using Equation 2 instead of Equation 1, one obtains the UD* heuristic.

6.3 Communication time evaluation

In the TIGHTLY-COUPLED model, all communications for the configuration are done in the same interval of time. We then adapt the communication time evaluation. 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 $E^{(S)}_{comm} = \max_{P_q \in S}\{E^{(P_q)}(n_q)\}$. 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: $E^{(S)}_{comm} = \max\left\{\max_{P_q \in S}\left\{E^{(P_q)}(n_q)\right\}, \frac{\sum_{P_q \in S} n_q}{n_{\text{com}}}\right\}$.

Let $P^{(P_q)}_{ND}(t)$ 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 $P^{(S)}_{comm} = \prod_{P_q \in S} P^{(P_q)}_{ND}(E^{(S)}_{comm})$. The expression for $P^{(S)}_{comm}$ 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.

6.4 Passive heuristics for the TIGHTLY-COUPLED model

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.

• **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 5: $P^{(S)}(q) = P^{(S\cup\{P_q\})}(W_q) \times P^{(S\cup\{P_q\})}_{comm}$ 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 $q_0 = \text{ArgMax}\left\{P^{(S)}(q)\right\}$. This natural idea of the most reliable workers has been used for scheduling independent tasks in [19, 1, 46].

• **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 $E^{(S\cup\{P_q\})}_{comm}$ 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 $E^{(S)}(q) = E^{(S\cup\{P_q\})}_{comm} + E^{(S\cup\{P_q\})}(W_q)$. We assign the next task to worker P_{q_0}, with $q_0 = \text{ArgMin}\left\{E^{(S)}(q)\right\}$. This idea of picking the fatest workers has been used for scheduling independent tasks in [34].

• **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 $q_0 = \text{ArgMax}\left\{\frac{P^{(S)}(q)}{t+E^{(S)}(q)}\right\}$. This general idea of trading off reliability for speed has been used in the context of independent tasks in many previous works [34, 1, 10, 28] and has been formulated in various ways (e.g., among the workers that have a high probability of successfully completing the extra

task pick the one that can do it the fatest [1], compute a reasonable deadline by which the application should complete so as to exclude workers that have a low probability of meeting that deadline and pick the fastest worker that can meet it [34], pick the worker with the highest expected number of delivered compute cycles [1]). The yield metric is a simple formulation of the same idea that we use in the context of tightly-coupled application.

• **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 $q_0 = \text{ArgMax}\left\{\frac{P^{(S)}(q)}{E^{(S)}(q)}\right\}$.

6.5 Proactive heuristics for the TIGHTLY-COUPLED model

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 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 3×4 proactive heuristics named C-H where $C \in \{\text{P, E, Y}\}$ and $H \in \{\text{IP, IE, IY, IAY}\}$, plus the RANDOM heuristic.

7 Experiments

We have evaluated the heuristics described in the previous section using a discrete-even simulator for the execution of application on volatile resources (The simulator is publicly available at `http://graal.ens-lyon.fr/~fdufosse/changing_platforms.tar.gz`). The simulator takes as input values for all the parameters listed in Section 3, and it assumes that temporal processor availability follows a Markov process.

Table 2: Parameter values for Markov experiments.

parameter	values
p	20
n	$5, 10, 20, 40$
n_{com}	$5, 10, 20$
w_{min}	$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$

For the simulation experiments, rather than fixing N, the 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 equivalent problem is simpler to instantiate since it does not require choosing meaningful N values, which would depend on the application and platform characteristics. We have executed all heuristics presented above for several problem instances. For each problem instance we compute the *degradation from best* (dfb) of each heuristic, i.e., the percentage relative difference between the makespan achieved by the heuristic and that achieved by the best heuristic, all for that particular instance. A value of zero means that the heuristic is best for the instance. We use this metric because makespans vary widely between instances depending on processor availability patterns. We also count how often, over all instances, each heuristic is the (or tied with the) best one, so that we can report on numbers of wins for each heuristics.

All our experiments are for $p = 20$ processors. The Markov chain that characterizes processor P_q's availability is defined as follows. We uniformly pick a random value 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 experimental scenario is defined by the above and by three parameters: n, the number of tasks per iteration, n_{com}, the constraint on the master's communication bandwidth, and the w_{min} parameter, which is used as follows. For each processor P_q, we pick w_q uniformly between w_{min} and $10 \times w_{min}$. T_{data} is set to w_{min}, meaning that the fastest possible processor has a computation-communication ratio of 1. T_{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 experimental scenarios for each of the possible instantiations of $(n, n_{\text{com}}, w_{min})$ given the values shown in Table 2. We must emphasize that our goal here is *not* to instantiate a representative model for a desktop grid and application, but rather to create arbitrary but simple synthetic experimental scenarios that will highlight inherent strengths and weaknesses of the heuristics. All heuristics are designed to achieve a given trade-off between availability, speed and reliability. Using these heurisics directly in real-world situations requires that empirical Markov approximations of the stochatic processes that determines processor availabilities be derived.

7.1 INDEPENDENT model

For each possible instantiation of the parameters in Table 2, we create 247 random experimental scenarios as described above. For each experimental scenario, we run 10 trials, varying the seed of the random number generator used to determine Markov state transitions. We compute average dfb values for each heuristic based over these 10 trials, for each experimental scenarios. The total number of generated problem instances is $4 \times 3 \times 10 \times 247 \times 10 = 296,400$.

Table 3 shows average dfb and number of wins results, averaged over all experimental scenarios and sorted by increasing dfb values, i.e., from best to worst. In spite of the averaging over all problem instances, the trends are clear. All four MCT algorithms perform best, followed closely behind by the UD, and then the LW algorithms. The random algorithms perform significantly worse. Regarding these algorithms, one can note that, expectedly, biasing the probability that a processor P_q is picked by w_q is a good idea

Table 3: Results over all problem instances

Algorithm	Average dfb	#wins
EMCT	4.77	80320
EMCT*	4.81	78947
MCT	5.35	73946
MCT*	5.46	70952
UD*	7.06	42578
UD	8.09	31120
LW*	11.15	28802
LW	12.74	19529
RANDOM1W	28.42	259
RANDOM2W	28.43	301
RANDOM4W	28.51	278
RANDOM3W	31.49	188
RANDOM3	44.01	87
RANDOM4	47.33	88
RANDOM1	47.44	36
RANDOM2	47.53	73
RANDOM	47.87	45

(i.e., RANDOMxW always outperforms RANDOMx). The other differences in the definitions of the random algorithms do not lead to significant performance differences. On average on all problem instances, EMCT algorithms have makespans 10% smaller than the MCT algorithms, which shows that taking into account the probability of state changes does lead to improved performance.

To provide more insight than the overall averages shown in Table 3, Figure 1 plots dfb results averaged for distinct w_{min} values, shown on the x-axis. We present only results for the four MCT heuristics and for those heuristics that do account for network contention (i.e., with a $*$), and leave out the random heuristics. Note that increasing w_{min} amounts to scaling the unit time, meaning that availability state transitions occur more often during the execution of a task. In other words, the right hand side of the x-axis in Figure 1 corresponds to more difficult problem instances. Indeed, the larger w_{min}, the higher the probability that a task's processor experiences a state transition. Therefore, as w_{min} increases, it becomes increasingly important to estimate the negative impacts of the $DOWN$ and $RECLAIMED$ states: the most powerful processor may

Table 4: Results for contention-prone experiments

Communication times $\times 5$

Algorithm	Average dfb
EMCT*	3.87
MCT*	4.10
UD*	5.23
EMCT	6.13
UD	6.42
MCT	7.70
LW*	8.76
LW	10.11

Communication times $\times 10$

Algorithm	Average dfb
UD*	2.76
UD	3.20
EMCT*	3.66
LW*	4.02
MCT*	4.22
LW	4.46
EMCT	8.02
MCT	15.50

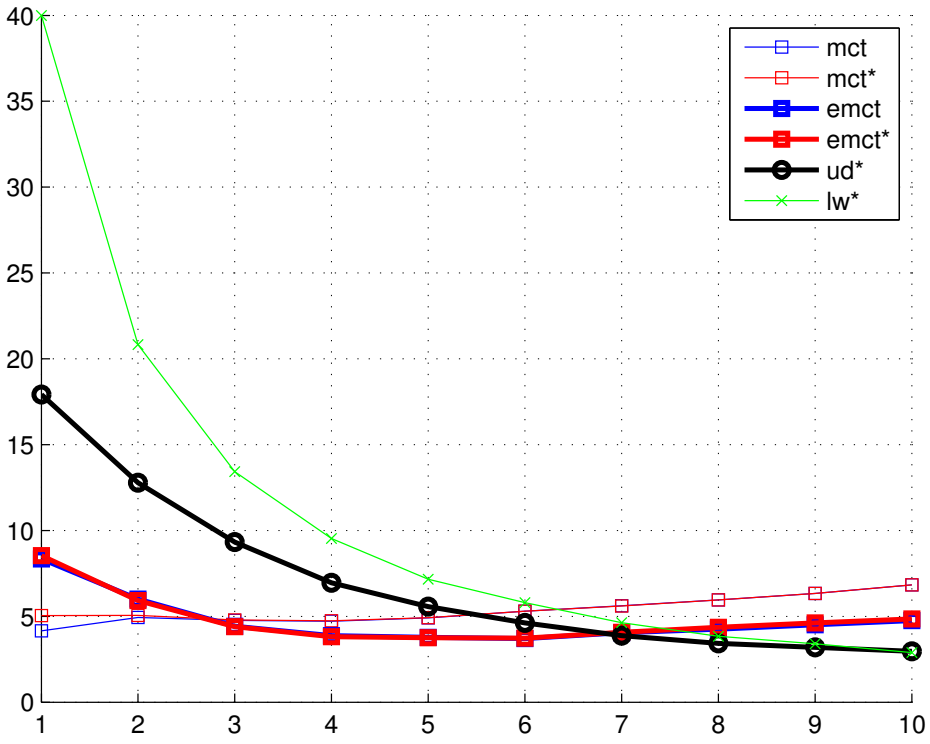

Figure 1: Averaged dfb results vs. w_{min}.

no longer be the best choice if it has a higher probability of going into the states $RECLAIMED$ or $DOWN$. The two EMCT algorithms take into account the probability that a processor enters the $RECLAIMED$ state. We see that they overtake the MCT algorithms when w_{min} becomes larger than 3. The UD and LW algorithms also take into account the probability that a processor goes $DOWN$. UD heuristics consistently outperform their LW counterparts. Also, UD (slightly) overtakes EMCT as soon as $w_{min} = 7$. We conclude that when the probability of state transitions rises one must use heuristics that explicitly take into account that processors can go in the states $RECLAIMED$ and $DOWN$.

In our results, we do not see much difference between the original versions of the heuristics and the versions that try to account for network contention, i.e., the heuristics that have a '$*$' in their names. Part of the reason may be that, as stated in Section 6.2.1, the correcting factor used to account for contention is a very coarse approximation. However, our experimental scenarios correspond to compute-intensive executions, meaning that processors typically spend much more time computing than communicating. We ran a set of experiments for $n = 20$, $n_{\text{com}} = 5$, and $w_{min} = 1$, but with $T_{\text{data}} = 5w_{min}$ and $T_{\text{prog}} = 25w_{min}$, i.e., with communication times 5 times larger than those in our base set of experimental scenarios. Results averaged over 100 such "contention-prone" experimental scenarios (each of which is ran for 10 trials) are shown in the left-hand side of Table 4. The right-hand side shows similar results for communication that are 10 times larger than those in our base set of scenarios. These results confirm that, as the scenario becomes more communication intensive, those algorithms that account for network contention outperform their counterparts.

7.2 TIGHTLY-COUPLED model

For the TIGHTLY-COUPLED model, 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 $w_{min} \in \{1, 2, \ldots, 10\}$.

7.2.1 Results for $m = 5$

Table 5: Results with $m = 5$ tasks.

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

Table 5 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 processor 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 processor 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 6: 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

7.2.2 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 6. 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 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 6 that Y-IE and P-IE experience exactly the same number of failures (141 failures out of the 3,000 instances).

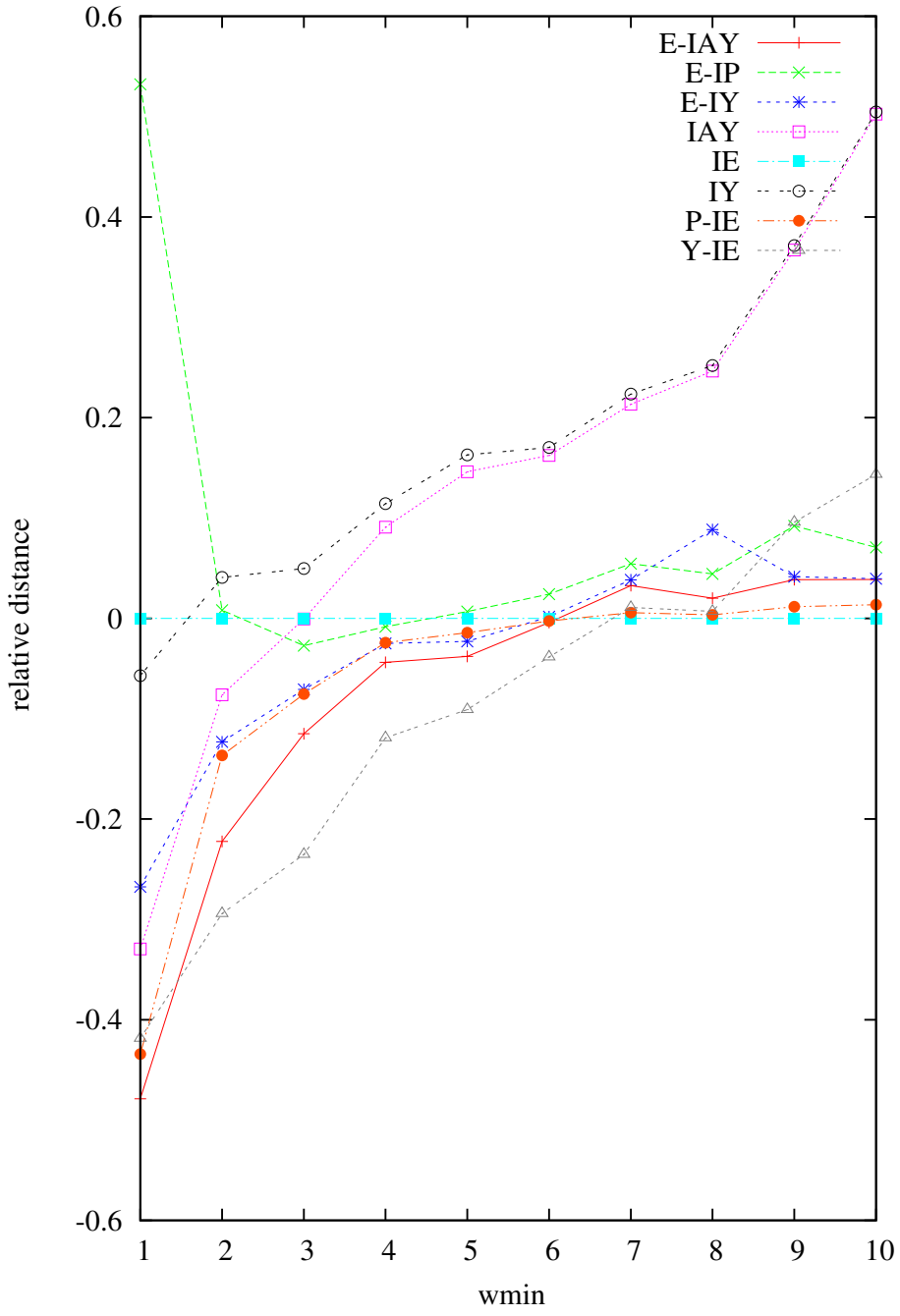

Figure 2: %diff for $m = 10$ tasks vs. w_{min}.

8 Conclusion

In this paper, we have studied the problem of scheduling master-worker iterative applications on volatile platforms in which processors can experience failures or be temporarily reclaimed by their owners. A unique aspect of our work is that we model the fact that communication between the master and the workers is subject to a bandwidth constraint, e.g., due to the limited capacity of the master's network card. Another key contribution is that we deal with both independent and tightly-coupled tasks, thereby covering a very large class of scientific applications.

In this context, we have made a theoretical contribution by characterizing the computational complexity of the offline problems (INDEPENDENT and TIGHTLY-COUPLED models), which turns out to be both NP-hard. Interestingly, without any bandwidth constraint, the INDEPENDENT problem becomes solvable in polynomial time. By assuming a Markov model of processor availability, we have been able to derive a closed-form formula for the expectation of the time needed by a volatile worker to complete a set of tasks in the INDEPENDENT model, and to provide an analytical approximation for the TIGHTLY-COUPLED model. We have then proposed several online scheduling heuristics. For the INDEPENDENT model, heuristics EMCT, EMCT*, UD, UD* use the evaluation of the expected computation time to make scheduling decisions. In the TIGHTLY-COUPLED model, all heuristics (expect random ones) also use this evaluation. Some heuristics also use a contention-correcting factor as a way to account for the constraint on the master's bandwidth (namely EMCT*, LW*, UD*). The evaluation of the heuristics in simulation has led to the following conclusions:

- Our failure-aware heuristics deliver better performance than classical heuristics a ssoon as the probability that a task is subject to processor state transitions becomes non negligible;
- Our contention-correcting factor improves performance on contention-prone platforms, and does not degrade performance otherwise;
- For the INDEPENDENT model, the EMCT* heuristic delivers overall good performance, leading to a 10% reduction over makespans achieved by MCT, the optimal algorithm for the contention-free offline case. However, EMCT* is outperformed by UD* in scenarios that exhibit very large state transition probabilities when compared to task duration, or a network with heavy contention.
- For the TIGHTLY-COUPLED model, the 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 the best approach is to use a proactive heuristic that selects processors to maximize expected execution time and changes configuration based on yield or probability of success.

Altogether, computing the expectation of the time needed by a set of workers to complete a given workload —either expressed as an exact closed form or an approximated expression— has enabled us to design heuristics that significantly outperform classical heuristics.

Acknowledgments.

Yves Robert is with Institut Universitaire de France. This work was supported in part by the ANR *RESCUE* project. A preliminary version of this paper (for the INDEPENDENT model) appears in IPDPS 2011. We would like to thank the reviewers for their comments and suggestions, which greatly helped improve the final version of the paper.

References

[1] 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*, pages 56–63, 2006.

[2] 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.

[3] E. Argollo, D. Rexachs, F. G. Tinetti, and E. Luque. Efficient execution of scientific computation on geographically distributed clusters. In *Proc. 7th Int. Conf. Applied Parallel Computing*, PARA'04, pages 691–698. Springer-Verlag, 2006.

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

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

[6] BOINC: Berkeley Open Infrastructure for Network Computing. `http://boinc.berkeley.edu`.

[7] 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.

[8] D. Buntinas, C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. *FGCS*, 24(1):73–84, 2008.

[9] C. Byrne. A unified treatment of some iterative algorithms in signal processing and image reconstruction. *Inverse Problems*, 20(1):103, 2004.

[10] 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*, 23:616–622, 2007.

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

[12] Y. Censor, D. Gordon, and R. Gordon. Component averaging: An efficient iterative parallel algorithm for large and sparse unstructured problems. *Parallel Computing*, 27(6):777 – 808, 2001.

[13] 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*, 27(5):606 – 613, 2011.

[14] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199 – 222, 1969.

[15] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *J. Par. and Distr. Comp.*, 63:597–610, 2003.

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

[17] J. R. de Souza, E. Argollo, A. Duarte, D. Rexachs, and E. Luque. Fault tolerant master-worker over a multi-cluster architecture. In *Proc. of ParCo 2005*, pages 465–472. NIC Series, Vol. 33, 2006.

[18] F. Dufossé. *Scheduling for Reliability: Complexity and Algorithms*. PhD thesis, Ecole Normale Supérieure de Lyon, 2011. Available on line at `http://graal.ens-lyon.fr/~fdufosse/these-Dufosse.pdf`.

[19] 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.

[20] T. Estrada, O. Fuentes, and M. Taufer. A distributed evolutionary method to design scheduling policies for volunteer computing. *SIGMETRICS Perf. Eval. Rev.*, 36(3):40–49, 2008.

[21] G. E. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proc. 7th EuroPVM/MPI*, pages 346–353. Springer-Verlag, 2000.

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

[23] W. Gropp. MPICH2: A New Start for MPI Implementations. In *PVM/MPI*, page 7, 2002.

[24] R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30:68–74, 1997.

[25] J. Håstad. Some optimal inapproximability results. In *STOC '97*, pages 1–10. ACM, 1997.

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

[27] A. Heddaya and K. Park. Mapping parallel iterative algorithms onto workstation networks. In *HPDC'94*, pages 211 –218, 1994.

[28] E. Heien, D. Anderson, and K. Hagihara. Computing Low Latency Batches with Unreliable Workers in Volunteer Computing Environments. *Journal of Grid Computing*, 7(4):501–518, 2009.

[29] B. Hong and V. K. Prasanna. Adaptive allocation of independent tasks to maximize throughput. *IEEE TPDS*, 18(10):1420–1435, 2007.

[30] J.-H. Hyun. An Effective Scheduling Method for More Reliable Execution on Desktop Grids. In *Proc. of the 12th IEEE International Conference on High Performance Computing and Communications*, 2010.

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

[32] 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.

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

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

[35] T. Leblanc, R. Anand, E. Gabriel, and J. Subhlok. VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes. In *Proc. of EuroPVM/MPI 2009*, pages 124–133. Springer-Verlag, 2009.

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

[37] 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.

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

[39] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Prediction of Resource Availability in Fine-Grained Cycle Sharing Systems Empirical Evaluation. *Journal of Grid Computing*, 5(2):173–195, 2007.

[40] G. L. G. Sleijpen and H. A. V. d. Vorst. A jacobi-davidson iteration method for linear eigenvalue problems. *SIAM Review*, 42(2):pp. 267–293, 2000.

[41] P. Stelling, C. DeMatteis, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2):117–128, 1999.

[42] J. C. Strikwerda. A probabilistic analysis of asynchronous iteration. *Linear Algebra and its Applications*, 349(1-3):125 – 154, 2002.

[43] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, 13(3):260–274, 2002.

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

[45] B. Uçar and C. Aykanat. Partitioning sparse matrices for parallel preconditioned iterative methods. *SIAM Journal on Scientific Computing*, 29(4):1683–1709, 2007.

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

[47] 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.

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

[49] 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.