HAL
open science

Scheduling Parallel Iterative Applications on Volatile Resources

Henri Casanova, Fanny Dufossé, Yves Robert, Frédéric Vivien

To cite this version:

Henri Casanova, Fanny Dufossé, Yves Robert, Frédéric Vivien. Scheduling Parallel Iterative Applications on Volatile Resources. [Research Report] 2010, pp.19. inria-00524784v1

HAL Id: inria-00524784

https://inria.hal.science/inria-00524784v1

Submitted on 8 Oct 2010 (v1), last revised 19 Oct 2010 (v2)

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

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

Scheduling Parallel Iterative Applications on Volatile Resources

Henry Casanova, Fanny Dufossé, Yves Robert and Frédéric Vivien
Ecole Normale Supérieure de Lyon and University of Hawai'i at Mānoa

October 2010

LIP Research Report RR-2008-31

Abstract

In this paper we study the execution of iterative applications on volatile processors such as those found on desktop grids. We develop master-worker scheduling schemes that attempt to achieve good trade-offs between worker speed and worker availability. 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 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 makes the design of scheduling algorithms more difficult. 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 problem in its off-line version, i.e., when processor availability behaviors are known in advance. Even with this knowledge, the problem is NP-hard, and cannot be approximated within a factor 8/7. Our second contribution is a closed-form formula for the expectation of the time needed by a worker to complete a set of tasks. This formula relies on a Markovian assumption for the temporal availability of processors, and is at the heart of some heuristics that aim at favoring "reliable" processors in a sensible manner. Our third contribution is a set of heuristics, which we evaluate in simulation. Our results provide insightful guidance to selecting the best strategy as a function of processor state availability versus average task duration.

1 Introduction

We study the problem of efficiently executing parallel applications on platforms that comprise volatile resources. More specifically we focus on iterative applications implemented using the master-worker paradigm. The master coordinates the computation of each iteration as the execution of a fixed number of independent tasks. A synchronization of all tasks occurs at the end of each iteration. 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., [1, 2, 3, 4]). 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 [5] and desktop grids [6]. In these platforms processors are contributed by resource owners that 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 [4], which is also the context of this work. 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 [7]. Unfortunately, the memoryless assumption is known to not hold in practice. Several authors have reported that the durations of availability intervals in production desktop grids are not sampled from exponential distributions [8, 9, 10]. 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*. Note that some authors have attempted to model processor availabilities using (non-memoryless) semi-Markov processes [11]. Faced with the lack of a good model for transitions between the *UP*, *DOWN*, and *RECLAIMED* states, and not knowing whether such a model would be tractable or not, for now we opt for the Markovian model. 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. The 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.

A unique aspect of this work is that we account for network bandwidth constraints for communication 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 [12], applies to concurrent data transfers implemented with multi-threading. One alternative is to simply not consider these constraints. However, 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. However, 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 an extremely low communication-to-computation ratio. 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 need 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 problem in its off-line version, i.e., when processor availability behaviors are known in advance. Even with this knowledge, the problem is NP-hard, and cannot be approximated within a factor 8/7. Next, relying on the Markov assumption for processor availability, we provide a closed-form formula for the expectation of the time needed by a worker to complete a set of tasks. This formula is 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 off-line 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 processor to complete a given workload. Heuristics for the on-line 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

There is a large literature on scheduling master-worker applications, or applications that consist of a sequence of iterations where each iteration can be executed in master-worker fashion [13, 14, 15]. In this work we focus on executions on volatile resources, such as desktop resources. The volatility of desktop or other resources is well documented and characterizations have been proposed [8, 9, 10]. Several authors have studied the master-worker (or "bag-of-tasks")

scheduling problem in the face of such volatility in the context of desktop grid computing, either at an Internet-wide scale or with an Enterprise [16, 17, 18, 19, 7, 20, 21, 22]. Most of these works propose simple greedy scheduling algorithms that rely on mechanisms to pick processors according to some criteria. These processor selection criteria include static ones (e.g., processor clock-rates or benchmark results), simple ones based on past host behavior [16, 18, 20], and more sophisticated ones based on statistical analysis of past host availability [21, 22, 19, 7]. In a global setting, the work in [17] includes time-zone as a criterion for processor selection. These criteria are used to rank processors, but also to exclude them from consideration [16, 18]. The work in [7] is particularly related to our own in that it uses a Markov model of processor availability (but not accounting for preemption).

Most of these works also advocate for task replication as a way to cope with volatile resources. Expectedly, injecting task replicas is sensible toward the end of application execution. Given the number of possible variants of scheduling algorithms, in [20] the authors propose a method to automatically instantiate the parameters that together define the behavior of a scheduling algorithm. Works published in this area are of a pragmatic nature, and few theoretical results have been sought or obtained (one exception is the work in [23]).

A key difference between our work and all the above is that we seek to develop scheduling algorithms that explicitly manage for the master's bandwidth. Limited master bandwidth is a known issue for desktop grid computing [24, 25, 26] and must therefore be addressed even though it complexifies the scheduling problem. To the best of our knowledge no previous work has made such an attempt.

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.

3.1 Application Model

We target an iterative application in which iterations entail the execution of a fixed number m of same-size independent 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.

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 that decides which processor computes which tasks and when, executes on a host that is always *UP* (otherwise a simple redundancy mechanism such as primary back-up [27] 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 [28]. The availabilities of processors evolve independently. For a processor 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. Indeed, 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. Note that 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 [12]. 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 [29]. 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 $n_{com} = \mathrm{BW}/\mathrm{bw}$ 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

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

Let P_q be a processor engaged in communication at time t, for receiving either the program or some data. In both cases, it does this with bandwidth bw. Hence the time for a worker to receive the program is $T_{\mathrm{prog}} = V_{\mathrm{prog}}/\mathrm{bw}$, and the time to receive the data is $T_{\mathrm{data}} = V_{\mathrm{data}}/\mathrm{bw}$.

3.3 Scheduling Model

Let $\mathit{config}(t)$ denote the set of processors enrolled for computing the m application tasks in an iteration, or *configuration*, at time t. 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 due to volatile availability. 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.

If a processor becomes *DOWN* at time t, the scheduler may simply use the remaining *UP* processors in *config*(t) to complete the iteration, or enroll a new processor. Even if all processors in *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 *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 results obtained for previously completed tasks are not lost because already sent back to the master. 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.

3.4 Problem Statement

The scheduling problem we address in this work is that of maximizing 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 off-line 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 off-line complexity of the problem. 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. We also identify a polynomial instance with $n_{com} = +\infty$, which highlights the impact of communication contention.

For the off-line 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 $\mathcal{S}_{q''}[j] = \mathcal{S}_q[i]$. 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 Off-Line denote the problem of minimizing the time to complete the first iteration, with same-speed processors:

Theorem 1. *Problem* Off-Line *is NP-hard.*

Proof. Consider the associated decision problem: given a number m of tasks, of computing cost w and communication cost T_{data}, a program of communication cost T_{prog}, and a platform

of p processors, with availability vectors $\mathcal{S}_q$, a bound n_{com} on the number of simultaneous communications, and a time limit N, does there exist a schedule that executes one iteration in time less than N? The problem is obviously in NP: given a set of tasks, a platform, a time limit and a schedule (of communications and computations), we can check the schedule and compute its completion time with polynomial complexity.

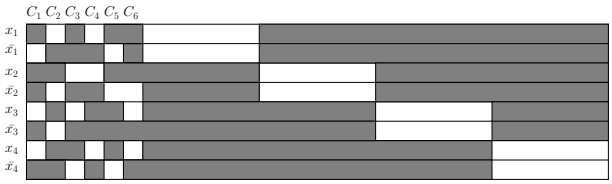

Figure 1: Proof of NP-completeness of Off-Line.

The proof follows from a reduction from 3SAT. Let $\mathcal{I}_1$ be an instance of 3SAT : given a set $U = \{x_1, ..., x_n\}$ of variables and a collection $\{C_1, ..., C_m\}$ of clauses, does there exist a truth assignment of U? We suppose that each variable is present in at least one clause.

We construct the following instance $\mathcal{I}_2$ of the Off-Line problem with m tasks and $p = 2n$ processors: $n_{com} = 1$, $T_{\text{prog}} = m$, $T_{\text{data}} = 0$, $w_i = w = 1$, $N = m(n+1)$ and $\forall i \in [1, n], \forall j \in [1, m]$, 1) if $x_i \in C_j$ then $\mathcal{S}_{2i-1}[j] = u$ else $\mathcal{S}_{2i-1}[j] = r$, 2) if $\bar{x}_i \in C_j$ then $\mathcal{S}_{2i}[j] = u$ else $\mathcal{S}_{2i}[j] = r$, 3) $\mathcal{S}_{2i}[mi+j] = \mathcal{S}_{2i-1}[mi+j] = u$ and 4) $\forall k \in [1, n], i \neq k$, $\mathcal{S}_{2k-1}[mi+j] = \mathcal{S}_{2k}[mi+j] = r$. The size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. Figure 1 illustrates this construction for $\mathcal{I}_1 = (\bar{x_1} \vee x_3 \vee x_4) \wedge (x_1 \vee \bar{x_2} \vee \bar{x_3}) \wedge (x_2 \vee x_3 \vee \bar{x_4}) \wedge (x_1 \vee x_2 \vee x_4) \wedge (\bar{x_1} \vee \bar{x_2} \vee \bar{x_4}) \wedge (\bar{x_2} \vee x_3 \vee x_4)$.

Suppose that $\mathcal{I}_1$ has a solution A with, for all $j \in [1, n]$, $x_j = A[j]$. For any $i \in [1, m]$, there exists at least one true literal of A in C_i. We pick one arbitrarily. Let x_j be the associated variable. Then during time-slot i, if $A[j] = 1$, processor P_{2i-1} will download (a fraction of) the program, while if $A[j] = 0$, processor P_{2i} will download it. During this time-slot, no other processor communicates with the master. Then, for all $\forall i \in [1, n]$:

- if $A[j] = 1$, between $mi + 1$ and $m(i + 1)$, P_{2i-1} completes the reception of the program and then executes as many tasks as possible, P_{2i} stays idle
- if $A[j] = 0$, then P_{2i-1} is idle and P_{2i} completes downloading its copy of the program and computes as many tasks as possible.

For all $i \in [1, 2n]$, let L_i be the number of communication time-slots for processor P_i between time-slots 1 and m. By the choice of the processor receiving the program at any time-slot $t \in [1, m]$, if $A[i] = 0$, then $L_{2i-1} = 0$, else $L_{2i} = 0$. Let, for all $i \in [1, n]$, $p(i) = 2i - A[i]$. Then, for all $i \in [1, n]$, between time $mi+1$ and $m(i+1)$, $P_{p(i)}$ is available and $P_{p(i)+2A[i]-1}$ is idle. $P_{p(i)}$ completes receiving its program at the latest at time $mi + T_{prog} - L_{p(i)} = m(i + 1) - L_{p(i)}$ and can execute $L_{p(i)}$ tasks before being reclaimed. Overall, the processors execute $X = \sum_{i=1}^{n} L_{p(i)}$ tasks. For any $j \in [1, m]$, by construction there is exactly one processor downloading the program during time-slot j. Consequently, $X = m$, and thus $\mathcal{I}_2$ has a solution.

Suppose now that $\mathcal{I}_2$ has a solution. As $n_{com} = 1$, for all $i \in [1, n]$, processors P_{2i-1} and P_{2i} receive the program during at m time slots before time m. After time m, processors P_{2i-1} and P_{2i} are only available between time $mi + 1$ and $m(i + 1)$. Then, at time N, the sum S of the time-slots spent in receiving the program on processors P_{2i-1} and P_{2i} is $S \leq 2m$. This means that at most one of these processors can execute tasks before time N. Among P_{2i-1} and P_{2i}, let $p(i)$ be the processor computing at least one task. If neither P_{2i-1} nor P_{2i} computes any task, let $p(i) = 2i$. Let A be an array of size n such that if $p(i) = 2i - 1$, then $A[i] = 1$ else $A[i] = 0$. We will prove that all clauses of $\mathcal{I}_1$ are satisfied with this assignment. Without loss of generality we assume that no communication is made to a processor that does not execute any task. Suppose

that for $i \in [1, m]$, a processor P_j with $j = p(k)$ receives a part of the program during time-slot i. Then, by definition of the function p, either $A[k] = 1$ and $x_k \in C_i$, or $A[k] = 0$ and $\bar{x_k} \in C_i$. This means that assignment A satisfies clause C_i. Let X be the number of true clauses with assignment A. For all $i \in [1, 2n]$, we define L_i as the number of communication time-slots for processor P_i between times 1 and m. Then, $X \geq \sum_{j=1}^{n} L_{p(i)}$. In addition, processor $P_{p(i)}$ completes the reception of the program at the latest at time $m(i+1) - L_i$, and then computes at most L_i tasks before being reclaimed at time $m(i+1)$. Overall, the processors compute m tasks. Then, $\sum_{j=1}^{n} L_{p(i)} \geq m$, and $X \geq m$. Consequently, all clauses are satisfied by A, i.e., $\mathcal{I}_1$ has a solution, which concludes the proof. □

Proposition 1. *Problem* Off-Line *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 $\epsilon > 0$ [30]. By construction of the proof of Theorem 1, problem Off-Line cannot be approximated with the same value. □

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

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

Proof. We send the program to processors as soon as possible, at the beginning of the execution. Then, task by task, we greedily assign the next task to the processor that can terminate its execution the soonest; this is the classical MCT (Minimum Completion Time) strategy, whose complexity is $m \times p$.

To show that this strategy is optimal, let S_1 be an optimal schedule, and let S_2 the MCT schedule. Let T_1 and T_2 be the associated completion times. We aim at proving that $T_2 = T_1$. We first modify the schedule S_1 as follows. Suppose that processor P_q begins a computation or a communication at time t, and that it is available but idle during time-slot $t - 1$. Then, we can shift forward the operation and execute it at time $t - 1$ without breaking any rules and without increasing the completion time of the current iteration. We repeat this modification as many times as possible, and finally obtain a schedule S_1' with completion time $T_1' = T_1$. Assume now that P_q executes i tasks under schedule S_1' and j under S_2. The first $\min\{i, j\}$ tasks are executed at the same time by S_1' and by S_2. Suppose that $T_2 > T_1$. Consider S_2 right before the allocation of the first task whose completion time is $t > T_1$. At this time, at least one processor P_{q_0} has strictly fewer tasks in S_2 than in S_1'. We can thus allocate a task to P_{q_0} with completion time $t \leq T_1$. The MCT schedule should have chosen the latter allocation, and we obtain a contradiction. The MCT schedule S_2 is thus optimal. □

The MCT algorithm is not optimal if $n_{com} < +\infty$. Consider an instance with $T_{\text{prog}} = T_{\text{data}} = 2$, two tasks ($m = 2$) and two identical processors ($p = 2$, $w_q = w = 2$). Suppose that $n_{com} = 1$, and that $\mathcal{S}_1 = [u, u, u, u, u, u, r, r, r]$ and $\mathcal{S}_2 = [r, u, u, u, u, u, u, u, u]$. The optimal schedule computes both tasks in time 9 one waits for one time-slot and then sends the program and data to P_2. However, MCT executes the first task on P_1, and is thus not optimal.

5 Computing the expectation of a workload

In this section, we first introduce a Markov model for processor availability, and then show how to compute the expected execution time of a processor to complete a given workload.

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 $W-1$ another 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, 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 2. $E^{(q)}(W) = W + (W-1) \times \frac{P_{u,r}^{(q)} P_{r,u}^{(q)}}{1 - P_{r,r}^{(q)}} \times \frac{1}{P_{u,u}^{(q)}(1 - P_{r,r}^{(q)}) + P_{u,r}^{(q)} P_{r,u}^{(q)}}$.

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

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

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 remarking that:
$E^{(q)}(W) = 1 + (W-1) \times E^{(q)}(up)$. □

6 On-line heuristics

6.1 Rationale

In this section, we propose heuristics to address the on-line version of the problem. Conceptually, we can envision 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*.

Dynamic heuristics that may change configuration on the fly, 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 would allow for the possibility of aggressively terminating ongoing tasks, at the risk for an iteration to never complete.

In our model, the dynamic strategy is the most appealing. Since tasks are executed one by one and independently on each processor, using a passive approach by which all m tasks are assigned once and for all without possible reassignment does not make sense. A proactive strategy would have little impact on the time to complete the iteration unless the last tasks are assigned to slow processors. In this case, these tasks should be terminated and assigned to faster processors, which could have significant benefit when m is small. A simpler, and popular, solution is to use only dynamic strategies but to *replicate* these last tasks on one or more hosts in the *UP* state, canceling all remaining replicas when one of them completes. 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, task replication is more beneficial when m is small.

In all the heuristics described hereafter, 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 [16] 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.

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 is due to a Markov process, as discussed in Section 5.

6.2 Random heuristics

The heuristics described in this section use randomness to select, among the processors that are in the *UP* state, which one will execute the next task. The simplest one, 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 to 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.

6.3 Greedy heuristics

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

6.3.1 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 appraoch, 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 its number of already assigned tasks (out of the $m - m'$ tasks), and let $CT(P_q, n_q)$ be the estimation of its completion time:

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

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

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 $\left\lceil \frac{n_{\text{active}}}{n_{com}} \right\rceil T_{\text{data}}$, 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:

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

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 2 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.

6.3.2 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*:

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

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.

6.3.3 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_{q0} 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:

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

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

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://navet.ics.hawaii.edu/~casanova/software/cp_simulator.tgz`). 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.

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 $P_{x,x}^{(q)}$ value (for $x = u, r, d$). We then set $P_{x,y}^{(q)}$ to $0.5 \times (1 - P_{x,x}^{(q)})$, 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_{com}, w_{min}) given the values shown in Table 1. We must emphasize that our goal here is *not* to instantiate a representative model for

Table 1: Parameter values for Markov experiments.

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

Table 2: 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

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.

For each possible instantiation of the parameters in Table 1, 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 2 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 (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 2, Figure 2 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

Table 3: Results for contention-prone experiments

Communication 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 ×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

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 2 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 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 that try to account for network contention, i.e., the heuristics thave have a $*$ in their names. Part of the reason may be that, as stated in Section 6.3.1, 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 $n = 20$, $n_{com} = 5$, 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 3. 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.

8 Conclusion

In this paper we have studied the problem of scheduling master-worker iterative applications on volatile platforms in which hosts 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. In this context we have made a theoretical contribution by characterizing the computational complexity of the off-line problem, which turns out to be NP-hard. Interestingly, without any bandwidth constraint, the problem becomes solvable in

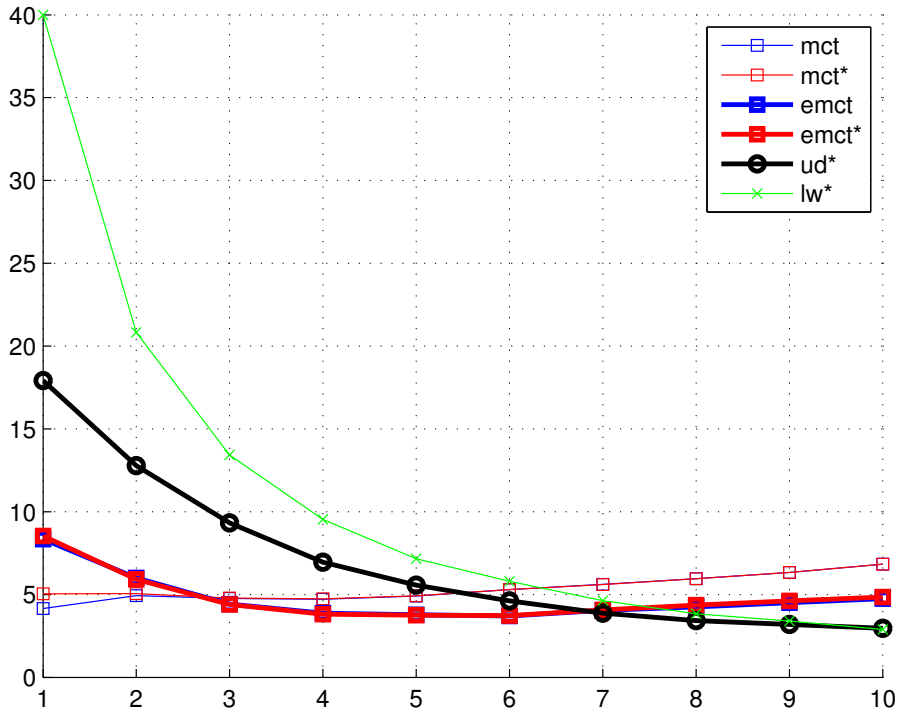

Figure 2: Averaged dbf results vs. w_{min}.

polynomial time. We have then proposed several online scheduling heuristics. 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. Some of our heuristics use this expectation for making scheduling decision (namely EMCT, EMCT*, UD, UD*). 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 our heuristics in simulation has led to the following conclusions:

- Our failure-aware heuristics deliver better performance than classical heuristics when 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;
- Our EMCT* heuristic delivers overall good performance, leading to a 10% reduction over the makespans achieved by MCT, the optimal algorithm for the contention-free offline case;
- EMCT* is outperformed by UD* in scenarios that exhibit from very large state transition probabilities when compare to task duraction, or a highly contented network.

The next step in this research is to challenge the Markov assumption for processor availability. As explained in Section 1, processor availability in desktop grid platforms is not Markovian. We see two possible avenues of research. First, we could rely on those stochastic models of processor availability that are available [8, 9, 10, 11], and evolve our algorithmic techniques, if at all possible, to account for those models. Second, given that no consensus has emerged regarding the correct models of processor availability (and that perhaps there is none), we could embark on a purey empirical study based on availability traces such as those available in the Failure

Trace Archive [31].

Acknowledgment: F. Dufossé, Y. Robert and F. Vivien are with the Université de Lyon. Y. Robert is with the Institut Universitaire de France. F. Vivien is with INRIA. This work was supported in part by the ANR *StochaGrid* project.

References

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

[2] 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*. NIC Series, Vol. 33, 2006, pp. 465–472.

[3] 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*. Springer-Verlag, 2009, pp. 124–133.

[4] 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*, vol. 24, no. 1, pp. 73–84, 2008.

[5] "BOINC: Berkeley Open Infrastructure for Network Computing," http://boinc.berkeley.edu.

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

[7] E. Byun, S. Choi, M. Baik, J. Gil, C. Park, and C. Hwang, "MJSA: Markov job scheduler based on availability in desktop grid computing environment," *FGCS*, vol. 23, no. 4, pp. 616–622, 2007.

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

[9] 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.

[10] 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.

[11] 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*, vol. 5, no. 2, pp. 173–195, 2007.

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

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

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

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

[16] D. Kondo, A. Chien, and H. Casanova, "Resource Management for Rapid Application Turnaround on Enterprise Desktop Grids," in *Proc. of SC'04*, 2004.

[17] 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.

[18] 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.

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

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

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

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

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

[24] 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.

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

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

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

[28] 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*, vol. 2, no. 2, pp. 117–128, 1999.

[29] W. Gropp, "MPICH2: A New Start for MPI Implementations," in *PVM/MPI*, 2002, p. 7.

[30] J. Håstad, "Some optimal inapproximability results," in *STOC '97*. ACM, 1997, pp. 1–10.

[31] D. Kondo, B. Javadi, A. Iosup, and D. Epema, "The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems," in *Proc. of CCGrid*, 2010.