Resilient application co-scheduling with processor redistribution

Anne Benoit, Loïc Pottier, Yves Robert

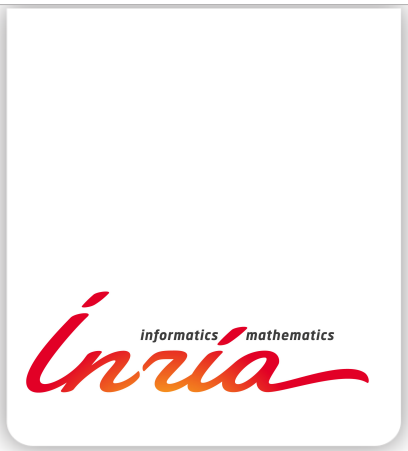
RESEARCH REPORT

N° 8795

October 2015

Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--8795--FR+ENG



Resilient application co-scheduling with processor redistribution

Anne Benoit*, Loïc Pottier*, Yves Robert*†

Project-Team ROMA

Research Report n° 8795 — October 2015 — 30 pages

Abstract: Recently, the benefits of co-scheduling several applications have been demonstrated in a fault-free context, both in terms of performance and energy savings. However, large-scale computer systems are confronted to frequent failures, and resilience techniques must be employed to ensure the completion of large applications. Indeed, failures may create severe imbalance between applications, and significantly degrade performance. In this paper, we propose to redistribute the resources assigned to each application upon the striking of failures, in order to minimize the expected completion time of a set of co-scheduled applications. First we introduce a formal model and establish complexity results. When no redistribution is allowed, we can minimize the expected completion time in polynomial time, while the problem becomes NP-complete with redistributions, even in a fault-free context. Therefore, we design polynomial-time heuristics that perform redistributions and account for processor failures. A fault simulator is used to perform extensive simulations that demonstrate the usefulness of redistribution and the performance of the proposed heuristics.

Key-words: Resilience; co-scheduling; redistribution; complexity results; heuristics; simulations.

* Ecole Normale Supérieure de Lyon & Inria, France
† University of Tennessee Knoxville, USA

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Ordonnancement d'applications concurrentes dans un contexte résilient avec redistribution de processeurs

Résumé : Récemment, les bénéfices de l'ordonnancement concurrent de plusieurs application ont été démontrés dans un contexte sans fautes, à la fois en terme de performance et de consommation énergétique. Cependant, les plateformes distribuées à grande échelle sont fréquemment confrontées à des pannes, et des techniques de résilience doivent être employées. En effet, les pannes peuvent créer des déséquilibres importants entre applications et ainsi dégrader les performances. Dans ce papier, nous proposons de redistribuer les ressources allouées à chaque application à chaque fois qu'une faute survient, dans le but de minimiser le temps de complétion d'un ensemble de tâches concurrentes. Dans un premier temps, nous introduisons le modèle formel et nous présentons des résultats de complexité. Quand aucune redistribution n'est permise, nous pouvons minimiser l'espérance du temps de complétion en temps polynomial, tandis que le problème devient NP-complet lorsque les redistributions sont permises, même dans un contexte sans fautes. Par conséquent, nous proposons des heuristiques polynomiales effectuant des redistributions, et prenant en compte les pannes des processeurs. Un simulateur de fautes est utilisé pour réaliser un nombre important de simulations qui démontrent l'utilité de la redistribution ainsi que les performances des heuristiques proposées.

Mots-clés : Resilience; ordonnancement concurrent; redistribution; résultats de complexité; heuristiques; simulations.

1 Introduction

With the advent of multicore platforms, HPC applications can be efficiently parallelized on a flexible number of processors. Usually, a speedup profile determines the performance of the application for a given number of processors. For instance, the applications in [1] were executed on a platform with up to 256 cores, and the corresponding execution times were reported. A perfectly parallel application has an execution time t_{seq}/p, where t_{seq} is the sequential execution time, and p is the number of processors. In practice, because of the overhead due to communications and to the inherently sequential fraction of the application, the parallel execution time is larger than t_{seq}/p. The speedup profile of the application is assumed to be known (or estimated) before execution, through benchmarking campaigns.

A simple scheduling strategy on HPC platforms is to execute each application in dedicated mode, assigning all resources to each application throughout its execution. However, it was shown recently that rather than using the whole platform to run one single application, both the platform and the users may benefit from *co-scheduling* several applications, hence minimizing the loss due to the fact that applications are not perfectly parallel. Sharing the platform between two applications leads to significant performance and energy savings [2], that are even more important when co-scheduling more than two applications simultaneously [3].

To the best of our knowledge, co-scheduling has been investigated so far only in the context of fault-free platforms. However, large-scale platforms are prone to failures. Indeed, for a platform with p processors, even if each node has an individual MTBF (Mean Time Between Failures) of 120 years, we expect a failure to strike every $120/p$ years, for instance every hour for a platform with $p = 10^6$ nodes. Failures are likely to destroy the load-balancing achieved by co-scheduling algorithms: if all applications were assigned resources by the co-scheduler so as to complete their execution approximately at the same time, the occurrence of a failure will significantly delay the completion time of the corresponding application. In turn, several failures may well create severe imbalance among the applications, thereby significantly degrading performance.

To cope with failures, the de-facto general-purpose error recovery technique in HPC is checkpoint and rollback recovery [4]. The idea consists in periodically saving the state of the application, so that when an error strikes, the application can be restored into one of its former states. The most widely used protocol is coordinated checkpointing, where all processes periodically stop computing and synchronize to write critical application data onto stable storage. The frequency at which checkpoints are taken should be carefully tuned, so that the overhead in a fault-free execution is not too important, but also so that the price to pay in case of failure remains reasonable. Young and Daly provide good approximations of the optimal checkpointing interval [5, 6].

This paper investigates co-scheduling on failure-prone platforms. Checkpointing helps to mitigate the impact of a failure on a given application, but it must be complemented by redistributions to re-balance the load among applications. Co-scheduling usually involves partitioning the applications into *packs*, and then scheduling each pack in sequence, as efficiently as possible. We focus on the second step, namely co-scheduling a given pack of applications that execute in parallel, and leave the partitioning for further work. This is because scheduling a given pack becomes a difficult endeavor with failures (and redistributions), while it was of linear complexity without failures. Given a pack, i.e., a set of parallel tasks that start execution simultaneously, there are two main opportunities for redistributing processors. First, when a task completes, the applications that are still running can claim its processors. Second, when a failure strikes a task, that task is slowed down. By adding more resources to it, we hope to reduce the overall completion time. However, we have to be careful, because each redistribution has a cost, which depends on the volume of data that is exchanged, and on the number of processors involved in

redistribution. In addition, adding processors to a task increases its probability to fail, so there is a trade-off to achieve in order to minimize the expected completion time of the pack.

The major contributions of this work are the following:

- the design of a detailed and comprehensive model for scheduling a pack of tasks on a failure-prone platform;
- the NP-completeness proof for the problem with redistributions;
- the design and assessment of several polynomial-time heuristics to deal with the general problem with failures and redistribution costs.

The rest of the paper is organized as follows. First, we discuss related work in Section 2. The model and the optimization problem are formally defined in Section 3. In Section 4 we expose the complexity results. We introduce some polynomial-time heuristics in Section 5, which are assessed through simulations using a fault generator in Section 6. Finally, we conclude and provide directions for future work in Section 7.

2 Related work

We first discuss related work on models for parallel tasks in Section 2.1, before surveying related work on resilience (Section 2.2). Finally, we discuss previous work on co-scheduling algorithms in Section 2.3.

2.1 Parallel task models

A parallel task is a task that may use several processors at the same time during its execution. In the literature, many parallel task models have been developed and several types of tasks have been defined.

In 1986, with the development of multiprocessor systems, Błażewicz et al. [7] have modeled the problem of scheduling a set of independent parallel tasks on identical processors. The number of processors assigned to each task was fixed during the execution. They showed that problem is NP-complete when the number of processors is not fixed. A task that has a fixed number of processors is called *rigid*.

In 1989, Du and Leung [8] have developed a model called the Parallel Task System, where a task is executed by one or more processors at the same time, but the number of processors assigned to one task cannot exceed a certain threshold. Contrarily to the Błażewicz's model, the number of processors is not fixed in advance, but once it is determined (between one and the threshold), it remains fixed during the execution. Such tasks are called *moldable*.

Finally, a *malleable* task can see its number of processors allocated vary during the execution. Błażewicz et al. [9] have designed approximation algorithms to solve the problem of scheduling independent malleable tasks. Malleable tasks are more flexible than rigid and moldable tasks, they can be implemented with data redistributions through processors (the technique used in this paper) or work stealing. In practice, changing the number of processors at runtime requires specific tools, frameworks and even dedicated programming languages like Cilk, which is using work stealing techniques [10].

Martín et al. [11] have developed an MPI extension, called Flex-MPI, which introduces malleability in MPI. Flex-MPI can achieve a load balancing among applications through a prediction model. The prediction model in Flex-MPI does not take into account resilience aspects.

One contribution of this work is to develop a complete model taking into account resilience aspects. We also provide heuristics able to re-assign processors to tasks that need them. We also show that the problem of finding a schedule that minimizes the execution times without redistribution costs and without failures is NP-complete (in the strong sense).

2.2 Resilience

One of the most used technique to handle fail-stop errors in HPC is checkpoint and rollback recovery [4]. The idea is to periodically save the system state, or the application memory footprint onto a stable storage. Then, after a downtime and a recovery time, the system can be restored into a former valid state (rollback step). Another technique to dealing with fail-stop errors is the process replication, which consists in replicating a process and even replicate communications. For instance, the project RedMPI [12] implements a process replication mechanism and quadruplicates each communication.

In this paper, we use a light-weight checkpointing protocol called the *double checkpointing algorithm* [13, 14]. This is an in-memory checkpointing protocol, which avoids the high overhead of disk checkpoints. Processors are paired: each processor has an associated processor called its *buddy processor*. When a processor stores its checkpoint file in its own memory, it also sends this file to its buddy, and the buddy does the same. Therefore, each processor stores two checkpoints, its own and that of its buddy. When a failure occurs, the faulty processor looses these two checkpoint files, and the buddy must re-send both checkpoints to the faulty node. If a second failure hits the buddy during this recovery period, we have a fatal failure and the system cannot be recovered.

2.3 Co-scheduling algorithms

This work provides an important extension to our previous work on co-schedules [2, 3], which already demonstrated that sharing the platform between two or more applications can lead to significant performance and energy savings. To the best of our knowledge, it is the first work to consider co-schedules and failures, and hence to use malleable tasks to allow redistributions of processors between applications.

However, we point out that co-scheduling with packs can be seen as the static counterpart of batch scheduling techniques, where jobs are dynamically partitioned into batches as they are submitted to the system (see [15] and the references therein). Batch scheduling is a complex online problem, where jobs have release times and deadlines, and when only partial information on the whole workload is known when taking scheduling decisions. On the contrary, co-scheduling applies to a set of tasks that are all ready for execution. In this paper, as already mentioned, we restrict to a single pack, because scheduling already becomes difficult for a single pack with failures and redistributions.

3 Framework

We consider a pack of n independent malleable tasks $\{T_1, \ldots, T_n\}$, and an execution platform with p identical processors subject to failures. The objective is to minimize the expected completion time of the last task. First, we define the fault model in Section 3.1. Then, we show how to compute the execution time of a task in Section 3.2, assuming that no redistribution has occurred. The redistribution mechanism and its associated cost are discussed in Section 3.3.

For convenience, a summary of notations is provided in Table 1.

3.1 Fault model

We consider fail-stop errors, which are detected instantaneously. To model the rate at which faults occur on one processor, we use an exponential probability law of parameter λ. The mean (or MTBF) of this law is $\mu = \frac{1}{\lambda}$. The MTBF of a task depends upon the number of processors

Parameters	
n	Number of tasks
p	Total number of processors available
μ	MTBF for one processor
λ	Parameter for the exponential law: $\lambda = 1/\mu$
Tasks	
$t_{i,j}$	Fault-free time needed to complete task T_i with j processors
m_i	Size of problem of task T_i (i.e., number of data)
$\sigma(i)$	Number of processors assigned to T_i
α	The fraction of remaining work after a failure or an application termination
t_e	The time when a task is over
$N_{i,j}^{\text{ff}}(\alpha)$	The number of checkpoints during the remaining work α for the task T_i with j processors in a fault-free case
Resilience	
$t_{i,j}^R(\alpha)$	Expected time to complete a fraction of work α for task T_i with j processors
t_i^U	Current expected finish time for the task T_i
$\tau_{i,j}$	The optimal checkpointing period for a task T_i on j processors
t_{lastR_i}	The time of last redistribution or last error of task T_i
$N_{i,j}$	The number of checkpoints for task T_i with j processors between t_{lastR_i} and the next event (t_e or t_f)
T_f	The task stroke by the fault at time t_f
T_{last}	The longest task in the schedule before the fault
t_f	Time when a fault occurs
p_f	The processor which is affected by a fault at time t_f

Table 1: List of notations.

it is using, hence changes whenever a redistribution occurs. Specifically, if task T_i is (currently) executed on j processors, its MTBF is $\mu_{i,j} = \frac{\mu}{j}$ (see for instance [16] for a proof). To recover from fail-stop errors, we use the double checkpointing scheme, or *buddy algorithm* [13, 14]. Therefore, the number of processors assigned to each task must be even.

We consider that tasks are divisible, and we do periodic checkpointing for each task. Formally, if task T_i is executed on j processors, there is a checkpoint every period of length $\tau_{i,j}$, with a cost $C_{i,j}$. The checkpoint cost $C_{i,j}$ depends on the number of processors assigned to the task: $C_{i,j} = C_i/j$. Here, C_i represents the sequential time to communicate critical data for T_i, which is equally partitioned across the j processors. As for the checkpointing period $\tau_{i,j}$ (which we can choose arbitrarily because tasks are divisible), we use Young's formula [17] and let

$$\tau_{i,j} = \sqrt{2\mu_{i,j}C_{i,j}} + C_{i,j}. \tag{1}$$

As $\tau_{i,j}$ is a first order approximation, the formula is valid only if $C_{i,j} \ll \mu_{i,j}$. When a fault strikes, there is first a downtime of duration D, and then a recovery period of duration $R_{i,j}$. We assume that $R_{i,j} = C_{i,j}$, while the downtime value D is platform-dependent and not application-dependent.

3.2 Execution time without redistribution

To compute the expected execution time of a schedule, we must be able to compute the expected execution time of a task T_i executed on j processors subject to failures. We first consider the case without redistribution. Let $t_{i,j}$ be the execution time of task T_i on j processors in a fault-free scenario. Let $t^R_{i,j}(\alpha)$ be the expected time required to compute a fraction α of the total work for task T_i on j processors, with $0 \leq \alpha \leq 1$. We need to consider a partial execution of T_i on j processors to prepare for the case with redistributions.

Recall that the execution of task T_i is periodic, and that the period $\tau_{i,j}$ depends only on the number of processors, but not on the remaining execution time (see Equation (1)). After a work of duration $\tau_{i,j}$, there is a checkpoint of duration $C_{i,j}$. In a fault-free execution, the time required to execute the fraction of work α is $\alpha t_{i,j}$, hence a total number of checkpoints of

$$N^{\text{ff}}_{i,j}(\alpha) = \left\lfloor \frac{\alpha t_{i,j}}{\tau_{i,j} - C_{i,j}} \right\rfloor. \tag{2}$$

Next, we have to estimate the expected execution time for each period of work between checkpoints. The expected time to execute successfully during T units of time with j processors (there are $T - C$ units of work and C units of checkpoint, where T is the period) is equal to $\left(\frac{1}{\lambda j} + D\right)(e^{\lambda j T} - 1)$ [16]. Therefore, in order to compute $t^R_{i,j}(\alpha)$, we compute the sum of the expected time for each period, plus the expected time for the last non-complete period. The last period is denoted τ_{last} and it is defined as:

$$\tau_{last} = \alpha t_{i,j} - N^{\text{ff}}_{i,j}(\alpha)(\tau_{i,j} - C_{i,j}). \tag{3}$$

The first $N^{\text{ff}}_{i,j}(\alpha)$ periods are equal (of length $\tau_{i,j}$), hence have the same expected time. Finally, we obtain:

$$t^R_{i,j}(\alpha) = e^{\lambda j R_{i,j}}\left(\frac{1}{\lambda j} + D\right)\left(N^{\text{ff}}_{i,j}(\alpha)(e^{\lambda j \tau_{i,j}} - 1) + (e^{\lambda j \tau_{last}} - 1)\right). \tag{4}$$

In a fault-free environment, it is natural to assume that the execution time is non-increasing with the number of processors. Here, this assumption would translate into the condition:

$$t^R_{i,j+1}(\alpha) \leq t^R_{i,j}(\alpha) \text{ for } 1 \leq i \leq n, 1 \leq j < p, 0 \leq \alpha \leq 1. \tag{5}$$

However, when we allocate more processors to a task, even though it further parallelize the work, the probability of failures increases, and hence the waste increases. Therefore, adding resources to an application is useful up to a threshold. After this threshold, we have $t^R_{i,j+1} \geq t^R_{i,j}$. In order to satisfy Equation (5), we restrict, according to the threshold, the number of processors assigned to each task:

$$t^R_{i,j}(\alpha) = \min\{\, t^R_{i,j-2}(\alpha)\,,\, t^R_{i,j}(\alpha)\,\}. \tag{6}$$

Another common assumption is that the work is non-decreasing with j: we assume that for $1 \leq i \leq n$, $1 \leq j < p$ and $0 \leq \alpha \leq 1$, $(j+1) \times t^R_{i,j+1}(\alpha) \geq j \times t^R_{i,j}(\alpha)$.

For convenience, we denote by t^U_i the current expected finish time of task T_i at any point of the execution. Initially, if task T_i is allocated to j processors, we have $t^U_i = t^R_{i,j}(1)$.

3.3 Redistributing processors

There are two major cases in which it may be useful to redistribute processors: (i) in a fault-free scenario, when a task ends, it releases processors that can be used to accelerate other tasks, and

(ii) when an error strikes, we may want to force the release of processors, so that we can assign them to the application that has been slowed down by the error.

We first consider a fault-free scenario in Section 3.3.1, and then we account for the checkpoint costs and for redistribution after failures in Section 3.3.2.

3.3.1 Fault-free scenario

We first consider that there is no failure, and hence no checkpoint is taken.

The goal is to explain how redistribution works.

Consider for instance that there are q available processors when task T_2 ends. We can allocate q_1 new processors to task T_1, and q_3 new processors to task T_3, such that $q_1+q_3 = q$ (see Figure 1).

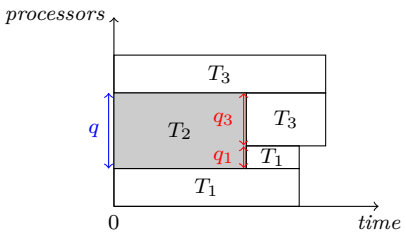


Figure 1: Redistribution at the end of a task.

Consider that a redistribution is done at time t_e (the end time of a task), and that task T_i, initially with j processors, now has $k = j + q > j$ processors. What will be the new finish time of T_i? The fraction of work already executed for T_i is $\frac{t_e}{t_{i,j}}$, because the task was supposed to finish at time $t_{i,j}$ (see Figure 2). The remaining fraction of work is $\alpha = 1 - \frac{t_e}{t_{i,j}}$, and the time required to complete this work with k processors is t', where $\frac{t'}{t_{i,k}} = \alpha$, hence

$$t' = \alpha t_{i,k} = \left(1 - \frac{t_e}{t_{i,j}}\right) t_{i,k}.$$

Furthermore, we need to add a redistribution cost: when moving from j to k processors, a task must redistribute its data between processors. Indeed, we have to send a fraction $\frac{1}{k\times j}$ of data from the initial j processors to the $q = k - j$ new processors. So we have a bipartite graph G with j nodes in one subspace and k in another. Before the redistribution, each j node has $\frac{k}{k\times j}$ data, after the redistribution the $q = k - j$ new processors will have $\frac{j}{k\times j}$. One processor can send $\frac{1}{k\times j}$ at the time, so we have j sending in the same time. One parallel dispatch is called a *round*. How many rounds are required? We can transform this problem into an edge coloring problem, with one color for one round (see Figure 3).

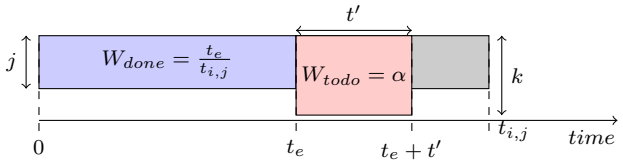


Figure 2: Work representation for task T_i at time t_e.

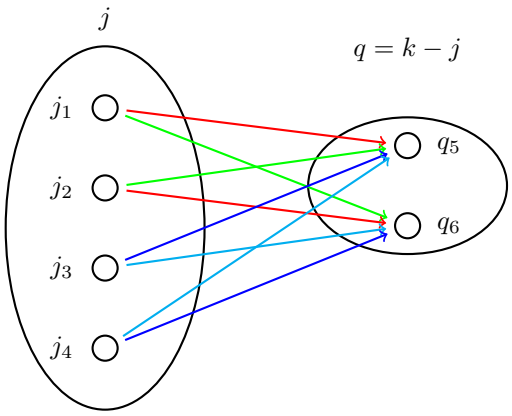


Figure 3: Bipartite graph G representing a redistribution from $j = 4$ to $k = 6$, with each round colored in one color. We have $\chi'(G) = \Delta(G) = 4$.

The number of rounds required is equal to the edge chromatic number $\chi'(G)$. Konig's theorem [18] states that the edge chromatic number is equal to the maximum degree in G so $\chi'(G) = \Delta(G)$ when G is bipartite. We can determine the $\Delta(G)$ value: each j and q node are connected, hence the maximum degree is equal to $\max(j, k-j)$. Therefore the number of rounds is equal to $\max(j, k-j)$. Finally, the redistribution cost is

$$RC_i^{j \to k} = \max(j, k-j) \times \frac{1}{k} \times \frac{m_i}{j}, \tag{7}$$

where m_i is the total data belonging to task T_i. Note that we perform a redistribution only if the cost of redistribution is lower than the benefit to allocate new processors to a task, i.e., if $t_{i,j} - (t_e + t') > RC_i^{j \to k}$.

3.3.2 Accounting for failures

If a fault strikes a task, that task needs to recover from the failure and to re-execute some work. Even if the tasks were initially well-balanced to minimize the total execution time, this task is now likely to exceed the expected execution time. If it becomes the longest task of the schedule, we try to give more processors to this task in order to reduce its finish time, hence redistributing processors.

Because we use the double checkpointing algorithm as resilience model, we consider processors by pairs. We aim at redistributing pairs of processors either when a task is finished, at time t_e (as in the fault-free scenario discussed in Section 3.3.1), or when a failure occurs, at time t_f.

In each case, we need to compute the remaining work and the new expected finish time of the tasks that have been affected by the event. Given a task T_i, we keep track of the time when the last redistribution or failure occurred for this task, denoted t_{lastR_i}. At time t (corresponding to the end of a task or to a failure), we know exactly how many checkpoints have been taken by task T_i executed on j processors since t_{lastR_i}, denoted by $N_{i,j}$:

$$N_{i,j} = \left\lfloor \frac{t - t_{lastR_i}}{\tau_{i,j}} \right\rfloor. \tag{8}$$

We begin with the application ending case: consider that an application finishes its execution at time t_e, hence releasing some processors. We consider giving some of these processors to a task T_i currently running on j processors. The fraction of work executed by T_i since the last redistribution is $\frac{t_e - t_{lastR_i} - N_{i,j}C_{i,j}}{t_{i,j}}$, because we have to remove the cost of the checkpoints, during which the application did not execute useful work.

We apply the same reasoning when a fault occurs. Consider that task T_i, running on j processors, is subject to a failure at time t_f. Therefore, T_i needs to recover from its last valid checkpoint, and the fraction of work executed by T_i corresponds to the number of entire periods completed since the last failure or redistribution t_{lastR_i}, each followed by a checkpoint. We can express it as $\frac{N_{i,j} \times (\tau_{i,j} - C_{i,j})}{t_{i,j}}$.

At time t_f, if we want to give extra processors to the faulty task, we may need to remove some processors from a running application. In this case, we perform a redistribution on an application T_i moving from j to k processors, with $k < j$. During the downtime and the recovery period of task T_i, the other tasks are stopped. Hence, the fraction of work executed by T_i can be computed as in the application ending case scenario: it is $\frac{t_f - t_{lastR_i} - N_{i,j}C_{i,j}}{t_{i,j}}$.

Similarly to the fault-free scenario, $RC_i^{j \to k}$ denotes the redistribution cost for task T_i. Redistribution can now add or remove processors to task T_i, and the cost is expressed as:

$$RC_i^{j \to k} = \max(\min(j,k), |k-j|) \times \frac{1}{k} \times \frac{m_i}{j}. \tag{9}$$

When a redistribution is done for task T_i at time t ($t = t_e$ or $t = t_f$), we start with a checkpoint before computing with a period $\tau_{i,k}$. Therefore, if a fault occurs, we do not have to redistribute again. Let α be the remaining fraction of work to be executed by T_i, that is 1 minus the sum of the fraction of work executed before t_{lastR_i} and the fraction of work expressed above (computed between t_{lastR_i} and t).

The new value of t_{lastR_i} becomes $t_{lastR_i} = t + D + R_{i,j} + RC_i^{j \to k} + C_{i,k}$ for a task on which a fault stroke (we need to account for the downtime and recovery), and $t_{lastR_i} = t + RC_i^{j \to k} + C_{i,k}$ for a task with no failure but on which we performed some redistribution. Finally, the expected finish time of T_i after the failure becomes $t_i^U = t_{lastR_i} + t_{i,k}^R(\alpha)$.

Similarly to the fault-free scenario, we give extra processors to an application only if the new expected finish time t_i^U is lower than the one with no redistribution, namely $t + t_{i,j}^R(\alpha)$.

4 Complexity results

We first consider the problem without redistribution in Section 4.1 and provide an optimal polynomial-time algorithm. Then, we prove that the problem becomes NP-complete with redistribution, even in a fault-free scenario (Section 4.2).

4.1 Without redistribution

Aupy et al. [3] designed a greedy algorithm to solve the problem with no redistribution (called Optimal-1-pack-schedule) in a fault-free scenario. This algorithm therefore works with $t_{i,j}$ values instead of $t_{i,j}^R$, and minimizes the execution time of the tasks. As a technical detail, it does not take into account the fact that the number of processors assigned to a task must always

be even in our setting because we use the double checkpointing algorithm. However, it is easy to extend this algorithm to solve the problem with failures. We prove below that the problem of minimizing the expected execution time without redistribution can be solved in polynomial time.

Theorem 1. *Given n tasks to be scheduled on p processors in a single pack, the problem of finding a schedule without redistribution that minimizes the expected execution time can be solved in polynomial time.*

Proof. We define a function σ such that $\sum_{i=1}^{n} \sigma(i) \leq p$, where $\sigma(i)$ is the number of processors assigned to T_i. A schedule with no redistribution corresponds to a function σ, because the number of processors remains identical throughout the whole execution.

The fraction of work that each task must compute is $\alpha = 1$, and we use the notation $T_i \preccurlyeq_\sigma^R T_j$ if $t_{i,\sigma(i)}^R(1) \leq t_{j,\sigma(j)}^R(1)$. Then, Algorithm 1 returns in polynomial time a schedule that minimizes the expected execution time. It greedily allocates processors to the longest task while its expected execution time can be decreased. If we cannot decrease the expected execution time of the longest task, then we cannot decrease the overall expected execution time, which is the maximum of the expected execution times of all tasks.

Algorithm 1: Optimal schedule with no redistribution.

```
1  procedure Optimal-Schedule(n, p) begin
2      for i = 1 to n do σ(i) := 2 ;
3      Let L be the list of tasks sorted in non-increasing values of ≼_σ^R;
4      p_available := p − 2n;
5      while p_available ≥ 2 do
6          T_i* := head(L);
7          L := tail(L);
8          p_max := σ(i*) + p_available;
9          if t^R_{i*,σ(i*)}(1) > t^R_{i*,p_max}(1) then
10             σ(i*) := σ(i*) + 2;
11             L := Insert T_i* in L according to its ≼_σ^R value;
12             p_available := p_available − 2;
13         else p_available := 0;
14     end
15     return σ;
16 end
```

The proof that this algorithm returns an optimal cost schedule is similar to the proof in [3]. We replace $t_{i,j}$ by $t_{i,j}^R(1)$, and instead of adding processors one-by-one, we add them two-by-two. Consequently, there are at most $(p - 2n)/2$ iterations. The complexity of Algorithm 1 is $O(p \times \log(n))$. □

Note that we added a test in Line 9 to check whether there is a hope to decrease the expected execution time of the longest time. If the expected execution time with $\sigma(i^\star)$ is equal to the expected execution time using all available processors, then we will not be able to decrease the expected execution time. Therefore, we prefer to keep these processors available so that we may use them if we want to perform some redistribution. We discuss heuristics performing redistribution in Section 5.

4.2 NP-completeness of the redistribution problem

To establish the problem complexity with redistribution, we consider the simple case with no failures. Therefore, redistributions occur only at the end of an application, and any application changes at most n times its number of processors, where n is the total number of applications. We further consider that the redistribution cost is null. Even in this simplified scenario, we prove that the problem is NP-complete:

Theorem 2. *Without redistribution costs and without failures, the problem of finding a schedule that minimizes the execution time is NP-complete (in the strong sense).*

Proof. We consider the associated decision problem: given a bound on the execution time D, is there a schedule with an execution time less than D? The problem is obviously in NP: given a schedule and a mapping, it is easy to check in polynomial time that it is valid by computing its execution time.

To establish the completeness, we use a reduction from 3-PARTITION [19]. We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3m$ positive integers $a_1, a_2, \ldots, a_{3m}$ such that for all $i \in \{1, \ldots, 3m\}$, $B/4 < a_i < B/2$ and with $\sum_{i=1}^{m} a_i = mB$, does there exist a partition $I_1, \ldots, I_m$ of $\{1, \ldots, 3m\}$ such that for all $j \in \{1, \ldots, m\}$, $|I_j| = 3$ and $\sum_{i \in I_j} a_i = B$? We let $D = \max_{1 \leq i \leq m}\{a_i\} + 1$ be the bound on the execution time. Note that $4D - B > D$.

We build an instance $\mathcal{I}_2$ of our problem, with $n = 4m$ applications and n processors. For $1 \leq i \leq 3m$, we have the following execution times: $t_{i,1} = a_i$, and $t_{i,j} = \frac{3a_i}{4}$ for $j > 1$ (these are *small* tasks, and the work is strictly larger when using more than one processor). The last m tasks are identical, with the following execution times: for $3m + 1 \leq i \leq 4m$, $t_{i,j} = \frac{4D-B}{j}$ for $1 \leq j \leq 4$, and $t_{i,j} = \frac{2}{9}(4D - B)$ for $j > 4$ (these are *large* tasks with a total work equal to $4D - B$ for $1 \leq j \leq 4$, and a strictly larger work when using more than four processors). It is easy to check that the execution times are non-increasing with j, and that the work $j \times t_{i,j}$ is non-decreasing with j for all tasks.

Clearly, the size of $\mathcal{I}_2$ is polynomial in the size of $\mathcal{I}_1$. We now show that instance $\mathcal{I}_1$ has a solution if and only if instance $\mathcal{I}_2$ does.

Suppose first that $\mathcal{I}_1$ has a solution. Let $I_k = \{a'_{1,k}, a'_{2,k}, a'_{3,k}\}$, for $k \in \{1, \ldots, m\}$. We build the following schedule for $\mathcal{I}_2$: initially, each task has a single processor. When a task T_i finishes its execution (at time a_i), with $1 \leq i \leq 3m$, its processor is given to task $3m + k$, given that $a_i \in I_k$. Task $3m + k$ is therefore assigned at most 4 processors, hence with an optimal work of $4D - B$, and it completes in time D. Indeed, these 4 processors have to complete a total work of $a'_{1,k} + a'_{2,k} + a'_{3,k} + 4D - B = 4D$, where the three tasks with $1 \leq i \leq 3m$ finish before D, and the last task can be parallelized with up to 4 processors without loosing any work (see Figure 4).

Suppose now that $\mathcal{I}_2$ has a solution. Initially, we have necessarily one processor per task, because there are exactly n processors and n tasks. Each task T_i, with $1 \leq i \leq 3m$, will complete before the remaining m tasks, because $4D - B > D > \max_{1 \leq i \leq 3m}\{a_i\}$. If the first $3m$ tasks use more than one processor or if the last m tasks use more than four processors, then the total work becomes strictly larger than $\sum_{i=1}^{3n} a_i + m \times (4D - B) = mB + 4mD - mB = nD$, and hence the deadline of D with n processors cannot be achieved. Therefore, each small task uses only one processor, and finishes before D. For $1 \leq k \leq m$, the large task $3m + k$ can use the remaining time of at most three small tasks, say $a'_{1,k}, a'_{2,k}, a'_{3,k}$. Therefore, we must have $4D - B + a'_{1,k} + a'_{2,k} + a'_{3,k} < 4D$ in order to complete within D, and hence $a'_{1,k} + a'_{2,k} + a'_{3,k} < B$. This is true for all triplets of small tasks, and because the work is tight, we must have an equality for each triplet, hence the solution to $\mathcal{I}_1$. □

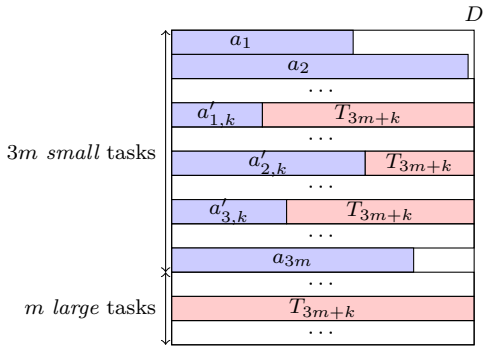


Figure 4: Illustration for the proof of Theorem 2.

5 Heuristics

Because the redistribution problem is NP-hard, we propose some polynomial-time heuristics to perform redistribution when a failure strikes or when an application ends. Before performing any redistribution, we need to choose an initial allocation of the p processors to the n tasks. We use the optimal algorithm with no redistribution discussed in Section 4.1 (Algorithm 1).

We first discuss the general structure of the heuristics in Section 5.1. Then, we explain how to redistribute available processors in Section 5.2, and the two strategies to redistribute when failures occur in Section 5.3.

5.1 General structure

All heuristics share the same skeleton (see Algorithm 2): we iterate over events (either a failure or an application termination) until the sum of work (the total work) is equal to zero.

If some tasks are still working for a previous redistribution, (i.e., the current time t is smaller than t_{lastR_i} for these tasks), then we exclude them for the next redistribution (Line 15), and add them back into the list of tasks after the redistribution. If an application ends, we redistribute available processors as will be discussed in Section 5.2. Then, if there is a failure, we calculate the new expected execution time of the faulty task (Line 26). Also, we remove from the list the tasks that end before t_{lastR_f}, and we release their processors (Line 28).

Afterwards, we have to choose between trying to redistribute or do nothing. If the faulty task is not the longest task, the total execution time has not changed since the last redistribution. Therefore, because it is the best execution time that we could reach, there is no need to try to improve it. However, if the faulty task is the longest task (Line 30), we apply a heuristic to redistribute processors (see Section 5.3).

5.2 Redistributing available processors

When a task ends, the idea is to redistribute the available processors in order to decrease the expected execution time. The easiest way to proceed consists in adding processors greedily to the task with the longest execution time, as was done in Algorithm 1 to compute an optimal schedule. This time, we further account for the redistribution cost, and update the values of α_i, t_{lastR_i} and t_i^U for each task i that encountered a redistribution. Therefore, this heuristic, called EndLocal (see Algorithm 3), returns a new distribution of processors.

Rather than using only local decisions to redistribute available processors at time t, it is possible to recompute an entirely new schedule, using the greedy algorithm Algorithm 1 again,

Algorithm 2: Algorithmic skeleton.

```
1  procedure Main(n, p)
2  begin
3      α and t_lastR are considered as global variables;
4      /* Initial schedule */;
5      σ := Optimal-Schedule(n, p);
6      for i = 1 to n do
7          α_i = 1; t_lastR_i = 0;
8          t_i^U = t^R_{i,σ(i)}(1);
9      end
10     Let L be the list of tasks sorted in non-increasing values of t_i^U;
11     /* While it remains work */;
12     while Σ_{i=1}^n α_i > 0 do
13         k := p − Σ_{i=1}^n σ(i) /* There are k unused processors */;
14         t := next incoming event;
15         for i = 1 to n do if t ≤ t_lastR_i then Remove temporarily T_i from L;
16         ;
17         if t is the end of task T_e then
18             α_e := 0 ;
19             Remove T_e from the list of tasks L;
20             σ := Redistrib-Available-Procs(L, t, k + σ(e), σ);
21         else if t is a failure striking task T_f then
22             /* Updating information about the faulty task T_f */
23             j := σ(f); N_{f,j} = ⌊(t − t_lastR_f)/τ_{f,j}⌋;
24             α_f := α_f − N_{f,j}(τ_{f,j} − C_{f,j})/t_{f,j};
25             t_lastR_f := t + D + R_{f,j};
26             t_f^U := t_lastR_f + t^R_{f,j}(α_f);
27             Update the position of T_f in the list L according to its new t_f^U value;
28             for i = 1 to n do if T_i finishes before t_lastR_f then Remove T_i and release σ(i)
               processors;
29             ;
30             if t_f^U = max_{1≤i≤n} t_i^U then
31                 σ := Apply-heuristic(L, t, f, σ);
32             end
33         end
34         ;
35         Put back the previous removed tasks in L;
36     end
37 end
```

Algorithm 3: Algorithm to perform a redistribution with k processors on n tasks at time t

```
1  procedure Redistrib-Available-Procs(L, t, k, σ)
2  begin
3      σ_init := σ;
4      while k ≥ 2 do
5          T_i := head(L); L := tail(L);
6          j := σ_init(i);
7          N_{i,j} = ⌊(t − t_{lastR_i})/τ_{i,j}⌋;
8          α_i^t := α_i − (t − t_{lastR_i} − N_{i,j}C_{i,j})/t_{i,j};
9          /* We first check whether T_i can be improved */
10         improvable := false; q := 2;
11         while q ≤ k do
12             t^E := t + RC_i^{j→σ(i)+q} + C_{i,σ(i)+q} + t^R_{i,σ(i)+q}(α_i^t);
13             if t^E < t_i^U then improvable := true; q := k + 1;
14             else q := q + 2;
15         end
16         if improvable then
17             σ(i) := σ(i) + 2;
18             t_i^U := t + RC_i^{j→σ(i)} + C_{i,σ(i)} + t^R_{i,σ(i)}(α_i^t);
19             L := Insert T_i in L according to its t_i^U value;
20             k := k − 2;
21         end
22     end
23     /* Updating α_i and t_{lastR_i} if needed */
24     for i = 1 to n do
25         j := σ_init(i);
26         if σ(i) ≠ j then
27             N_{i,j} = ⌊(t − t_{lastR_i})/τ_{i,j}⌋;
28             α_i := α_i − (t − t_{lastR_i} − N_{i,j}C_{i,j})/t_{i,j};
29             t_{lastR_i} := t + RC_i^{j→σ(i)} + C_{i,σ(i)};
30         end
31     end
32     return σ;
33 end
```

but further accounting for the cost of redistributions. This heuristic is called EndGreedy. Now, we need to compute the remaining fraction of work for each task, and we obtain an estimation of the expected finish time when each task is mapped on two processors. Similarly to Algorithm 1, we then add two processors to the longest task while we can improve it, accounting for redistribution costs.

Note that we effectively update the values of α_i and t_{lastR_i} for task T_i only if a redistribution was done for this task. It may happen that the algorithm assigns the same number of processors as was used before. Therefore, we keep the updated value of the fraction of work in a temporary variable α_i^t and update the value if needed at the end of the procedure.

This approach is very similar to IteratedGreedy (Algorithm 5) described in the next section, except that there is no faulty task.

5.3 Redistributing when there is a failure

Similarly to the previous case of redistributing available processors, we propose two heuristics to redistribute in case of failures. The first one, ShortestTasksFirst, takes only local decisions. First, we allocate the k available processors (if any) to the faulty task if that task is improvable. Then, if the faulty task is still improvable, we try to take processors from shortest tasks (denoted T_s) in the schedule, and give these processors to the faulty task, until the faulty task is no longer improvable, or there are no more processors to take from other tasks. We take processors from a task only if its new execution time is smaller than the execution time of the faulty task (see Algorithm 4).

The second heuristic, IteratedGreedy, uses a modified version of the greedy algorithm that initializes the schedule (Algorithm 1) each time there is a failure, while accounting for the cost of redistributions. This is done similarly to the redistribution of EndGreedy explained in Section 5.2, except that we need to handle the faulty task differently to update the values of α_f and t_{lastR_f} (see Algorithm 5).

6 Simulations

To show the efficiency of the heuristics defined in Section 5, we have performed extensive simulations. The simulation settings are discussed in Section 6.1, and results are presented in Section 6.2. Note that the code is publicly available at `http://graal.ens-lyon.fr/~abenoit/code/redistrib`, so that interested readers can experiment with their own parameters.

6.1 Simulation settings

To evaluate the quality of the heuristics, we conduct several simulations, using realistic parameters. The first step is to generate a fault distribution: we use an existing fault simulator developed in [20, 21]. In our case, we use this simulator with an exponential law of parameter λ. The second step is to generate a fault-free execution time for each task (the $t_{i,j}$ value). We use a *synthetic* model to generate the execution time in order to represent a large set of scientific applications. The task model that we use is a classical one, similar to the one used in [3]. For a problem of size m, we define the sequential time: $t(m, 1) = 2 \times m \times log_2(m)$. Then we can define the parallel execution time on q processors:

$$t(m, q) = f \times t(m, 1) + (1 - f)\frac{t(m, 1)}{q} + \frac{m}{q} log_2(m). \tag{10}$$

The parameter f is the sequential fraction of time, we fix it to $f = 0.08$. So 92% of time is considered as parallel. The factor $\frac{m}{q} log_2(m)$ represents the overhead due to communications

Algorithm 4: ShortestTasksFirst

```
1  procedure ShortestTasksFirst (L, t, f, σ) begin
2      σ_init := σ;
3      /* Compute α_i^t */
4      for i = 1 to n do
5          if i ≠ f then α_i^t := α_i − (t − t_lastR_i − ⌊(t − t_lastR_i)/τ_{i,σ(i)}⌋ C_{i,σ(i)})/t_{i,σ(i)} ;
6          else
7              α_f^t := α_f;
8          end
9      end
10     k := p − Σ_{i=1}^{n} σ(i) /* There are k available processors */;
11     improvable := false;
12     while k ≥ 2 do
13         improvable := false; q := 2;
14         q_max := q;
15         while q ≤ k do
16             t^E := t + RC_f^{σ_init(f)→σ_init(f)+q} + C_{f,σ_init(f)+q} + t^R_{f,σ_init(f)+q}(α_f);
17             if t^E < t_f^U then improvable := true; q_max := q; q := k + 1;
18             else q := q + 2;
19         end
20         if improvable then
21             σ(f) := σ(f) + q_max;
22             t_f^U := t + RC_f^{σ_init(f)→σ(f)} + C_{f,σ(f)} + t^R_{f,σ(f)}(α_f);
23             k := k − q_max;
24         end
25     end
26     /* Taking processors from shortest application */;
27     while improvable do
28         Let T_s be the shortest task such that σ(s) ≥ 4; improvable := false; q := 2;
29         while q ≤ σ(s) − 2 do
30             t_f^E := t + RC_f^{σ_init(f)→σ(f)+q} + C_{f,σ(f)+q} + t^R_{f,σ(f)+q}(α_f);
31             t_s^E := t + RC_s^{σ_init(s)→σ(s)−q} + C_{s,σ(s)−q} + t^R_{s,σ(s)−q}(α_s^t);
32             if t_f^E < t_f^U and t_s^E < t_f^U then improvable := true; q := σ(s) + 1;
33             else q := q + 2;
34         end
35         if improvable then
36             σ(f) := σ(f) + 2; σ(s) := σ(s) − 2;
37             t_f^U := t + RC_f^{σ_init(f)→σ(f)} + C_{f,σ(f)} + t^R_{f,σ(f)}(α_f);
38             t_s^U := t + RC_s^{σ_init(s)→σ(s)} + C_{s,σ(s)} + t^R_{s,σ(s)}(α_s^t);
39             if t_s^U > t_f^U then improvable := false;
40         end
41     end
42     /* Updating α_i and t_lastR_i if needed */
43     for i = 1 to n do
44         if σ(i) ≠ σ_init(i) then
45             α_i := α_i^t;
46             t_lastR_i := t + RC_i^{σ_init(i)→σ(i)} + C_{i,σ(i)};
47         end
48     end
49     return σ;
50 end
```

Algorithm 5: IteratedGreedy

```
1  procedure IteratedGreedy (L, t, f, σ) begin
2      σ_init := σ;
3      for i = 1 to n do
4          if i ≠ f then α_i^t := α_i − (t − t_lastR_i − ⌊(t − t_lastR_i)/τ_{i,σ(i)}⌋ C_{i,σ(i)})/t_{i,σ(i)} ;
5          α_f^t := α_f;
6          σ(i) ← 2;
7          if σ(i) ≠ σ_init(i) then t_i^U = t + RC_i^{σ_init→σ(i)} + C_{i,σ(i)} + t^R_{i,σ(i)}(α_i^t);
8      end
9      Let L be the list of tasks sorted in non-increasing values of t_i^U;
10     p_available := p − 2n;
11     while p_available ≥ 2 do
12         T_i := head(L); L := tail(L);
13         p_max := σ(i) + p_available;
14         improvable := false; q := 2;
15         while σ(i) + q ≤ p_max do
16             if σ(i) + q = σ_init(i) then t^E := t_lastR_i + t^R_{i,σ(i)+q}(α_i);
17             else t^E := t + t^R_{i,σ(i)+q}(α_i^t) + RC_i^{σ_init(i)→σ(i)+q} + C_{i,σ(i)+q};
18             if t^E < t_i^U then improvable := true; q := p_max + 1;
19             else q := q + 2;
20         end
21         if improvable then
22             σ(i) := σ(i) + 2;
23             if σ(i) = σ_init(i) then t_i^U := t_lastR_i + t^R_{i,σ(i)}(α_i);
24             else
25                 t_i^U := t + RC_i^{σ_init→σ(i)} + C_{i,σ(i)} + t^R_{i,σ(i)}(α_i^t);
26             end
27             L := Insert T_i in L according to its t_i^U value;
28             p_available := p_available − 2;
29         end
30         else p_available := 0;
31     end
32     /* Updating t_lastR_i and α_i if needed */
33     for i = 1 to n do
34         if σ(i) ≠ σ_init(i) then
35             α_i := α_i^t;
36             t_lastR_i := t + RC_i^{σ_init(i)→σ(i)} + C_{i,σ(i)};
37         end
38     end
39     return σ;
40 end
```

and synchronization of data. Finally, we have $t_{i,j}(m_i) = t(m_i, j)$ where $t_{i,j}(m_i)$ is the execution time for task T_i with a problem of size m_i on j identical processors.

Finally, we assign to each task T_i a random value for the number of data m_i such that: $m_{inf} \leq m_i \leq m_{sup}$. If $m_{inf} \ll m_{sup}$ then the data distribution between tasks is very heterogeneous. On the contrary, if m_{inf} is close to m_{sup}, the data distribution is homogeneous, in other words all tasks have (almost) the same execution time. Unless stated otherwise, we set $m_{inf} = 1500000$ and $m_{sup} = 2500000$ to have execution times long enough so that several failures strike during execution. With such a value for m_{sup}, the longest execution time in a fault-free execution is around 100 days. The cost of checkpoints for a task T_i with j processors is $C_{i,j} = C_i/j$, where C_i is proportional to the memory footprint of the task. We have $C_i = m_i \times c$, where c is the time needed to checkpoint one data unit of m_i. The default value is $c = 1$, unless stated otherwise. Finally, the MTBF of a single processor is fixed to 100 years, unless stated otherwise.

In the following section, we vary the number of processors, the number of tasks, the checkpointing cost and the data distribution to study their impact on performance. Note that we assume that a failure can strike during checkpoints but not during downtime, recovery and while the processor is performing some redistribution.

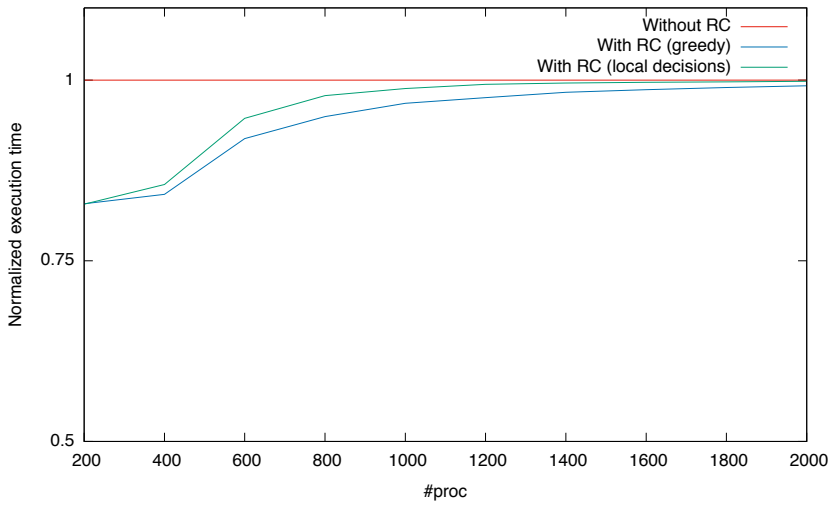
6.2 Results

To evaluate the heuristics, we execute each heuristic $x = 50$ times and we compute the average *makespan*, i.e., the longest execution time in the pack. We compare the makespan obtained by the heuristics to the makespan (i) in a fault context without any redistribution (worst case), and (ii) in a fault-free context with redistributions (best case). RC stands for redistribution in the graphs. We normalize the results by the makespan obtained in a fault context without any redistribution, which is expected to be the worst case. The execution in a fault-free setting provides us an optimistic value of the execution of the application in the ideal case where no failures occur.

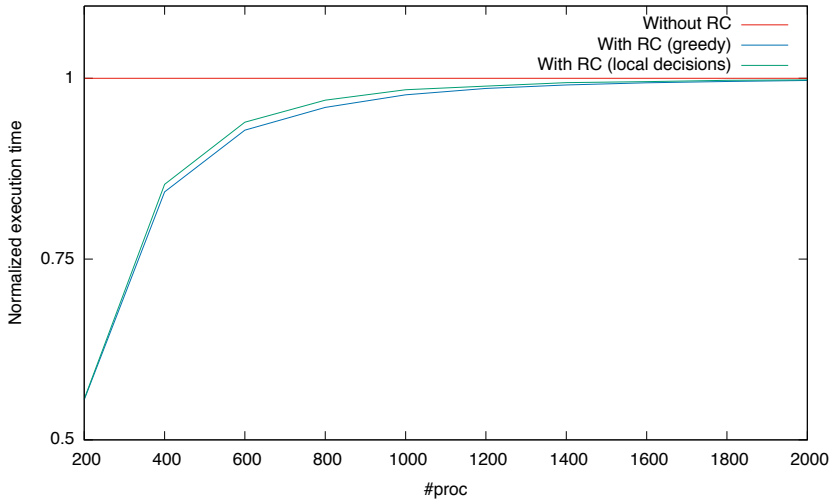
We consider four heuristics: IteratedGreedy-EndGreedy where we greedily recompute a new schedule at each task termination and each failure; IteratedGreedy-EndLocal where we use Algorithm 3 at each task termination, but IteratedGreedy in case of failures; ShortestTasksFirst-EndGreedy where we greedily recompute a new schedule at each task termination, but use ShortestTasksFirst in case of failures; and ShortestTasksFirst-EndLocal where we only use the local variants.

Performance in a fault-free context. Figure 5 shows the impact of redistribution in a fault-free context with 100 tasks, where we vary the number of processors from 200 to 2000. In this case, we compare EndLocal with EndGreedy (see Section 5.2). The two heuristics have a very similar behavior, leading to a gain of a least 20% with less than 500 processors, and a slightly better gain for the EndGreedy global heuristic. When the number of processors increases, the efficiency of both heuristics decrease to converge to the performance without redistribution. Indeed, there are then enough processors so that each application does not make use of extra processors released by ending applications. In the heterogeneous context (with $m_{inf} = 1500$), the gain due to redistribution is even better.

Figure 6 shows the impact of redistribution in a fault-free context with 1000 tasks, we vary the number of processors from 2000 to 5000. We compare EndLocal with EndGreedy, the two heuristics have a similar behavior. As showed in Figure 5, the redistribution is more efficient in the heterogeneous context (with $m_{inf} = 1500$).

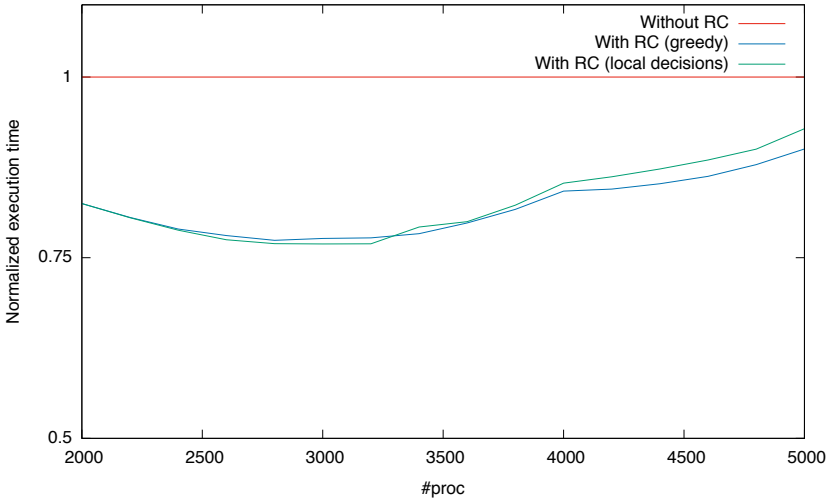


(a) $m_{inf} = 1500000$.

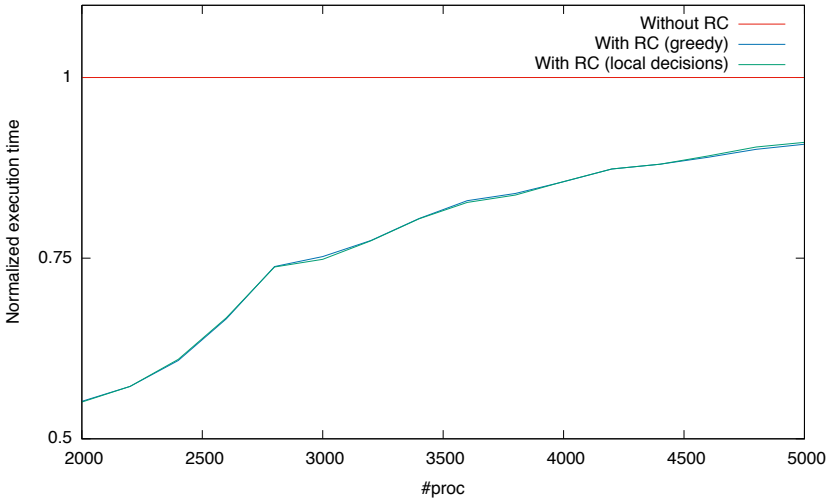


(b) $m_{inf} = 1500$.

Figure 5: Performance of redistribution in a fault-free context with $m_{sup} = 2500000$.



(a) $m_{inf} = 1500000$.



(b) $m_{inf} = 1500$.

Figure 6: Performance of redistribution in a fault-free context with $m_{sup} = 2500000$.

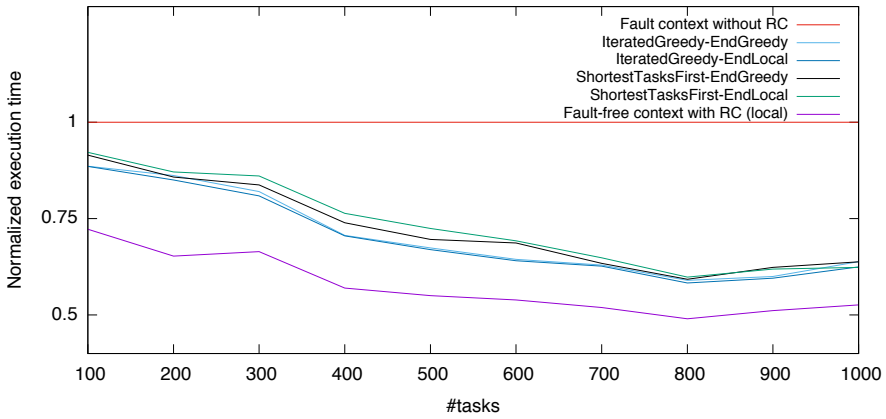


Figure 7: Impact of n with $p = 5000$ processors.

Impact of n. Figure 7 shows the impact of the number of tasks n when the number of processors is fixed to 5000. The results show that having more tasks increases efficiency of both heuristics. With $n = 1000$, we obtain a gain of more than 40% due to redistribution. The reason is that when n increases, the number of processors assigned to each task decreases, then heuristics have more flexibility to redistribute.

Note that, as expected, IteratedGreedy is better than ShortestTasksFirst, because it recomputes a complete new schedule at each fault, instead of just allocating available processors and processors from shortest tasks to the faulty task. Using EndGreedy with IteratedGreedy does not improve the performance, while EndGreedy is useful with ShortestTasksFirst, hence showing that complete redistributions are useful, even when only performed at the end of a task.

Impact of p. Figure 8 shows the impact of the number of processors p when the number of tasks is fixed. We vary p between 200 and 5000 processors. The results show that having more processors decreases the efficiency of both heuristics, but there is always a gain of at least 10% thanks to redistributions.

The same observations hold, i.e., the use of EndGreedy vs EndLocal impacts only ShortestTasksFirst. In average, with IteratedGreedy, we obtain a gain of 25%, while ShortestTasksFirst provides a gain around 15% when it is not combined with EndGreedy.

This figure also allows us to observe the impact of the MTBF on performance. Indeed, the MTBF is set to 100 years for each processor, but the overall MTBF for a task ($\mu_{i,j}$ value) decreases when the number of processors increases, so the gain obtained by heuristics decreases due to the increasing number of failures.

Heuristic behaviors. Figure 9 compares IteratedGreedy and ShortestTasksFirst, when combined with EndLocal, on a single execution. We depict both the evolution of the makespan (see Figure 9a) and the standard deviation, in terms of number of processors (see Figure 9b). IteratedGreedy is clearly superior in terms of makespan, and this can be explained by the fact that it allocates more processors to the longest task more quickly than ShortestTasksFirst, hence resulting in a larger standard deviation. Because ShortestTasksFirst

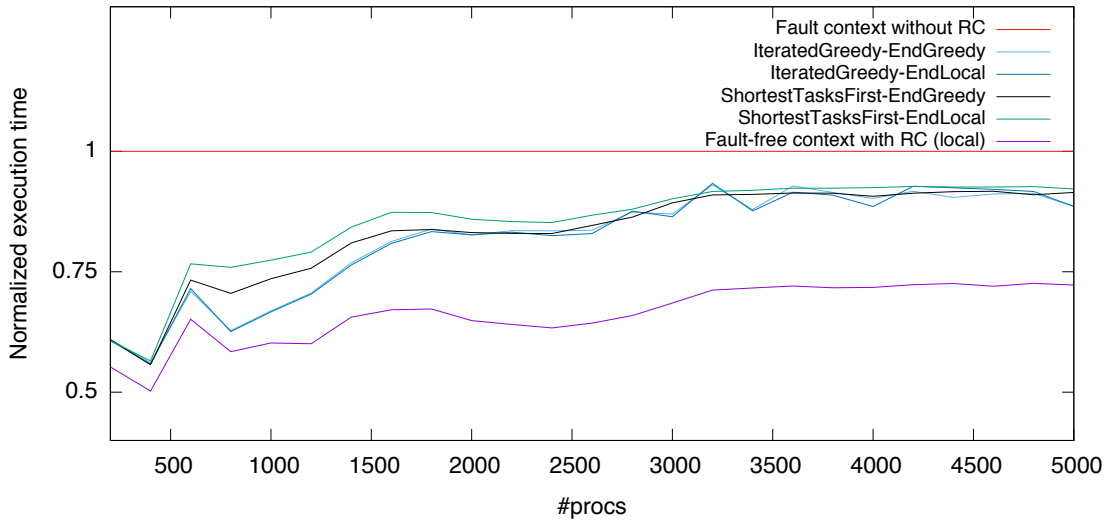


Figure 8: Impact of p with $n = 100$ tasks.

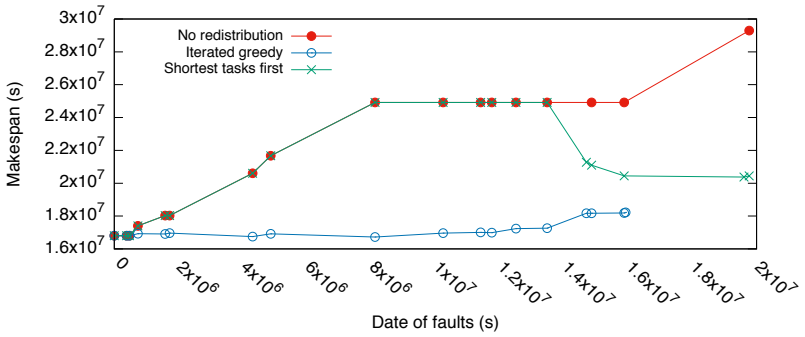
takes only local decisions, it needs more time before enough processors are given to the longest task.

Impact of MTBF. Figure 11 shows the impact of the MTBF on redistribution. We vary the MTBF of a single processor between 5 years and 125 years. When the MTBF decreases, the number of failures increases, consequently the performance of both heuristics decreases. The performance of IteratedGreedy is very linked to the MTBF value. Indeed, it tends to favor a heterogeneous distribution of processors (i.e., tasks with many processors and tasks with few processors). If a task is executed on many processors, its MTBF becomes very small and this task will be hit by more failures, hence it becomes even worse than without redistribution!

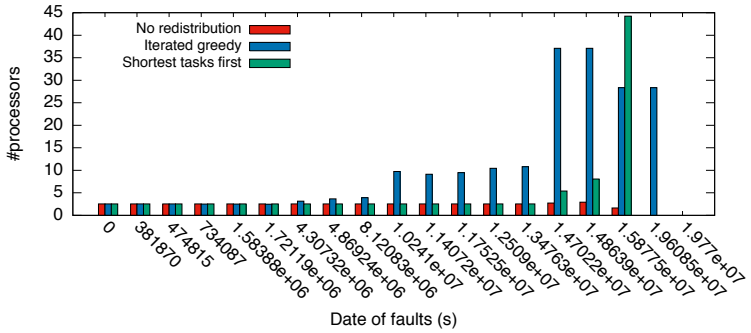
Impact of checkpointing cost. Figure 12 shows the impact of the checkpointing cost on a platform with 100 tasks and 1000 processors. To do so, we multiply the checkpointing cost by c in Figure 12 (recall that c is the time needed to checkpoint one data unit). When c decreases, the performance of the heuristics increases and the gap between the execution time in a fault-free context and a fault context becomes small. Indeed, if checkpoints are cheap, a lot of checkpoints can be taken, and the average time lost due to failures decreases. We see the same effect on Figure 13.

Impact of the sequential fraction of time. Figure 14 shows the impact of the sequential fraction of time. We vary f from 0 (tasks are fully parallel) to 0.5 (50% of the time is sequential). The results show that when tasks are more parallel, the redistribution is more efficient. This result is expected, because if tasks are not parallel, there is no gain when trying to allocate more processors to help them complete.

Summary. To conclude, we note that IteratedGreedy achieves better performance than ShortestTasksFirst, mainly because it rebuilds a complete schedule at each fault, which is



(a) Makespan at each failure handled.



(b) Standard deviation at each failure handled.

Figure 9: Heuristic behaviors with $n = 100$, $p = 1000$, MTBF of 50 years, for a single execution.

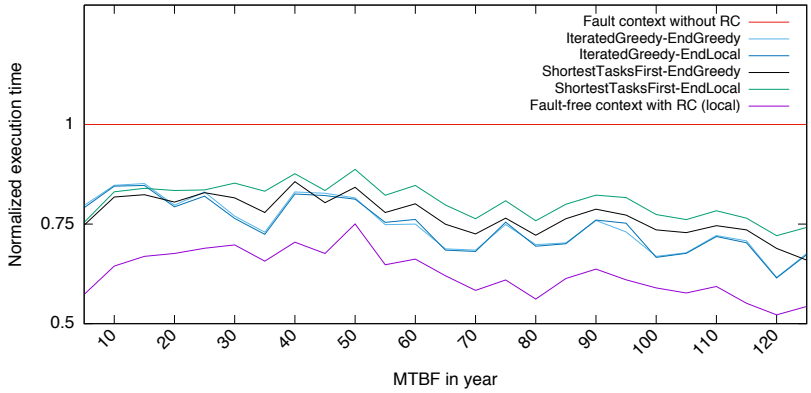


Figure 10: Impact of MTBF with $n = 100$ and $p = 1000$.

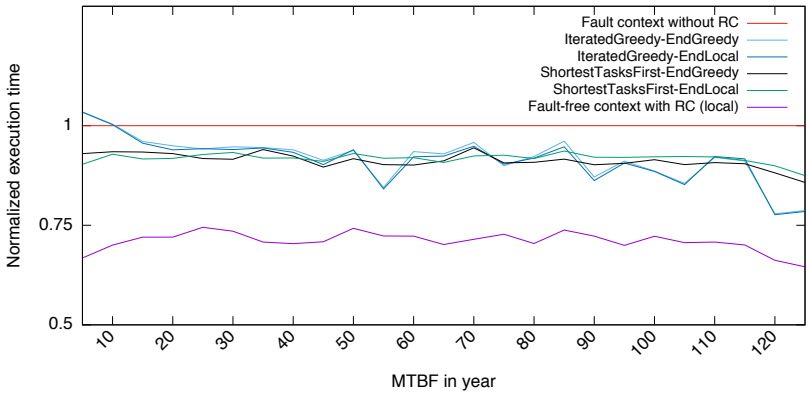


Figure 11: Impact of MTBF with $n = 100$ and $p = 5000$.

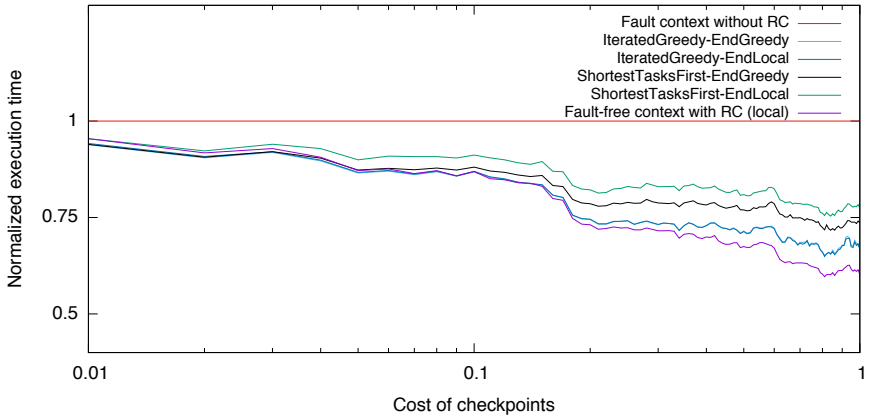
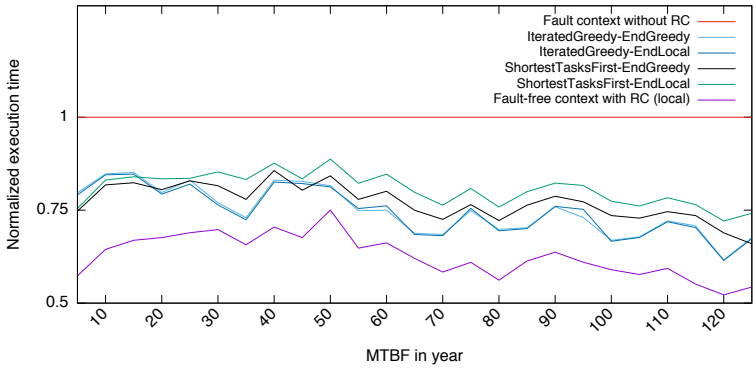
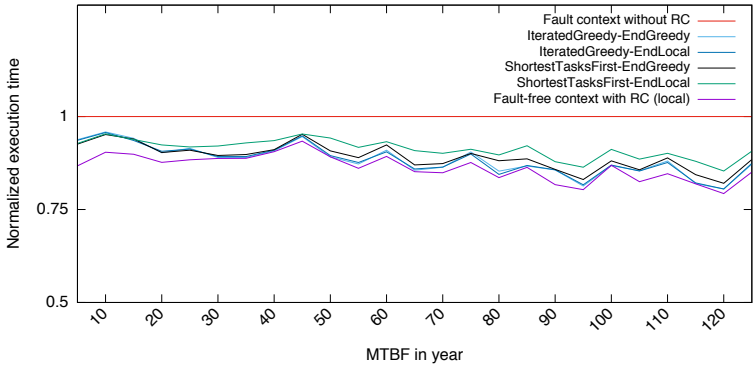


Figure 12: Impact of checkpointing cost.

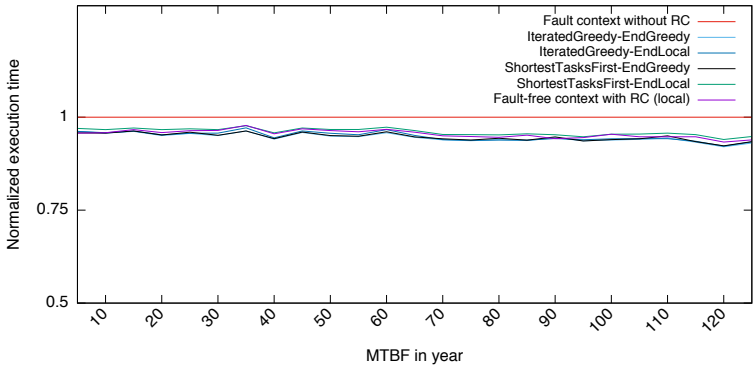
very efficient but also very costly. Nevertheless, when the MTBF is low (around 10 years or less), ShortestTasksFirst becomes better than IteratedGreedy. In a fault context, we gain flexibility from the failures and we can achieve a better load balance. We observe that the ratio between the number of tasks and the number of processors is important, because too many processors for few tasks leads to a deterioration of performance. We also show that the cost of checkpointing and the fraction of sequential time have a significant impact on performance. Note that all four heuristics run within a few seconds, while the total execution time of the application takes several days, hence even the more costly combination IteratedGreedy-EndGreedy incurs a negligible overhead.



(a) Original checkpoint cost $c = 1$.



(b) Checkpoint cost $c = 0.1$.



(c) Checkpoint cost $c = 0.01$.

Figure 13: Impact of checkpointing cost with $n = 100$ and $p = 1000$.

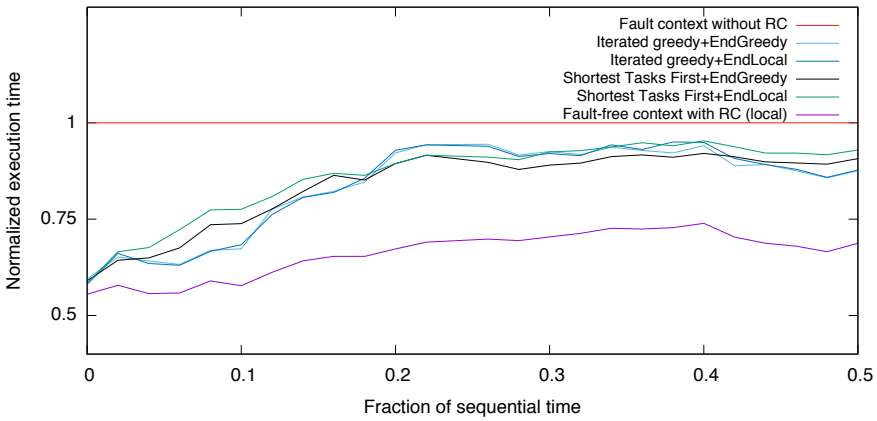


Figure 14: Impact of the sequential fraction of time with $n = 100$ and $p = 1000$ when $0 \leq f \leq 0.5$.

7 Conclusion

In this paper, we have designed a detailed and comprehensive model for scheduling a pack of tasks on a failure-prone platform with processor redistributions. We have introduced a greedy polynomial-time algorithm that returns the optimal solution when there are failures but no processor redistribution is allowed. We have shown that the problem of finding a schedule that minimizes the execution time when accounting for redistributions is NP-complete in the strong sense, even when there are no redistribution costs and no failures. Finally, we have provided several polynomial-time heuristics to redistribute efficiently processors at each failure or when an application ends its execution and releases processors. The heuristics are tested through a simulator that generates faults, and the results demonstrate their usefulness: a significant improvement of the execution time can be achieved thanks to the redistributions.

Further work will consider partitioning the tasks into several consecutive packs (rather than one) and conduct further simulations in this context. We also plan to investigate the complexity of the online redistribution algorithms in terms of competitiveness. It would also be interesting to deal not only with fail-stop errors, but also with silent errors. This would require to add verification mechanisms to detect such errors.

References

[1] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, USA, Research Report 5574, September 2009.

[2] M. Shantharam, Y. Youn, and P. Raghavan, "Speedup-aware co-schedules for efficient workload management," *Parallel Processing Letters*, vol. 23, no. 02, p. 1340001, 2013. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S012962641340001X

[3] G. Aupy, M. Shantharam, A. Benoit, Y. Robert, and P. Raghavan, "Co-scheduling algorithms for high-throughput workload execution," *Journal of Scheduling*, vol. To appear, 2015. [Online]. Available: http://arxiv.org/abs/1304.7793

[4] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sep. 2002. [Online]. Available: http://doi.acm.org/10.1145/568522.568525

[5] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Comm. of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.

[6] J. T. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *FGCS*, vol. 22, no. 3, pp. 303–312, 2004.

[7] J. Blazewicz, M. Drabowski, and J. Weglarz, "Scheduling multiprocessor tasks to minimize schedule length," *Computers, IEEE Transactions on*, vol. C-35, no. 5, pp. 389–393, May 1986.

[8] J. Du and J. Y.-T. Leung, "Complexity of scheduling parallel task systems," *SIAM Journal on Discrete Mathematics*, vol. 2, no. 4, pp. 473–487, 1989. [Online]. Available: http://dx.doi.org/10.1137/0402042

[9] J. Blazewicz, M. Machowiak, G. Mounié, and D. Trystram, "Approximation algorithms for scheduling independent malleable tasks," in *Euro-Par 2001 Parallel Processing*, ser. Lecture Notes in Computer Science, R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, Eds. Springer Berlin Heidelberg, 2001, vol. 2150, pp. 191–197. [Online]. Available: http://dx.doi.org/10.1007/3-540-44681-8_29

[10] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223. [Online]. Available: http://doi.acm.org/10.1145/277650.277725

[11] G. Martín, D. E. Singh, M.-C. Marinescu, and J. Carretero, "Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration," *Parallel Computing*, vol. 46, pp. 60 – 77, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819115000642

[12] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell, "Detection and correction of silent data corruption for large-scale high-performance computing," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 78:1–78:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389102

[13] X. Ni, E. Meneses, and L. Kale, "Hiding Checkpoint Overhead in HPC Applications with a Semi-Blocking Algorithm," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, Sept 2012, pp. 364–372.

[14] J. Dongarra, T. Hérault, and Y. Robert, "Performance and reliability trade-offs for the double checkpointing algorithm," *International Journal of Networking and Computing*, vol. 4, no. 1, pp. 23–41, 2014. [Online]. Available: http://www.ijnc.org/index.php/ijnc/article/view/71

[15] N. Muthuvelu, I. Chai, E. Chikkannan, and R. Buyya, "Batch resizing policies and techniques for fine-grain grid tasks: The nuts and bolts," *J. Information Processing Systems*, vol. 7, no. 2, 2011.

[16] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*. Springer International Publishing, 2015.

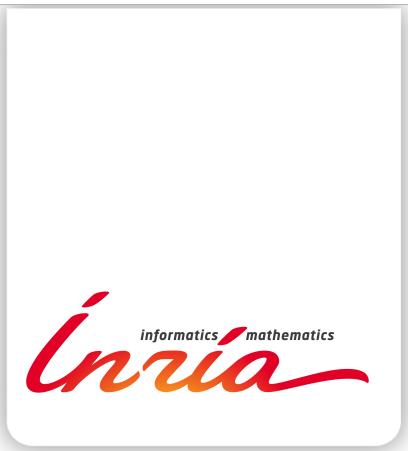
[17] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974. [Online]. Available: http://doi.acm.org/10.1145/361147.361115

[18] J. A. Bondy and U. S. R. Murty, *Graph theory with applications*. North Holland, 1976.

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co, 1979.

[20] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, "Checkpointing strategies for parallel jobs," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, Nov 2011, pp. 1–11.

[21] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, "Unified model for assessing checkpointing protocols at extreme-scale," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 17, pp. 2772–2791, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3173



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399