HAL
open science

Resilient co-scheduling of malleable applications

Anne Benoit, Loïc Pottier, Yves Robert

To cite this version:

Anne Benoit, Loïc Pottier, Yves Robert. Resilient co-scheduling of malleable applications. International Journal of High Performance Computing Applications, 2017, 10.1177/1094342017704979 . hal-01670153

HAL Id: hal-01670153

https://hal.science/hal-01670153v1

Submitted on 21 Dec 2017

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.

Resilient co-scheduling of malleable applications*

Anne Benoit[a], Loïc Pottier[a], Yves Robert[a,b]

[a] *Laboratoire LIP, École Normale Supérieure de Lyon, France*
[b] *University of Tennessee Knoxville, USA*

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 by frequent failures, and resilience techniques must be employed for large applications to execute efficiently. Indeed, failures may create severe imbalance between applications, and significantly degrade performance. In this paper, we aim at minimizing the expected completion time of a set of co-scheduled applications. We propose to redistribute the resources assigned to each application upon the occurrence of failures, and upon the completion of some applications, in order to achieve this goal. First, we introduce a formal model and establish complexity results. The problem is NP-complete for malleable applications, 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.

Keywords: Resilience; co-scheduling; redistribution; complexity results; heuristics; 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 studies. 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, thereby minimizing the loss due to the fact that applications are not perfectly parallel. Sharing the platform between two applications already leads to significant performance and energy savings [2], which become even more important when the number of co-scheduled applications increases [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 likely to occur 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.

Email addresses: `Anne.Benoit@ens-lyon.fr` (Anne Benoit), `Loic.Pottier@ens-lyon.fr` (Loïc Pottier), `Yves.Robert@inria.fr` (Yves Robert)

*A short version of this work appeared in *Proceedings of ICPP'16, Philadelphia, August 2016*, as "Resilient application co-scheduling with processor redistribution."

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 occurs, 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 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 endeavour with failures (and redistributions), while it was of linear complexity without failures. Also, designing efficient pack scheduling algorithms is needed whenever there are relatively few applications that can be all scheduled simultaneously, and it is a prerequisite before tackling the general problem. Given a pack, i.e., a set of parallel applications that start execution simultaneously, there are two main opportunities for redistributing processors. First, when an application completes, the applications that are still running can claim its processors. Second, when a failure strikes an application, that application is delayed. By adding more resources to it, we can reduce its final 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 an application 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: (i) the design of a detailed and comprehensive model for scheduling a pack of applications on a failure-prone platform; (ii) the NP-completeness proof for the problem with redistributions; (iii) 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

Parallel application models. A parallel application is an application that may use several processors during its execution. Note that the scheduling literature uses the term *parallel tasks* rather than *parallel application*. Many parallel application models have been developed, and several types of applications 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 applications on identical processors. The number of processors assigned to each application was fixed during the execution. They showed that the problem is NP-complete when the number of processors is not fixed. An application that has a fixed number of processors is called *rigid*. In 1989, Du and Leung [8] developed a model called the Parallel Task System, where an application is executed by one or more processors at the same time, but the number of processors assigned to one application 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 applications are called *moldable*. Finally, a *malleable* application can have its number of allocated processors vary during the execution. Błażewicz et al. [9] have designed approximation algorithms to solve the problem of scheduling independent malleable applications. Malleable applications are more flexible than rigid and moldable applications, and they can be implemented with data redistribution techniques (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 [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 applications that need them. We also show that the problem of finding a schedule that minimizes the execution time with fixed redistribution costs and without failures is NP-complete (in the strong sense).

Resilience. One of the most used techniques 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 for dealing with fail-stop errors is process replication, which consists in replicating a process and even replicating 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 loses 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 (which happens with very low probability), we have a fatal failure and the system cannot be recovered.

Co-scheduling algorithms. This work provides an important extension to our previous work on co-schedules [3], which already demonstrated that sharing the platform between two or more applications can lead to significant performance and energy savings [2]. To the best of our knowledge, it is the first work to consider co-schedules and failures, and hence to use malleable applications 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 where only partial information on the whole workload is known when taking scheduling decisions. On the contrary, co-scheduling applies to a set of applications 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 applications $\{T_1, \ldots, T_n\}$, and an execution platform with p identical processors subject to failures. We assume $n \leq 2p$ due to the use of the double checkpointing model. The objective is to minimize the expected completion time of the last application. First, we define the fault model in Section 3.1. Then, we show how to compute the execution time of an application in Section 3.2, assuming that no redistribution has occurred. The redistribution mechanism and its associated cost are discussed in Section 3.3. Finally, the objective function is detailed in Section 3.4.

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 an application depends upon the number of processors it is using, hence changes whenever a redistribution occurs. Specifically, if application T_i is (currently) executed on j processors, its MTBF is $\mu_{i,j} = \frac{\mu}{j}$ (see [16, Proposition 1.2] 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

application must be even. We enforce periodic checkpointing for each application. Formally, if application T_i is executed on j processors, there is a checkpoint every period of length $\tau_{i,j}$, with a cost $C_{i,j}$.

We now explain how to compute the cost $C_{i,j}$ of a checkpoint when application T_i executes with j processors. Recall that we use in-memory checkpointing. Let m_i be the memory footprint (total data size) of application T_i. Each of the j processors holds $\frac{m_i}{j}$ data, which it must send to its buddy processor. The time to communicate a message of size s is $\beta + \frac{s}{\tau}$, where β is a start-up latency and τ the link bandwidth. We derive that $C_{i,j} = \frac{m_i}{j\tau} + \beta$.

As for the checkpointing period $\tau_{i,j}$, we use Young's formula [17] and let

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

Because $\tau_{i,j}$ is a first order approximation, the formula is valid only if $C_{i,j} \ll \mu_{i,j}$. When a fault occurs, 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 first have to compute the expected execution time of an application T_i executed on j processors subject to failures. We first consider the case without redistribution (but taking failures into account). Let $t_{i,j}$ be the execution time of application 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 application T_i on j processors, with $0 \leq \alpha \leq 1$. We need to consider such a partial execution of T_i on j processors to prepare for the case with redistributions.

Recall that the execution of application 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 $\tau_{i,j} - C_{i,j}$, 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. We are able to calculate the expectation of one period of work according to an MTBF value and a number of processors. 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 (possibly incomplete) period. This last period is denoted as $\tau_{last}(\alpha)$ and defined as $\tau_{last}(\alpha) = \alpha t_{i,j} - N^{\text{ff}}_{i,j}(\alpha)(\tau_{i,j} - C_{i,j})$.

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:

$$t^R_{i,j}(\alpha) = e^{\lambda j R_{i,j}}\left(\frac{1}{\lambda j} + D\right)\left(N^{\text{ff}}_{i,j}(\alpha)(e^{\lambda j \tau_{i,j}} - 1) + (e^{\lambda j \tau_{last}(\alpha)} - 1)\right). \tag{3}$$

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:

$$t^R_{i,j+1}(\alpha) \leq t^R_{i,j}(\alpha) \text{ for } 1 \leq i \leq n,\ 1 \leq j < p,\ 0 \leq \alpha \leq 1. \tag{4}$$

However, when we allocate more processors to an application, even though the work is further parallelized, the probability of failures increases, and the corresponding waste increases as well. 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 (4), we restrict the number of processors assigned to each application, and never assign more processors than the previous threshold. In other words, if T_i is already assigned j processors, we consider assigning more processors to it only if $t^R_{i,j+1} \leq t^R_{i,j}$. Formally, this defines a maximum number of processors,

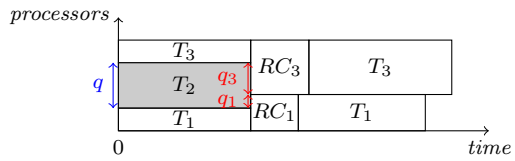


Figure 1: Redistribution at the end of an application, where RC_i represents the redistribution cost for task T_i.

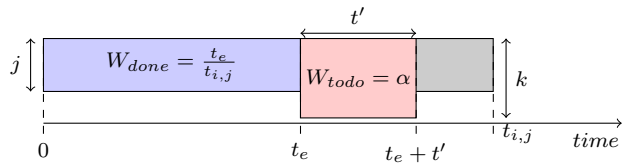


Figure 2: Work representation for application T_i at time t_e.

$j_{max}(i)$, for each application T_i: $j_{max}(i) = \min_{1 \le j \le p}\{j \text{ such that } t^R_{i,k} \ge t^R_{i,j} \text{ for all } k > j\}$, and we assume that $t^R_{i,j+1} \le t^R_{i,j}$ for all $j < j_{max}(i)$.

Another common assumption for malleable applications is that the work is non-decreasing when the number of processors increases [9]: this amounts to saying that no super-linear speed-up is possible. Hence, we assume here that for $1 \le i \le n$, $1 \le j < p$ and $0 \le \alpha \le 1$, $(j+1) \times t^R_{i,j+1}(\alpha) \ge j \times t^R_{i,j}(\alpha)$.

For convenience, we denote by t^U_i the current expected finish time of application T_i at any point of the execution. Initially, if application 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 for which it may be useful to redistribute processors: (1) in a fault-free scenario, when an application ends, it releases processors that can be used to accelerate other applications, and (2) when an error occurs, we may want to force the release of processors, so that we can assign more processors 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. Finally, we discuss the case of consecutive redistributions in Section 3.3.3.

3.3.1. Fault-free scenario

We first consider a simplified scenario without checkpoint (nor failure), in order to explain how redistribution works. Consider for instance that q processors are released when application T_2 ends. We can allocate q_1 new processors to application T_1, and q_3 new processors to application T_3, where $q_1 + q_3 = q$ (see Figure 1). This redistribution will take some time (redistribution cost RC_i, detailed below), after which T_1 and T_3 will resume execution, and we first need to compute the new expected completion time for their remaining fraction of work.

Consider that a redistribution is conducted at time t_e (the end time of an application), and that application 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 application 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 $k = j + q$ processors, the application T_i must redistribute its data across the processors. The application keeps its initial j processors, which now hold too much data, and enrolls $q = k - j$ new processors, which have no data yet. Recall that m_i is the memory footprint (total data size) of application T_i. Each of the original j processors initially holds $\frac{m_i}{j}$ data and will keep only $\frac{m_i}{k}$ after the redistribution; it sends $\frac{m_i}{jk}$ data to each of the newly enrolled q processors, thereby keeping $\frac{m_i}{j} - (k-j)\frac{m_i}{jk} = \frac{m_i}{k}$ data. In turn, each new processor receives $\frac{m_i}{jk}$ data from j processors and duly gets $\frac{m_i}{k}$ data in the end.

What is the best schedule for such a redistribution, and what time does it require? We first account for a constant start-up overhead S, paid for initiating the redistribution call. Then we adopt a realistic one-port communication model [18] where a processor can send and receive at most one message at any time-step. Independent communications, involving distinct sender/receiver pairs, can take place in parallel: however, two messages sent by the same processor will be serialized. Recall that the time to communicate a message of size s is $\beta + \frac{s}{\tau}$. To schedule the redistribution, we build a bipartite graph G with j nodes on

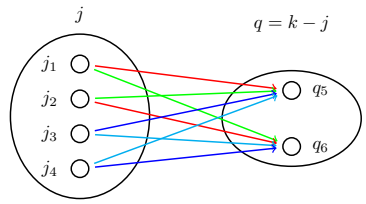


Figure 3: Bipartite graph G representing a redistribution from $j = 4$ to $k = 6$ processors, with each communication round colored. We have $\chi'(G) = \Delta(G) = 4$.

the left and q nodes on the right. In the one-port model, there can be up to j simultaneous communications (each of size $\frac{m_i}{jk}$) involving j distinct processor pairs. Let us call a *round* such a set of simultaneous (independent) communications. How many rounds are required to schedule the redistribution? We transform this problem into an edge coloring problem, with one color for one round (see Figure 3). The number of rounds required is equal to the edge chromatic number $\chi'(G)$. Konig's theorem [19] states that this edge chromatic number is equal to the maximum degree in G so $\chi'(G) = \Delta(G)$ when G is bipartite. Clearly, we have here $\Delta(G) = \max(j, k - j)$. Therefore, the number of rounds is equal to $\max(j, k - j)$, and the redistribution cost is $RC_i^{j \to k} = S + \max(j, k - j) \times \left(\frac{m_i}{jk\tau} + \beta\right)$.

Needless to say, we would perform a redistribution if the cost of redistribution is lower than the benefit of allocating new processors to the application, i.e., if $t_{i,j} - (t_e + t') > RC_i^{j \to k}$.

3.3.2. Accounting for failures

When struck by a fault, an application needs to recover from the failure and to re-execute some work. While the application loads were well-balanced initially in order to minimize total execution time, now the faulty application is likely to exceed its expected execution time. If it becomes the longest application of the schedule, we try to assign it more processors so as to reduce its completion time, hence redistributing processors.

Because we use the double checkpointing algorithm as the resilience model, we consider processors by pairs. We aim at redistributing pairs of processors either when an application is finished, at time t_e (as in the fault-free scenario discussed in Section 3.3.1), or when a failure occurs, say at time t_f. In each case, we need to compute the remaining work, and the new expected completion time of the applications that have been affected by the event. Given an application T_i, we keep track of the time when the last redistribution or failure occurred for this application, denoted as t_{lastR_i}. At time t (corresponding to the end of an application or to a failure), we know exactly how many checkpoints have been taken by application T_i executed on j processors since t_{lastR_i}, and we let this number be $N_{i,j}$:

$$N_{i,j} = \left\lfloor \frac{t - t_{lastR_i}}{\tau_{i,j}} \right\rfloor . \tag{5}$$

We begin with the case of an application completion: consider that an application finishes its execution at time t_e, hence releasing some processors. We consider assigning some of these processors to an application 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 for the second case, when a fault occurs. In this case, we need to consider the application T_i where the failure stroke, and other applications $T_{i'}$ from which we would remove some processors (in order to give them to T_i).

- Consider that application T_i is running on j processors and 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, consider application $T_{i'}$, on which we perform a redistribution, moving from j' to $j' - q$ processors, with $q > 0$. The fraction of work executed by $T_{i'}$ can be computed as in the application ending case scenario: it is $\frac{t_f - t_{lastR_{i'}} - N_{i',j'} C_{i',j'}}{t_{i',j'}}$.

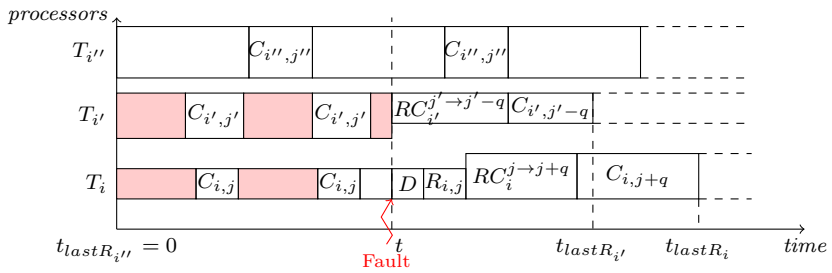


Figure 4: Example of redistribution when a fault strikes application T_i. The colored rectangles correspond to useful work done by T_i and $T_{i'}$ before the failure. $T_{i''}$ is not affected by the failure as it does not perform a redistribution.

Finally, for any application subject to a redistribution or a failure, let α_i 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).

Similarly to the fault-free scenario, $RC_i^{j\to k}$ denotes the redistribution cost for application T_i when moving from j to k processors. Redistribution can now add ($k > j$) or remove ($k < j$) processors to application T_i, and the cost is expressed as:

$$RC_i^{j\to k} = S + \max(\min(j,k), |k-j|) \times \left(\frac{m_i}{kj\tau} + \beta\right). \tag{6}$$

We are now ready to compute the new values of t_{lastR_i} for all applications subject to a failure or a redistribution, and we illustrate the different scenarios in Figure 4. Let t be the time of the event (end of application $t = t_e$, or failure $t = t_f$), and consider that a redistribution is done either for a faulty application T_i or for another application $T_{i'}$. After a redistribution, we always start by taking a checkpoint before computing with the new period. Therefore, if a fault occurs, we do not have to redistribute again.

For the faulty application T_i, the new value of t_{lastR_i} hence becomes $t_{lastR_i} = t + D + R_{i,j} + RC_i^{j\to k} + C_{i,k}$ (we need to account for the downtime and recovery). However, if $T_{i'}$ is performing a redistribution but it was not struck by a failure, it can start the redistribution at time t: either it is getting new processors that are available following the end of an application, or it is using fewer processors and can perform its redistribution. In all cases, we have $t_{lastR_{i'}} = t + RC_{i'}^{j'\to k'} + C_{i',k'}$. Note that we can have processors involved simultaneously in two redistributions, as they will only receive data from the other processors of the faulty application T_i, and send data to the other processors of the non-faulty application $T_{i'}$. We assume that sends and receives can be done in parallel without slowdown.

Finally, the expected finish time of an application T_i for which we have updated t_{lastR_i} becomes $t_i^U = t_{lastR_i} + t_{i,k}^R(\alpha_i)$, where k is the new number of processors on which T_i is executed, and α_i the remaining fraction of work. 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.

3.3.3. When multiple redistributions overlap

Here, we deal with the problem of chaining redistributions. If another event (application completion or fault) occurs during the current redistribution, we cannot enroll the processors that have not yet finished the current redistribution. On Figure 5, at the end of the application T_3, there are no available applications to whom we may try to give its processors. T_1 will be able to start a new redistribution at time-step t_1, and T_2 at time-step r_2.

3.4. Objective function

We can now state the objective function: Given n malleable applications $\{T_1, \ldots, T_n\}$, their speedup profiles, and an execution platform with p identical processors subject to failures with individual rate λ, CoSched aims at minimizing the maximum of the expected completion times of the applications. Redistributions are allowed only when an application completes execution or is struck by a failure (with a cost specified in Section 3.3).

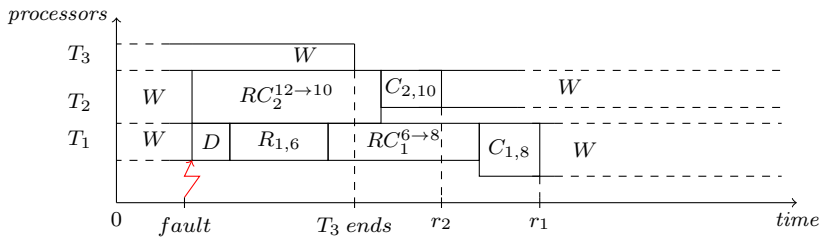


Figure 5: Illustration of two consecutive events.

4. Complexity results

We first consider the CoSched problem without redistributions and provide an optimal polynomial-time algorithm. Then, we prove that the problem becomes NP-complete with redistributions, even in a fault-free scenario.

Aupy et al. [3] designed a greedy algorithm to solve the problem with no redistribution, in a fault-free scenario. Their algorithm (called Optimal-1-pack-schedule) therefore works with $t_{i,j}$ values instead of $t^R_{i,j}$, and minimizes the execution time of the applications. As a minor detail, it does not take into account the fact that the number of processors assigned to an application must always be even in our setting, because we use the double checkpointing algorithm. It is not difficult to extend this algorithm to solve the problem with failures, but still without redistributions: the idea is to give initially two processors per applications, to sort them by expected execution time, and to greedily give two extra processors to the longest application, if it decreases its expected execution time. This algorithm is called Opt-NoRedistrib. We can therefore prove the following theorem (see [20] for the proof).

Theorem 1. *The* CoSched *problem without redistributions can be solved in polynomial time $O(n)$, where n is the number of applications.*

We show through a few examples the difficulty of CoSched when redistributions are allowed, even when there are no failures. The first example shows that the previous algorithm Opt-NoRedistrib is no longer optimal. Consider two applications T_1 and T_2 and three processors, and further assume that there is no cost for redistribution. We use the following speedup profiles:

$$T_1 = \begin{cases} t_{1,1} = 10, & w_{1,1} = 10 \\ t_{1,2} = 9, & w_{1,2} = 18 \\ t_{1,3} = 6, & w_{1,3} = 18 \end{cases} \qquad T_2 = \begin{cases} t_{2,1} = 6, & w_{2,1} = 6 \\ t_{2,2} = 3, & w_{2,2} = 6 \end{cases}$$

where $w_{i,j}$ represents the work for application i with j processors, i.e., $w_{i,j} = j \times t_{i,j}$.

Opt-NoRedistrib initially assigns one processor to each application, and then the remaining one to the longest application T_1. At time 6, when T_2 finishes and releases its processor, we redistribute T_1 over the three processors. At time 6, the application T_1 has done 2/3 of its work, it remains $1/3 \times t_{1,3} = 1/3 \times 6 = 2$ time units with 3 processors, therefore T_1 ends at time $6+2 = 8$ (see Figure 6a). We obtain a smaller makespan if we do not use Opt-NoRedistrib but instead the variant GreedySpeedupProfile, where remaining processors are allocated to the application with the best speedup profile. In the example, GreedySpeedupProfile initially allocates the third processor to T_2 because the execution time with two processors is divided by two, i.e., perfect speedup with $w_{2,2}/w_{2,1} = 1$. Then T_2 finishes at time 3. At this time, T_1 has still to complete 7/10 of its load, so the remaining time for T_1 is equal to $7/10 \times t_{1,3} = 7/10 \times 6 = 4.2$. The makespan in this second configuration becomes $3 + 4.2 = 7.2$, which is better!

Since Opt-NoRedistrib is no longer optimal, a natural question is whether GreedySpeedupProfile is optimal. The following example answers negatively. Consider the following speedup profiles (with two applications and three processors as before):

$$T_1 = \begin{cases} t_{1,1} = 10, & w_{1,1} = 10 \\ t_{1,2} = 6, & w_{1,2} = 12 \\ t_{1,3} = 5, & w_{1,3} = 15 \end{cases} \qquad T_2 = \begin{cases} t_{2,1} = 6, & w_{2,1} = 6 \\ t_{2,2} = 3, & w_{2,2} = 6 \end{cases}$$

(a) Opt-NoRedistrib uses largest execution time to allocate processors

(b) GreedySpeedupProfile uses best speedup profile to allocate processors

Figure 6: Examples: coordinates are execution time (x-axis) and processors (y-axis).

GreedySpeedupProfile allocates two processors to T_2 (best speedup profile) and one processor to T_1. So at time 3, the application T_2 completes and its two processors are given to T_1. The execution time for T_1 is $3 + 7/10 \times 5 = 6.5$. But if we allocate two processors to T_1 and one to T_2, we finish both applications at time 6 without any redistribution!

Intuitively, these little examples show that CoSched seems to be of combinatorial nature when redistributions are taken into account, even with zero cost.

To establish the complexity of the problem with redistributions, 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 a constant equal to S, i.e., we let $\beta = 0$ and $\tau = +\infty$ in Equation (6). Even in this simplified scenario, the problem is NP-complete:

Theorem 2. *With constant redistribution costs and without failures,* CoSched *is NP-complete (in the strong sense).*

The proof is available in the associated research report [20]. Note that we conjecture that CoSched remains NP-complete with zero redistribution cost. This is because of the combinatorial exploration suggested by the examples. But this remains an open problem!

5. Heuristics

In this section, we introduce polynomial-time heuristics to solve the general CoSched problem with both failures and redistributions. Before performing any redistribution, we need to choose an initial allocation of the p processors to the n applications. We use the optimal algorithm without redistribution discussed in Section 4 (Opt-NoRedistrib).

We first discuss the general structure of the heuristics. Then, we explain how to redistribute available processors, and the two strategies to redistribute when failures occur. The pseudo-codes for all algorithms are available in the companion research report [20].

General structure. All heuristics share the same skeleton: we iterate over each event (either a failure or an application termination) until total remaining work is equal to zero. If some applications are still working for a previous redistribution, (i.e., the current time t is smaller than t_{lastR_i} for these applications), then we exclude them for the next redistribution, and add them back into the list of applications after the current redistribution is completed. If an application ends, we redistribute available processors as will be discussed in Section 5. Then, if there is a failure, we calculate the new expected execution time of the faulty application. Also, we remove from the list the applications that end before t_{lastR_f}, and we release their processors.

Afterwards, we have to choose between trying to redistribute or do nothing. If the faulty application is not the longest application, 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 application is the longest application, we apply a heuristic to redistribute processors (see below).

Redistribution when an application ends. When an application ends, the idea is to redistribute the processors that it releases in order to decrease the expected execution time. The easiest way to proceed consists in adding processors greedily to the application with the longest execution time, as was done in Opt-NoRedistrib 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 application i that encountered a redistribution. Therefore, this heuristic, called EndLocal, 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 Opt-NoRedistrib again, 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 application, and we obtain an estimation of the expected finish time when each application is mapped on two processors. Similarly to Opt-NoRedistrib, we then add two processors to the longest application while we can improve it, accounting for redistribution costs.

Note that we effectively update the values of α_i and t_{lastR_i} for application T_i only if a redistribution was conducted for this application. 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 it whenever needed at the end of the procedure.

Redistribution when there is a failure. Similarly to the case of an application ending, we propose two heuristics to redistribute in case of failures. The first one, ShortestApplicationsFirst, takes only local decisions. First, we allocate the k available processors (if any) to the faulty application if that application is improvable. Then, if the faulty application is still improvable, we try to take processors from shortest applications (denoted T_s) in the schedule, and give these processors to the faulty application, until the faulty application is no longer improvable, or there are no more processors to take from other applications. We take processors from an application only if its new execution time is smaller than the execution time of the faulty application.

The second heuristic, IteratedGreedy, uses a modified version of the greedy algorithm that initializes the schedule (Opt-NoRedistrib) each time there is a failure, while accounting for the cost of redistributions. This is done similarly to the redistribution of EndGreedy explained above, except that we need to handle the faulty application differently to update the values of α_f and t_{lastR_f}.

6. Simulations

To assess 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 [21, 22]. 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 application (the $t_{i,j}$ value). We use a *synthetic* model to generate the execution profiles in order to represent a large set of scientific applications. The application 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:

$$t(m,q) = f \times t(m,1) + (1-f)\frac{t(m,1)}{q} + \frac{m}{q}log_2(m). \tag{7}$$

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

Finally, we assign to each application 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 applications is very heterogeneous. On the contrary, if m_{inf} is close to m_{sup}, the data distribution is homogeneous, in other words all applications have (almost) the same execution time. Unless stated otherwise, we set $m_{inf} = 1,500,000$ and $m_{sup} = 2,500,000$ to have execution times long enough so that several failures are likely to strike during execution. With such a value for m_{sup}, the longest execution time in a fault-free execution is around 100 days. We also consider two different data distribution cases, (i) very heterogeneous with $m_{inf} = 1,500$, and (ii) homogeneous with $m_{inf} = 2,499,000$.

The cost of checkpoints for an application T_i with j processors is $C_{i,j} = C_i/j$, where C_i is proportional to the memory footprint of the application. 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. The synchronisation cost value S is fixed to $S = 0$ for all following experiments. 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 applications, the checkpointing cost and the data distribution, in order 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

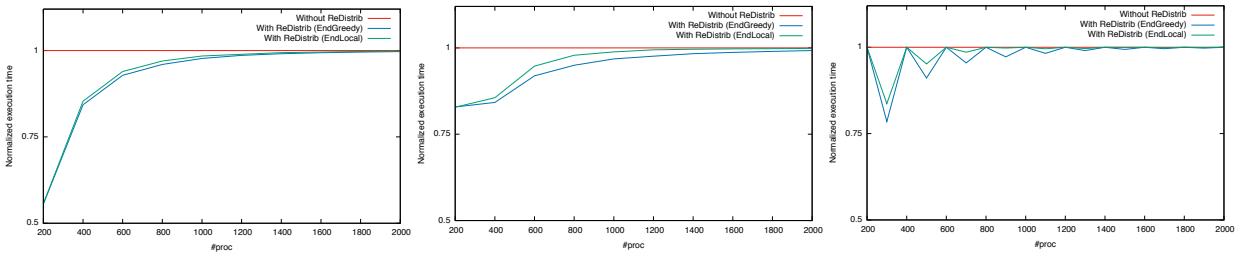
To evaluate the heuristics, we execute each heuristic $x = 50$ times and we compute the average *makespan*, i.e., the longest execution time in the pack. We compare the makespan obtained by the heuristics to the makespan (i) in a faulty context without any redistribution (worst case), and (ii) in a fault-free context with redistributions (best case). We normalize the results by the makespan obtained in a faulty 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 application termination and each failure; IteratedGreedy-EndLocal where we use EndLocal at each application termination, but IteratedGreedy in case of failures; ShortestApplications-First-EndGreedy where we greedily recompute a new schedule at each application termination, but use ShortestApplicationsFirst in case of failures; and ShortestApplicationsFirst-EndLocal where we only use the local variants.

Performance in a fault-free context. Figure 7 shows the impact of redistribution in a fault-free context with 100 applications, where we vary the number of processors from 200 to 2000. In this case, we compare EndLocal with EndGreedy (see Section 5). 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 decreases to converge to the performance without redistribution. Indeed, there are then enough processors so that each application does not make use of the extra processors released by ending applications. In the heterogeneous context (with $m_{inf} = 1500$), the gain due to redistribution is even larger.

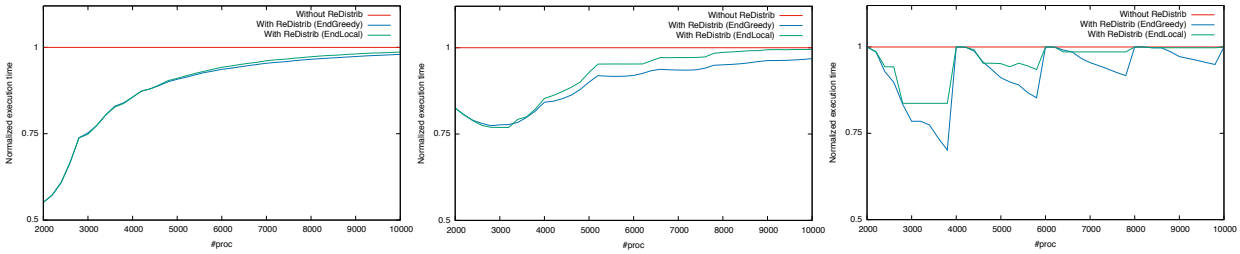
Figure 8 shows the impact of redistribution in a fault-free context with 1000 applications, we vary the number of processors from 2000 to 10000. We compare EndLocal with EndGreedy, the two heuristics have a similar behavior. As showed in Figure 7, the redistribution is more efficient in the heterogeneous context (with $m_{inf} = 1500$).

In the homogeneous case (Figure 8c), the results clearly show how the number of processors is directly linked to the efficiency of redistribution. If p/n is even, as each application has almost the same size, we assign p/n processors to each application. Consequently each application will finish at the same time and redistributions have very limited use. When p is not divisible by n or if p/n is odd, at least one application will have more processors than the others. So, at least one application will finish before the others and the redistribution will be more efficient. We note that the redistribution has a larger impact when few processors are involved, this is due to the fact that the speedup is better with fewer processors (sublinear speedup). In



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 7: Performance of redistribution in a fault-free context with $m_{sup} = 2500000$.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

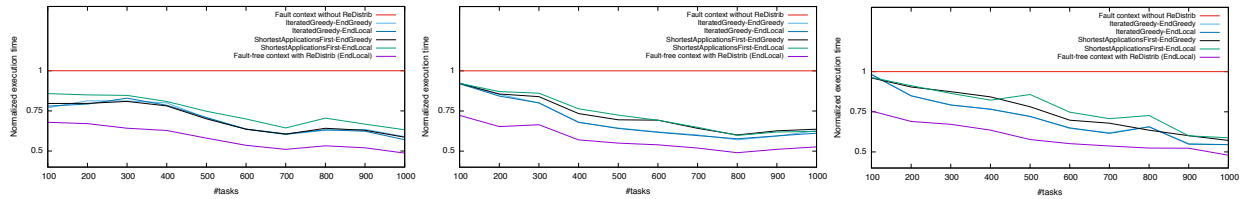
Figure 8: Performance of redistribution in a fault-free context with $m_{sup} = 2500000$.

other words, it is more useful to upgrade from 2 processors to 4 processors rather than from 500 to 502, in terms of speedup.

Impact of n. Figure 9 shows the impact of the number of applications n when the number of processors is fixed to 5000. The results show that having more applications increases the efficiency of both heuristics. With $n = 1000$, we obtain a gain of more than 40% due to redistributions. The reason is that when n increases, the number of processors assigned to each application decreases, then heuristics have more flexibility to redistribute.

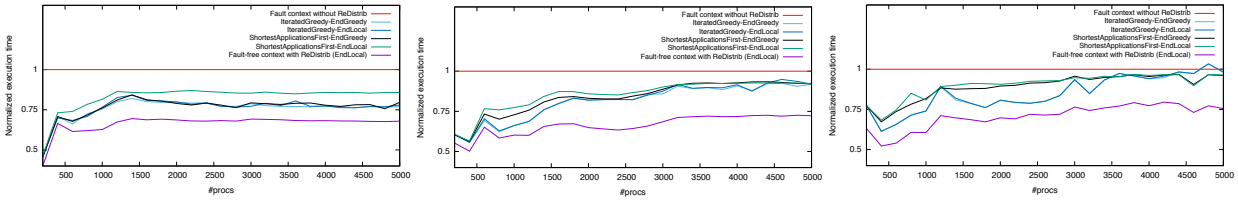
Note that, as expected, IteratedGreedy is better than ShortestApplicationsFirst, because it recomputes a complete new schedule at each fault, instead of just allocating available processors from shortest applications to the faulty application. Using EndGreedy with IteratedGreedy does not improve the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

We also observe that results in the heterogeneous cases are slightly better than in the homogeneous case, but the difference in the homogeneous case when $n = 1000$ is very tiny due to the large number of applications (i.e., fewer processors allocated to applications so the redistribution is more efficient).



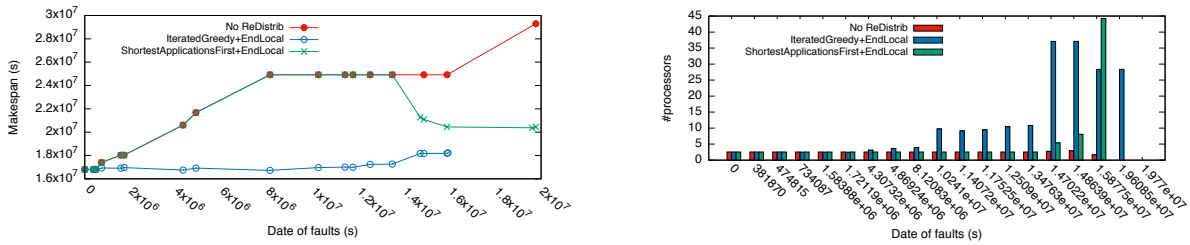
(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 9: Impact of n with $p = 5000$ processors.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 10: Impact of p with $n = 100$ applications and $m_{sup} = 2500000$.



(a) Makespan at each failure dealt with. (b) Standard deviation at each failure dealt with.

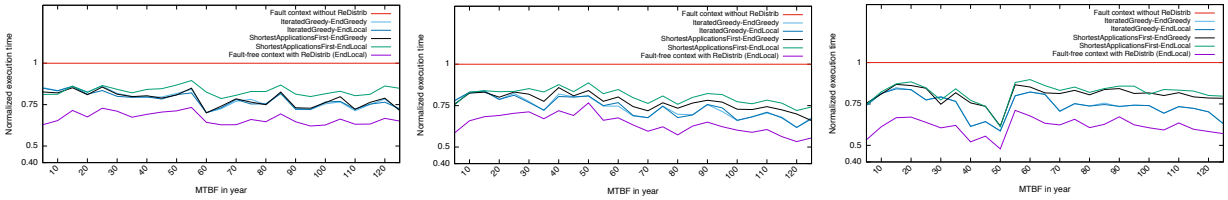
Figure 11: Heuristic behaviors with $n = 100$, $p = 1000$, MTBF of 50 years, for a single execution.

Impact of p. Figure 10 shows the impact of the number of processors p when the number of applications is fixed. We vary p between 200 and 5000 processors. The results show that having more processors decreases the efficiency of both heuristics, but, in the heterogeneous cases, there is always a gain of at least 10% thanks to redistributions. As noted in the fault-free case, the redistribution is more efficient when the data distribution is very heterogeneous (Figure 10a). On the contrary, in the homogeneous case (Figure 10c) the redistribution is less efficient (gain around 10%). The same observations hold, i.e., the use of EndGreedy vs EndLocal impacts only ShortestApplicationsFirst. In average, with IteratedGreedy, we obtain a gain of 25%, while ShortestApplicationsFirst 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 an application ($\mu_{i,j}$ value) decreases when the number of processors increases, so the gain obtained by the heuristics decreases due to the increasing number of failures.

Heuristic behaviors. Figure 11 compares IteratedGreedy and ShortestApplicationsFirst, when combined with EndLocal, on a single execution. We depict both the evolution of the makespan (see Figure 11a) and the standard deviation, in terms of number of processors (see Figure 11b). IteratedGreedy is clearly superior in terms of makespan, and this can be explained by the fact that it allocates more processors to the longest application, earlier in time than ShortestApplicationsFirst, hence resulting in a larger standard deviation. Because ShortestApplicationsFirst takes only local decisions, it takes more time before enough processors are given to the longest application.

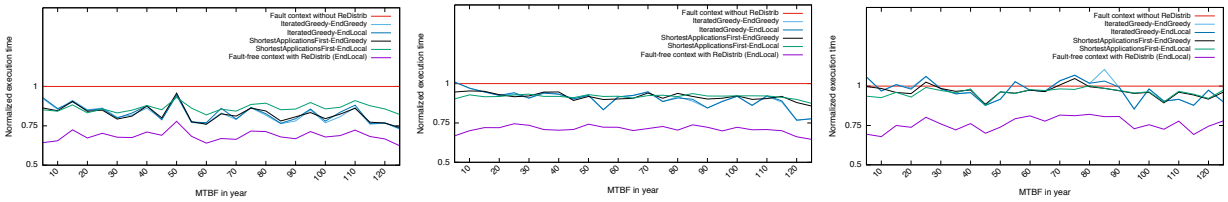
Impact of MTBF. Figures 12 and 13 show the impact of the MTBF on the performance of redistributions. We vary the MTBF of a single processor between 5 years and 125 years. When the MTBF decreases, the number of failures increases, consequently the performance of both heuristics decreases. In Figure 12, the performance of IteratedGreedy is closely linked to the MTBF value. Indeed, it tends to favor a heterogeneous distribution of processors (i.e., applications with many processors and applications with few processors). If an application is executed on many processors, its MTBF becomes very small and this application will be hit by more failures, hence it becomes even worse than without redistribution!

We observe the same result in Figure 13, especially in the homogeneous case (Figure 13c). This effect is even amplified due the number of processors ($p = 5000$) which directly decreases the MTBF and deteriorate the performance (increasing number of faults).



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 12: Impact of MTBF with $n = 100$, $p = 1000$, and $m_{sup} = 2500000$.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

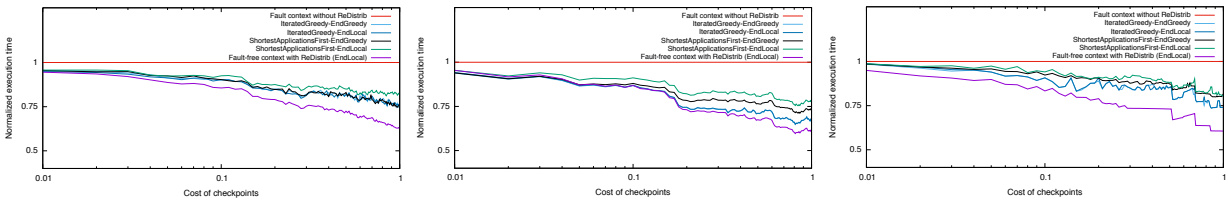
Figure 13: Impact of MTBF with $n = 100$, $p = 5000$, and $m_{sup} = 2500000$.

Impact of checkpointing cost. Figure 14 shows the impact of the checkpointing cost on a platform with 100 applications and 1000 processors. To do so, we multiply the checkpointing cost by c in Figure 14 (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 observe that when the checkpointing cost c tends to 1, the checkpointing costs are more important and the redistribution (specially IteratedGreedy) becomes more unstable. This effect is amplified in a homogeneous context, because applications and checkpoints are larger than in a heterogeneous context. We see the same effect on Figure 15.

Impact of the sequential fraction of time. Figure 16 shows the impact of the sequential fraction of time. We vary f from 0 (applications are fully parallel) to 0.5 (50% of the time is sequential). The results show that when applications are more parallel, the redistribution is more efficient. This result is expected, because if applications are not parallel, there is less gain when trying to allocate more processors to help them complete.

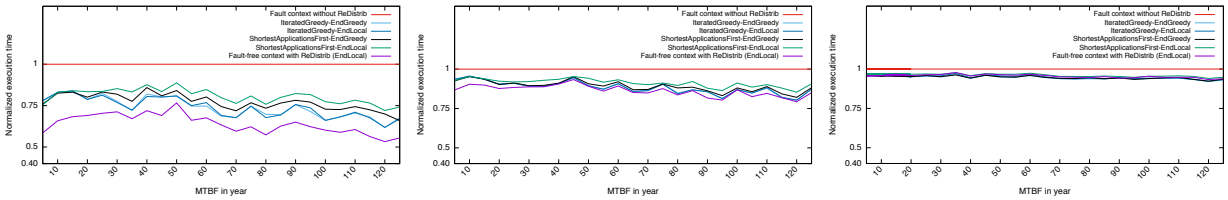
In the homogeneous case (Figure 16c), the IteratedGreedy heuristic is worse than the result without redistribution when f is greater than 0.3. It is due to the fact that all applications are large and not fully parallel, so when we greedily recompute a new schedule at each fault, we might deteriorate the performance.

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



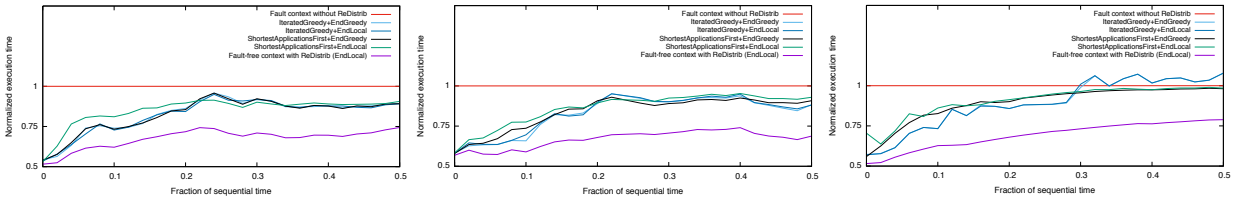
(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 14: Impact of checkpointing cost.



(a) Original checkpoint cost $c = 1$. (b) Checkpoint cost $c = 0.1$. (c) Checkpoint cost $c = 0.01$.

Figure 15: Impact of checkpointing cost with $n = 100$, $p = 1000$, and $m_{inf} = 1500000$.



(a) $m_{inf} = 1500$. (b) $m_{inf} = 1500000$. (c) $m_{inf} = 2499000$.

Figure 16: Impact of the sequential fraction of time with $n = 100$ and $p = 1000$ when $0 \leq f \leq 0.5$.

also costly. Nevertheless, when the MTBF is low (around 10 years or less), ShortestApplicationsFirst becomes better than IteratedGreedy. In a faulty context, we gain flexibility from the failures and we can achieve a better load balance. We observe that the ratio between the number of applications and the number of processors plays an important role, because having too many processors for few applications leads to a deterioration of performance (especially in a homogeneous context).

About the data distributions, we observe that the best context to take advantage of redistributions is a heterogeneous context with large and short applications. In the homogeneous context, when we assign the same weight to each application, redistributions become much less interesting. We also show that the cost of checkpointing and the fraction of sequential time have a significant impact on performance.

Finally, we point out 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.

7. Conclusion

In this paper, we have designed a detailed and comprehensive model for scheduling a pack of applications 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 with constant 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 extensive simulations, and the results demonstrate their usefulness: a significant improvement of the execution time can be achieved thanks to the redistributions.

There remains to validate the model on a real system by conducting real experiments, even though this is beyond the scope of this paper. Further work will also consider partitioning the applications into several consecutive packs (rather than one) and conduct further simulations (and experiments) in this context. On the theoretical side, we 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 adding 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, R. W. Numrich, Improving Performance via Mini-applications, Research Report 5574, Sandia National Laboratories, USA (September 2009).

[2] M. Shantharam, Y. Youn, P. Raghavan, Speedup-aware co-schedules for efficient workload management, Parallel Processing Letters 23 (02) (2013) 1340001. doi:10.1142/S012962641340001X.

[3] G. Aupy, M. Shantharam, A. Benoit, Y. Robert, P. Raghavan, Co-scheduling algorithms for high-throughput workload execution, Journal of Scheduling To appear.

[4] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, D. B. Johnson, A survey of rollback-recovery protocols in message-passing systems, ACM Computing Surveys 34 (3) (2002) 375–408.

[5] J. W. Young, A first order approximation to the optimum checkpoint interval, Comm. of the ACM 17 (9) (1974) 530–531.

[6] J. T. Daly, A higher order estimate of the optimum checkpoint interval for restart dumps, FGCS 22 (3) (2004) 303–312.

[7] J. Blazewicz, M. Drabowski, J. Weglarz, Scheduling multiprocessor tasks to minimize schedule length, Computers, IEEE Transactions on C-35 (5) (1986) 389–393. doi:10.1109/TC.1986.1676781.

[8] J. Du, J. Y.-T. Leung, Complexity of scheduling parallel task systems, SIAM Journal on Discrete Mathematics 2 (4) (1989) 473–487. doi:10.1137/0402042.

[9] J. Blazewicz, M. Machowiak, G. Mounié, D. Trystram, Approximation algorithms for scheduling independent malleable tasks, in: R. Sakellariou, J. Gurd, L. Freeman, J. Keane (Eds.), Euro-Par 2001 Parallel Processing, Vol. 2150 of LNCS, Springer Berlin Heidelberg, 2001, pp. 191–197.

[10] M. Frigo, C. E. Leiserson, 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, PLDI '98, ACM, New York, NY, USA, 1998, pp. 212–223. doi:10.1145/277650.277725.

[11] G. Martín, D. E. Singh, M.-C. Marinescu, J. Carretero, Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration, Parallel Computing 46 (2015) 60 – 77. doi:http://dx.doi.org/10.1016/j.parco.2015.04.003.

[12] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, R. Brightwell, Detection and correction of silent data corruption for large-scale high-performance computing, in: Proceedings of SC'12, 2012, pp. 78:1–78:12.

[13] X. Ni, E. Meneses, L. Kale, Hiding Checkpoint Overhead in HPC Applications with a Semi-Blocking Algorithm, in: Proceedings of CLUSTER'12, 2012, pp. 364–372. doi:10.1109/CLUSTER.2012.82.

[14] J. Dongarra, T. Hérault, Y. Robert, Performance and reliability trade-offs for the double checkpointing algorithm, International Journal of Networking and Computing 4 (1) (2014) 23–41.

[15] N. Muthuvelu, I. Chai, E. Chikkannan, R. Buyya, Batch resizing policies and techniques for fine-grain grid tasks: The nuts and bolts, Journal of Information Processing Systems 7 (2).

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

[17] J. W. Young, A first order approximation to the optimum checkpoint interval, Commun. ACM 17 (9) (1974) 530–531. doi:10.1145/361147.361115.

[18] P. B. Bhat, C. S. Raghavendra, V. K. Prasanna, Efficient collective communication in distributed heterogeneous systems, JPDC 63 (3) (2003) 251–263.

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

[20] A. Benoit, L. Pottier, Y. Robert, Resilient application co-scheduling with processor redistribution, Research report RR-8795, INRIA, available at graal.ens-lyon.fr/~abenoit (2015).

[21] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, F. Vivien, Checkpointing strategies for parallel jobs, in: Proceedings of SC'11, 2011, pp. 1–11.

[22] G. Bosilca, A. Bouteiller, E. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, D. Zaidouni, Unified model for assessing checkpointing protocols at extreme-scale, Concurrency and Computation: Practice and Experience 26 (17) (2014) 2772–2791. doi:10.1002/cpe.3173.

Anne Benoit received the PhD degree from Institut National Polytechnique de Grenoble in 2003, and the Habilitation à Diriger des Recherches (HDR) from École Normale Supérieure de Lyon (ENS Lyon) in 2009. She is currently an associate professor in the Computer Science Laboratory LIP at ENS Lyon, France. She is the author of 38 papers published in international journals, and 78 papers published in international conferences. She is the advisor of 8 PhD theses. Her research interests include algorithm design and scheduling techniques for parallel and distributed platforms, and also the performance evaluation of parallel systems and applications, with a focus on energy awareness and resilience. She is Associate Editor of IEEE TPDS, JPDC, and SUSCOM. She is the program chair of several workshops and conferences, in particular she is program chair for HiPC'2016, program co-chair for ICPP'2017, and technical papers chair for SC'2017. She is a senior member of the IEEE, and she has been elected a Junior Member of Institut Universitaire de France in 2009.

Loïc Pottier completed his master at the University of Versailles in 2015, and then moved to École Normale Supérieure de Lyon (ENS Lyon), where he is currently a PhD candidate under the supervision of Anne Benoit and Yves Robert. As part of completing his PhD, he also spent three months as visiting student at Argonne National Laboratory, where he worked with Swann Perarneau. His main topics of interest include co-scheduling, fault tolerance, and scheduling techniques for large scale platforms.

Yves Robert received the PhD degree from Institut National Polytechnique de Grenoble. He is currently a full professor in the Computer Science Laboratory LIP at ENS Lyon. He is the author of 7 books, 150 papers published in international journals, and 220 papers published in international conferences. He is the editor of 11 book proceedings and 13 journal special issues. He is the advisor of 30 PhD theses. His main research interests are scheduling techniques and resilient algorithms for large-scale platforms. Yves Robert served on many editorial boards, including IEEE TPDS and JPDC. He was the program chair of HiPC'2006 in Bangalore, IPDPS'2008 in Miami, ISPDC'2009 in Lisbon, ICPP'2013 in Lyon and HiPC'2013 in Bangalore. He is a Fellow of the IEEE. He has been elected a Senior Member of Institut Universitaire de France in 2007 and renewed in 2012. He has been awarded the 2014 IEEE TCSC Award for Excellence in Scalable Computing, and the 2016 IEEE TCPP Outstanding Service Award. He holds a Visiting Scientist position at the University of Tennessee Knoxville since 2011.