Resilient application co-scheduling with processor redistribution

Anne Benoit*, Loïc Pottier*, Yves Robert*†
*Ecole Normale Supérieure de Lyon & Inria, France
†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 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.**

***Index Terms*—Resilience; co-scheduling; redistribution; complexity results; heuristics; simulations.**

I. 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.

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 [7], [8] to allow redistributions of processors between applications. More related work on models for parallel applications and resilience are discussed in the companion research report [9].

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 [10] 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.

The rest of the paper is organized as follows. The model and the optimization problem are formally defined in Section II. In Section III, we expose the complexity results. We introduce some polynomial-time heuristics in Section IV, which are assessed through simulations using a fault generator in Section V. Finally, we conclude and provide directions for future work in Section VI.

II. Framework

We consider a pack of n independent malleable applications $\{T_1, \dots, T_n\}$, and an execution platform with p identical processors subject to failures. We assume $n \leq p$ in failure-free problems and $2n \leq p$ when accounting for failures, 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 II-A. Then, we show how to compute the execution time of an application in Section II-B, assuming that no redistribution has occurred. The redistribution mechanism and its associated cost are discussed in Section II-C, and the objective function is detailed in Section II-D.

A. 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 [11, Proposition 1.2] for a proof).

To recover from fail-stop errors, we use a light-weight checkpointing protocol called the *double checkpointing algorithm*, or *buddy algorithm* [12], [13]. 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 (which happens with very low probability), we have a fatal failure and the system cannot be recovered. Note that 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. 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 [14] 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 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.

B. 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 $\left(\frac{1}{\lambda j} + D\right)(e^{\lambda j T} - 1)$ [11]. 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 τ_{last} and defined as:

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

Note that τ_{last} is depending on α because τ_{last} represents the incomplete fraction of $\tau_{i,j} - C_{i,j}$ at the end of the application. 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 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 (5), 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, $j_{max}(i)$, for each application T_i:

$$j_{max}(i) = \min_{1 \leq j \leq p} \{ j \text{ such that } t^R_{i,k} \geq t^R_{i,j} \text{ for all } k > j \}, \tag{6}$$

and we assume that $t^R_{i,j+1} \leq t^R_{i,j}$ 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 [7]: this amounts to say that no super-linear speed-up is possible. Hence, we assume here 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 application T_i at any point of the execution. Initially, if application T_i is allocated to j processors, we have $t^U_i = t^R_{i,j}(1)$.

C. 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 strikes, 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.

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 $q_1 + q_3 = q$. 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 1). 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 m_i data across the processors. The application keeps its initial j processors, which now hold

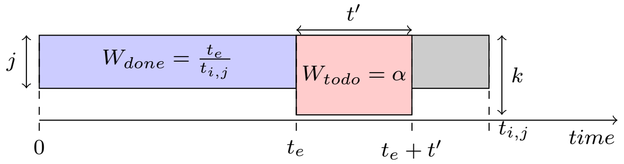

Figure 1: Work representation for application T_i at time t_e.

too much data, and enrolls $q = k - j$ new processors, which have no data yet. 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 [15] 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 the left and q nodes on the right, and we count the number of rounds required to schedule the redistribution. Thanks to Konig's theorem [16], we obtain a number of rounds equal to $\max(j, k-j)$ (see [9] for details), and the redistribution cost is

$$RC_i^{j \to k} = S + \max(j, k-j) \times \left(\frac{m_i}{jk\tau} + \beta\right). \qquad (7)$$

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

$$t_{i,j} - (t_e + t') > RC_i^{j \to k}.$$

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 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 II-C1), 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 $N_{i,j}$:

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

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 $\frac{t_f - t_{lastR_{i'}} - N_{i',j'}C_{i',j'}}{t_{i',j'}}$.

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:

$$RC_i^{j \to k} = S + \max(\min(j,k), |k-j|) \times \left(\frac{m_i}{kj\tau} + \beta\right). \qquad (9)$$

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 2. 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

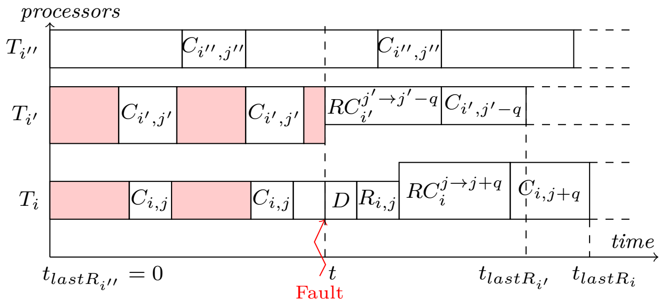

Figure 2: Example of redistribution when a fault strikes application T_i. The colored rectangles correspond to useful work done by T_i and $T_{i'}$ before the failure. $T_{i''}$ is not affected by the failure, since it does not perform a redistribution.

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 is is using less 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.

Note that we consider that we cannot enroll processors that have not yet finished the current redistribution, i.e., if an event happens between t and $t_{lastR_{i'}}$ in Figure 2, the processors involved in T_i and $T_{i'}$ cannot be considered for a new redistribution.

D. Objective function

We can now state the objective function:

Definition 1 (CoSched). *Given n malleable applications $\{T_1, \dots, T_n\}$, their speedup profiles, and an execution platform with p identical processors subject to failures with individual rate λ, minimize 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 II-C).*

III. Complexity results

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

A. Without redistributions

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 $t_{i,j}^R$, 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:

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

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 unique function σ, because the number of processors remains identical throughout the whole execution.

The fraction of work that each application 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 application while its expected execution time can be decreased. If we cannot decrease the expected execution time of the longest application, then we cannot decrease the overall expected execution time, which is the maximum of the expected execution times of all applications.

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 8 to check whether there is hope to decrease the expected execution time of the longest application. If $T_{i^\star}$ has reached its maximum enrollment with $\sigma(i^\star)$ processors (according to the threshold $j_{max}(i^\star)$ defined with Equation (6)), then we cannot decrease its expected execution time. In the following,

Algorithm 1: Optimal schedule with no redistribution.

```
procedure Optimal-Schedule($n$, $p$) begin
    for $i = 1$ to $n$ do  $\sigma(i) := 2$ ;
    Let $L$ be the list of applications sorted in
      non-increasing values of $\preccurlyeq_{\sigma}^{R}$;
    $p_{available} := p - 2n$;
    while $p_{available} \geq 2$ do
        $T_{i^\star} := head(L)$;
        $L := tail(L)$;
        if $\sigma(i^\star) < j_{max}(i^\star)$ then
            $\sigma(i^\star) := \sigma(i^\star) + 2$;
            $L :=$ Insert $T_{i^\star}$ in $L$ according to its $\preccurlyeq_{\sigma}^{R}$ value;
            $p_{available} := p_{available} - 2$;
        else $p_{available} := 0$;
    end
    return $\sigma$;
end
```

in such situations, we aim at making good use of extra processors through redistributions.

B. With redistributions

We can easily build examples to show the difficulty of CoSched when redistributions are allowed, even when there are no failures: (i) Algorithm 1 is no longer optimal because it may give processors to an application with a poor speedup profile (i.e., it does not gain much from the additional processors); and (ii) the greedy variant where remaining processors are allocated to the application with the best speedup profile can also lead to non-optimal schedules (see the companion research report [9] for details). 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 (9). 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).*

Proof. We consider the associated decision problem: given a bound on the execution time D, is there a schedule whose expected execution time does not exceed D? The problem is obviously in NP: with n applications, there are at most $n-1$ redistributions, hence n intervals during which processor assignment remains constant for all applications. Given a schedule and the list of resources assigned to each application within these n intervals, it is easy to check in polynomial time that it is valid and that its execution time does not exceed the bound D.

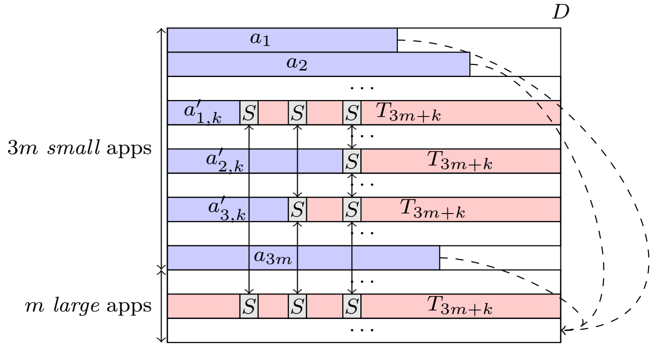

Figure 3: Illustration for the proof of Theorem 2.

To establish the completeness, we use a reduction from 3-PARTITION [17] with distinct integers (which remains strongly NP-complete [18, Corollary 7]). We consider an instance $\mathcal{I}_1$ of 3-PARTITION: given an integer B and $3m$ distinct 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 $\sum_{i=1}^{3m} a_i = mB$, 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 $\sum_{i \in I_j} a_i = B$? Letting $M = \max_{1 \leq i \leq 3m}(a_i)$, we can assume w.lo.g. that $B \leq 3M$, otherwise there is no solution to $\mathcal{I}_1$.

We build an instance $\mathcal{I}_2$ of our problem, with $n = 4m$ applications and $p = n$ processors. We let $D = 3M + 2$ be the bound on the execution time. For each redistribution, each application whose processor number changes, simply pays the constant overhead $S = \frac{1}{9m} < 1$ (communication costs are set to zero). 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* applications, and the work is strictly larger when using more than one processor). The last m applications are identical, with the following execution times: for $3m+1 \leq i \leq 4m$, $t_{i,j} = \frac{4D-B-9S}{j}$ for $1 \leq j \leq 4$, and $t_{i,j} = \frac{2}{9}(4D - B - 9S)$ for $j > 4$ (these are *large* applications with a total work equal to $4D - B - 9S$ 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 applications. Note that $4D - B - 9S > D$. 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 application has a single processor. When an application T_i finishes its execution (at time a_i), with $1 \leq i \leq 3m$, its processor is redistributed to application T_{3m+k}, given that $a_i \in I_k$. Both the single processor of T_i and each currently enrolled processor of T_{3m+k} pay a time overhead S for this redistribution, see Figure 3 for an illustration. Because the a_i's are all distinct, the successive redistributions occur at different time-steps, and the redistribution intervals of size S do not overlap. Each application T_{3m+k} starts with 1 processor and proceeds first with 2 processors (then paying an overhead S for its single processor before the redistribution), then with 3 processors (then paying an overhead S for each of its two processors before the redistribution), and finally with 4 processors (then paying an overhead S for each of its three processors before the redistribution) for some time in the end of its execution, because $M + S < D$. The total overhead due to the redistributions involving the three small tasks giving resources to T_{3m+k} is therefore $9S$. Now, each application T_{3m+k} always executes with an optimal work profile, and actually completes its execution in time D. Indeed, the 4 processors finally assigned to T_{3m+k} have to complete a total work of $a'_{1,k} + a'_{2,k} + a'_{3,k} + 4D - B - 9S = 4D - 9S$, and there are exactly $3(D - S) + D - 6S = 4D - 9S$ time slots available for computations. Again, because $M + S < D$, all small applications also complete before the deadline, and we have a solution to $\mathcal{I}_2$.

Suppose now that $\mathcal{I}_2$ has a solution. Initially, we have one processor per application, because there are exactly n processors and n applications. We first show that each small application T_i terminates before the end of the schedule, and that its processor must be redistributed. Indeed, $a_i \leq M < D$, and if we do not redistribute the processor assigned to T_i when it completes, then this processor stays idle for $D - a_i > D - M$ time steps. But the total work to execute is at least $\sum_{i=1}^{3m} a_i + m \times (4D - B - 9S) = m(4D - 9S)$, assuming perfect parallelism. If the remaining $n - 1$ processors work all the time, they contribute for $(n-1)D$. If the processor assigned to T_i works at most M time-steps, we must have $m(4D - 9S) \leq (n-1)D + M$, or equivalently $9mS \geq D - M$. But $D - M > 2$, and $9mS \leq 1$ by definition of S, a contradiction.

Because the a_i's are distinct, the $3m$ redistributions at the end of the $3m$ small tasks do not overlap. The first m redistributions involve at least another application running on one processor, which also looses S time-steps. The next m redistributions involve at least another application running on two processors, which costs $2S$ work units, and finally the last m redistributions involve at least another application running on three processors, hence costing $3S$ work units. Altogether, we have at least $9mS$ work units for redistribution costs. But the total work is at least $nD - 9mS$, and the area of the computing window is nD. This means that we pay exactly $9mS$ for redistributions, and that all the work is perfectly parallel. We now draw two consequences:

- When a small task completes, the redistribution of its processor involves a single other application (otherwise we would end with strictly more than $9mS$ redistribution overhead).
- This processor is redistributed to a large application, because all the work is perfectly parallel.

There are $3m$ processors to redistribute to m large applications, and none of them can receive more than 3 processors, again because all the work is perfectly parallel. Hence, each large application is assigned exactly 3 new processors throughout its execution. Formally, for $1 \leq k \leq m$, the large application T_{3m+k} receives processors from 3 small applications T_i with $i \in I_k = \{a'_{1,k}, a'_{2,k}, a'_{3,k}\}$, for $k \in \{1, \ldots, m\}$. The total work of these four processors is $4D - B - 9S + a'_{1,k} + a'_{2,k} + a'_{3,k}$ and there are $4D - 9S$ available time-steps for them. Hence $a'_{1,k} + a'_{2,k} + a'_{3,k} \leq B$. This is true for all triplets of small applications, and because $\sum_{i=1}^{3m} a_i = mB$, we must have an equality for each triplet, hence the solution to $\mathcal{I}_1$. □

Remark. 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!

IV. 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 III-A (Algorithm 1).

We first discuss the general structure of the heuristics in Section IV-A. Then, we explain how to redistribute available processors in Section IV-B, and the two strategies to redistribute when failures occur in Section IV-C. The pseudo-codes for all algorithms are available in the companion research report [9].

A. 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 IV-B. 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 Section IV-C).

B. 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 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 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 the greedy algorithm Algorithm 1 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 Algorithm 1, 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.

C. 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 (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 IV-B, except that we need to handle the faulty application differently to update the values of α_f and t_{lastR_f}.

V. Simulations

To assess the efficiency of the heuristics defined in Section IV, we have performed extensive simulations. The simulation settings are discussed in Section V-A, and results are presented in Section V-B. 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.

A. 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 [19], [20]. 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:

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

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 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: $m_{inf} \leq m_i \leq m_{sup}$. 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 $m_{inf} = 2,499,000$, and detailed results for these distributions are available in [9].

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.

Note that we assume that a failure can strike during checkpoints but not during downtime, recovery and while the processor is performing some redistribution.

B. Results

To evaluate the heuristics, we execute each heuristic 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

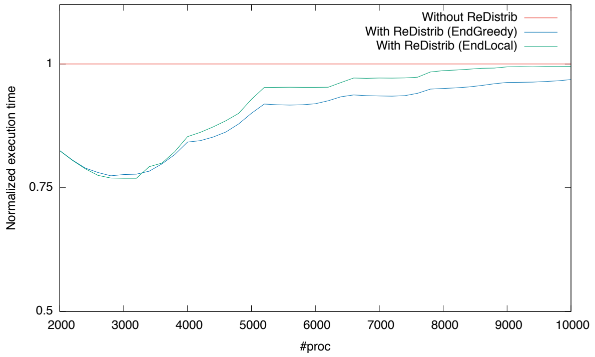

Figure 4: Redistribution in a fault-free context.

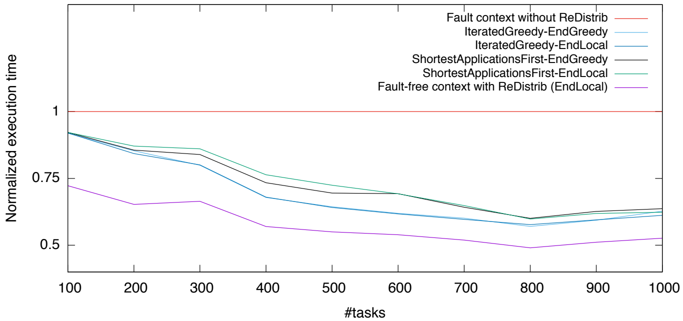

Figure 5: Impact of n with $p = 5000$ processors.

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 all four possible combinations of EndLocal or EndGreedy with ShortestApplicationsFirst or IteratedGreedy.

a) Performance in a fault-free context: Figure 4 shows the impact of redistribution in a fault-free context with 1000 applications, where we vary the number of processors from 2000 to 10000. In this case, we compare EndLocal with EndGreedy (see Section IV-B). The two heuristics have a very similar behavior, leading to a gain of more than 20% with less than 4000 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 (see [9]).

b) Impact of n: Figure 5 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.

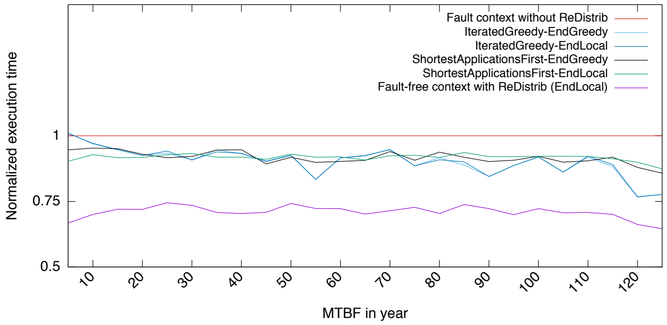

Figure 6: Impact of MTBF with $n = 100$ and $p = 5000$.

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. Similar results can be observed in the homogeneous and heterogeneous cases, and similar conclusions are drawn when varying p for a fixed value of n (see [9]).

c) Impact of MTBF: Figure 6 shows 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. 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!

d) Impact of checkpointing cost: Figure 7 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 7 (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.

Additionally, we show in [9] that the sequential fraction of time f of the applications also has an impact on performance: as expected, when applications are more parallel, the redistribution is more efficient.

e) Summary: Altogether, we observe that IteratedGreedy achieves better performance than Shortestap-

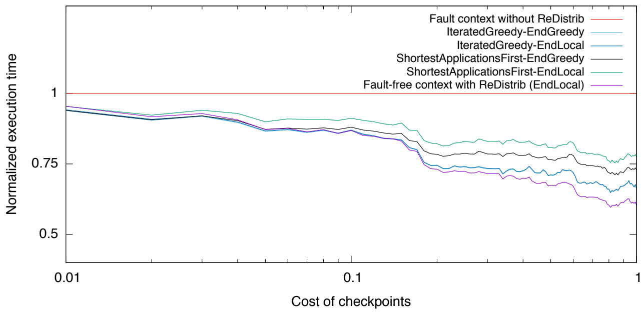

Figure 7: Impact of checkpointing cost.

plicationsFirst, mainly because it rebuilds a complete schedule at each fault, which is very efficient but 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. 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.

VI. 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.

Further work will consider partitioning the applications into several consecutive packs (rather than one) and conduct additional 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 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, 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.

[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.

[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 Computing Surveys*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

[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. Machowiak, G. Mounié, and D. Trystram, “Approximation algorithms for scheduling independent malleable tasks,” in *Euro-Par 2001 Parallel Processing*, ser. LNCS, R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, Eds. Springer Berlin Heidelberg, 2001, vol. 2150, pp. 191–197.

[8] 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.

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

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

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

[12] X. Ni, E. Meneses, and L. Kale, “Hiding Checkpoint Overhead in HPC Applications with a Semi-Blocking Algorithm,” in *Proceedings of CLUSTER’12*, Sept 2012, pp. 364–372.

[13] 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.

[14] J. W. Young, “A first order approximation to the optimum checkpoint interval,” *Commun. ACM*, vol. 17, no. 9, pp. 530–531, Sep. 1974.

[15] P. B. Bhat, C. S. Raghavendra, and V. K. Prasanna, “Efficient collective communication in distributed heterogeneous systems,” *JPDC*, vol. 63, no. 3, pp. 251–263, 2003.

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

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

[18] H. Hulett, T. G. Will, and G. J. Woeginger, “Multigraph realizations of degree sequences: Maximization is easy, minimization is hard,” *Op. Research Letters*, vol. 36, no. 5, pp. 594–596, 2008.

[19] M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien, “Checkpointing strategies for parallel jobs,” in *Proceedings of SC’11*, Nov 2011, pp. 1–11.

[20] 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.