Co-Scheduling High-Performance Computing Applications

Guillaume Aupy, Anne Benoit, Loïc Pottier, Padma Raghavan, Yves Robert, Manu Shantharam

Big Data: Management, Architecture, and Processing

Contents

CHAPTER 1

Co-Scheduling High Performance Computing Applications

Guillaume Aupy
Vanderbilt University, USA

Anne Benoit
LIP, ENS Lyon, France

Loic Pottier
LIP, ENS Lyon, France

Padma Raghavan
Vanderbilt University, USA

Yves Robert
LIP, ENS Lyon, France and University of Tennessee Knoxville, USA

Manu Shantharam
San Diego Supercomputer Center, USA

CONTENTS

BIG DATA applications play an increasing role in High Performance Computing (HPC). They are perfect candidates for co-scheduling, as they obey flexible speedup models, alternating I/O operations and intensive computation phases. In this chapter, we discuss 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 all applications. Co-scheduling usually involves partitioning the applications into *packs*, and then scheduling each pack in sequence, as efficiently as possible. The objective is therefore to determine a partition into packs, and an assignment of processors to applications, that minimize the sum of the execution times of the packs. On the theoretical side, we assess the problem complexity. On the practical side, we design several polynomial-time heuristics to deal with the general problem with failures and redistribution costs. The proposed heuristics show very good performance while executing in very short time, hence validating the approach.

1.1 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 [23] 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.

Big data applications play an increasing role in HPC. They are perfect candidates for co-scheduling, as they obey flexible speedup models a la BSP, alternating I/O operations and intensive computation phases. 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 [31], which become even more important when the number of co-scheduled applications increases [1].

Furthermore, 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 [22], 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 [17]. 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. The Young and Daly formulas [32, 13] provide good approximations of the optimal checkpointing interval.

In this chapter, we discuss 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. The objective is therefore to determine a partition into packs, and an assignment of processors to applications, that minimize the sum of the execution times of the packs. 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 run-

ning can claim its processors. Second, when a failure strikes a task, that task 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 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:

1. the NP-completeness proof for the general partitioning problem with $k \geq 3$ tasks per pack in a fault-free context, and an approximation algorithm;
2. the design of a detailed and comprehensive model for scheduling a given pack of tasks on a failure-prone platform;
3. an optimal algorithm to assign processors to applications when the tasks that form a pack are given and when no redistributions can be done;
4. the NP-completeness proof for the problem with redistributions;
5. the design and assessment of several polynomial-time heuristics to deal with the general problem with failures and redistribution costs. These heuristics show very good performance while executing in very short time, hence validating the approach.

The chapter is organized as follows. We discuss related work in Section 1.2. The problem is then formally defined in Section 1.3. Theoretical results are presented in Section 1.4, exhibiting the problem complexity, discussing subproblems and optimal solutions, and providing an approximation algorithm. Building upon these results, several polynomial-time heuristics are described and thoroughly evaluated in Section 1.5. Finally we conclude and discuss future work in Section 1.6.

1.2 RELATED WORK

In this chapter, we deal with pack scheduling for parallel tasks, aiming at makespan minimization (recall that the makespan is the total execution time). The corresponding problem with sequential tasks (tasks that execute on a single processor) is easy to solve for the makespan minimization objective: simply make a pack out of the largest p tasks, and proceed likewise while there remain tasks. Note that the pack scheduling problem with sequential tasks has been widely studied for other objective functions, see [9] for various job cost functions, and [30] for a survey. Back to the problem with sequential tasks and the makespan objective, Koole and Righter in [26] deal with the case where the execution time of each task is unknown but defined by a probabilistic distribution. They improve the result of Deb and Serfozo [14], who considered the stochastic problem with identical jobs. Ikura et al. [24] solve the makespan minimization problem where tasks have identical execution times, but different release times and deadlines; they assume agreeable deadlines, meaning that if a task has an earlier release time than another, it also has an earlier deadline.

Koehler et al. [25] propose a linear time solution to this last problem, and further give a $O(n^3)$ solution to the problem of minimizing the number of packs while achieving optimal makespan.

We focus next on the problem of co-scheduling parallel tasks in Section 1.2.1, and then we discuss related work on resilience in Section 1.2.2.

1.2.1 Parallel tasks

To the best of our knowledge, the problem with parallel tasks has not been studied as such. However, it was introduced by Dutot et al. in [16] as a moldable-by-phase model to approximate the moldable problem. The moldable task model is similar to the pack-scheduling model, but without the additional constraint (pack constraint) that the execution of new tasks cannot start before all tasks in the current pack are completed. Dutot et al. in [16] provide an optimal polynomial-time solution for the problem of pack scheduling identical independent tasks, using a dynamic programming algorithm. This is the only instance of pack-scheduling with parallel tasks that we found in the literature.

In practice, pack scheduling is really useful as shown by recent results. Li et al. [27] propose a framework to predict the energy and performance impacts of power-aware MPI task aggregation. Frachtenberg et al. [19] show that system utilization can be improved through their schemes to co-schedule jobs based on their load-balancing requirements and inter-processor communication patterns. Shantharam et al. [31] study co-scheduling based on speed-up profiles, similar to our work, but packs can have only one or two tasks; still, they report faster workload completion and corresponding savings in system energy.

Several publications [3, 10, 21] consider co-scheduling at a single multicore node, when contention for resources by co-scheduled tasks leads to complex tradeoffs between energy and performance measures. Chandra et al. [10] predict and utilize inter-thread cache contention at a multicore in order to improve performance. Hankendi and Coskun [21] show that there can be measurable gains in energy per unit of work through the application of their multi-level co-scheduling technique at runtime, which is based on classifying tasks according to specific performance measures. Bhaduria and McKee [3] consider local search heuristics to co-schedule tasks in a resource-aware manner at a multicore node to achieve significant gains in thread throughput per watt.

These publications demonstrate that complex tradeoffs cannot be captured through the use of the speed-up measure alone, without significant additional measurements to capture performance variations from cross-application interference at a multicore node. Additionally, and following [31] where packs have one or two tasks only, we expect significant benefits even when we aggregate only across multicore nodes because speed-ups suffer due to the longer latencies of data transfer across nodes. We can therefore project savings in energy

as being commensurate with the savings in the time to complete a workload through co-scheduling. Hence, we only test configurations where no more than a single application can be scheduled on a multicore node.

One could ask, given a set of n tasks to schedule, why schedule them in packs rather than globally? A global schedule would avoid the gaps incurred by some processors between the end of a pack and the beginning of the next pack, thereby potentially decreasing the makespan. However, there are several reasons to prefer pack scheduling. First, a global schedule is very hard to construct. Best-known heuristics greedily assign a new task to a set of processors as soon as this set terminates execution, thereby constraining the number of resources to be the same for the new task as for the last task. Our co-schedule does not suffer from this rigidity in processor assignment decisions. Secondly, the cost of scheduling itself is greatly reduced with pack scheduling. The scheduler launches a set of tasks and transfers corresponding input data only at the beginning of a pack. No overhead is paid until all tasks in the pack return, and a new pack is executed.

1.2.2 Resilience

One of the most used technique to handle fail-stop errors in HPC is checkpoint and rollback recovery [17]. The idea is to periodically save the system state, or the application memory footprint onto a stable storage. Then, after a downtime and a recovery time, the system can be restored into a former valid state (rollback step). Another technique to dealing with fail-stop errors is process replication, which consists in replicating a process and even replicate communications. For instance, the project RedMPI [18] implements a process replication mechanism and quadruplicates each communication.

In this chapter, we use a light-weight checkpointing protocol called the *double checkpointing algorithm* [29, 15]. 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.

To the best of our knowledge, this is the first work to consider co-schedules and failures, and hence to use malleable applications [5, 20] to allow redistributions of processors between applications. More related work on models for parallel applications and resilience are discussed in [2].

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 [28] 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. When considering failures, we restrict to a single pack, because scheduling already becomes difficult for a single pack with failures and redistributions.

1.3 PROBLEM DEFINITION

The application consists of n independent tasks $T_1, \ldots, T_n$. The target execution platform consists of p identical processors, and each task T_i can be assigned an arbitrary number $\sigma(i)$ of processors, where $1 \leq \sigma(i) \leq p$. The objective is to minimize the total execution time by co-scheduling several tasks onto the p resources. Note that the approach is agnostic of the granularity of each processor, which can be either a single CPU or a multicore node.

1.3.1 Speedup profiles

Let $t_{i,j}$ be the execution time of task T_i with j processors, and $work(i,j) = j \times t_{i,j}$ be the corresponding work. We assume the following for $1 \leq i \leq n$ and $1 \leq j < p$:

$$\text{Weakly decreasing execution time: } t_{i,j+1} \leq t_{i,j} \tag{1.1}$$

$$\text{Weakly increasing work: } work(i,j+1) \geq work(i,j) \tag{1.2}$$

Equation (1.1) implies that execution time is a non-increasing function of the number of processors. Equation (1.2) states that efficiency decreases with the number of enrolled processors: in other words, parallelization has a cost! As a side note, we observe that these requirements make good sense in practice: many scientific tasks T_i are such that $t_{i,j}$ first decreases (due to load-balancing) and then increases (due to communication overhead), reaching a minimum for $j = j_0$; we can always let $t_{i,j} = t_{i,j_0}$ for $j \geq j_0$ by never actually using more than j_0 processors for T_i.

Remarks. Determining j_0 for a given application is a challenge by itself. In most cases, it is obtained by profiling and interpolation. Also, in case of an imperfect knowledge of execution-time profiles, it is possible to use curve-fitting techniques to construct near complete knowledge, and then use this constructed knowledge. We treat the same application with two different problem sizes as two different applications (their execution time profiles could potentially be different). Thus, sensitivity of runtime to different parameters that could change runtime profiles are inherently taken care of.

1.3.2 Co-schedules

A co-schedule partitions the n tasks into groups (called *packs*), so that (i) all tasks from a given pack start their execution at the same time; and (ii) two

tasks from different packs have disjoint execution intervals. For instance, in the example of Figure 1.1, the two first packs have three tasks, the third pack has only one task, and the last pack has two tasks. The execution time, or *cost*, of a pack is the maximal execution time of a task within that pack, and the cost of a co-schedule is the sum of the costs of all packs.

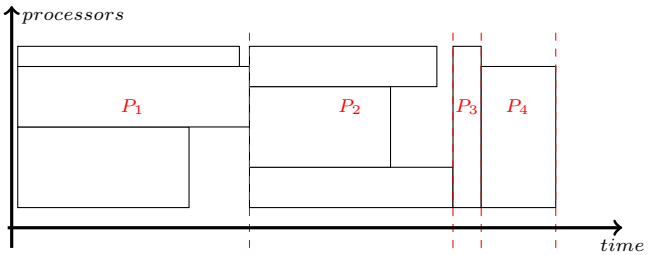

FIGURE 1.1 A co-schedule with four packs P_1 to P_4.

1.3.3 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 [22, 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* [29, 15]. 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 com-

municate 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 [32] and let

$$\tau_{i,j} = \sqrt{2\mu_{i,j}C_{i,j}} + C_{i,j}. \tag{1.3}$$

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.

1.3.4 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). Recall that $t_{i,j}$ is 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 (1.3)). 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

$$N^{\text{ff}}_{i,j}(\alpha) = \left\lfloor \frac{\alpha t_{i,j}}{\tau_{i,j} - C_{i,j}} \right\rfloor. \tag{1.4}$$

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 $\left(\frac{1}{\lambda j} + D\right)(e^{\lambda j T} - 1)$ [22]. 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:

$$\tau_{last} = \alpha t_{i,j} - N^{\text{ff}}_{i,j}(\alpha)(\tau_{i,j} - C_{i,j}). \tag{1.5}$$

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:

$$t^R_{i,j}(\alpha) = e^{\lambda j R_{i,j}}\left(\frac{1}{\lambda j} + D\right)\left(N^{\text{ff}}_{i,j}(\alpha)(e^{\lambda j \tau_{i,j}} - 1) + (e^{\lambda j \tau_{last}} - 1)\right). \tag{1.6}$$

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:

$$t^R_{i,j+1}(\alpha) \leq t^R_{i,j}(\alpha) \text{ for } 1 \leq i \leq n,\ 1 \leq j < p,\ 0 \leq \alpha \leq 1. \tag{1.7}$$

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 (1.7), 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:

$$j_{max}(i) = \min_{1 \leq j \leq p} \{j \text{ such that } t^R_{i,k} \geq t^R_{i,j} \text{ for all } k > j\}, \tag{1.8}$$

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 [5]: this amounts to say that no super-linear speed-up is possible, as stated earlier for the fault-free scenario. 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 $t^U_i = t^R_{i,j}(1)$.

1.3.5 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

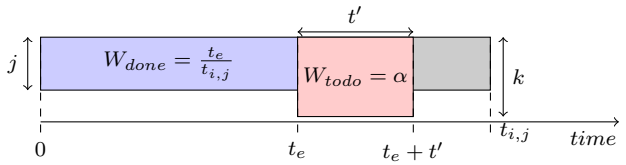

FIGURE 1.2 Work representation for application T_i at time t_e.

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 1.2). 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 m_i 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. 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 [4] 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 [6], we obtain a number of rounds equal to $\max(j, k-j)$ (see [2] for details), and the redistribution cost is

$$RC_i^{j \to k} = S + \max(j, k-j) \times \left(\frac{m_i}{jk\tau} + \beta\right). \tag{1.9}$$

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 above), 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 $N_{i,j}$:

$$N_{i,j} = \left\lfloor \frac{t - t_{lastR_i}}{\tau_{i,j}} \right\rfloor. \tag{1.10}$$

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).

1. 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 $\frac{N_{i,j} \times (\tau_{i,j} - C_{i,j})}{t_{i,j}}$.
2. 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:

$$RC_i^{j\to k} = S + \max(\min(j,k), |k-j|) \times \left(\frac{m_i}{kj\tau} + \beta\right). \tag{1.11}$$

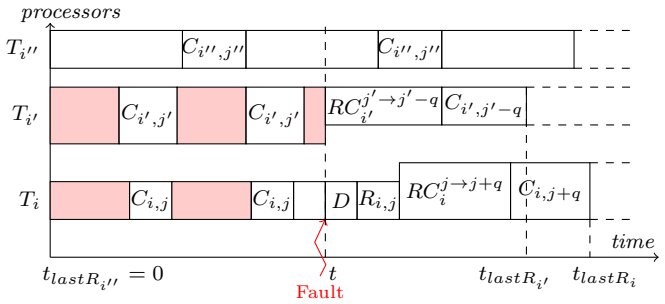

FIGURE 1.3 Example of redistribution when a fault strikes application T_i. The colored rectangles correspond to useful work done by T_i and $T_{i'}$ before the failure. $T_{i''}$ is not affected by the failure, since it does not perform a redistribution.

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 1.3. 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 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 applica-

tion 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 1.3, the processors involved in T_i and $T_{i'}$ cannot be considered for a new redistribution.

1.3.6 Optimization problems

We consider two optimization problems.

The general one is studied in a fault-free context with no redistribution, and builds packs with at most k tasks. The most general problem is when $k = p$, but, in some frameworks, we may have an upper bound $k < p$ on the maximum number of tasks within each pack.

Definition 1.1 (k-in-p-CoSchedule) *Given a fixed constant $k \leq p$, find a co-schedule with at most k tasks per pack that minimizes the execution time.*

When considering failures and redistributions, we focus on a single pack made of n applications:

Definition 1.2 (Resilient-CoSched-1pack) *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, 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 1.3.5).*

1.4 THEORETICAL ANALYSIS

In this section, we first focus on the problem in a failure-free scenario, where no checkpoints are taken. We discuss the complexity of the problem in Section 1.4.1, by exhibiting polynomial and NP-complete instances. Next we discuss how to optimally schedule a set of k tasks in a single pack (Section 1.4.2), both in the failure-free scenario and when accounting for failures but not doing any redistributions. Then, focusing again on the failure-free scenario, we explain how to compute the optimal solution of k-IN-p-CoSchedule (in expected exponential cost) in Section 1.4.3, and we provide an approximation algorithm in Section 1.4.4. Finally, we prove the NP-completeness of the problem Resilient-CoSched-1pack when considering failures and performing redistributions in Section 1.4.5.

1.4.1 Complexity

Theorem 1.1 *The* 1-IN-p-CoSchedule *and* 2-IN-p-CoSchedule *problems can both be solved in polynomial time.*

Proof: This result is obvious for 1-IN-p-COSCHEDULE: each task is assigned exactly p processors (see Equation (1.1)) and the minimum execution time is $\sum_{i=1}^{n} t_{i,p}$.

The proof is more involved for 2-IN-p-COSCHEDULE, and we start with the 2-IN-2-COSCHEDULE problem to get an intuition. Consider the weighted undirected graph $G = (V, E)$, where $|V| = n$, each vertex $v_i \in V$ corresponding to a task T_i. The edge set E is the following: (i) for all i, there is a loop on v_i of weight $t_{i,2}$; (ii) for all $i < i'$, there is an edge between v_i and $v_{i'}$ of weight $\max(t_{i,1}, t_{i',1})$. Finding a perfect matching of minimal weight in G leads to the optimal solution to 2-IN-2-COSCHEDULE, which can thus be solved in polynomial time.

For the 2-IN-p-COSCHEDULE problem, the proof is similar, the only difference lies in the construction of the edge set E: (i) for all i, there is a loop on v_i of weight $t_{i,p}$; (ii) for all $i < i'$, there is an edge between v_i and $v_{i'}$ of weight $\min_{j=1..p}(\max(t_{i,p-j}, t_{i',j}))$. Again, a perfect matching of minimal weight in G gives the optimal solution to 2-IN-p-COSCHEDULE. We conclude that the 2-IN-p-COSCHEDULE problem can be solved in polynomial time.

Theorem 1.2 *When $k \geq 3$, the k-IN-p-COSCHEDULE problem is strongly NP-complete.*

The proof can be found in [1]. It is based on a reduction from 3-PARTITION. Note that the 3-IN-p-COSCHEDULE problem is NP-complete, and the 2-IN-p-COSCHEDULE problem can be solved in polynomial time, hence 3-IN-3-COSCHEDULE is the simplest problem whose complexity remains open.

1.4.2 Scheduling a pack of tasks

In this section, we discuss how to optimally schedule a set of k tasks in a single pack: the k tasks $T_1, \ldots, T_k$ are given, and we search for an assignment function $\sigma : \{1, \ldots, k\} \to \{1, \ldots, p\}$ such that $\sum_{i=1}^{k} \sigma(i) \leq p$, where $\sigma(i)$ is the number of processors assigned to task T_i. Such a schedule is called a 1-pack-schedule, and its *cost* is $\max_{1 \leq i \leq k} t_{i,\sigma(i)}$. In Algorithm 1 below, we use the notation $T_i \preccurlyeq_\sigma T_j$ if $t_{i,\sigma(i)} \leq t_{j,\sigma(j)}$:

Theorem 1.3 *Given k tasks to be scheduled on p processors in a single pack, Algorithm 1 finds a 1-pack-schedule of minimum cost in time $O(p \log(k))$.*

In this greedy algorithm, we first assign one processor to each task, and while there are processors that are not processing any task, we select the task with the longest execution time and assign an extra processor to this task. Algorithm 1 performs $p - k$ iterations to assign the extra processors. We denote by $\sigma^{(\ell)}$ the current value of the function σ at the end of iteration ℓ. For convenience, we let $t_{i,0} = +\infty$ for $1 \leq i \leq k$. We start with the following lemma:

Lemma: At the end of iteration ℓ of Algorithm 1, let $T_{i^\star}$ be the first task

Algorithm 1: Finding the optimal 1-pack-schedule σ of k tasks in the same pack.

```
procedure Optimal-1-pack-schedule(T_1, ..., T_k)
begin
    for i = 1 to k do
        σ(i) ← 1;
    end
    Let L be the list of tasks sorted in non-increasing values of ≼_σ;
    p_available := p − k;
    while p_available ≠ 0 do
        T_i* := head(L);
        L := tail(L);
        σ(i*) ← σ(i*) + 1;
        p_available := p_available − 1;
        L := Insert T_i* in L according to its ≼_σ value;
    end
    return σ;
end
```

of the sorted list, i.e., the task with longest execution time. Then, for all i, $t_{i^\star,\sigma^{(\ell)}(i^\star)} \leq t_{i,\sigma^{(\ell)}(i)-1}$.

Proof: Let $T_{i^\star}$ be the task with longest execution time at the end of iteration ℓ. For tasks such that $\sigma^{(\ell)}(i) = 1$, the result is obvious since $t_{i,0} = +\infty$. Let us consider any task T_i such that $\sigma^{(\ell)}(i) > 1$. Let $\ell' + 1$ be the last iteration when a new processor was assigned to task T_i: $\sigma^{(\ell')}(i) = \sigma^{(\ell)}(i) - 1$ and $\ell' < \ell$. By definition of iteration $\ell' + 1$, task T_i was chosen because $t_{i,\sigma^{(\ell')}(i)}$ was greater than any other task, in particular $t_{i,\sigma^{(\ell')}(i)} \geq t_{i^\star,\sigma^{(\ell')}(i^\star)}$. Also, since we never remove processors from tasks, we have $\sigma^{(\ell')}(i) \leq \sigma^{(\ell)}(i)$ and $\sigma^{(\ell')}(i^\star) \leq \sigma^{(\ell)}(i^\star)$. Finally, $t_{i^\star,\sigma^{(\ell)}(i^\star)} \leq t_{i^\star,\sigma^{(\ell')}(i^\star)} \leq t_{i,\sigma^{(\ell')}(i)} = t_{i,\sigma^{(\ell)}(i)-1}$.

Proof of Theorem 1.3. Let σ be the 1-pack-schedule returned by Algorithm 1 of cost $c(\sigma)$, and let $T_{i^\star}$ be a task such that $c(\sigma) = t_{i^\star,\sigma(i^\star)}$. Let σ' be a 1-pack-schedule of cost $c(\sigma')$. We prove below that $c(\sigma') \geq c(\sigma)$, hence σ is a 1-pack-schedule of minimum cost:

1. If $\sigma'(i^\star) \leq \sigma(i^\star)$, then $T_{i^\star}$ has fewer processors in σ' than in σ, hence its execution time is larger, and $c(\sigma') \geq c(\sigma)$.
2. If $\sigma'(i^\star) > \sigma(i^\star)$, then there exists i such that $\sigma'(i) < \sigma(i)$ (since the total number of processors is p in both σ and σ'). We can apply the previous Lemma at the end of the last iteration, where $T_{i^\star}$ is the task of maximum execution time: $t_{i^\star,\sigma(i^\star)} \leq t_{i,\sigma(i)-1} \leq t_{i,\sigma'(i)}$, and therefore $c(\sigma') \geq c(\sigma)$.

Finally, the time complexity is obtained as follows: first we sort k elements, in time $O(k \log k)$. Then there are $p-k$ iterations, and at each iteration, we insert

an element in a sorted list of $k-1$ elements, which takes $O(\log k)$ operations (use a heap for the data structure of L).

Note that it is easy to compute an optimal 1-pack-schedule using a dynamic-programming algorithm: the optimal cost is $c(k,p)$, which we compute using the recurrence formula

$$c(i,q) = \min_{1 \le q' \le q} \{\max(c(i-1, q-q'), t_{i,q'})\}$$

for $2 \le i \le k$ and $1 \le q \le p$, initialized by $c(1,q) = t_{1,q}$, and $c(i,0) = +\infty$. The complexity of this algorithm is $O(kp^2)$. However, we can significantly reduce the complexity of this algorithm by using Algorithm 1.

With failures. It is not difficult to extend this algorithm to solve the problem with failures, but still without redistributions:

Theorem 1.4 *The* RESILIENT-COSCHED-1PACK *problem without redistributions can be solved in polynomial time $O(n)$, where n is the number of applications.*

We replace $t_{i,j}$ by $t^R_{i,j}(1)$, and instead of adding processors one-by-one, we add them two-by-two.

1.4.3 Optimal solution of k-IN-p-COSCHEDULE

In this section, we sketch two methods to find the optimal solution to the general k-IN-p-COSCHEDULE problem. This can be useful to solve some small-size instances, albeit at the price of a cost exponential in the number of tasks n.

The first method is to generate all possible partitions of the tasks into packs. This amounts to computing all partitions of n elements into subsets of cardinality at most k. For a given partition of tasks into packs, we use Algorithm 1 to find the optimal processor assignment for each pack, and we can compute the optimal cost for the partition. We still have to calculate the minimum of these costs among all partitions.

The second method is to cast the problem in terms of an integer linear program:

Theorem 1.5 *The following integer linear program characterizes the k-IN-p-COSCHEDULE problem, where the unknown variables are the $x_{i,j,b}$'s (Boolean variables) and the y_b's (rational variables), for $1 \le i, b \le n$ and $1 \le j \le p$:*

$$\begin{array}{ll} \textit{Minimize} \sum_{b=1}^{n} y_b & \textit{subject to} \\ \textit{(i)} \ \sum_{j,b} x_{i,j,b} = 1, & 1 \le i \le n \\ \textit{(ii)} \ \sum_{i,j} x_{i,j,b} \le k, & 1 \le b \le n \\ \textit{(iii)} \ \sum_{i,j} j \times x_{i,j,b} \le p, & 1 \le b \le n \\ \textit{(iv)} \ x_{i,j,b} \times t_{i,j} \le y_b, & 1 \le i, b \le n, 1 \le j \le p \end{array} \tag{1.12}$$

Proof: The $x_{i,j,b}$'s are such that $x_{i,j,b} = 1$ if and only if task T_i is in the pack b

and it is executed on j processors; y_b is the execution time of pack b. Since there are no more than n packs (one task per pack), $b \leq n$. The sum $\sum_{b=1}^{n} y_b$ is therefore the total execution time ($y_b = 0$ if there are no tasks in pack b). Constraint (i) states that each task is assigned to exactly one pack b, and with one number of processors j. Constraint (ii) ensures that there are not more than k tasks in a pack. Constraint (iii) adds up the number of processors in pack b, which should not exceed p. Finally, constraint (iv) computes the cost of each pack.

1.4.4 Approximation algorithm

In this section, we introduce PACK-APPROX, a 3-approximation algorithm for the p-IN-p-COSCHEDULE problem: if COST_{OPT} is the optimal solution, and $\text{COST}_{\text{algo}}$ is the output of the algorithm, we guarantee that $\text{COST}_{\text{algo}} \leq 3\text{COST}_{\text{OPT}}$. The design principle of PACK-APPROX is the following: we start from the assignment where each task is executed on one processor, and use Algorithm 2 to build a first solution. Algorithm 2 is a greedy heuristic that builds a co-schedule when each task is pre-assigned a number of processors for execution. Then we iteratively refine the solution, adding a processor to the task with longest execution time, and re-executing Algorithm 2. Here are details on both algorithms:

Algorithm 2. The k-IN-p-COSCHEDULE problem with processor pre-assignments remains strongly NP-complete (use a similar reduction as in the proof of Theorem 1.2). We propose a greedy procedure in Algorithm 2 that is similar to the First Fit Decreasing Height algorithm for strip packing [11]. The output is a co-schedule with at most k tasks per pack, and the complexity is $O(n \log(n))$ (dominated by sorting).

Algorithm 3. We iterate the calls to Algorithm 2, adding a processor to the task with longest execution time, until: (i) either the task of longest execution time is already assigned p processors, or (ii) the sum of the work of all tasks is greater than p times the longest execution time. The algorithm returns the minimum cost found during execution. The complexity of this algorithm is $O(n^2 p)$ in the simplest version presented here: in the $O(np)$ calls to Algorithm 2, we do not need to re-sort the list but we maintain it sorted instead, thus each call except the first one has linear cost. The complexity can be reduced to $O(n \log(n) + np)$ using standard algorithmic techniques [12].

Theorem 1.6 PACK-APPROX *is a 3-approximation algorithm for the* p-IN-p-COSCHEDULE *problem.*

The involved proof can be found in [1].

Minimum resource requirement. We conclude this section on theoretical analysis in a fault-free scenario by the following remark. We point out that all results can be extended to deal with the variant of the problem where each task T_i has a minimum compute node requirement m_i. Such a requirement is typically provided by the user. In that variant, Equation (1.2) is defined

Algorithm 2: Creating packs of size at most k, when the number $\sigma(i)$ of processors per task T_i is fixed.

```
procedure MAKE-PACK(n, p, k, σ)
begin
    Let L be the list of tasks sorted in non-increasing values of
     execution times t_{i,σ(i)};
    while L ≠ ∅ do
        Schedule the current task on the first pack with enough
         available processors and fewer than k tasks. Create a new pack
         if no existing pack fits;
        Remove the current task from L;
    end
    return the set of packs
end
```

only for j greater than m_i. For all previous algorithms, the difference lies in the preliminary step where one assigns one processor to each task: one would now assign m_i processors to task i, for all i. The number of total steps in the algorithms becomes smaller (because there are fewer processors available). One should note that with this constraint, all results (Theorems 1.1 to 1.6) are still valid, and proofs are quite similar.

1.4.5 With redistributions

We can easily build examples to show the difficulty of RESILIENT-COSCHED-1PACK 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 [2] for details). Intuitively, these little examples show that RESILIENT-COSCHED-1PACK 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 (1.11). Even in this simplified scenario, the problem is NP-complete:

Theorem 1.7 *With constant redistribution costs and without failures,* RESILIENT-COSCHED-1PACK *is NP-complete (in the strong sense).*

The involved reduction comes from 3-PARTITION, and can be found in [2].

Algorithm 3: PACK-APPROX

```
procedure PACK-APPROX(T_1, ..., T_n)
begin
    COST = +∞;
    for j = 1 to n do σ(j) ← 1;
    for i = 0 to n(p − 1) − 1 do
        Let A_tot(i) = Σ_{j=1}^{n} t_{j,σ(j)} σ(j);
        Let T_{j*} be one task that maximizes t_{j,σ(j)};
        Call MAKE-PACK (n, p, p, σ);
        Let COST_i be the cost of the co-schedule;
        if COST_i < COST then COST ← COST_i;
        if ( A_tot(i)/p > t_{j*,σ(j*)} ) or (σ(j*) = p) then
        return COST ;                                  /* Exit loop */
        else σ(j*) ← σ(j*) + 1;          /* Add a processor to T_{j*} */
    end
    return COST;
end
```

Remark. We conjecture that RESILIENT-COSCHED-1PACK remains NP-complete with zero redistribution cost. This is because of the combinatorial exploration suggested by the examples. But this remains an open problem!

1.5 HEURISTICS AND SIMULATIONS

In this section, we introduce and evaluate polynomial-time heuristics to solve the general RESILIENT-COSCHED-1PACK 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, Algorithm 1. Note that heuristics for the k-IN-p-COSCHEDULE general problem can be found in [1], together with their evaluation.

We first discuss the general structure of the heuristics in Section 1.5.1. Then, we explain how to redistribute available processors in Section 1.5.2, and the two strategies to redistribute when failures occur in Section 1.5.3. The pseudo-codes for all algorithms are available in [2]. The simulation settings are discussed in Section 1.5.4, and results are presented in Section 1.5.5.

1.5.1 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 1.5.2. 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 1.5.3).

1.5.2 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.

1.5.3 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 1.5.2, except that we need to handle the faulty application differently to update the values of α_f and t_{lastR_f}.

1.5.4 Simulation settings

To assess the efficiency of the heuristics, we have performed extensive simulations. 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.

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 [8, 7]. 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 [1]. 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:

$$t(m,q) = f \times t(m,1) + (1-f)\frac{t(m,1)}{q} + \frac{m}{q} log_2(m). \tag{1.13}$$

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 [2].

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.

1.5.5 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 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.

Performance in a fault-free context. Figure 1.4 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 1.5.2). 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 [2]).

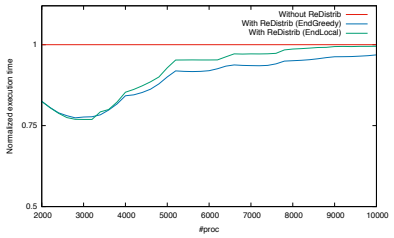

FIGURE 1.4 Redistribution in a fault-free context.

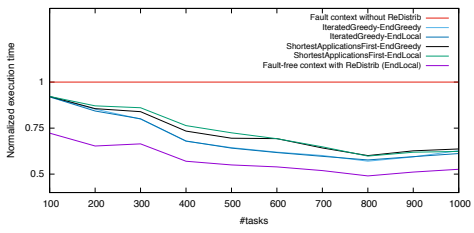

FIGURE 1.5 Impact of n with $p = 5000$ processors.

Impact of n. Figure 1.5 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.

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 [2]).

Impact of MTBF. Figure 1.6 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!

Impact of checkpointing cost. Figure 1.7 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 1.7 (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.

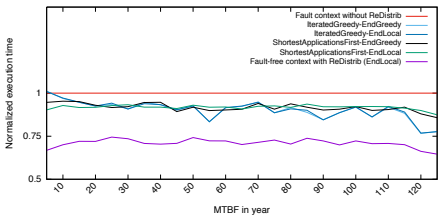

FIGURE 1.6 Impact of MTBF with $n = 100$ and $p = 5000$.

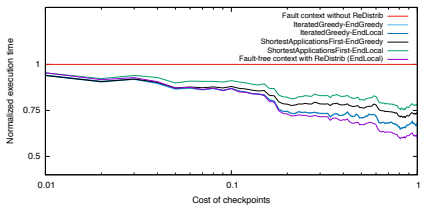

FIGURE 1.7 Impact of checkpointing cost.

Additionally, we show in [2] 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.

Summary. Altogether, we observe that IteratedGreedy achieves better performance than ShortestApplicationsFirst, 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.

1.6 CONCLUSION

In this chapter, we have provided theoretical results to assess the complexity of the general partitioning problem in a fault-free scenario; the problem is NP-complete when a pack can contain at least 3 tasks, and we have provided an approximation algorithm. When accounting for failures, we have designed a detailed and comprehensive model for scheduling a single pack of applications on a failure-prone platform, with processor redistributions. We have introduced a greedy polynomial-time algorithm that returns the optimal solution (for a single pack) when there are failures but no processor redistribution is allowed, or in a fault-free scenario. 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.

Bibliography

[1] G. Aupy, M. Shantharam, A. Benoit, Y. Robert, and P. Raghavan. Co-scheduling algorithms for high-throughput workload execution. *Journal of Scheduling*, To appear, 2015.

[2] A. Benoit, L. Pottier, and Y. Robert. Resilient application co-scheduling with processor redistribution. Research report RR-8795, INRIA, 2015. Available: `graal.ens-lyon.fr/~abenoit`.

[3] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proc. 24th ACM Int. Conf. on Supercomputing ICS '10*. ACM, 2010.

[4] P. B. Bhat, C. S. Raghavendra, and V. K. Prasanna. Efficient collective communication in distributed heterogeneous systems. *Journal of Parallel and Distributed Computing*, 63(3):251–263, 2003.

[5] J. Blazewicz, M. Machowiak, G. Mounie, and D. Trystram. Approximation algorithms for scheduling independent malleable tasks. In R. Sakellariou, J. Gurd, L. Freeman, and J. Keane, editors, *Euro-Par 2001 Parallel Processing*, volume 2150 of *Lecture Notes in Computer Science*, pages 191–197. Springer Berlin Heidelberg, 2001.

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

[7] 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*, 26(17):2772–2791, 2014.

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

[9] P. Brucker, A. Gladky, H. Hoogeveen, M. Y. Kovalyov, C. Potts, T. Tautenhahn, and S. Van De Velde. Scheduling a batching machine. *J. Scheduling*, 1:31–54, 1998.

[10] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA 11*, pages 340–351. IEEE, 2005.

[11] E. G. Coffman Jr, M. R. Garey, D. S. Johnson, and R. E. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4):808–826, 1980.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2009.

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

[14] R. K. Deb and R. F. Serfozo. Optimal control of batch service queues. *Advances in Applied Probability*, pages 340–361, 1973.

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

[16] P.-F. Dutot et al. Scheduling parallel tasks: Approximation algorithms. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, 2003.

[17] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[18] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[19] E. Frachtenberg, D. Feitelson, F. Petrini, and J. Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE. Trans. Parallel Distributed Systems*, 16(11):1066–1077, 2005.

[20] 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*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[21] C. Hankendi and A. Coskun. Reducing the energy cost of computing through efficient co-scheduling of parallel workloads. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 994–999, 2012.

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

[23] 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. Research Report 5574, Sandia National Laboratories, USA, September 2009.

[24] Y. Ikura and M. Gimple. Efficient scheduling algorithms for a single batch processing machine. *Operations Research Letters*, 5(2):61–65, 1986.

[25] F. Koehler and S. Khuller. Optimal batch schedules for parallel machines. In *Proceedings of the 13th Annual Algorithms and Data Structures Symposium*, 2013.

[26] G. Koole and R. Righter. A stochastic batching and scheduling problem. *Probability in the Engineering and Informational Sciences*, 15(04):465–479, 2001.

[27] D. Li, D. S. Nikolopoulos, K. Cameron, B. R. de Supinski, and M. Schulz. Power-aware MPI task aggregation prediction for high-end computing systems. In *IPDPS 10*, pages 1–12, 2010.

[28] N. Muthuvelu, I. Chai, E. Chikkannan, and R. Buyya. Batch resizing policies and techniques for fine-grain grid tasks: The nuts and bolts. *J. Information Processing Systems*, 7(2), 2011.

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

[30] C. N. Potts and M. Y. Kovalyov. Scheduling with batching: a review. *European Journal of Operational Research*, 120(2):228–249, 2000.

[31] M. Shantharam, Y. Youn, and P. Raghavan. Speedup-aware co-schedules for efficient workload management. *Parallel Processing Letters*, 23(2), 2013.

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