Resilient scheduling algorithms for large-scale platforms

Valentin Le Fèvre

Numéro National de Thèse : 2020LYSEN019

THÈSE de DOCTORAT de L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale N°512 :
École doctorale Informatique et Mathématiques

Spécialité de doctorat : Informatique

Soutenue publiquement le 18/06/2020, par :

Valentin Le Fèvre

Resilient scheduling algorithms for large-scale platforms

Algorithmes d'ordonnancement tolérants aux fautes pour les plates-formes à grande échelle

---

Devant le jury composé de :

Olivier Beaumont	Directeur de recherche INRIA, INRIA Bordeaux Sud-Ouest	*Examinateur*
Anne Benoit	Maîtresse de conférences, ENS de Lyon, LIP	*Co-encadrante*
Henri Casanova	Professeur, Université d'Hawai'i	*Rapporteur*
Amina Guermouche	Maitresse de conférences, Télécom Sud-Paris	*Examinatrice*
Rami Melhem	Professeur, Université de Pittsburgh	*Rapporteur*
Yves Robert	Professeur des universités, ENS de Lyon, LIP	*Directeur*

Remerciements

Merci Sci-hub

Contents

Chapter 1

Introduction

In the recent years, many scientific advances in physics, chemistry, biology and more have been achieved thanks to high performance computing (HPC). For example, to study magnetic field properties, physicists need to go deep into tiny details of fluid dynamics, which is an intense computational task even for current machines [138]. In order to compute faster, processors and more generally units of computation have seen their number of transistors increase, as observed by Moore's law. However, physical limits (thermal limits, density and more [128, 123]) made constructors unable to keep increasing the number of transistors with the same rate as before. The new plan since reaching these limits has been to increase the number of components in a machine in order to split the computation across all the units of computation. State-of-the-art computing platforms have all grown to very large scales. For example, the current fastest machine according to the Top500 in November 2019 [181] is Summit, a platform with more than 2.4 millions of cores, for a peak performance of 2×10^{17} Flop/s.

However, this increase in the number of components brings one major problem: **resilience**. It was reported to be among the top ten challenges of exascale computing (i.e., 10^{18} Flop/s) by the Advanced Scientific Computing Advisory Committee (ASCAC) [135]. Resilience can be defined by "ensuring correct scientific computation in face of faults, reproducibility, and algorithm verification challenges". Why is this challenge relevant? Resilience is mandatory so that most (if not all) applications can finish and deliver non-erroneous results. Indeed, if one processor has a lifetime of 100 years (which is already optimistic), then a machine with 100,000 processors will experience a failure every 9 hours. More generally, if we consider a mean time before failures (MTBF) μ for one processor, a machine using p such components will have a MTBF of $\frac{\mu}{p}$ [106, Proposition 1.2], i.e., the MTBF of the platform decreases linearly with the number of components. Since applications running on such systems can last for a day up to a few weeks, solutions need to be provided to recover from two sources of errors: fail-stop and silent errors.

Silent errors are errors that strike while an application is running and are unnoticed by the system. Such errors can be bit-flips in the memory or faulty Arithmetic Logic Units (ALU). They can be caused by lots of factors including cosmic radiation [147].

The main challenge with these faults is to detect them as they only modify the data or output of an algorithm. Verification mechanisms exist and need to be carefully used to detect (and correct if possible) this kind of errors. *Fail-stop errors*, in contrast to silent errors, are automatically detected because they result in the complete stop of an application. They can come from either a dead component, or a bug in the code of the application that provokes a segmentation fault, for instance. If they are simpler to detect than silent errors, fail-stop errors make the progress of an application lost and thus cannot be "corrected". The standard approach to deal with these errors is **checkpointing**, which consists in regularly saving the progress of the application.

Checkpointing. This de facto technique works as follows: in order to avoid restarting an application from scratch (and probably reach an infinite execution time by starting over and over), the idea is to save the state of an application regularly. When an error is detected (either a fail-stop or silent error), we can roll back to that saved state so that we can start recomputing from that state. When silent errors are taken into account, a verification mechanism can be added just before taking the checkpoint to ensure that the saved progress is not erroneous and that the application can be safely restored from that point.
The main question is now: when should we save the state of the application so that it completes the earliest possible in average? If we perform a checkpoint too often, we will lose some time overall due to spending too much time saving data instead of computing. However, if we do not save regularly enough, the risk of having to re-execute a huge part of the application increases and can lead to some degraded performance as well. Young [200] and Daly [58] proposed a model for fail-stop errors and a simple equation to optimize the total execution time:

$$\sqrt{2C\mu},$$

where C is the time of writing a checkpoint and μ is the MTBF of the platform (depending on the number of components as seen above)[1]. This formula applies to divisible applications (applications that can be pre-empted at anytime). In this thesis we present extensions of this formula for various checkpointing strategies: Chapter 2 deals with multi-level checkpointing instead of single-level checkpointing; Chapters 6 and 7 extend the formula when replication (see next paragraph) is added. Chapter 4 also tries to address the problem of checkpointing a general workflow with tasks that are atomic and cannot be pre-empted at any time. For workflows that are linear, Toueg and Babaoglu [183] gave an optimal dynamic programming algorithm (that we extend with replication in Chapter 5) but no optimal algorithm exists for generic workflows.

Replication. The principle of replication is to execute some work several times in parallel and each running copy is called a replica: if one replica fails (for any reason),

[1] A similar equation exists for minimizing the energy consumption (another big challenge in [135]) instead of the execution time. We address this problem in Section 7.8.

the other replicas are still safe and can be used. Hence, an application crashes or is uncorrectable only if all the replicas have failed. By definition, this is less likely to happen than when having no copy, but we still need to use a checkpoint/restart strategy to handle these cases [159, 75, 205, 82, 46, 110]. The number of replicas determine how strong the replication is: for example, with two replicas (*duplication*), we can tolerate one fail-stop error on one of the two replicas and still succeed. However, if a silent error strikes, we can detect it by looking at the results of the replicas but we would be unable to guess which replica was not corrupted by the error. Instead, if we use three replicas (*triplication*), we can tolerate up to one fail-stop error on two different replicas, and we can also correct a silent error by using a majority rule. The main drawback of replication is that it uses more components to execute the same effective amount of work, but we show in Part II that duplication still leads to better performance than no replication in some cases.

In this thesis, we review a set of techniques using checkpointing in Part I. In Chapter 2, we derive a Young/Daly-like formula for multi-level checkpointing. Chapter 3 studies how enrolling more processors than necessary can be efficient for a checkpointed application, while Chapter 4 proposes a generic workflow checkpointing scheme.

We combine replication with checkpointing in Part II. In Chapter 5, we design an optimal dynamic programming algorithm for placing checkpoints and choosing the replicated tasks in linear workflows. Chapter 6 extends the Young/Daly formula to an application duplicated on heterogeneous platforms. Chapter 7 extends the formula for homogeneous core-based duplication.

Finally, we investigate some scheduling results for HPC in Part III. Chapter 8 studies, theoretically and experimentally, several scheduling heuristics in a non-reliable context. The main question is to know if algorithms that were designed for job scheduling without failures [184] are still efficient when some tasks have to be re-executed. Another problem with the recent supercomputers is that more and more data are produced faster while the I/O bandwidth increases at a slower pace. For instance, when Los Alamos National Laboratory moved from Cielo to Trinity, the peak performance moved from 1.4 Petaflops to 40 Petaflops ($\times$28) while the I/O bandwidth moved to 160 GB/s to 1.45TB/s (only $\times$9) [124]. To tackle this challenge, Chapter 9 investigates scheduling at the I/O level.

Part I

Advanced checkpointing techniques

Chapter 2

Towards optimal multi-level checkpointing

We provide a framework to analyze multi-level checkpointing protocols, by formally defining a k-level checkpointing pattern. We provide a first-order approximation to the optimal checkpointing period, and show that the corresponding overhead is in the order of $\sum_{\ell=1}^{k} \sqrt{2\lambda_\ell C_\ell}$, where λ_ℓ is the error rate at level ℓ, and C_ℓ the checkpointing cost at level ℓ. This nicely extends the classical Young/Daly formula on single-level checkpointing. Furthermore, we are able to fully characterize the shape of the optimal pattern (number and positions of checkpoints), and we provide a dynamic programming algorithm to determine the optimal subset of levels to be used. Finally, we perform simulations to check the accuracy of the theoretical study and to confirm the optimality of the subset of levels returned by the dynamic programming algorithm. The results nicely corroborate the theoretical study, and demonstrate the usefulness of multi-level checkpointing with the optimal subset of levels. The work in this chapter is joint work with Anne Benoit, Aurélien Cavelan, Yves Robert and Hongyang Sun, and has been published in *Transactions on computers* (TC) [J1].

2.1 Introduction

Checkpointing is the de-facto standard resilience method for HPC platforms at extreme-scale. However, the traditional single-level checkpointing method suffers from significant overhead, and multi-level checkpointing protocols now represent the state-of-the-art technique. These protocols allow different levels of checkpoints to be set, each with a different checkpointing overhead and recovery ability. Typically, each level corresponds to a specific fault type, and is associated to a storage device that is resilient to that type. For instance, a two-level system would deal with (i) transient memory errors (level 1) by storing key data in main memory; and (ii) node failures (level 2) by storing key data in stable storage (remote redundant disks).

In this chapter, we deal with fail-stop errors only. We consider a very general scenario, where the platform is subject to k levels of faults, numbered from 1 to k. Level ℓ is associated with an error rate λ_ℓ, a checkpointing cost C_ℓ, and a recovery cost R_ℓ. A fault at level ℓ destroys all the checkpoints of lower levels (from 1 to $\ell - 1$

included) and implies a roll-back to a checkpoint of level ℓ or higher. Similarly, a recovery of level ℓ will restore data from all lower levels. Typically, fault rates are decreasing and checkpoint/recovery costs are increasing when we go to higher levels: $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k$, $C_1 \leq C_2 \leq \cdots \leq C_k$, and $R_1 \leq R_2 \leq \cdots \leq R_k$.

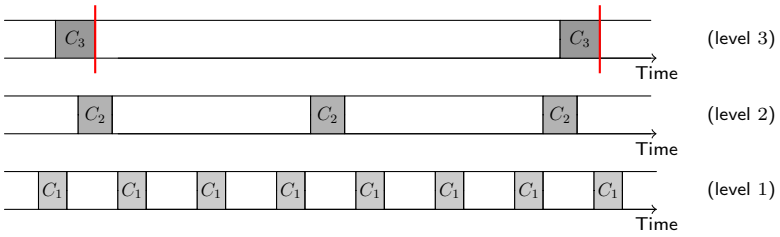


Figure 2.1: Independent checkpointing periods for three levels of faults: no synchronization between checkpoint levels.

The idea of multi-level checkpointing is that checkpoints are taken for each level of faults, but at different periods. Intuitively, the less frequent the faults, the longer the checkpointing period: this is because the risk of a failure striking is lower when going to higher levels; hence the expected re-execution time is lower too; one can safely checkpoint less frequently, thereby reducing failure-free overhead (checkpointing is useless in the absence of fault). There are several natural approaches to implement multi-level checkpointing. The first option is to use *independent checkpointing periods* for each level, as illustrated in Figure 2.1 with $k = 3$ levels. This option raises several difficulties, the most prominent one being overlapping checkpoints. Typically, we need to checkpoint different levels in sequence (e.g., writing into memory before writing onto disk), so we would need to delay some checkpoints, which might not be possible in some environments, and which would introduce irregular periods. The second option is to synchronize all checkpoint levels by nesting them inside a *periodic pattern* that repeats over time, as illustrated in Figure 2.2(a). In this figure, the **pattern** has five computational **segments**, each followed by a level-1 checkpoint. A segment is a chunk of work between two checkpoints, and a pattern consists in segments and checkpoints. The second and fifth level-1 checkpoints are followed by a level-2 checkpoint. Finally, the pattern ends with a level-3 checkpoint. When using patterns, a checkpoint at level ℓ is always preceded by checkpoints at all lower levels 1 to $\ell - 1$, which makes good sense in practice (e.g., with two levels, main memory and disk, one writes the data into memory before transferring it to disk).

Using periodic patterns simplifies the orchestration of checkpoints at all levels. In addition, repeatedly applying the same pattern is optimal for on-line scheduling problems, or for jobs running a very long (even infinite) time on the platform. Indeed, in this scenario, we seek the best pattern, i.e., the one whose overhead is minimal. The *overhead* of a pattern is the price per work unit to pay for resilience in the pattern; hence minimizing overhead is equivalent to optimizing platform throughput. For a pattern $\mathrm{P}(W)$ with W units of work (the cumulated length of all its segments), the overhead

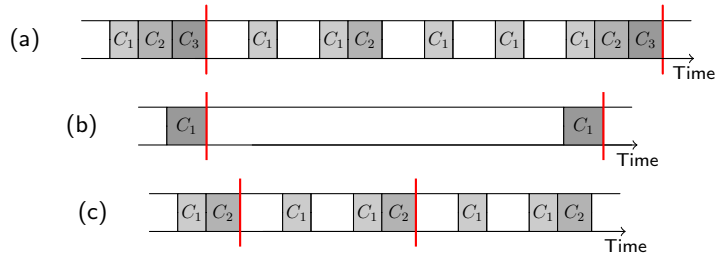


Figure 2.2: Checkpointing patterns (highlighted using red bars) with (a) $k = 3$, (b) $k = 1$, and (c) $k = 2$ levels.

$H(\mathrm{P(W)})$ is defined as the ratio of the pattern's expected execution time $\mathbb{E}(\mathrm{P(W)})$ over its total work W minus 1:

$$H(\mathrm{P}(W)) = \frac{\mathbb{E}(\mathrm{P}(W))}{W} - 1. \tag{2.1}$$

If there were neither checkpoint nor fault, the overhead would be zero. Determining the optimal pattern (with minimal overhead), and then repeatedly using it until job completion, is the optimal approach with Exponential failure distributions and long-lasting jobs. Indeed, once a pattern is successfully executed, the optimal strategy is to re-execute the same pattern. This is because of the memoryless property of exponential distributions: the history of failures has no impact on the solution, so if a pattern is optimal at some point in time, it stays optimal later in the execution, because we have no further information about the amount of work still to be executed.

The difficulty of characterizing the optimal pattern dramatically increases with the number of levels. How many checkpoints of each level should be used, and at which locations inside the pattern? What is the optimal length of each segment? With one single level (see Figure 2.2(b)), there is a single segment of length W, and the Young/Daly formula [200, 58] gives $W^{\mathrm{opt}} = \sqrt{\frac{2C_1}{\lambda_1}}$. The minimal overhead is then $H^{\mathrm{opt}} = \sqrt{2\lambda_1 C_1} + O(\lambda_1)$.

With two levels, the pattern still has a simple shape, with N segments followed by a level-1 checkpoints, and ended by a level-2 checkpoint (see Figure 2.2(c)). Recent work [63] shows that all segments have same length in the optimal pattern, and provides mathematical equations that can be solved numerically to compute both the optimal length W^{opt} of the pattern and its optimal number of segments. However, no closed-form expression is available, neither for W^{opt}, nor for the minimal overhead H^{opt}.

With three levels, no optimal solution is known. The pattern shape becomes quite complicated. Coming back to Figure 2.2(a), we identify two sub-patterns ending with a level-2 checkpoint. The first sub-pattern has 2 segments while the second one has 3. The memoryless property does not imply that all sub-patterns are identical, because

the state after completing the first sub-pattern is not the same as the initial state when beginning the execution of the pattern. In the general case with k levels, the shape of the pattern will be even more complicated, with different-shaped sub-patterns (each ended by a level $k-1$ checkpoint). In turn, each sub-pattern may have different-shaped sub-sub-patterns (each ended by a level $k-2$ checkpoint), and so on. The major contribution of this work is to provide an analytical characterization of the optimal pattern with an arbitrary number k of checkpointing levels, with closed-form formulas for the pattern length W^{opt}, the number of checkpoints at each level, and the optimal overhead H^{opt}. In particular, we obtain the following beautiful result:

$$H^{\text{opt}} = \sum_{\ell=1}^{k} \sqrt{2\lambda_\ell C_\ell} + O(\Lambda), \tag{2.2}$$

where $\Lambda = \sum_{\ell=1}^{k} \lambda_\ell$. However, we point out that this analytical characterization relies on a first-order approximation, so it is valid only when resilience parameters C_ℓ and R_ℓ are small in front of the platform Mean Time Between Failures (MTBF) $\mu = 1/\Lambda$. Also, the optimal pattern has rational number of segments, and we use rounding to derive a practical solution. Still, Equation (2.2) provides a lower bound on the optimal overhead, and this bound is met very closely in all our experimental scenarios.

Finally, in many practical cases, there is no obligation to use all available checkpointing levels. For instance, with $k = 3$ levels, one may choose among four possibilities: level 3 only, levels 1 and 3, levels 2 and 3, and all levels 1, 2 and 3. Of course, we still have to account for all failure types, which translates into the following:

- level 3: use $\lambda_3 \leftarrow \lambda_1 + \lambda_2 + \lambda_3$;
- levels 1 and 3: use λ_1 and $\lambda_3 \leftarrow \lambda_2 + \lambda_3$;
- levels 2 and 3: use $\lambda_2 \leftarrow \lambda_1 + \lambda_2$ and λ_3;
- all levels: use λ_1, λ_2 and λ_3.

Our analytical characterization of the optimal pattern leads to a simple dynamic programming algorithm for selecting the optimal subset of levels.

The rest of this chapter is organized as follows. Section 2.2 is the heart of the chapter and shows how to compute the optimal pattern as well as the optimal subset of levels. Section 2.3 is devoted to simulations assessing the accuracy of the first-order approximation. Section 2.4 surveys the related work. Finally, Section 2.5 provides concluding remarks and hints for future work.

2.2 Computing the optimal pattern

This section computes the optimal multi-level checkpointing pattern. We first state our assumptions in Section 2.2.1, and then analyze the simple case with $k = 2$ levels in Section 2.2.2, before proceeding to the general case in Section 2.2.3. Finally, the algorithm to compute the optimal subset of levels is described in Section 2.2.4.

2.2.1 Assumptions

In this chapter, we assume that failures from different levels are independent[1]. For each level ℓ, the arrival of failures follows *Poisson* process with error rate λ_ℓ. In order to deal with the interplay of failures from different levels, we make use of the following well-known properties of independent Poisson processes [87, Chapter 2.3].

Property 1. *During the execution of a segment with length w, let X_ℓ denote the time when the first level-ℓ error strikes. Thus, X_ℓ is a random variable following an Exponential distribution with parameter λ_ℓ, for all $\ell = 1, 2, \ldots, k$.*

(1). Let X denote the time when the first error (of any level) strikes. We have $X = \min\{X_1, X_2, \ldots, X_k\}$, which follows an Exponential distribution with parameter $\Lambda = \sum_{\ell=1}^{k} \lambda_\ell$. The probability of having an error (from any level) in the segment is therefore $P(X \leq w) = 1 - e^{-\Lambda w}$.

(2). Given that an error (from any level) strikes during the execution of the segment, the probability that the error belongs to a particular level is proportional to the error rate of that level, i.e., $P(X = X_\ell | X \leq w) = \frac{\lambda_\ell}{\Lambda}$, for all $\ell = 1, 2, \ldots, k$.

Moreover, we assume that error rates of different levels are of the same order, i.e., $\lambda_\ell = \Theta(\Lambda)$ for all $\ell = 1, 2, \ldots, k$, and that errors only strike during the computations, while checkpointing and recovery are error-free. Indeed, the durations of checkpoints and recoveries are generally small compared to the pattern length, so the probability of a failure striking during these operations is low. It has been shown in [20] that removing this assumption does not impact the first-order approximation of the pattern overhead.

2.2.2 Optimal two-level pattern

We start by analyzing the two-level pattern shown in Figure 2.2(b). The goal is to determine a first-order approximation to the optimal pattern length W, the number n of level-1 checkpoints in the pattern, as well as the length $w_i = \alpha_i W$ of the i-th segment, for all $1 \leq i \leq n$, where $\sum_{i=1}^{n} \alpha_i = 1$.

2.2.2.1 With a single segment

We first consider a special case of the two-level pattern, in which only a single segment is present, i.e., $n = 1$. The result establishes the order of the optimal pattern length W^{opt}, which will be used later for analyzing the general case. Recall that $\Lambda = \lambda_1 + \lambda_2$ and, for convenience, let us also define $C = C_1 + C_2$. The following proposition shows the expected time of such a pattern with fixed length W.

[1] In practice, failures from different checkpointing levels can exhibit potential correlation [104, 62]. Consideration of correlated failures is beyond the scope of this work.

Proposition 1. *The expected execution time of a two-level pattern with a single segment and fixed length W is*

$$\mathbb{E} = W + C + \frac{1}{2}\Lambda W^2 + O(\max\{\Lambda^2 W^3, \Lambda W\}).$$

Proof. We can express the expected execution time of the pattern recursively as follows:

$$\begin{aligned}\mathbb{E} = P\left(\mathbb{E}^{\text{lost}}(W,\Lambda) + \frac{\lambda_1}{\Lambda}\left(R_1 + \mathbb{E}\right) + \frac{\lambda_2}{\Lambda}\Big(R_2 + R_1 + \mathbb{E}\Big)\right) \\ + (1-P)\left(W + C\right),\end{aligned} \tag{2.3}$$

where $P = 1 - e^{-\Lambda W}$ denotes the probability of having a failure (either level-1 or level-2) during the execution of the pattern based on Property 1.1, and $\mathbb{E}^{\text{lost}}(w_i, \Lambda)$ denotes the expected time lost when such a failure occurs. In this case, and based on Property 1.2, if the failure belongs to level 1, which happens with probability $\frac{\lambda_1}{\Lambda}$, we can recover from the latest level-1 checkpoint (R_1). Otherwise, the failure belongs to level 2 with probability $\frac{\lambda_2}{\Lambda}$, and we need to first recover from the latest level-2 checkpoint (R_2) before restoring the level-1 checkpoint (R_1). In both cases, the entire pattern needs to be re-executed again. Finally, if no error (of any level) strikes, which happens with probability $1 - P$, the pattern is completed after W time of execution followed by the time C to perform the two checkpoints, which are assumed to be error-free.

From [106, Equation (1.13)], the expected time lost when executing a segment of length W with error rate Λ is

$$\mathbb{E}^{\text{lost}}(W,\Lambda) = \frac{1}{\Lambda} - \frac{W}{e^{\Lambda W} - 1}. \tag{2.4}$$

Substituting Equation (2.4) into Equation (2.3) and solving for $\mathbb{E}$, we get:

$$\mathbb{E} = \left(e^{\Lambda W} - 1\right)\left(\frac{1}{\Lambda} + R_1 + \frac{\lambda_2}{\Lambda}R_2\right) + C_1 + C_2, \tag{2.5}$$

which is an exact formula on the expected execution time of the pattern. Now, using Taylor series to expand $e^{\Lambda W} = 1 + \Lambda W + \frac{\Lambda^2 W^2}{2} + O(\Lambda^3 W^3)$ while assuming $W = \Theta(\Lambda^{-x})$, where $0 < x < 1$, we can re-write Equation (2.5) as

$$\begin{aligned}\mathbb{E} = W + \frac{1}{2}\Lambda W^2 + C_1 + C_2 + O(\Lambda^2 W^3) \\ + \left(\Lambda W + \frac{\Lambda^2 W^2}{2} + O(\Lambda^3 W^3)\right)\left(R_1 + \frac{\lambda_2}{\Lambda}R_2\right).\end{aligned}$$

Since recovery costs (R_1, R_2) are assumed to be constants, and error rates $(\lambda_1, \lambda_2, \Lambda)$

are in the same order, the expected execution time can be expressed as follows:

$$\mathbb{E} = W + C_1 + C_2 + \frac{1}{2}\Lambda W^2 + O(\Lambda^2 W^3) + O(\Lambda W),$$

which completes the proof of the proposition. □

From Proposition 1, the expected execution overhead of the pattern can be derived as

$$H = \frac{C}{W} + \frac{1}{2}\Lambda W + O(\max\{\Lambda^2 W^2, \Lambda\}).$$

Assume that the platform MTBF $\mu = 1/\Lambda$ is large in front of the resilience parameters, and consider the first two terms of H: the overhead is minimized when the pattern has length $W = \Theta(\Lambda^{-1/2})$, and in that case both terms are in the order of $\Theta(\Lambda^{1/2})$, so we have $H = \Theta(\Lambda^{1/2}) + O(\Lambda)$. Indeed, the last term $O(\Lambda^2 W^2) = O(\Lambda)$ becomes negligible compared to $\Theta(\Lambda^{1/2})$. Hence, the optimal pattern length W^{opt} can be obtained by balancing the first two terms in H, which gives

$$W^{\text{opt}} = \sqrt{\frac{2C}{\Lambda}} = \Theta(\Lambda^{-1/2}), \tag{2.6}$$

and the optimal execution overhead becomes

$$H^{\text{opt}} = \sqrt{2\Lambda C} + O(\Lambda). \tag{2.7}$$

Remarks. Unlike in single-level checkpointing, the checkpoint to roll back to in a two-level pattern depends on which type of error strikes first. Under first-order approximation and assuming that the resilience parameters are small compared to the platform MTBF and pattern length, the formulas shown in Equations (2.6) and (2.7) reduce exactly to Young/Daly's classical result by aggregating the error rates and checkpointing costs of both levels.

2.2.2.2 With multiple segments

We now consider the general two-level pattern with multiple segments, and derive the optimal pattern parameters. As in the single-segment case, we start with a proposition showing the expected time to execute a two-level pattern with fixed parameters.

Proposition 2. *The expected execution time of a given two-level pattern is*

$$\mathbb{E} = W + nC_1 + C_2 + \frac{1}{2}\left(\lambda_1 \sum_{i=1}^{n} \alpha_i^2 + \lambda_2\right) W^2 + O(\Lambda^{1/2}).$$

Proof. We first prove the following result (by induction) on the expected time $\mathbb{E}_i$ to execute the i-th segment of the pattern (up to the level-1 checkpoint at the end of the

segment):

$$\mathbb{E}_i = w_i + C_1 + \frac{\lambda_1}{2} w_i^2 + \lambda_2 \left(\frac{w_i^2}{2} + \sum_{j=1}^{i-1} w_j w_i \right) + O(\Lambda^{1/2}). \tag{2.8}$$

According to the result with a single segment, we know that the optimal pattern length and hence the segment length are in the order of $O(\Lambda^{-1/2})$, which implies that $\mathbb{E}_i = w_i + O(1)$.

For the ease of analysis, we assume that there is a hypothetical segment at the beginning of the pattern with length $w_0 = 0$ (hence no need to checkpoint). For this segment, we have $\mathbb{E}_0 = w_0 = 0$, satisfying Equation (2.8). Suppose the claim holds up to $\mathbb{E}_{i-1}$. Then, $\mathbb{E}_i$ can be recursively expressed as follows:

$$\begin{aligned} \mathbb{E}_i = P_i \Bigg(& \mathbb{E}^{\text{lost}}(w_i, \Lambda) + \frac{\lambda_1}{\Lambda} \left(R_1 + \mathbb{E}_i \right) \\ & + \frac{\lambda_2}{\Lambda} \Big(R_2 + R_1 + \sum_{j=1}^{i-1} \mathbb{E}_j + \mathbb{E}_i \Big) \Bigg) \\ & + (1 - P_i)(w_i + C_1), \end{aligned} \tag{2.9}$$

where $P_i = 1 - e^{-\Lambda w_i}$ denotes the probability of having a failure (either level-1 or level-2) during the execution of the segment, and $\mathbb{E}^{\text{lost}}(w_i, \Lambda)$ denotes the expected time lost when such a failure occurs.

Equation (2.9) is very similar to Equation (2.3), except when a level-2 failure occurs we need to re-execute all the segments (up to segment i) that have been executed so far. Following the derivation of Proposition 1 and applying $\mathbb{E}_j = w_j + O(1)$ for $j = 1, 2, \ldots, i-1$, we can derive the first-order approximation of $\mathbb{E}_i$ as follows:

$$\begin{aligned} \mathbb{E}_i &= w_i + C_1 + \frac{1}{2} \left(\lambda_1 w_i^2 + \lambda_2 w_i^2 + 2\lambda_2 w_i \sum_{j=1}^{i-1} \mathbb{E}_j \right) + O(\Lambda^{1/2}) \\ &= w_i + C_1 + \frac{1}{2} \left(\lambda_1 w_i^2 + \lambda_2 w_i^2 + 2\lambda_2 w_i \sum_{j=1}^{i-1} \Big(w_j + O(1) \Big) \right) + O(\Lambda^{1/2}) \\ &= w_i + C_1 + \frac{1}{2} \left(\lambda_1 w_i^2 + \lambda_2 \Big(w_i^2 + 2 \sum_{j=1}^{i-1} w_j w_i \Big) \right) + O(\Lambda^{1/2}). \end{aligned} \tag{2.10}$$

Since the level-2 checkpoint at the end of the pattern is also assumed to be error-

free, we can compute the expected execution time of the pattern as

$$\begin{aligned}\mathbb{E} &= \sum_{i=1}^{n} \mathbb{E}_i + C_2 \\ &= W + nC_1 + C_2 + \frac{1}{2}\left(\lambda_1 \sum_{i=1}^{n} \alpha_i^2 + \lambda_2\right) W^2 + O(\Lambda^{1/2}),\end{aligned}$$

since $\sum_{i=1}^{n} w_i^2 + 2\sum_{i=1}^{n}\sum_{j=1}^{i-1} w_j w_i = (\sum_{i=1}^{n} w_i)^2 = W^2$. □

Theorem 1. *A first-order approximation to the optimal two-level pattern is characterized by*

$$n^{opt} = \sqrt{\frac{\lambda_1}{\lambda_2} \cdot \frac{C_2}{C_1}}, \tag{2.11}$$

$$\alpha_i^{opt} = \frac{1}{n^{opt}} \quad \forall i = 1, 2, \ldots, n^{opt}, \tag{2.12}$$

$$W^{opt} = \sqrt{\frac{n^{opt} C_1 + C_2}{\frac{1}{2}\left(\frac{\lambda_1}{n^{opt}} + \lambda_2\right)}}, \tag{2.13}$$

where n^{opt} is the number of segments, $\alpha_i^{opt} W^{opt}$ is the length of the i-th segment, and W^{opt} is the pattern length.
The optimal pattern overhead is

$$H^{opt} = \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} + O(\Lambda). \tag{2.14}$$

Proof. For a given pattern with a fixed number n of segments, $\sum_{i=1}^{n} \alpha_i^2$ is minimized subject to $\sum_{i=1}^{n} \alpha_i = 1$ when $\alpha_i = \frac{1}{n}$ for all $i = 1, 2, \ldots, n$. Hence, we can derive the expected execution overhead from Proposition 2 as follows:

$$H = \frac{nC_1 + C_2}{W} + \frac{1}{2}\left(\frac{\lambda_1}{n} + \lambda_2\right) W + O(\Lambda). \tag{2.15}$$

For a given n, the optimal work length can then be computed from Equation (2.15), and it is given by $W^{\text{opt}} = \sqrt{\frac{nC_1 + C_2}{\frac{1}{2}\left(\frac{\lambda_1}{n} + \lambda_2\right)}}$. In that case, the execution overhead becomes

$$H = \sqrt{2\left(\frac{\lambda_1}{n} + \lambda_2\right)(nC_1 + C_2)} + O(\Lambda), \tag{2.16}$$

which is minimized as shown in Equation (2.14) when n satisfies Equation (2.11). Indeed, $2\left(\frac{\lambda_1}{n^{\text{opt}}} + \lambda_2\right)(n^{\text{opt}} C_1 + C_2) = 2\lambda_1 C_1 + 2\lambda_2 C_2 + 4\sqrt{\lambda_1 \lambda_2 C_1 C_2} = (\sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_1 C_1})^2$. In practice, since the number of segments can only be a positive integer, the optimal solution is either $\max(1, \lfloor n^{\text{opt}} \rfloor)$ or $\lceil n^{\text{opt}} \rceil$, whichever leads to a smaller value

of the convex function H as shown in Equation (2.16). □

Remarks. Consider the example given in [63] with $C_1 = R_1 = 20$, $C_2 = R_2 = 50$, $\lambda_1 = 2.78 \times 10^{-4}$ and $\lambda_2 = 4.63 \times 10^{-5}$. The optimal solution[2] provided by [63] gives $n^{\text{opt}} = 3.83$, $W^{\text{opt}} = 1362.49$ and $H^{\text{opt}} = 0.1879$, while Theorem 1 suggests $n^{\text{opt}} = 3.87$, $W^{\text{opt}} = 1378.27$ and $H^{\text{opt}} = 0.1735$, which is quite close to the exact optimum. The difference in overhead is due to the negligence of lower-order terms in the first-order approximation. We point out that the solution provided by [63] relies on numerical methods to solve rather complex mathematical equations, whose convergence is not always guaranteed, and it is only applicable to two levels. Our result, on the other hand, is able to provide fast and good approximation to the optimal solution when the error rates are sufficiently small, and it can be readily extended to an arbitrary number of levels, as shown in the next section.

2.2.3 Optimal k-level pattern

In this section, we derive the first-order approximation to the optimal k-level pattern by determining its length W, the number N_ℓ of level-ℓ checkpoints for all $1 \leq \ell \leq k$, as well as the positions of all checkpoints in the pattern.

2.2.3.1 Observations

Before analyzing the optimal pattern, we make several observations. First, we can obtain the orders of the optimal length and pattern overhead as shown below (recall that $\Lambda = \sum_{\ell=1}^{k} \lambda_\ell$).

Observation 1. *Consider the simplest k-level pattern with a single segment of length W. We can conduct the same analysis as in Section 2.2.2.1 to show that the optimal pattern length satisfies $W^{opt} = \Theta(\Lambda^{-1/2})$, and the corresponding overhead satisfies $H^{opt} = \Theta(\Lambda^{1/2})$.*

From the analysis of the two-level pattern, we can also observe that the overall execution overhead of any pattern comes from two distinct sources defined below.

Observation 2. *There are two types of execution overheads for a pattern:*

(1). Error-free overhead, *denoted as o_{ef}, is the total cost of all the checkpoints placed in the pattern. For a given set of checkpoints, the error-free overhead is completely determined regardless of their positions in the pattern.*

(2). Re-executed fraction overhead, *denoted as o_{re}, is the expected fraction of work that needs to be re-executed due to errors. The re-executed fraction overhead depends on both the set of checkpoints and their positions.*

[2]The original optimal solution of [63] considers faults in checkpointing but not during recoveries. We adapt its solution to exclude faults in checkpointing so to be consistent with the model in this chapter for a fair comparison. The results reported herein are based on this modified solution.

For example, in the two-level pattern with n level-1 checkpoints and given values of α_i for all $i = 1, 2, \ldots, n$, the two types of overheads are given by $o_{\text{ef}} = nC_1 + C_2$ and $o_{\text{re}} = \frac{1}{2}\left(f_1 \sum_{i=1}^{n} \alpha_i^2 + f_2\right)$, where $f_\ell = \frac{\lambda_\ell}{\Lambda}$ for $\ell = 1, 2$. Assuming that checkpoints at all levels have constant costs and that the error rates at all levels are in the same order, then both o_{ef} and o_{re} can be considered as constants, i.e., $o_{\text{ef}} = O(1)$ and $o_{\text{re}} = O(1)$.

A trade-off exists between these two types of execution overheads, since placing more checkpoints generally reduces the re-executed work fraction when an error strikes, but it can adversely increase the overhead when the execution is error-free. Therefore, in order to achieve the best overall overhead, a resilience algorithm must seek an optimal balance between o_{ef} and o_{re}.

For a given pattern with fixed overheads o_{ef} and o_{re}, we can make the following observation based on Propositions 1 and 2, which partially characterizes the optimal pattern.

Observation 3. *For a given pattern (with fixed o_{ef} and o_{re}), the expected execution time is given by*

$$\mathbb{E} = \underbrace{W + o_{\text{ef}}}_{\substack{\text{error-free} \\ \text{execution time}}} + \underbrace{\Lambda W}_{\substack{\text{expected} \\ \text{\# errors}}} \cdot \underbrace{o_{\text{re}} W}_{\substack{\text{re-executed work} \\ \text{in case of error}}} + O(\Lambda^{1/2}), \tag{2.17}$$

and the optimal pattern length and the resulting expected execution overhead of the pattern are

$$W^{opt} = \sqrt{\frac{o_{\text{ef}}}{\Lambda \cdot o_{\text{re}}}}, \tag{2.18}$$

$$H^{opt} = 2\sqrt{\Lambda \cdot o_{\text{ef}} \cdot o_{\text{re}}} + O(\Lambda). \tag{2.19}$$

Equation (2.19) shows that the trade-off between o_{ef} and o_{re} is manifested as the product of the two terms. Hence, in order to determine the optimal pattern, it suffices to find the pattern parameters (e.g., n and α_i) that minimize $o_{\text{ef}} \cdot o_{\text{re}}$.

2.2.3.2 Analysis

We now extend the analysis to derive the optimal multi-level checkpointing patterns. Generally, for a k-level pattern, each computational segment $s^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ can be uniquely identified by its level ℓ as well as its position $\langle i_{k-1}, \ldots, i_\ell \rangle$ within the multi-level hierarchy. For instance, in a four-level pattern, the segment $s^{(2)}_{1,3}$ denotes the third level-2 segment inside the first level-3 segment of the pattern (see Figure 2.3). Note that a segment can contain multiple sub-segments at the lower levels (except for bottom-level segments) and is a sub-segment of a larger segment at a higher level (except for top-level segments). The entire pattern can be denoted as $s^{(k)}$, which is the only segment at level k.

For any segment $s^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ at level ℓ, where $1 \leq \ell \leq k$, let $w^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ denote its length. Hence, we have $w^{(\ell+1)}_{i_{k-1},\ldots,i_{\ell+1}} = \sum_{i_\ell} w^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ and $w^{(k)} = W$. Also, let $n^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ denote

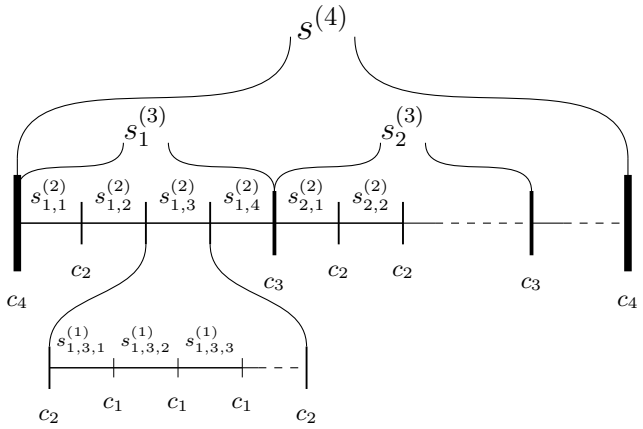


Figure 2.3: Example of a 4-level pattern. Here, we let $c_\ell = C_1|C_2|\cdots|C_\ell$ denote the succession of checkpoints from level 1 to level ℓ.

the number of sub-segments contained by $s^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ at the lower level $\ell - 1$. We have $n^{(1)}_{i_{k-1},\ldots,i_1} = 1$ for all $i_{k-1}, \ldots, i_1$. For convenience, we further define

$$\alpha^{(\ell)}_{i_{k-1},\ldots,i_\ell} = \frac{w^{(\ell)}_{i_{k-1},\ldots,i_\ell}}{W}$$

as the fraction of the length of segment $s^{(\ell)}_{i_{k-1},\ldots,i_\ell}$ inside the pattern, and define N_ℓ to be the total number of level-ℓ segments in the entire pattern. Therefore, we have $N_k = 1$, $N_{k-1} = n^{(k)}$, and in general

$$N_\ell = \sum_{i_{k-1},\ldots,i_{\ell+1}} n^{(\ell+1)}_{i_{k-1},\ldots,i_{\ell+1}}.$$

The following proposition shows the expected time to execute a given k-level pattern.

Proposition 3. *The expected execution time of a given k-level pattern is*

$$\mathbb{E} = W + \sum_{\ell=1}^{k-1} N_\ell C_\ell + C_k + \frac{W^2}{2}\left(\sum_{\ell=1}^{k} \lambda_\ell \sum_{i_{k-1},\ldots,i_\ell} \left(\alpha^{(\ell)}_{i_{k-1},\ldots,i_\ell}\right)^2\right) + O(\Lambda^{1/2}).$$

Proof. We show that the expected time to execute any segment $s^{(h)}_{i_{k-1},\ldots,i_h}$ at level h, where $1 \le h \le k$, satisfies the following (without counting the time to execute all the check-

points inside the segment):

$$\begin{aligned}\mathbb{E}^{(h)}_{i_{k-1},\dots,i_h} = w^{(h)}_{i_{k-1},\dots,i_h} + \frac{W^2}{2}\left(\sum_{\ell=1}^{h}\lambda_\ell \sum_{i_{h-1},\dots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\dots,i_\ell}\right)^2\right) \\ + \Lambda_{[h+1,k]}\left(\frac{\left(w^{(h)}_{i_{k-1},\dots,i_h}\right)^2}{2} + w^{(h)}_{i_{k-1},\dots,i_h}\sum_{j_h=1}^{i_h-1}\mathbb{E}^{(h)}_{i_{k-1},\dots,j_h}\right) \\ + w^{(h)}_{i_{k-1},\dots,i_h}\sum_{\ell=h+2}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\dots,j_{\ell-1}} + O(\Lambda^{1/2}),\end{aligned} \tag{2.20}$$

where $\Lambda_{[x,y]} = \sum_{\ell=x}^{y}\lambda_\ell$ and, if $x > y$, we define $\Lambda_{[x,y]} = 0$. The proposition can then be proven by setting $\mathbb{E} = \mathbb{E}^{(k)} + \sum_{\ell=1}^{k-1} N_\ell C_\ell + C_k$, since checkpoints are assumed to be error-free.

We now prove Equation (2.20) by induction on the level h. For the base case, i.e., when $h = 1$, consider a segment $s^{(1)}_{i_{k-1},\dots,i_1}$ at the first level. Following the proof of Proposition 2 (in particular, Equation (2.9)), we can express its expected execution time $\mathbb{E}^{(1)}_{i_{k-1},\dots,i_1}$, as

$$\begin{aligned}\mathbb{E}^{(1)}_{i_{k-1},\dots,i_1} = P^{(1)}_{i_{k-1},\dots,i_1}\Bigg(&\mathbb{E}^{\text{lost}}\left(w^{(1)}_{i_{k-1},\dots,i_1},\Lambda\right) \\ &+ \frac{\lambda_1}{\Lambda}\left(R_1 + \mathbb{E}^{(1)}_{i_{k-1},\dots,i_1}\right) \\ &+ \frac{\lambda_2}{\Lambda}\left(\sum_{j=1}^{2}R_j + \sum_{j_1=1}^{i_1}\mathbb{E}^{(1)}_{i_{k-1},\dots,j_1}\right) \\ &+ \frac{\lambda_3}{\Lambda}\left(\sum_{j=1}^{3}R_j + \sum_{j_2=1}^{i_2-1}\mathbb{E}^{(2)}_{i_{k-1},\dots,j_2} + \sum_{j_1=1}^{i_1}\mathbb{E}^{(1)}_{i_{k-1},\dots,j_1}\right) \\ &\quad\vdots \\ &+ \frac{\lambda_k}{\Lambda}\left(\sum_{j=1}^{k}R_j + \sum_{j_{k-1}=1}^{i_{k-1}-1}\mathbb{E}^{(k-1)}_{j_{k-1}} + \sum_{j_{k-2}=1}^{i_{k-2}-1}\mathbb{E}^{(k-2)}_{i_{k-1},j_{k-2}} + \cdots + \sum_{j_1=1}^{i_1}\mathbb{E}^{(1)}_{i_{k-1},\dots,j_1}\right)\Bigg) \\ &+ \left(1 - P^{(1)}_{i_{k-1},\dots,i_1}\right)w^{(1)}_{i_{k-1},\dots,i_1},\end{aligned} \tag{2.21}$$

where $\Lambda = \sum_{\ell=1}^{k}\lambda_\ell$ is the total rate of all error sources, and $P^{(1)}_{i_{k-1},\dots,i_1} = 1 - e^{\Lambda\cdot w^{(1)}_{i_{k-1},\dots,i_1}}$ denotes the probability of having an error (from any level) during the execution of the

segment. Simplifying Equation (2.21) and solving for $\mathbb{E}^{(1)}_{i_{k-1},\ldots,i_1}$ we get:

$$\begin{aligned}
\mathbb{E}^{(1)}_{i_{k-1},\ldots,i_1} &= w^{(1)}_{i_{k-1},\ldots,i_1} + \frac{W^2}{2}\Lambda_{[1,k]}\left(\alpha^{(1)}_{i_{k-1},\ldots,i_1}\right)^2 \\
&\quad + w^{(1)}_{i_{k-1},\ldots,i_1}\sum_{\ell=2}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\ldots,j_{\ell-1}} + O(\Lambda^{1/2}) \\
&= w^{(1)}_{i_{k-1},\ldots,i_1} + \frac{W^2}{2}\lambda_1\left(\alpha^{(1)}_{i_{k-1},\ldots,i_1}\right)^2 \\
&\quad + \Lambda_{[2,k]}\left(\frac{\left(w^{(1)}_{i_{k-1},\ldots,i_1}\right)^2}{2} + w^{(1)}_{i_{k-1},\ldots,i_1}\sum_{j_1=1}^{i_1-1}\mathbb{E}^{(1)}_{i_{k-1},\ldots,j_1}\right) \\
&\quad + w^{(1)}_{i_{k-1},\ldots,i_1}\sum_{\ell=3}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\ldots,j_{\ell-1}} + O(\Lambda^{1/2}),
\end{aligned}$$

which satisfies Equation (2.20).

Suppose Equation (2.20) holds up to any segment $s^{(h)}_{i_{k-1},\ldots,i_h}$ at level h. Following the proof of Proposition 2 (in particular, the derivation of Equation (2.10)), we can show by induction that $\mathbb{E}^{(h)}_{i_{k-1},\ldots,i_h} = w^{(h)}_{i_{k-1},\ldots,i_h} + O(1)$. Hence, for segment $s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}$ at level $h+1$, we have the Equation (2.22).

Hence, Equation (2.20) also holds for any segment at level $h+1$. This completes the proof of the proposition.

$$\begin{aligned}
\mathbb{E}^{(h+1)}_{i_{k-1},\ldots,i_{h+1}} &= \sum_{i_h}\mathbb{E}^{(h)}_{i_{k-1},\ldots,i_h} \\
&= \sum_{i_h} w^{(h)}_{i_{k-1},\ldots,i_h} + \frac{W^2}{2}\left(\sum_{\ell=1}^{h}\lambda_\ell\sum_{i_h,\ldots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\ldots,i_\ell}\right)^2\right) \\
&\quad + \Lambda_{[h+1,k]}\sum_{i_h}\left(\frac{\left(w^{(h)}_{i_{k-1},\ldots,i_h}\right)^2}{2} + w^{(h)}_{i_{k-1},\ldots,i_h}\sum_{j_h=1}^{i_h-1} w^{(h)}_{i_{k-1},\ldots,j_h}\right) \\
&\quad + \sum_{i_h} w^{(h)}_{i_{k-1},\ldots,i_h}\sum_{\ell=h+2}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\ldots,j_{\ell-1}} + O(\Lambda^{1/2})
\end{aligned}$$

$$\begin{aligned}
\mathbb{E}^{(h+1)}_{i_{k-1},\dots,i_{h+1}} &= w^{(h+1)}_{i_{k-1},\dots,i_{h+1}} + \frac{W^2}{2}\left(\sum_{\ell=1}^{h}\lambda_\ell \sum_{i_h,\dots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\dots,i_\ell}\right)^2\right) + \Lambda_{[h+1,k]}\frac{\left(w^{(h+1)}_{i_{k-1},\dots,i_{h+1}}\right)^2}{2} \\
&\quad + w^{(h+1)}_{i_{k-1},\dots,i_{h+1}} \sum_{\ell=h+2}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\dots,j_{\ell-1}} + O(\Lambda^{1/2}) \\
&= w^{(h+1)}_{i_{k-1},\dots,i_{h+1}} + \frac{W^2}{2}\left(\sum_{\ell=1}^{h+1}\lambda_\ell \sum_{i_h,\dots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\dots,i_\ell}\right)^2\right) \\
&\quad + \Lambda_{[h+2,k]}\left(\frac{\left(w^{(h+1)}_{i_{k-1},\dots,i_{h+1}}\right)^2}{2} + w^{(h+1)}_{i_{k-1},\dots,i_{h+1}}\sum_{j_{h+1}=1}^{i_{h+1}-1}\mathbb{E}^{(h+1)}_{i_{k-1},\dots,j_{h+1}}\right) \\
&\quad + w^{(h+1)}_{i_{k-1},\dots,i_{h+1}} \sum_{\ell=h+3}^{k}\Lambda_{[\ell,k]}\sum_{j_{\ell-1}=1}^{i_{\ell-1}-1}\mathbb{E}^{(\ell-1)}_{i_{k-1},\dots,j_{\ell-1}} + O(\Lambda^{1/2}). \qquad (2.22)
\end{aligned}$$

□

Proposition 3 shows that, for a given k-level checkpointing pattern, the error-free overhead o_{ef} and the re-executed fraction overhead o_{re} are given as follows:

$$o_{\text{ef}} = \sum_{\ell=1}^{k-1} N_\ell C_\ell + C_k, \qquad (2.23)$$

$$o_{\text{re}} = \frac{1}{2}\sum_{\ell=1}^{k} f_\ell \sum_{i_{k-1},\dots,i_\ell}\left(\alpha^{(\ell)}_{i_{k-1},\dots,i_\ell}\right)^2, \qquad (2.24)$$

where $f_\ell = \frac{\lambda_\ell}{\Lambda}$. According to Observation 3, it remains to find parameters of the pattern such that $o_{\text{ef}} \cdot o_{\text{re}}$ is minimized.

To derive the optimal pattern, we first consider the case where o_{ef} is fixed, i.e., the set of checkpoints is given. The following proposition shows the optimal value of o_{re}.

Proposition 4. *For a k-level checkpointing pattern, suppose the number N_ℓ of checkpoints at each level ℓ is given, i.e., the error-free overhead o_{ef} is fixed (as in Equation* (2.23)*). Then, the optimal value of the re-executed work overhead is given by*

$$o^{opt}_{\text{re}} = \frac{1}{2}\left(\sum_{\ell=1}^{k-1}\frac{f_\ell}{N_\ell} + f_k\right), \qquad (2.25)$$

and it is obtained when all the checkpoints of each level are equally spaced in the pattern.

Proof. According to Equation (2.24), which shows the value of o_{re} for the entire pattern, we can define the corresponding overhead for each level-h segment $s^{(h)}_{i_{k-1},\dots,i_h}$ recursively

as follows:

$$o_{\text{re}}\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) = \frac{f_h}{2} \cdot \left(\alpha^{(h)}_{i_{k-1},\ldots,i_h}\right)^2 + \sum_{i_{h-1}} o_{\text{re}}\left(s^{(h-1)}_{i_{k-1},\ldots,i_{h-1}}\right),$$

with $o_{\text{re}}\left(s^{(0)}_{i_{k-1},\ldots,i_0}\right) = 0$ by definition.

For each segment $s^{(h)}_{i_{k-1},\ldots,i_h}$, we also define $N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)$ to be the total number of level-ℓ segments it contains, with $\ell \leq h$. We will show that the optimal value $o^{\text{opt}}_{\text{re}}\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)$ for the segment satisfies:

$$o^{\text{opt}}_{\text{re}}\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) = \frac{1}{2}\left(\sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)}\right)\left(\alpha^{(h)}_{i_{k-1},\ldots,i_h}\right)^2, \tag{2.26}$$

and it is achieved when its level-ℓ checkpoints are equally spaced, for all $\ell \leq h-1$. The proposition can then be proven by setting $o^{\text{opt}}_{\text{re}} = o^{\text{opt}}_{\text{re}}\left(s^{(k)}\right)$, since $N_\ell\left(s^{(k)}\right) = N_\ell$, $N_k = 1$, and $\alpha^{(k)} = 1$.

Now, we prove Equation (2.26) by induction on the level h. For the base case, i.e., when $h = 1$, we have $o_{\text{re}}\left(s^{(1)}_{i_{k-1},\ldots,i_1}\right) = \frac{f_1}{2} \cdot \left(\alpha^{(1)}_{i_{k-1},\ldots,i_1}\right)^2$ by definition, and it satisfies Equation (2.26), because $N_1\left(s^{(1)}_{i_{k-1},\ldots,i_1}\right) = 1$. Suppose Equation (2.26) holds for any segment $s^{(h)}_{i_{k-1},\ldots,i_h}$ at level h. Then, for segment $s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}$ at level $h+1$, we have:

$$\begin{aligned} o_{\text{re}}\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right) &= \frac{f_{h+1}}{2} \cdot \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2 + \sum_{i_h} o^{\text{opt}}_{\text{re}}\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) \\ &= \frac{f_{h+1}}{2} \cdot \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2 + \frac{1}{2}y, \end{aligned} \tag{2.27}$$

where $y = \sum_{i_h} x^{(h)}_{i_{k-1},\ldots,i_h} \cdot \left(\alpha^{(h)}_{i_{k-1},\ldots,i_h}\right)^2$, and $x^{(h)}_{i_{k-1},\ldots,i_h} = \sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)}$. To minimize $o_{\text{re}}\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)$ as shown in Equation (2.27), it suffices to solve the following minimization problem:

$$\begin{aligned} &\text{minimize} \quad y = \sum_{i_h} x^{(h)}_{i_{k-1},\ldots,i_h} \cdot \left(\alpha^{(h)}_{i_{k-1},\ldots,i_h}\right)^2, \\ &\text{subject to} \quad \sum_{i_h} \alpha^{(h)}_{i_{k-1},\ldots,i_h} = \alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}. \end{aligned}$$

Since y is clearly a convex function of $\alpha^{(h)}_{i_{k-1},\ldots,i_h}$, we can readily get, using Lagrange

multiplier [32], the minimum value of y as follows:

$$y_{\min} = \frac{1}{\sum_{i_h} 1/x^{(h)}_{i_{k-1},\ldots,i_h}} \cdot \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2, \tag{2.28}$$

which is obtained at

$$\widetilde{\alpha}^{(h)}_{i_{k-1},\ldots,i_h} = \frac{1/x^{(h)}_{i_{k-1},\ldots,i_h}}{\sum_{j_h} 1/x^{(h)}_{i_{k-1},\ldots,j_h}} \cdot \alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}. \tag{2.29}$$

Let us define $z = \sum_{i_h} 1/x^{(h)}_{i_{k-1},\ldots,i_h}$. We now need to solve the following maximization problem:

$$\begin{aligned} &\text{maximize } \; z = \sum_{i_h} \frac{1}{\sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)}}, \\ &\text{subject to } \sum_{i_h} N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) = N_\ell\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right), \forall \ell = 1,\ldots,h. \end{aligned}$$

Again, z is a convex function of $N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right)$, and it can be shown to be maximized when

$$N_\ell\left(s^{(h)}_{i_{k-1},\ldots,i_h}\right) = \frac{N_\ell\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)}{n^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}}, \quad \forall \ell = 1,\ldots,h,$$

which gives $\widetilde{\alpha}^{(h)}_{i_{k-1},\ldots,i_h} = \frac{1}{n^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}} \cdot \alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}$ according to Equation (2.29). This implies that all level-ℓ checkpoints are also equally spaced inside segment $s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}$, for all $\ell \leq h$. The maximum value of z in this case is

$$z_{\max} = \frac{1}{\sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)}},$$

and the optimal value of $y_{\min}$ according to Equation (2.28) is then given by

$$\begin{aligned} y^{\text{opt}}_{\min} &= \frac{1}{z_{\max}} \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2 \\ &= \left(\sum_{\ell=1}^{h} \frac{f_\ell}{N_\ell\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)}\right) \left(\alpha^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)^2. \end{aligned}$$

Substituting $y^{\text{opt}}_{\min}$ into Equation (2.27), we get the optimal value of $o_{\text{re}}\left(s^{(h+1)}_{i_{k-1},\ldots,i_{h+1}}\right)$ as

follows:

$$o_{\text{re}}^{\text{opt}}\left(s_{i_{k-1},\ldots,i_{h+1}}^{(h+1)}\right) = \frac{f_{h+1}}{2}\cdot\left(\alpha_{i_{k-1},\ldots,i_{h+1}}^{(h+1)}\right)^2 + \frac{1}{2}y_{\min}^{\text{opt}}$$
$$= \frac{1}{2}\left(\sum_{\ell=1}^{h+1}\frac{f_\ell}{N_\ell\left(s_{i_{k-1},\ldots,i_h}^{(h+1)}\right)}\right)\left(\alpha_{i_{k-1},\ldots,i_{h+1}}^{(h+1)}\right)^2.$$

This shows that Equation (2.26) also holds for segment $s_{i_{k-1},\ldots,i_{h+1}}^{(h+1)}$ at level $h+1$ and, hence, completes the proof of the proposition. □

We are now ready to characterize the optimal k-level pattern. The result is stated in the following theorem.

Theorem 2. *A first-order approximation to the optimal k-level pattern and its overhead are characterized by*

$$W^{opt} = \sqrt{\frac{2\left(\sum_{\ell=1}^{k-1}N_\ell^{opt}C_\ell + C_k\right)}{\sum_{\ell=1}^{k-1}\frac{\lambda_\ell}{N_\ell^{opt}} + \lambda_k}}, \tag{2.30}$$

$$N_\ell^{opt} = \sqrt{\frac{\lambda_\ell}{C_\ell}\cdot\frac{C_k}{\lambda_k}}, \quad \forall \ell = 1,2,\ldots,k-1, \tag{2.31}$$

$$H^{opt} = \sum_{\ell=1}^{k}\sqrt{2\lambda_\ell C_\ell} + O(\Lambda). \tag{2.32}$$

Proof. From Observation 3, Equation (2.23) and Proposition 4, we know that the optimal pattern can be obtained by minimizing the following function:

$$F = o_{\text{ef}}\cdot o_{\text{re}}^{\text{opt}} = \frac{1}{2}\left(\sum_{\ell=1}^{k-1}N_\ell C_\ell + C_k\right)\left(\sum_{\ell=1}^{k-1}\frac{f_\ell}{N_\ell} + f_k\right).$$

We first compute the optimal number of checkpoints at each level using a *two-phase iterative* method. Towards this end, let us define

$$o_{\text{ef}}(h) = \sum_{\ell=h}^{k-1}N_\ell C_\ell + C_k,$$
$$o_{\text{re}}^{\text{opt}}(h) = \frac{1}{2}\left(\sum_{\ell=h}^{k-1}\frac{f_\ell}{N_\ell} + f_k\right).$$

In the first phase, we set initially

$$F(1) = o_{\text{ef}}(1)\cdot o_{\text{re}}^{\text{opt}}(1).$$

The optimal value of N_1 that minimizes $F(1)$ can then be obtained by setting

$$\begin{aligned}\frac{\partial F(1)}{\partial N_1} &= C_1 o_{\text{re}}^{\text{opt}}(1) - o_{\text{ef}}(1)\frac{f_1}{2N_1^2} \\ &= C_1\left(\frac{f_1}{2N_1} + o_{\text{re}}^{\text{opt}}(2)\right) - \left(N_1 C_1 + o_{\text{ef}}(2)\right)\frac{f_1}{2N_1^2} \\ &= C_1 o_{\text{re}}^{\text{opt}}(2) - o_{\text{ef}}(2)\frac{f_1}{2N_1^2} = 0,\end{aligned}$$

which gives $N_1^{\text{opt}} = \sqrt{\frac{f_1}{C_1}\cdot\frac{o_{\text{ef}}(2)}{2o_{\text{re}}^{\text{opt}}(2)}}$. Substituting it into $F(1)$ and simplifying, we can get the value of F after the first iteration as

$$F(2) = \frac{1}{2}\left(\sqrt{f_1 C_1} + \sqrt{o_{\text{ef}}(2)\cdot o_{\text{re}}^{\text{opt}}(2)}\right)^2.$$

Repeating the above process, we can get the optimal value of F after $k-1$ iterations as

$$F^{\text{opt}} = F(k) = \frac{1}{2}\left(\sum_{\ell=1}^{k}\sqrt{f_\ell C_\ell}\right)^2, \tag{2.33}$$

and the optimal value of N_ℓ as

$$N_\ell^{\text{opt}} = \sqrt{\frac{f_\ell}{C_\ell}\cdot\frac{o_{\text{ef}}(\ell+1)}{2o_{\text{re}}^{\text{opt}}(\ell+1)}}, \quad \forall \ell = 1, 2, \ldots, k-1. \tag{2.34}$$

In the second phase, we first compute from Equation (2.34)

$$\begin{aligned}N_{k-1}^{\text{opt}} &= \sqrt{\frac{f_{k-1}}{C_{k-1}}\cdot\frac{o_{\text{ef}}(k)}{2o_{\text{re}}^{\text{opt}}(k)}} \\ &= \sqrt{\frac{f_{k-1}}{C_{k-1}}\cdot\frac{C_k}{f_k}} \\ &= \sqrt{\frac{\lambda_{k-1}}{C_{k-1}}\cdot\frac{C_k}{\lambda_k}}.\end{aligned}$$

Substituting it into N_{k-2}^{opt}, we obtain:

$$\begin{aligned}
N_{k-2}^{\text{opt}} &= \sqrt{\frac{f_{k-2}}{C_{k-2}} \cdot \frac{o_{\text{ef}}(k-1)}{2o_{\text{re}}^{\text{opt}}(k-1)}} \\
&= \sqrt{\frac{\lambda_{k-2}}{C_{k-2}} \cdot \frac{N_{k-1}^{\text{opt}} C_{k-1} + C_k}{\frac{\lambda_{k-1}}{N_{k-1}^{\text{opt}}} + \lambda_k}} \\
&= \sqrt{\frac{\lambda_{k-2}}{C_{k-2}} \cdot \frac{\sqrt{\frac{\lambda_{k-1}}{\lambda_k} C_{k-1} C_k} + C_k}{\sqrt{\lambda_{k-1}\lambda_k \frac{C_{k-1}}{C_k}} + \lambda_k}} \\
&= \sqrt{\frac{\lambda_{k-2}}{C_{k-2}} \cdot \frac{C_k \left(\sqrt{\frac{\lambda_{k-1}}{\lambda_k} \cdot \frac{C_{k-1}}{C_k}} + 1\right)}{\lambda_k \left(\sqrt{\frac{\lambda_{k-1}}{\lambda_k} \cdot \frac{C_{k-1}}{C_k}} + 1\right)}} \\
&= \sqrt{\frac{\lambda_{k-2}}{C_{k-2}} \cdot \frac{C_k}{\lambda_k}}.
\end{aligned}$$

Repeating the above process iteratively, we can compute the optimal values of N_ℓ^{opt} for $\ell = k-3, \ldots, 2, 1$, as given in Equation (2.31) by using values of $N_{k-1}^{\text{opt}}, \ldots, N_{\ell+1}^{\text{opt}}$.

The optimal pattern length, according to Equation (2.18), can be expressed as $W^{\text{opt}} = \sqrt{\frac{o_{\text{ef}}}{\Lambda \cdot o_{\text{re}}^{\text{opt}}}}$, which turns out to be Equation (2.30) with the optimal values of N_ℓ^{opt}.

The optimal overhead, according to Equations (2.19) and (2.33), can be expressed as $H^{\text{opt}} = 2\sqrt{\Lambda \cdot F^{\text{opt}}} + O(\Lambda)$, which gives rise to Equation (2.32). This completes the proof of the theorem. □

Since Proposition 4 shows that all the checkpoints of each level are equally spaced in the pattern, we can readily obtain the following corollary.

Corollary 1. *In an optimal k-level pattern, the number of level-ℓ checkpoints between any two consecutive level-$(\ell+1)$ checkpoints is given by*

$$n_\ell^{opt} = \frac{N_\ell^{opt}}{N_{\ell+1}^{opt}} = \sqrt{\frac{\lambda_\ell}{\lambda_{\ell+1}} \cdot \frac{C_{\ell+1}}{C_\ell}}. \tag{2.35}$$

for all $\ell = 1, \ldots, k-1$.

Remarks. The optimal *k*-level pattern derived in this section has a *rational* number of segments, while the optimal *integer* solution could be much harder to compute. In Section 2.3, we use rounding to derive a practical solution. Still, Equation (2.32)

provides a lower bound on the optimal overhead, which is met very closely in all our experimental scenarios.

2.2.4 Optimal subset of levels

The preceding section characterizes the optimal pattern by using k levels of checkpoints. In many practical cases, there is no obligation to use all available levels. This section addresses the problem of selecting the optimal subset of levels in order to minimize the overall execution overhead.

2.2.4.1 Checkpoint cost models

So far, we have assumed that all the checkpoint costs are fixed under a multi-level checkpointing scheme. In practice, the checkpoint costs may vary depending upon the implementation, and upon the subset of selected levels. In order to determine the optimal subset, we identify the following two checkpoint cost models:

- **Fixed independent costs.** The checkpoint cost C_ℓ at level ℓ is the cost paid to save data at level ℓ, *independently* of the subset of levels used. In this model, the checkpoint costs stay the same for all possible subsets.
- **Incremental costs.** The checkpointing cost C_ℓ at level ℓ is the *additional* cost paid to save data when going from level $\ell - 1$ to ℓ. In this model, the checkpoint cost at a particular level depends on the subset of levels selected.

For example, with $k = 2$ levels and $C_1 = 10, C_2 = 20$, two subsets are possible: $\{1, 2\}$ and $\{2\}$. In the fixed independent cost model, these costs will stay unchanged regardless of the subset chosen. In the incremental cost model, since C_2 is the *additional* cost paid after C_1 is done, when using subset $\{2\}$, i.e., only placing level-2 checkpoints in the pattern, we need to adjust its cost as $C_2' = 10 + 20 = 30$. In both cases, once the subset is decided, the checkpoint costs at the selected levels can be computed and therefore considered as fixed constants. The theoretical analysis presented in Section 2.2.3 can then be used to compute the optimal pattern.

But how to determine the optimal subset of levels? Consider again the example with $k = 2$ levels. In the incremental cost model, Equation (2.32) suggests that the optimal solution (ignoring lower-order terms) uses both levels if and only if

$$\begin{aligned} \sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} &\leq \sqrt{2\left(\lambda_1 + \lambda_2\right)\left(C_1 + C_2\right)} \\ \Leftrightarrow \quad 0 &\leq \left(\sqrt{\lambda_1 C_2} - \sqrt{\lambda_2 C_1}\right)^2, \end{aligned}$$

which is always true when assuming $\lambda_1 \geq \lambda_2$ and $C_1 \leq C_2$. We can easily apply the same argument to show that the optimal subset must contain *all* levels available as long as all checkpoint costs are positive.

In the fixed independent cost model, however, it is not clear whether all available levels should be used. Consider the same example with $k = 2$ levels, and define $\alpha = \frac{\lambda_2}{\lambda_1}$ and $\beta = \frac{C_2}{C_1}$. The optimal solution uses both levels if and only if

$$\begin{aligned} &\sqrt{2\lambda_1 C_1} + \sqrt{2\lambda_2 C_2} \leq \sqrt{2\left(\lambda_1 + \lambda_2\right) C_2} \\ \Leftrightarrow \quad &4\alpha\beta \leq (\beta - 1)^2, \end{aligned}$$

which is not true when $\alpha = 0.5$ and $\beta = 2$. In this case, using only level-2 checkpoints leads to a smaller overhead.

2.2.4.2 Dynamic programming algorithm

In the fixed independent cost model, the optimal subset of levels in a general k-level pattern could well depend on the checkpoint costs and error rates of different levels. One can enumerate all 2^{k-1} possible subsets and select the one that leads to the smallest overhead. The following theorem presents a more efficient dynamic programming algorithm when the number k of levels is large.

Theorem 3. *Suppose there are k levels of checkpoints available and their costs are fixed. Then, the optimal subset of levels to use can be obtained by dynamic programming in $O(k^2)$ time.*

Proof. Let $\mathcal{S}^{\text{opt}}(h) \subseteq \{0, 1, \ldots, h\}$ denote the optimal subset of levels used by a pattern that is capable of handling errors up to level h, and let $H^{\text{opt}}(h)$ denote the corresponding optimal overhead (ignoring lower-order terms) incurred by the pattern. Define $\mathcal{S}^{\text{opt}}(0) = \emptyset$ and $H^{\text{opt}}(0) = 0$. Recall that $\Lambda_{[x,y]} = \sum_{\ell=x}^{y} \lambda_\ell$. We can compute $H^{\text{opt}}(h)$ using the following dynamic programming formulation:

$$H^{\text{opt}}(h) = \min_{0 \leq \ell \leq h-1} \left\{ H^{\text{opt}}(\ell) + \sqrt{2\Lambda_{[\ell+1,h]} C_h} \right\}, \tag{2.36}$$

and the optimal subset is $\mathcal{S}^{\text{opt}}(h) = \mathcal{S}^{\text{opt}}(\ell^{\text{opt}}) \bigcup \{h\}$, where ℓ^{opt} is the value of ℓ that yields the minimum $H^{\text{opt}}(h)$.

The optimal subset of levels to handle all k levels of errors is then given by $\mathcal{S}^{\text{opt}}(k)$ with the optimal overhead $H^{\text{opt}}(k)$. The complexity is clearly quadratic in the total number of levels. □

2.3 Simulations

In this section, we conduct a set of simulations whose goal is threefold: (i) to verify the accuracy of the first-order approximation; (ii) to confirm the optimality of the subset of levels found by the dynamic programming algorithm; and (iii) to evaluate the performance of our approach and to compare it with other multi-level checkpointing algorithms. After introducing the simulation setup in Section 2.3.1, we proceed in two steps. First, in Section 2.3.2, we instantiate the model with realistic parameters

from the literature and run simulations for all possible subsets of levels and roundings. Then, in Section 2.3.3, we instantiate the model with different test cases from the recent work of Di et al. [62, 63] on multilevel checkpointing and compare the overheads obtained with three approaches: (a) Young/Daly's classical formula; (b) our first-order approximation formula; and (c) Di et al.'s iterative/optimal algorithm. The simulator code is publicly available at `http://perso.ens-lyon.fr/aurelien.cavelan/multilevel.zip`, so that interested readers can experiment with it and instantiate the model with parameters of their own choice.

2.3.1 Simulation setup

Checkpoint and recovery costs both depend on the volume of data to be saved, and are mostly determined by the hardware resource used at each level. As such, we assume that recovery cost for a given level is equivalent to the corresponding checkpointing cost, i.e., $R_\ell = C_\ell$ for $1 \leq \ell \leq k$ (unless specified otherwise). This a common assumption [140, 62], even though in practice the recovery cost can be expected to be *somewhat* smaller than the checkpoint cost [62, 63]. All costs are fixed and independent (as discussed in Section 2.2.4.1).

The simulator is fed with k levels of errors and their MTBFs $\mu_\ell = 1/\lambda_\ell$, as well as the resilience parameters C_ℓ and R_ℓ. For each of the 2^{k-1} possible subsets of levels (the last level is always included), we take the optimal pattern given in Theorem 2 and Corollary 1, and then try all possible roundings (floor and ceiling) based on the optimal (rational) number of checkpoints (n_ℓ^{opt} given in Equation (2.35)). For each rounding, we compare the following three overheads:

- **Simulated overhead**, obtained by running the simulation 10000 times and averaging the results;
- **Corresponding theoretical overhead**, obtained from Equations (2.19), (2.23) and (2.25) using the integer solution that corresponds to the rounding;
- **Theoretical lower bound**, obtained from Equation (2.32) with the optimal rational solution.

In the following, we associate Young/Daly's classical formula, defined as $W^{opt} = \sqrt{\frac{2C}{\Lambda}}$, with the highest checkpointing level available, i.e., $C = C_k$. Note that in this case, Young/Daly's formula and Equation (2.30) can be used interchangeably, and the corresponding theoretical overhead is obtained with $H^{opt} = \sqrt{2\Lambda C}$.

2.3.2 Assessing accuracy of first-order approximation

In this section, we run simulations with two sets of parameters, described in Table 2.1. For each set of parameters, we consider all possible subsets of levels. Then, for each subset, we compute the optimal pattern length and number of checkpoints to be used at each level. We show the accuracy of our approach in both scenarios, and we confirm the optimality of the subset of levels returned by the dynamic programming algorithm.

Table 2.1: Sets of parameters (A) and (B) used as inputs for simulations.

Set	From	Level	1	2	3	4
(A)	Moody et al. [140]	C (s)	0.5	4.5	1051	-
		MTBF (s)	5.00e6	5.56e5	2.50e6	-
(B)	Balaprakash et al. [14]	C (s)	10	30	50	150
		MTBF (s)	3.60e4	7.20e4	1.44e5	7.20e5

2.3.2.1 Using set of parameters (A)

The first set of parameters (shown in set (A) of Table 2.1) corresponds to the Coastal platform, a medium-sized HPC system of 1104 nodes at the Lawrence Livermore National Laboratory (LLNL). The Coastal platform has been used to evaluate the Scalable Checkpoint/Restart (SCR) library by Moody et al. [140], who provided accurate measurements for the checkpoint costs using real applications (given in the first row of Table 2.1). There are $k = 3$ levels of checkpoints. First-level checkpoints are written to the local RAMs of the nodes, and this is the fastest method ($0.5s$). Second-level checkpoints are also written to local RAMs, but small sets of nodes collectively compute and store parity redundancy data, which takes a little while longer ($4.5s$). Lastly, Lustre is used to store third-level checkpoints onto the parallel file system, which takes significantly longer time ($1051s$). Failures were analyzed in [140], and the error rates are given in the second row of Table 2.1. Note that the error rate at level 2 is higher than those of levels 1 and 3.

Results: Table 2.2 and Figure 2.4 present the simulation results. Table 2.2 shows, from left to right, the subset of levels used, the number of checkpoints computed by our first-order approximation formula for each possible rounding (N_1, N_2, N_3), the corresponding optimal pattern length ($W^{\text{opt}}(s)$), the simulated overhead (Sim. Ov.), the corresponding theoretical overhead (Th. Ov.), the absolute and relative differences of these two overheads (Ab. Diff. = 100 $\times$ (Sim. Ov. - Th. Ov.), and Rel. Diff. = 100 $\times$ (Sim. Ov. - Th. Ov.)/Sim. Ov.), and finally the theoretical lower bound for this subset (Th. L.B.).

With $k = 3$, there are four possible subsets of levels, and both the best simulated overhead and the corresponding theoretical overhead are achieved for the subset $\{2, 3\}$, with $N_2 = 35$ and $N_3 = 1$ (highlighted in **bold** in the table). First, the difference between the simulated and theoretical overheads is very small, with a difference $< 0.7\%$ in absolute values, and a relative difference ranging from 2.9% (for subset $\{1, 2, 3\}$) to 8.14% (for subset $\{3\}$), which shows the accuracy of the first-order approximation for this set of parameters. The simulated overhead is always higher than the theoretical one, which is expected, because the first-order approximation ignores some lower-order terms. Next, we observe that, for each subset, all roundings of the number of checkpoints yield similar overheads on this platform, and the difference between the best and worst roundings is almost negligible.

Table 2.2: Simulation results using set of parameters (A).

Levels	N_1	N_2	N_3	W^{opt} (s)	Sim. Ov.	Th. Ov.	Abs. Diff.	Rel. Diff.	Th. L.B.
{3}	-	-	1	2.96e4	7.74e-2	7.11e-2	0.63%	8.14%	7.11e-2
{1,3}	14	-	1	3.09e4	7.40e-2	6.85e-2	0.55%	7.43%	6.85e-2
	13	-	1	3.09e4	7.39e-2	6.85e-2	0.54%	7.31%	
{2,3}	-	35	1	7.27e4	**3.44e-2**	**3.33e-2**	0.11%	3.20%	**3.33e-2**
	-	34	1	7.25e4	3.46e-2	**3.33e-2**	0.13%	3.76%	
{1,2,3}	33	33	1	7.27e4	3.46e-2	3.35e-2	0.11%	3.18%	3.35e-2
	32	32	1	7.24e4	3.45e-2	3.35e-2	0.10%	2.90%	

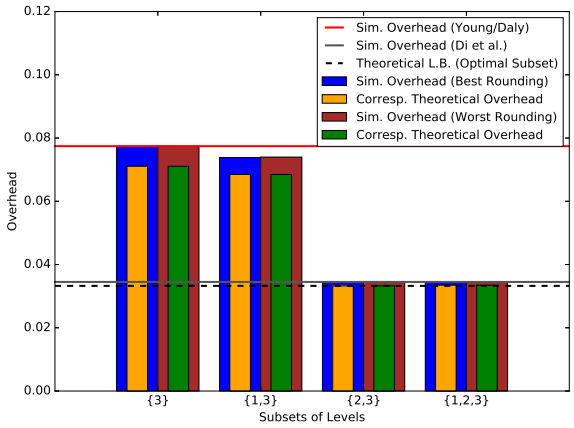


Figure 2.4: Simulated and (corresponding) theoretical overheads for all possible subsets of levels with the best and worst roundings for each subset using set of parameters (A).

Furthermore, using the best subset ($\{2,3\}$) improves the overhead by over 50% compared to using level-3 checkpoints alone (as in Young/Daly's result). This is indeed the subset returned by the dynamic programming algorithm, and the result matches closely the minimum theoretical lower bound. Finally, comparing our result to the one obtained by the optimal two-level algorithm by Di et al. [63] on this best subset, we see that the simulated overheads are similar under the optimal subset, as the patterns found using both approaches share the same number of checkpoints and the pattern lengths are also almost identical.

2.3.2.2 Using set of parameters (B)

The second set of parameters correspond to the execution of the LAMMPS application on the large BG/Q platform Mira at the Argonne National Laboratory (ANL) [14]. The parameters are presented in set (B) of Table 2.1. In this setting, the Fault Tolerance

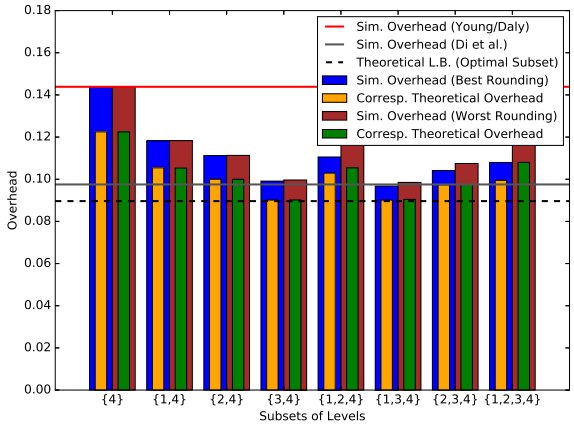


Figure 2.5: Simulated and (corresponding) theoretical overheads for all possible subsets of levels with the best and worst roundings for each subset using set of parameters (B).

Interface (FTI) [17] was used, which has four checkpoint levels ($k = 4$): Local checkpoint; Local checkpoint + Partner-copy; Local checkpoint + Reed-Solomon coding; and PFS-based checkpoint. The MTBFs correspond to the failure rates typically observed for petascale HPC applications [140, 17, 62].

Results: Table 2.3 and Figure 2.5 present the simulation results for this set of parameters. There are 8 possible subsets of levels. As before, we observe that the theoretical overhead is always slightly smaller than the simulated one, with an absolute difference of less than 2%, and a relative difference between 6-14%, demonstrating the accuracy of the model. Again, the results are very close to the theoretical lower bound. For this platform, the simulated overheads vary from 9.68% (with optimal subset of levels $\{1,3,4\}$ found by the dynamic programming algorithm) to 14.3% (with level-4 checkpoints alone). For a given subset of levels, the rounding does not play a significant role, as W^{opt} is also adjusted accordingly (increased or decreased) as a result of rounding. For instance, we observe that, for subset $\{1,2,3,4\}$, the numbers of checkpoints at levels 1 and 2 are halved for the third rounding compared to the first rounding in Table 2.3, but W^{opt} is also reduced by 31%, so that for the same amount of work, the number of checkpoints does not change by much. We can also see that the pattern length W^{opt} for the smallest overhead is around 10400s, but only 2450s for the largest overhead. In fact, the largest pattern lengths are obtained for the *highest cumulated checkpoint cost*, which turns out to be 830s for $\{2,3,4\}$ with $N_2 = 16, N_3 = 4, N_4 = 1$, and for $\{1,2,3,4\}$ with $N_1 = 24, N_2 = 8, N_3 = 4$ and $N_4 = 1$. This is because using more checkpoints both increases the error-free overhead and reduces the time lost due to re-executions upon errors. As a consequence, and to mitigate the aforementioned overhead, the length of the pattern increases (e.g., $W^{\text{opt}} = 17000s$ for $\{2,3,4\}$ and $W^{\text{opt}} = 16600s$ for $\{1,2,3,4\}$). And the converse is also true: when using fewer checkpoints, the error-free overhead decreases and the time lost upon errors increases. In

Table 2.3: Simulation results using set of parameters (B).

Levels	N_1	N_2	N_3	N_4	W^{opt} (s)	Sim. Ov.	Th. Ov.	Abs. Diff.	Rel. Diff.	Th. L.B.
{4}	-	-	-	1	2.45e3	1.43e-1	1.22e-1	1.9%	13.3%	1.22e-1
{1,4}	5	-	-	1	3.79e3	1.18e-1	1.05e-1	1.3%	11.0%	1.05e-1
	4	-	-	1	3.61e3	1.18e-1	1.05e-1	1.3%	11.0%	
{2,4}	-	5	-	1	6.00e3	1.11e-1	1.00e-1	1.1%	9.9%	1.00e-1
{3,4}	-	-	11	1	1.55e4	9.96e-2	9.02e-2	0.94%	9.44%	9.01e-2
	-	-	10	1	1.44e4	9.91e-2	9.01e-2	0.90%	9.08%	
{1,2,4}	9	3	-	1	6.41e3	1.11e-1	1.03e-1	0.8%	7.2%	1.02e-1
	6	2	-	1	5.21e3	1.13e-1	1.04e-1	0.9%	8.0%	
	6	3	-	1	5.84e3	1.11e-1	1.03e-1	0.8%	7.2%	
	4	2	-	1	4.74e3	1.17e-1	1.05e-1	1.2%	10.3%	
{1,3,4}	21	-	7	1	1.58e4	9.72e-2	8.99e-2	0.73%	7.51%	**8.96e-2**
	18	-	6	1	1.40e4	9.82e-2	**8.98e-2**	0.84%	8.55%	
	14	-	7	1	1.04e4	**9.68e-2**	9.01e-2	0.67%	6.92%	
	12	-	6	1	1.26e4	9.85e-2	9.04e-2	0.81%	8.22%	
{2,3,4}	-	16	4	1	1.70e4	1.07e-1	9.75e-2	0.95%	8.9%	9.68e-2
	-	12	3	1	1.36e4	1.04e-1	9.73e-2	0.67%	6.4%	
	-	12	4	1	1.47e4	1.05e-1	9.68e-2	0.82%	7.8%	
	-	9	3	1	1.17e4	1.05e-1	9.75e-2	0.75%	7.1%	
{1,2,3,4}	24	8	4	1	1.66e4	1.09e-1	1.00e-1	0.9%	8.2%	9.92e-2
	18	6	3	1	1.32e4	1.08e-1	9.99e-2	0.81%	7.5%	
	12	4	4	1	1.15e4	1.11e-1	1.03e-1	0.8%	7.2%	
	9	3	3	1	9.17e3	1.14e-1	1.05e-1	0.9%	7.9%	
	16	8	4	1	1.51e4	1.08e-1	9.95e-2	0.85%	7.9%	
	12	6	3	1	1.20e4	1.09e-1	1.00e-1	0.9%	8.3%	
	8	4	4	1	1.05e4	1.16e-1	1.05e-1	1.1%	9.5%	
	6	3	3	1	8.33e3	1.19e-1	1.08e-1	1.1%	9.2%	

order to compensate, the pattern length decreases (e.g., $W^{\text{opt}} = 8330s$ for $\{1,2,3,4\}$ with $N_1 = 6, N_2 = 3, N_3 = 3$ and $N_4 = 1$).

We note that, in this case, our first-order solution slightly outperforms the iterative method by Di et al. [62] on multi-level checkpointing (with a simulated overhead of 9.68e-2 compared to 9.75e-2). The reason is that their algorithm computes a solution under the independent checkpointing model, i.e., checkpoints at different levels are taken according to different independent periods. However, it is not clear how such a model can be implemented in practice due to the difficulties as explained in Section 2.1 and the different options to rollback to a checkpoint in case of a fault. Therefore, we transformed their result to a pattern-based solution by rounding the different numbers of checkpoints obtained using their algorithm to create equal number of checkpoints at level $\ell - 1$ between two consecutive level-ℓ checkpoints. Although the best rounding is selected here for comparison, the result can still change drastically the number of checkpoints computed by their initial rational solution without changing the pattern length, thus increasing the overhead.

2.3.3 Comparing performance of different approaches

In this section, we conduct simulations using settings from Di et al.'s recent work on multilevel checkpointing, which comprises two cases with four levels [62] and eight cases with two levels [63], thus covering a wide range of configurations. For each case, we compare the performance of three different approaches: (a) Young/Daly's classical formula; (b) our first-order approximation formula; and (c) Di et al.'s iterative algorithm.

Table 2.4: Set of parameters (C) used as input for simulations.

Set (C), from Di et al. [62]					
	Level	1	2	3	4
Case #A	C (s)	8	10	80	90
	R (s)	8	10	80	90
	MTBF (s)	2160	1440	8640	21600
Case #B	C (s)	1	20	60	70
	R (s)	1	10	30	35
	MTBF (s)	864	864	1080	1440

2.3.3.1 Using set of parameters (C)

We first run simulations for Cases #A and #B, whose parameters are presented in Table 2.4. These parameters are based on the FTI multilevel checkpointing model and

have been used by Di et al. [62] to evaluate the performance of their approach. Note that the recovery cost is about half that of the checkpointing cost in Case #B.

In their work, Di et al. considered independent checkpointing periods, as opposed to the nested method based on periodic patterns (as discussed in Section 2.1). Although they provided an optimal solution, an iterative approach was used to compute it numerically in contrast to the simple formula we propose in this work. Recall that using independent checkpointing periods allows checkpoints at different levels to be taken simultaneously, which can hardly be done in practice. Adapting their solution to our model results in rational numbers of checkpoints, and we again use rounding to resolve this issue. We find that, using the best roundings for both approaches, their solution turns out to be very similar to ours (with the same number of checkpoints, and close periods with $< 1\%$ difference).

Results: Figure 2.6 presents the overheads for both cases. First, we observe that Di et al.'s optimal iterative algorithm has almost identical performance to our solution, with a simulated overhead around 45% for Case #A and 140% for Case #B under both approaches. However, using Young/Daly's formula to checkpoint only at the highest level yields significantly worse overheads (around 90% for Case #A and 170% for Case #B). Overall, our solution is as good as Di et al.'s optimal numerical one (but has much less complexity), and it is up to 45% better than Young/Daly's formula in Case #A and 30% better in Case #B.

Note that the corresponding theoretical overhead of our solution is close to the simulated one for Case #A, but starts to diverge for Case #B. This is because first-order approximation is only accurate when the resilience parameters and pattern length are small compared to the MTBF, which is no longer true for Case #B. Specifically, we have:

- In Case #A, the optimal subset of levels is $\{2, 4\}$. The optimal pattern has length $W^{\text{opt}} = 1052s$ and consists of $N_2 = 8$ level-2 checkpoints followed by $N_4 = 1$ level-4 checkpoint, meaning that we have a level-2 checkpoint every $131.5s$ of computation. So a level-2 checkpoint is saved every $141.5s$ and a level-4 checkpoint is saved every 1222s. On the other hand, the combined MTBF for errors at levels 1 and 2 (handled by level-2 checkpoints) is $864s$ and the combined MTBF for errors at levels 3 and 4 (handled by level-4 checkpoints) is $6171s$. Hence, we have $\frac{141.5}{864} = 0.164$ and $\frac{1222}{6171} = 0.198$, which are reasonably small, making our solution accurate.

- In Case #B, the optimal subset of levels is $\{1, 4\}$, and the optimal pattern has $W^{\text{opt}} = 223s$, $N_1 = 5$ and $N_4 = 1$. Thus, we have a level-1 checkpoint every $44.6s$ of computation. So a level-1 checkpoint is saved every $45.6s$ and a level-4 checkpoint is saved every $298s$. The MTBF for errors at level 1 is $864s$ and the combined MTBF for errors at levels 2, 3 and 4 (handled by level-4 checkpoints) is $360s$. Thus, we have $\frac{44.6}{864} = 0.052$, which is fine, but $\frac{298}{360} = 0.828$, which is too high and essentially makes the first-order solution inaccurate.

Table 2.5: Set of parameters (D) used as input for simulations.

Set (D), from Di et al. [63]							
	Level	1	2		Level	1	2
Case 1	C (s)	20	50	Case 5	C (s)	10	40
	MTBF (s)	3600	21600		MTBF (s)	432	2160
Case 2	C (s)	20	50	Case 6	C (s)	100	20
	MTBF (s)	1728	8640		MTBF (s)	432	2160
Case 3	C (s)	20	100	Case 7	C (s)	40	200
	MTBF (s)	864	4320		MTBF (s)	288	1440
Case 4	C (s)	10	40	Case 8	C (s)	50	300
	MTBF (s)	864	4320		MTBF (s)	216	1440

Despite the difference between the theoretical and simulated overheads under Case #B, the proximity of our solution to Di et al.'s optimal numerical solution nevertheless shows the usefulness of first-order approximation for determining the optimal multi-level checkpointing patterns.

2.3.3.2 Using set of parameters (D)

Finally, we run simulations for eight cases, whose parameters are presented in Table 2.5. These parameters have been used by Di et al. [63] to evaluate their two-level checkpointing model, and as such, each case consists of only two checkpointing levels. In their work, the authors proposed an optimal solution by solving complex mathematical equations using numerical method. Again, for each case, we compare the simulated overheads obtained with the three different approaches.

In this set of parameters, the MTBF has a large variation, ranging from more than 1 hour (Case 1) to less than 4 minutes (Case 8). Similarly, the checkpointing costs vary from 10s (Cases 4 and 5) to 300s (Case 8). Note that Cases 7 and 8 have both very short MTBFs and very high checkpointing costs, resulting in a lot of errors and recoveries. In particular, the checkpointing cost at level 2 in Case 8 (300s) is larger than the MTBF at level 1 (216s).

Results: Figure 2.7 presents the simulation results for the eight cases. First, we observe that the optimal algorithm by Di et al. only yields a slightly better simulated overhead compared to our simple first-order approximation solution (by less than 2% in Cases 1 to 6). However, our solution always improves significantly over Young/-Daly's formula, from 2% (Case 1) up to 100% (Case 6). Due to their short MTBFs, Cases 7 and 8 stand out and incur much higher overheads compared to the first six cases (thus their results are presented in a separate plot). Still, considering Case 8, we are able to improve over Young/Daly's solution by as much as 2500% (in abso-

lute value of the overhead), and we are off the optimal simulated overhead by only 300%. In addition, Figure 2.7 shows the theoretical overheads obtained both with our formula and the solution provided by Di et al. in [63]. As expected, our first-order approximation remains accurate when the MTBF is large, as in Cases 1, 2 and 4. However, it becomes less accurate with shorter MTBFs and higher error rates, especially in Cases 7 and 8 (which do not represent healthy HPC platforms).

2.3.4 Summary of results

From the simulation results, we conclude that first-order approximation remains a valuable performance model for evaluating checkpointing solutions in HPC systems (as long as the error rates stay reasonably low). We have demonstrated, through an extensive set of simulations with a wide range of parameters, the usefulness of multi-level checkpointing (over using only one level of checkpoints) with significantly reduced overheads. The results also corroborate the analytical study by showing the benefit of selecting an optimal subset of levels among all the levels available. Overall, our approach achieves the optimal or near-optimal performance in almost all cases, except when the MTBF is too small, in which case even the optimal solution yields an unacceptably high overhead (e.g., Case 8 of Table 2.5).

2.4 Related work

Given the checkpointing cost and platform MTBF, classical formulas due to Young [200] and Daly [58] are well known to determine the optimal checkpointing period in the single-level checkpointing scheme. However, this method suffers from the intrinsic limitation that the cost of checkpointing/recovery grows with failure probability, and becomes unsustainable at large scale [82, 30] (even with diskless or incremental checkpointing [151]).

To reduce the I/O overhead, various two-level checkpointing protocols have been studied. Vaidya [185] proposed a two-level recovery scheme that tolerates a single node failure using a local checkpoint stored on a parter node. If more than one failure occurs during any local checkpointing interval, the scheme resorts to the global checkpoint. Silva and Silva [164] advocated a similar scheme by using memory to store local checkpoints, which is protected by XOR encoding. Di et al. [63] analyzed a two-level checkpointing pattern, and proved equal-length segments in the optimal solution. They also provided mathematical equations that can be solved numerically to compute the optimal pattern length and number of segments. Benoit et al. [20] relied on disk checkpoints to cope with fail-stop failures and memory checkpoints coupled with error detectors to handle silent data corruptions. They derived first-order approximation formulas for the optimal pattern length and the number of memory checkpoints between two disk checkpoints.

Some authors have also generalized two-level checkpointing to account for an arbitrary number of levels. Moody et al. [140] implemented this approach in a three-level

Scalable Checkpoint/Restart (SCR) library. They relied on a rather complex Markov model to recursively compute the efficiency of the scheme. Bautista-Gomez et al. [17] designed a four-level checkpointing library, called Fault Tolerance Interface (FTI), in which partner-copy and Reed-Solomon coding are employed as two intermediate levels between local and global disks. Based on FTI, Di et al. [62] proposed an iterative method to compute the optimal checkpointing interval for each level with prior knowledge of the application's total execution time. Hakkarinen and Chen [97] considered multi-level diskless checkpointing for tolerating simultaneous failures of multiple processors. Balaprakash et al. [14] studied the trade-off between performance and energy for general multi-level checkpointing schemes.

While all of these works relied on numerical methods to compute the checkpointing intervals at different levels, this work is the first to provide explicit formulas on the optimal parameters in a multi-level checkpointing protocol (up to first-order approximation as in Young/Daly's classical result).

2.5 Conclusion

This chapter has studied multi-level checkpointing protocols, where different levels of checkpoints can be set; lower levels deal with frequent errors that can be recovered at low cost (for instance with a memory copy), while higher levels allow us to recover from all errors, such as node failures (for instance with a copy in stable storage). We consider a general scenario with k levels of faults, and we provide explicit formulas to characterize the optimal checkpointing pattern, up to first-order approximation. The overhead turns out to be of the order of $\sum_{\ell=1}^{k} \sqrt{2\lambda_\ell C_\ell}$, which elegantly extends Young/Daly's classical formula.

The first-order approximation to the optimal k-level checkpointing pattern uses rational numbers of checkpoints, and we prove that all segments should have equal lengths. We corroborate the theoretical study by an extensive set of simulations, demonstrating that greedily rounding the rational values leads to an overhead very close to the lower bound. Furthermore, we provide a dynamic programming algorithm to determine those levels that should be selected, and the simulations confirm the optimality of the subset of levels returned by the dynamic programming algorithm.

The problem of finding a first-order optimal pattern with an integer number of segments to minimize the overhead remains open. It may well be the case that such an integer pattern is not periodic at each level and uses different-length segments. However, the good news is that the rounding of the rational solution provided in this chapter seems quite efficient in practice.

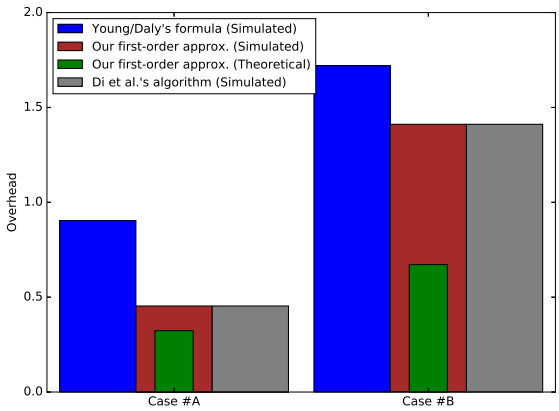


Figure 2.6: Performance comparison of the three different approaches using two cases from Di et al. [62].

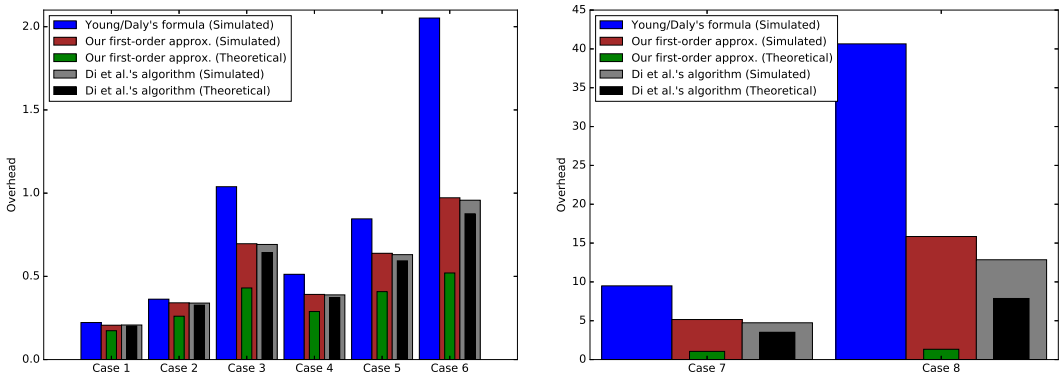


Figure 2.7: Performance comparison of the three different approaches using 8 cases from Di et al. [63].

Chapter 3

Comparing the performance of rigid, moldable and grid-shaped applications on failure-prone HPC platforms

In this chapter, we compare the performance of different approaches to tolerate failures for applications executing on large-scale failure-prone platforms. We study (i) Rigid applications, which use a constant number of processors throughout execution; (ii) Moldable applications, which can use a different number of processors after each restart following a fail-stop error; and (iii) GridShaped applications, which are moldable applications restricted to use rectangular processor grids (such as many dense linear algebra kernels). We start with checkpoint/restart, the de-facto standard approach. For each application type, we compute the optimal number of failures (i.e. that maximizes the yield of the application) to tolerate before relinquishing the current allocation and waiting until a new resource can be allocated, and we determine the optimal yield that can be achieved. For GridShaped applications, we also investigate *Application Based Fault Tolerance* (ABFT) techniques and perform the same analysis, computing the optimal number of failures to tolerate and the associated yield. We instantiate our performance model with realistic applicative scenarios and make it publicly available for further usage. We show that using spare nodes grants a much better yield than currently used strategies that restart after each failure. Moreover, the yield is similar for Rigid, Moldable and GridShaped applications, while the optimal number of failures to tolerate is very high even for a short wait time. Finally, Moldable applications have the advantage to restart less frequently than Rigid applications. The work in this chapter is joint work with Thomas Hérault, Yves Robert, Aurélien Bouteiller, Atsushi Hori, George Bosilca and Jack Dongarra, and has been published in *Parallel Computing* (ParCo) [J5].

3.1 Introduction

Consider a long-running job that requests N processors from the batch scheduler. Resilience to fail-stop errors[1] is typically provided by a Checkpoint/Restart (C/R) mechanism, the de-facto standard approach for High-Performance Computing (HPC) applications. After each failure on one of the nodes used by the application, the application restarts from the last checkpoint but the number of available processors decreases, assuming the application can continue execution after a failure (e.g., using ULFM [28]). Until which point should the execution proceed before requesting a new allocation with N fresh resources from the batch scheduler?

The answer depends upon the nature of the application. For a RIGID application, the number of processors must remain constant throughout the execution. The question is then to decide the number F of processors (out of the N available initially) that will be used as spares. With F spares, the application can tolerate F failures. The application always executes with $N - F$ processors: after each failure, then it restarts from the last checkpoint and continues executing with $N - F$ processors, the faulty processor having been replaced by a spare. After F failures, the application stops when the $(F + 1)$st failure strikes, and relinquishes the current allocation. It then asks for a new allocation with N processors, which takes a *wait time*, D, to start (as other applications are most likely using the platform concurrently). The optimal value of F obviously depends on the value of D, in addition to the application and resilience parameters. The wait time typically ranges from several hours to several days if the platform is over-subscribed (up to 10 days for large applications on the *K*-computer [196]). The metric to optimize here is the (expected) application yield, which is the fraction of useful work per second, averaged over the N resources, and computed in steady-state mode (expected value for multiple batch allocations of N resources).

For a MOLDABLE application, the problem is different: here we assume that the application can use a different number of processors after each restart. The application starts executing with N processors; after the first failure, the application recovers from the last checkpoint and is able to continue with only $N - 1$ processors, albeit with a slowdown factor $\frac{N-1}{N}$. After how many failures F should the application decide to stop[2] and accept to produce no progress during D, in order to request a new allocation? Again, the metric to optimize is the application yield.

Finally, consider an application which must have a given shape (or a set of given shapes) in terms of processor layout. Typically, these shapes are dictated by the application algorithm. In this chapter, we use the example of a GRIDSHAPED application, which is required to execute on a rectangular processor grid whose size can dynamically be chosen. Most dense linear algebra kernels (matrix multiplication, LU, Cholesky and QR factorizations) are GRIDSHAPED applications, and perform more ef-

[1] We use the terms *fail-stop error* and *failure* indifferently.

[2] Another limit is induced by the total application memory Mem_{tot}. There must remain at least ℓ live processors such that $Mem_{tot} \leq \ell \times Mem_{ind}$, where Mem_{ind} is the memory of each processor. We ignore this contraint in this chapter but it would be straightforward to take it into account.

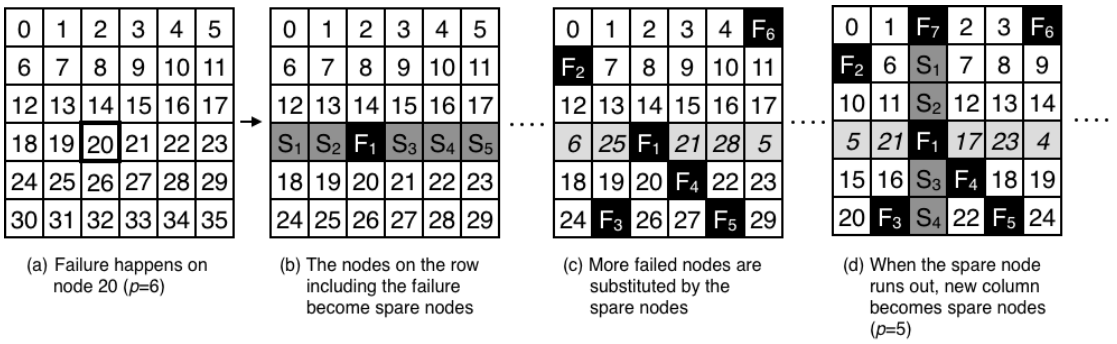


Figure 3.1: Example of node failures substituted by spare nodes in a 2-D GridShaped application.

ficiently on square processor grids than on elongated rectangle ones. The application starts with a (logical) square $p \times p$ grid of $N = p^2$ processors. After the first failure, execution continues on a $p \times (p-1)$ rectangular grid, keeping $p-1$ processors as spares for the next $p-1$ failures (Figure 3.1, b). After p failures, the grid is shrunk again to a $(p-1) \times (p-1)$ square grid (see Figure 3.1(d)), and so on. We address the same question: after how many failures F should the application stop working on a smaller processor grid and request a new allocation, in order to optimize the application yield?

Many GridShaped applications can also be protected from failures by using *Algorithm-Based Fault Tolerant* techniques (ABFT), instead of Checkpoint/Restart (C/R). ABFT is a widely used approach for linear algebra kernels [109, 31]. We present how we can model ABFT techniques instead of C/R and we perform the same analysis: we compute the optimal number of failures to tolerate before relinquishing the allocation, as well as the associated yield.

Altogether, the major contribution of this chapter is to present a detailed performance model and to provide analytical formulas for the expected yield of each application type. We instantiate the model for several applicative scenarios, for which we draw comparisons across application types. Our model is publicly available [166] so that more scenarios can be explored. Notably, we qualify the optimal number of spares for the optimal yield, and the optimal length of a period between two full restarts; it also qualifies how much the yield and total work done within a period are improved by deploying Moldable applications w.r.t. Rigid applications. Finally, for GridShaped applications, it compares the use of C/R and ABFT under various frameworks. Our main result is that using spare nodes grants a significantly higher yield for every kind of application, even for short wait times. We also show that the number of failures to tolerate before resubmitting the application is very high, meaning that it is possible that the application never needs to be resubmitted. Finally, we show the advantage of Moldable applications: while the yield obtained is similar for Rigid and Moldable applications, Moldable applications can tolerate more failures and thus

restart more rarely than RIGID ones. This means that a MOLDABLE application is more likely to terminate before being resubmitted.

The rest of the chapter is organized as follows. Section 3.2 is devoted to formally defining the performance model. Section 3.3 provides formulas for the yield of RIGID, MOLDABLE and GRIDSHAPED applications using the C/R approach, and for the yield of GRIDSHAPED applications using the ABFT approach . All these formulas are instantiated through the applicative scenarios in Section 3.4, to compare the different results. Section 3.5 provides an overview of related work. Finally, Section 3.6 provides final remarks and hints for future work.

3.2 Performance model

This section reviews the key parameters of the performance model. Some assumptions are made to simplify the computation of the yield. We discuss possible extensions in Section 3.6.

3.2.1 Application/platform framework

We consider perfectly parallel applications that execute on homogeneous parallel platforms. Without loss of generality, we assume that each processor has unit speed: we only need to know that the total amount of work done by p processors within T seconds requires $\frac{p}{q}T$ seconds with q processors.

3.2.2 Mean Time Between Failures (MTBF)

Each processor is subject to failures which are IID (independent and identically distributed) random variables[3] following an Exponential probability distribution of mean μ_{ind}, the individual processor MTBF. Then the MTBF of a section of the platform comprised of i processors is given by $\mu_i = \frac{\mu_{ind}}{i}$ [106].

3.2.3 Checkpoints

Processors checkpoint periodically, using the optimal Young/Daly period [200, 58]: for an application using i processors, this period is $\sqrt{2C_i\mu_i}$, where C_i is the time to checkpoint with i processors. We consider two cases to define C_i. In both cases, the overall application memory footprint is considered constant at Mem_{tot}, so the size of individual checkpoints is inversely linear with the number of participating/surviving processors. In the first case, the I/O bandwidth is the bottleneck (which is often the case in HPC platforms – it takes only a few processors to saturate the I/O bandwidth); then the checkpoint cost is constant and given by $C_i = \frac{Mem_{tot}}{\tau_{io}}$, where τ_{io} is the aggregated I/O bandwidth. In the second case, the processor network card is

[3] In datacenters, failures can actually be correlated in space or in time. But to the best of our knowledge, there is no currently available method to analyze their impact without the IID assumption.

the bottleneck, and the checkpoint cost is inversely proportional to number of active processors: $C_i = \frac{Mem_{tot}}{\tau_{xnet} \times i}$, where τ_{xnet} is the available network card bandwidth, i.e. the bandwidth available for one and only one processor, and $\frac{Mem_{tot}}{i}$ the checkpoint size.

We denote the recovery time with i processors as R_i. For all simulations we use $R_i = C_i$, assuming that the read and write bandwidths are identical.

3.2.4 Wait Time

Job schedulers allocate nodes to given applications for a given time. They aim at optimizing multiple criteria, depending on the center policy. These criteria include fairness (balancing the job requests between users or accounts), platform utilization (minimizing the number of resources that are idling), and job makespan (providing the answer as fast as possible). Combined with a high resource utilization (node idleness is usually in the single digit percentage for a typical HPC platform), a job has to wait a *Wait Time* (D) between its submission and the beginning of its execution.

Job schedulers implement the selection based on the list of submitted jobs, each job defining how many processors it needs and for how long. That definition is, in most cases, unchangeable: an application may use less resource than what it requested, but the account will be billed for the requested resource, and it will not be able to re-dimension the allocation during the execution.

Thus, if after some failures, an application has not enough resources left to efficiently complete, it will have to relinquish the allocation, and request a new one. During the wait time D, the application does not execute any computation to progress towards completion: its yield is zero during D seconds.

3.2.5 Objective.

We consider a long-lasting application that requests a resource allocation with N processors. We aim at deriving the optimal number of failures F that should be tolerated before paying the wait time and requesting a new allocation. We aim at maximizing the *yield* $\mathcal{Y}$ of the application, defined as the fraction of time during the allocation length and wait time where the N resources perform useful work. More precisely, the yield is defined by the following formula:

$$\mathcal{Y} = \frac{\text{total time spent computing for all processors}}{\text{number of processors } \times \text{ total execution time of application}}.$$

. Of course a spare does not perform useful work when idle, processor do not compute when checkpointing or recovering, re-execution odes not account for actual work, and no processor is active during wait time. All this explains that the yield will always be smaller than 1. We will derive the value of F that maximizes $\mathcal{Y}$ for the three application types using C/R (and both C/R and ABFT for GridShaped applications).

3.3 Expected yield

This section is the core of the chapter. We compute the expected yield for each application type, Rigid (Section 3.3.1), Moldable (Section 3.3.2) and GridShaped (Section 3.3.3), using the C/R approach, and compare it with ABFT for GridShaped in Section 3.3.4.

3.3.1 Rigid application

We first consider a Rigid application that can be parallelized at compile-time to use any number of processors but cannot change this number until it reaches termination. There are N processors allocated to the application. We use $N - F$ for execution and keep F as spares. The execution is protected from failures by checkpoints of duration C_{N-F}. Each failure striking the application will incur an in-place restart of duration R_{N-F}, using a spare processor to replace the faulty one. However, when the $(F+1)^{st}$ failure strikes, the job will have to stop and perform a full restart, waiting for a new allocation of N processors to be granted by the job scheduler.

We define $\mathcal{T}_R$ as the expected duration of an execution period until the $(F+1)^{st}$ failure strikes. The first failure is expected to strike after μ_N seconds, the second failure μ_{N-1} seconds after the first one, and so on. We relinquish the allocation after $F+1$ failures and wait some time D. As faults can also happen during the checkpoint and the recovery, this means that:

$$\mathcal{T}_R = \sum_{i=N}^{N-F} \mu_i + D. \tag{3.1}$$

What is the total amount of work $\mathcal{W}_R$ computed during a period $\mathcal{T}_R$? During the sub-period of length μ_i, there are $\frac{\mu_i}{\sqrt{2C_{N-F}\mu_{N-F}}}$ checkpoints, each of length C_{N-F}. The failure hits one of live processors, either a working processor or a spare. In both cases, the number of live processors decreases. if the failure hits a spare, it has no immediate impact on the application, except that the number of available spares decreases. If the failure hits a working processor, which happens with probability $\frac{N-F}{i}$, some work is lost, and a restart is needed. During each sub-period, and weighting the cost by the probability of the failure hittingt a working processor during that sub-period, the work lost by each processor by the end of the sub-period is in average $\frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2} \cdot \frac{N-F}{i}$ (see [106] for further details). Each time there is a failure, the next sub-period is thus started by a restart R_{N-F} with probability $\frac{N-F}{i+1}$, except for the first sub-period which always starts by a restart (it corresponds to reading input data at the beginning of the allocation). All in all, during the sub-period of length μ_i with $i \neq N$, each processor works during

$$\frac{1}{1+\frac{C_{N-F}}{\sqrt{2C_{N-F}\mu_{N-F}}}} \cdot \left(\mu_i - R_{N-F} \cdot \frac{N-F}{i+1} - \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2} \cdot \frac{N-F}{i} \right)$$

seconds. The first fraction corresponds to the proportion of the time that is used for useful computations and not for checkpointing. This fraction is actually: $\frac{\text{period time}}{\text{period time+checkpoint time}} = \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{\sqrt{2C_{N-F}\mu_{N-F}}+C_{N-F}}$ which is equivalent to the former fraction after simplification.
Finally, each processor works during

$$\frac{1}{1+\frac{C_{N-F}}{\sqrt{2C_{N-F}\mu_{N-F}}}} \cdot \left(\mu_N - R_{N-F} - \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2} \cdot \frac{N-F}{i}\right)$$

seconds in the first sub-period of length μ_N as it always starts by reading the initial data.

There are $N-F$ processors at work, hence, re-arranging terms, we obtain that

$$\mathcal{W}_R = \frac{N-F}{1+\frac{C_{N-F}}{\sqrt{2C_{N-F}\mu_{N-F}}}} \cdot \sum_{i=N}^{N-F} \left(\mu_i - (R_{N-F} + \frac{\sqrt{2C_{N-F}\mu_{N-F}}}{2}) \cdot \frac{N-F}{i}\right) \tag{3.2}$$

Indeed, the factor for R_{N-F} is $\frac{N-F}{i+1}$ for all subperiods except the first one, i.e. $N-F \leq i \leq N-1$, which means it is equivalent to $\frac{N-F}{i}$ with $N-F+1 \leq i \leq N$. Moreover, $\frac{N-F}{N-F} = 1$ which is the corresponding factor for the first subperiod, so by summing all the terms we get to $\sum_{i=N}^{N-F} R_{N-F} \cdot \frac{N-F}{i}$.

During the whole duration $\mathcal{T}_R$ of the period, in the absence of failures and protection techniques, the application could have used all the N processors to compute continuously. Thus the effective yield with protection for the application during $\mathcal{T}_R$ is reduced to $\mathcal{Y}_R$:

$$\mathcal{Y}_R = \frac{\mathcal{W}_R}{N \cdot \mathcal{T}_R}$$

3.3.2 Moldable application

We now consider a MOLDABLE application that can use a different number of processors after each restart. The application starts executing with N processors. After the first failure, the application recovers from the last checkpoint and is able to continue with only $N-1$ processors, after paying the restart cost R_{N-1}, albeit with a slowdown factor $\frac{N-1}{N}$ of the parallel work per time unit. After $F+1$ failures, the application stops, just as it was the case for a RIGID application.

We define $\mathcal{T}_M$ as the expected duration of an execution period until the $(F+1)^{st}$ failure strikes. The length of a period is

$$\mathcal{T}_M = \sum_{i=N}^{N-F} \mu_i + D, \tag{3.3}$$

the same as for RIGID applications.

However, for the total amount of work $\mathcal{W}_M$ during a period, things are slightly different. To compute the total amount of work $\mathcal{W}_M$ during a period $\mathcal{T}_M$, we proceed as before and consider each sub-period. During the sub-period of length μ_i, there are $\frac{\mu_i}{\sqrt{2C_i\mu_i}}$ checkpoints, each of length C_i. There is also a restart R_i at the beginning of each sub-period, and the average time lost is $\frac{\sqrt{2C_i\mu_i}}{2}$. The probability that the failure strikes a working processor is always 1, because all alive processors are working during each sub-period. Overall, there are i processors at work during the sub-period of length μ_i, and each of them actually works during

$$\frac{\mu_i - R_i \ - \frac{\sqrt{2C_i\mu_i}}{2}}{1 + \frac{C_i}{\sqrt{2C_i\mu_i}}}$$

seconds. Altogether, we derive that

$$\mathcal{W}_M = \sum_{i=N}^{N-F} i \times \frac{\mu_i - R_i \ - \frac{\sqrt{2C_i\mu_i}}{2}}{1 + \frac{C_i}{\sqrt{2C_i\mu_i}}} \tag{3.4}$$

The yield of the MOLDABLE application is then:

$$\mathcal{Y}_M = \frac{\mathcal{W}_M}{N \cdot \mathcal{T}_M}$$

3.3.3 GridShaped application

Next, we consider a GRIDSHAPED application, defined as a moldable execution which requires a rectangular processor grid. Here we mean a *logical* grid, i.e. a layout of processes whose communication pattern is organized as a 2D grid, not a *physical* processor grid where each processor has four neighbors directly connected to it. Indeed, there is little hope to use physical grids today. Some modern architectures have a multi-dimensional torus as physical interconnexion network, but the job scheduler never guarantees that allocated nodes are adjacent, let alone are organized along an actual torus. This means that the actual time to communicate with a logically adjacent processor is variable, depending upon the length of the path that connects them, and also upon the congestion of the links within that path (these links are likely to be shared by other paths). Other architecture communicate through a hierarchical interconnexion switch, hence a 2D processor grid is not meaningful for such architectures. Altogether, this explains that one targets logical process grids, not physical processor grids. Now why do the application needs a process grid? State-the-art linear algebra kernels such as matrix product, LU, QR and Cholesky factorizations, are most efficient when the data is partitioned across a logical grid of processes, preferably a square, or

at least a balanced rectangle of processes [150]. This is because the algorithms are based upon outer-product matrix updates, which are most efficiently implemented on (almost) square grids. Say you start with 64 working processors, arranged as a 8×8 process grid. When one processor fails, the squarest grid would be $63 = 9 \times 7$, and then after a second failure we get $62 = 31 \times 2$ which is way too elongated to be efficient. After the first failure, it is more efficient to use a 8×7 grid and keep 7 spares; then we use spares for the next 7 failures, after which we shrink to a 7×7 grid (and keep 7 spares), and so on.

For the analysis, assume that the application starts with a square $p \times p$ grid of $N = p^2$ processors. After the first failure, execution continues on a $p \times (p-1)$ rectangular grid, keeping $p-1$ processors as spares for the next $p-1$ failures. After p failures, the grid is shrunk again to a $(p-1) \times (p-1)$ square grid, and the execution continues on this reduced-size square grid. After how many failures F should the application stop, in order to maximize the application yield?

The derivation of the expected length of a period and of the total work is more complicated for GridShaped than for Rigid and Moldable. To simplify the presentation, we outline the computation of the yield only for values of F of the form $F = 2pf - f^2$, hence $p^2 = F + (p-f)^2$, meaning that we stop shrinking and request a new allocation when reaching a square grid of size $(p-f) \times (p-f)$ for some value of $f < p$ to be determined. Obviously, we could stop after any number of faults F, and the publicly available software [166] shows how to compute the optimal value of F without any restriction.

We start by computing an auxiliary variable: on a $(p_1-1) \times p_2$ grid with $p_1 \geq p_2$, the expected time to move from $p_2 - 1$ spare nodes to no spare nodes will be denoted by $T_G(p_1, p_2)$. It means that the number of computing nodes never changes and is $(p_1-1)p_2$. It always starts with a restart $R_{(p_1-1)p_2}$, because having $p_2 - 1$ spare nodes means that a failure just occurred on one of the $p_1 p_2$ processors that were working just before that failure, and we had to remove a row from the process grid. As previously this time is the sum of all the intervals between each failure, namely:

$$T_G(p_1, p_2) = \sum_{i=p_1 p_2 - 1}^{(p_1-1)p_2} \mu_i.$$

Going from p^2 processors down to $(p-f)^2$ processors thus require a time

$$\mathcal{T}_G = \mu_{p^2} + \sum_{g=0}^{f-1} \big(T_G(p-g, p-g) + T_G(p-g, p-g-1) \big) + D.$$

We simply add the time before the first failure and the wait time to the time needed to move from a grid of size p^2 to $(p-1)^2$, to $(p-2)^2$, ..., to $(p-f)^2$.

Similarly, we define the auxiliary variable $W_G(p_1, p_2)$ as the parallel work when moving from a $(p_1-1) \times p_2$ grid with $p_2 - 1$ spare nodes to a $(p_1-1) \times p_2$ grid with no spare node, where $p_1 \geq p_2$. There are $(p_1-1)p_2$ processors working during all

sub-periods. Without restart and re-execution, this work is $(p_1-1)p_2 \cdot \sum_{i=p_1p_2-1}^{(p_1-1)p_2} \mu_i$. Any failure which hits one of the working processors calls for a restart $R_{(p_1-1)p_2}$ and incurs some lost work: $\frac{\sqrt{2C_{(p_1-1)p_2}}}{2}$ in average. The first sub-period starts with a restart $R_{(p_1-1)p_2}$,because the application (distributed on a grid of $p_1 \times p_2$ was previously hit by a failure, except if this is the beginning of a new allocation (which case will be dealt with later on). Then, for all other sub-periods, a restart is taken if one of $(p_1-1)p_2$ computing processors was hit by a failure. This means that the sub-period with i processors alive (of length μ_i) starts with a restart $R_{(p_1-1)p_2}$ with probability $\frac{(p_1-1)p_2}{i+1}$, for $i \leq p_1p_2-2$. Similarly, for all sub-periods with i processors alive, we lose the expected compute time $\frac{\sqrt{2C_{(p_1-1)p_2}\mu_{(p_1-1)p_2}}}{2}$ with probability $\frac{(p_1-1)p_2}{i}$. Finally, the checkpoint period evolves with the number of processors, just as for MOLDABLE applications. We derive the following formula:

$$
\begin{aligned}
W_G(p_1,p_2) = & \frac{(p_1-1)p_2}{1+\frac{C_{(p_1-1)p_2}}{\sqrt{2C_{(p_1-1)p_2}\mu_{(p_1-1)p_2}}}} \quad \times \\
& \Bigg(\mu_{p_1p_2-1} - R_{(p_1-1)p_2} - \frac{\sqrt{2C_{(p_1-1)p_2}\mu_{(p_1-1)p_2}}}{2} \cdot \frac{(p_1-1)p_2}{p_1p_2-1} \\
& + \sum_{i=p_1p_2-2}^{(p_1-1)p_2} \left(\mu_i - R_{(p_1-1)p_2} \cdot \frac{(p_1-1)p_2}{i+1} - \frac{\sqrt{2C_{(p_1-1)p_2}\mu_{(p_1-1)p_2}}}{2} \cdot \frac{(p_1-1)p_2}{i} \right) \Bigg)
\end{aligned}
$$

Going from p^2 processors down to $(p-f)^2$ processors thus corresponds to a total work

$$
\begin{aligned}
\mathcal{W}_G = & \frac{p^2}{1+\frac{C_{p^2}}{\sqrt{2C_{p^2}\mu_{p^2}}}} \cdot \left(\mu_{p^2} - R_{p^2} - \frac{\sqrt{2C_{p^2}\mu_{p^2}}}{2} \right) \\
& + \sum_{g=0}^{f-1} \left(W_G(p-g,p-g) + W_G(p-g,p-g-1) \right)
\end{aligned}
$$

We use the previously computed function just as we did with the time and we add the work done during the first sub-period on the initial grid of size p^2 (this is the special case for the beginning of an allocation that was mentioned above). Its computation is similar to that of other subperiods.

The yield of the GRIDSHAPED application is then:

$$\mathcal{Y}_G = \frac{\mathcal{W}_G}{N \cdot \mathcal{T}_G}$$

where $N = p^2$.

3.3.4 ABFT for GridShaped

Finally, in this section, we investigate the impact of using *Algorithm-Based Fault Tolerant* techniques, or ABFT, instead of Checkpoint:Restart (C/R). Just as before, we build a performance model that uses first-order approximations. In particular, we do not consider overlapping failures, thereby allowing for a failure-free reconstruction of lost data after a failure. This first-order approximation is accurate up to a few percent, whenever the failure rate is not too high, or more precisely, when the MTBF remains an order of magnitude higher than resilience parameters [106]. Note that this is the case for state-of-the-art platforms, but may prove otherwise whenever millions of nodes are assembled in the forthcoming years.

Consider a matrix factorization on a $p \times p$ grid. The matrix is of size $n \times n$ and is partitioned into tiles of size $b \times b$. These tiles are distributed in a 2D block-cyclic fashion across processors. Letting $n = pbr$, each processor initially holds r^2 tiles. Every set of p consecutive tiles in the matrix is checksummed into a new tile, which is duplicated for resilience. These two new tiles are added to the right border of the matrix and will be distributed in a 2D block-cyclic fashion across processors, just as the original matrix tiles. In other words, we add $2pr^2$ new tiles, extending each tile row of the matrix (there are pr such tile rows) with $2r$ new tiles. Fig. 3.2 illustrates this scheme: the white area represents the original user matrix of size $n \times n$, split in tiles of size $b \times b$, and distributed over a $p \times p$ process grid. The number in each tile represents the rank that hosts a given tile of the matrix. There are two groups of tile-columns of checksums: the light grey ones checksum the right end of the matrix, and the dark grey ones checksum the left part of the matrix. For a bigger matrix, more groups would be added, each group accumulate the sum of p consecutive tile-columns of the matrix. In each group, there are two tile-columns: the checksum and its replica. These new tiles will be treated as regular matrix tiles by the ABFT algorithm, which corresponds to a ratio $\frac{2pr^2}{p^2r^2} = \frac{2}{p}$ of extra work, and to a failure-free slowdown factor $1 + \frac{2}{p}$ [31].

Now, if a processor crashes, we finish the current step of the factorization on all surviving processors, and then reconstruct the tiles of the crashed processor as follows:

- For each tile lost, there are $p - 1$ other tiles (the ones involved in the same checksum as the lost tile) and at least one checksum tile (maybe two if the crashed processor did not hold any of the two checksum tiles for that lost tile). This is what is needed to enable the reconstruction. We solve a linear system of size b and reconstruct the missing tile for a time proportional to $b^3 + pb^2$.
- We do this for the r^2 tiles of the crashed processor, for a total time of $O(r^2(b^3 + pb^2)\tau_a)$, where τ_a is the time to perform a floating-point operation.

Doing so, we reconstruct the same tile as if we had completed the factorization step without failure.

Then, there are two cases: the ABFT algorithm relies on a process grid, so the application behaves similarly to a GridShaped application. If spare nodes are available,

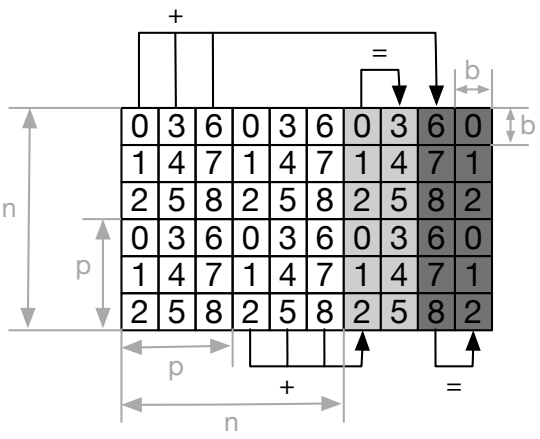


Figure 3.2: Example of data redundancy and checksuming in an ABFT factorization. Each white square represents a matrix tile; numbers in the square represent the rank on which this tile is hosted; grey tiles represent the checksums and their replica (symbolized by the $=$ arrow). In this case, $p = 3$, $n = 6b$.

one of them is selected and inserted within the process grid at the place of the crashed processor, at a cost of communicating $O(r^2b^2)$ matrix coefficients (the amount of data held by the faulty processor). If, on the other hand, there are no spare nodes, we have to start the redistribution of the matrix onto a $p \times (p-1)$ grid. The distribution is operated in parallel across the p grid rows. Within a processor row, most of the tiles will have to change owner to keep enforcing a 2D block-cyclic distribution, which implies $O(\frac{n^2}{p}) = O(r^2pb^2)$ communications as we redistribute everything on every row. Since all rows operate in parallel, the time for the redistribution is $O(r^2pb^2\tau_c)$, where τ_c is the time to communicate a floating point number. Altogether, the total cost to recover from a failure is $O(r^2b^2(b\tau_a + p(\tau_a + \tau_c)))$.

In the following, we compute the expected yield of a linear algebra kernel protected by ABFT. Again, we consider numbers of failures of the form $F = 2pf - f^2$ so that $p^2 = F + (p-f)^2$. Again, we could stop after any number of faults F, and the publicly available software [166] shows how to do so.

We first compute the expected time between two full restarts:

$$\mathcal{T}_{ABFT} = \sum_{i=p^2}^{(p-f)^2} \mu_i + D.$$

As previously, we tolerate failures up to reaching a processor grid size of $(p-f) \times (p-f)$, each inter-arrival time being the MTBF of the platform with the corresponding number of alive processors.

Now, we define the auxiliary variable $W_{ABFT}(p_1, p_2)$ as the parallel work when

moving from a $(p_1-1)\times p_2$ grid with p_2-1 spare nodes to a $(p_1-1)\times p_2$ grid with no spare node. There are $(p_1-1)p_2$ processors working during all sub-periods, just as it was the case for GridShaped applications. A failure-free execution would imply a parallel work of $\frac{1}{1+\frac{2}{p}}\sum_{i=p_1p_2-1}^{(p_1-1)p_2}\mu_i$ which is the total time of computation divided by overhead added with the checksum tiles. However, for each failure we need to reconstruct the lost tiles and either pay a redistribution cost (during the first sub-period where we just reduced the size of the grid because we had no spare) or a communication cost to send data to one of the p_2-1 spare nodes (all other sub-periods where we select a spare). This happens if and only if the failure stroke a working processor just as in the GridShaped case, i.e. with probability $\frac{(p_1-1)p_2}{i+1}$. In the end, since there are $(p_1-1)p_2$ processors at work, we get the following formula:

$$W_{ABFT}(p_1,p_2)=\frac{(p_1-1)p_2}{1+\frac{2}{p}}\left(\mu_{p_1p_2-1}-RD_{p_1}+\sum_{i=p_1p_2-2}^{(p_1-1)p_2}(\mu_i-RP\cdot\frac{(p_1-1)p_2}{i+1})\right),$$

where RP is the cost to replace a faulty processor by a spare, namely

$$RP=r^2(b^3+pb^2)\tau_a+r^2b^2\tau_c,$$

and RD_i is the cost to reistribute data, namely

$$RD_i=r^2(b^3+pb^2)\tau_a+\frac{n^2}{i}\tau_c.$$

To compute these values, we proceed as follows:

- The reconstruction of the lost tiles always takes $r^2(b^3+pb^2)$ floating-point operations, and the enrollment of the spare requires that it receives r^2b^2 floating-point values, which directly leads to the value of RP, which is independent of the number of processors;
- However, when we redistribute the tiles to shrink the grid, the time needed increases as the number of processors decreases because each of them has to gather more data: it requires $O(\frac{n^2}{i})$ communications when redistributing from a $i\times j$ grid of processors to a $(i-1)\times j$ grid, or similarly from a $j\times i$ grid to a $j\times(i-1)$ grid, where $j=i$ or $j=i-1$.

Overall, going from a $p\times p$ grid to a $(p-f)\times(p-f)$ grid corresponds to a total work of $\mathcal{W}_{ABFT}$. At the beginning of the allocation, we need to read the input data, then we wait for the first failure to happen (giving a work of $\mu_{p^2}-R_{p^2}$ during the first sub-period that we divide by the overhead added by the checksum tiles) and then we use our auxiliary variables to decrease the size of the grid step by step. This leads to the following:

$$\mathcal{W}_{ABFT}=\frac{p^2(\mu_{p^2}-R_{p^2})}{1+\frac{2}{p}}+\sum_{i=0}^{f-1}\left(W_{ABFT}(p-i,p-i)+W_{ABFT}(p-i,p-1-i)\right).$$

The yield of the application protected with ABFT is then:

$$\mathcal{Y}_{ABFT} = \frac{\mathcal{W}_{ABFT}}{N \cdot \mathcal{T}_{ABFT}}$$

where $N = p^2$.

3.4 Applicative scenarios

We consider several applicative scenarios in this section. We start with a platform inspired from existing ones in Section 3.4.1, then we study the impact of several key parameters in Section 3.4.2. Finally, we compare ABFT and C/R for a GridShaped application in Section 3.4.3.

3.4.1 Main scenario

As a main applicative scenario using C/R, we consider a platform with 22,250 nodes (150^2), with a node MTBF of 20 years, and an application that would take 2 minutes to checkpoint (at 22,250 nodes). In other words, we let $N = 22,500$, $\mu_{ind} = 20y$ and $C_i = C = 120s$. These values are inspired from existing platforms: the Titan supercomputer at OLCF [95], for example, holds 18,688 nodes, and experiences a few node failures per day, implying a node MTBF between 18 and 25 years. The filesystem has a bandwidth of 1.4TB/s, and nodes altogether aggregate 100TB of memory, thus a checkpoint that would save 30% of that system should take in the order of 2 minutes to complete. In other words, $C_i = C = 120$ seconds for all $i \leq 18,688$.

Figure 3.3 shows the yield that can be expected if doing a full restart after an optimal number of failures, as a function of the wait time, for the three kind of applications considered (Rigid, Moldable and GridShaped). We also plot the expected yield when the application experiences a full restart after each failure (NoSpare). First, one sees that the three approaches that avoid paying the cost of a wait time after every failure experience a comparable yield, while the performance of the NoSpare approach quickly degrades to a small efficiency (30% when the wait time is around 14h).

The zoom box to differentiate the Rigid, Moldable and GridShaped yield shows that the Moldable approach has a slightly higher yield than the other ones, but only for a minimal fraction of the yield. This is expected, as the Moldable approach takes advantage of all living processors, while the GridShaped and Rigid approaches sacrifice the computing power of the spare nodes waiting for the next failure. However, the size of the gain is small to the point of being negligible. The GridShaped approach experiences a yield whose behavior changes in steps: it starts with a constant slope, under the Rigid yield, until the wait time reaches 8h at which point both Rigid and GridShaped yields are the same. The slope of GridShaped then becomes smaller, exhibiting a better yield than Rigid and slowly reaching the yield of Moldable. If we extend the wait time, or change the configuration to experience more phase changes (as

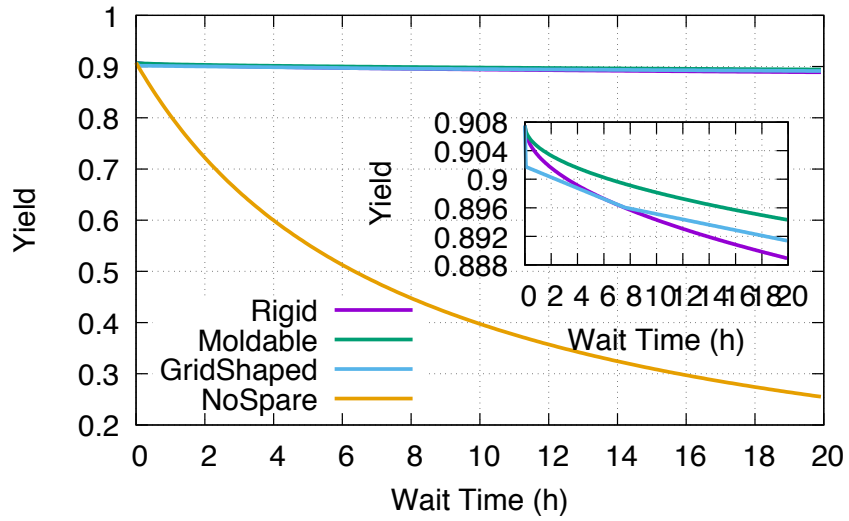


Figure 3.3: Optimal yield as function of the wait time, for the different types of applications.

is done in Section 3.4.2 below), the yield of GridShaped would reach the same value as the yield of Moldable, at which point the slope of GridShaped would change again and become higher. This phenomenon is explained by the next figures.

Figure 3.4 shows the number of failures after which the application should do a full restart, to obtain an optimal yield, as a function of the wait time, for the three kind of applications considered. We observe that this optimal is quickly reached: even with long wait times (e.g. 10h), 170 to 250 failures (depending on the method) should be tolerated within the allocation before relinquishing it. This is small compared to the number of nodes: less than 1% of the resource should be dedicated as spares for the Rigid approach, and after losing 1% of the resource, the Moldable approach should request a new allocation.

This is remarkable, taking into account the poor yield obtained by the approach that does not tolerate failures within the allocation. Even with a small wait time (assuming the platform would be capable of re-scheduling applications that experience failures in less than 2h), Figure 3.3 shows that the yield of the NoSpare approach would decrease to 70%. This represents a waste of 30%, which is much higher than the recommended waste of 10% for resilience in the current HPC platforms recommendations [42, 55]. Comparatively, keeping only 1% of additional resources (within the allocation) would allow to maintain a yield at 90%, for every approach considered.

The GridShaped approach experiences steps that correspond to using all the spares created when redeploying the application over a smaller grid before relinquishing the allocation. As illustrated in Figure 3.3, the yield evolves in steps, changing the slope of a linear approximation radically when redeploying over a smaller grid. This has

for consequence that the maximal yield is always at a slope change point, thus at the frontier of a new grid size. It is still remarkable that even with very small wait times, it is more beneficial to use spares (and thus to lose a full row of processors) than to redeploy immediately.

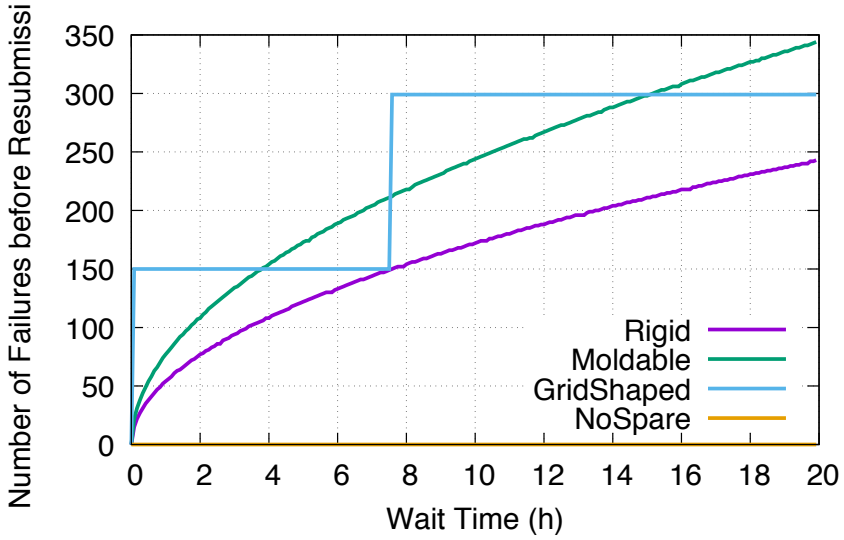


Figure 3.4: Optimal number of failures tolerated between two full restarts, as function of the wait time, for the different types of applications.

Figure 3.5 shows the maximal length of an allocation: after such duration, the job will have to fully restart in order to maintain the optimal yield. This figure illustrates the real difference between the Rigid and Moldable approaches: although both approaches are capable of extracting the same yield, the Moldable approach can do so with significantly longer periods between full restarts. This is important when considering real life applications, because this means that the applications using a Moldable approach have a higher chance to complete before the first full restart, and overall will always complete in a lower number of allocations than the Rigid approach.

Finally, Figure 3.6 shows an upper limit of the duration of the wait time in order to guarantee a given yield for the three applications. In particular, we see that to reach a yield of 90%, an application which would restart its job at each fault would need that restart to be done in less than 6 minutes whereas the Rigid and GridShaped approaches need a full restart in less than 3 hours approximately. This bound goes up to 7 hours for the Moldable approach. In comparison, with a wait time of 1 hour, the yield obtained using NoSpare is only 80%. This shows that, using these parameters, it seems impossible to guarantee the recommended waste of 10% without tolerating (a small) number of failures before rescheduling the job.

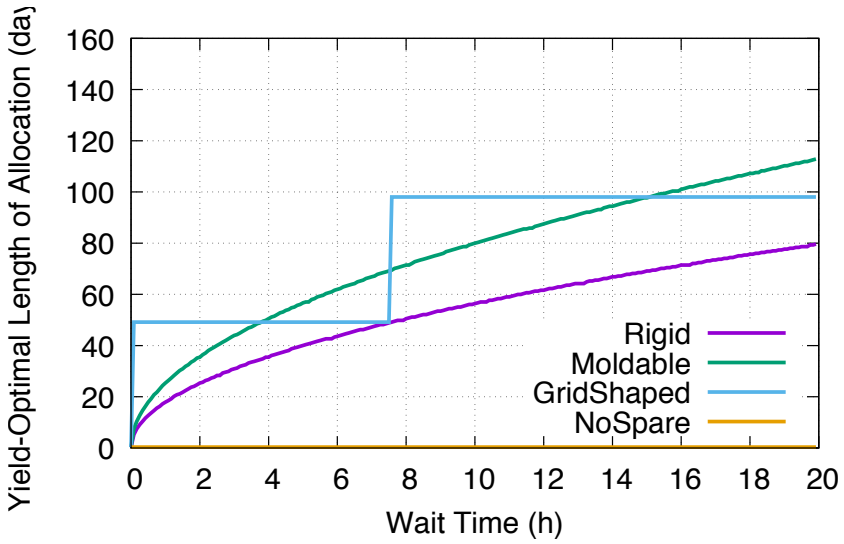


Figure 3.5: Optimal length of allocations, for the different types of applications.

3.4.2 Varying key parameters

We performed a full-factorial 4 level design simulation to assess the impact of key parameters. We tried all combinations of MTBF (5 years, 10 years, 20 years, 50 years), checkpointing cost (2 minutes, 10 minutes, 30 minutes, 60 minutes) and application size ($50 \times 50 = 2500$, $150 \times 150 = 22500$, $250 \times 250 = 62500$, $350 \times 350 = 122500$). Not all results are presented for conciseness, but they all give very similar results compared to the main scenario of Section 3.4.1.

Figure 3.7 shows the yield and the corresponding allocation length for different values of the MTBF, when using the largest application size $N = 350 \times 350$. The top subfigure is for $\mu_{ind} = 5$ years while the bottom subfigure is for $\mu_{ind} = 50$ years. The checkpoint cost is $C_i = C = 10$ minutes. As expected, the yield increases when the MTBF increases. However, the variation of the allocation length is a bit different. At first, it decreases with the MTBF (for example, with a wait time of 10 hours, it decreases from around 150 days to around 100 days when μ_{ind} decreases from 50 years to 20 years). This is because the optimal number of faults allowed is not much higher when $\mu_{ind} = 20$ years, thus it decreases the overall allocation length. However, when we reach limit behaviours with short node MTBF, the number of failures to tolerate explodes and increases the allocation length. We can also see that the allocation length for GridShaped applications tends to follow that of a Moldable application when μ_{ind} decreases.

Figure 3.8 shows the optimal number of faults to tolerate for the four different application sizes (with $\mu_{ind} = 20$ years and $C_i = C = 10$ minutes). We can see from this experiment that the number of tolerated failures stays within a small percentage of the total number of processors. In particular, the optimal number of failures allowed

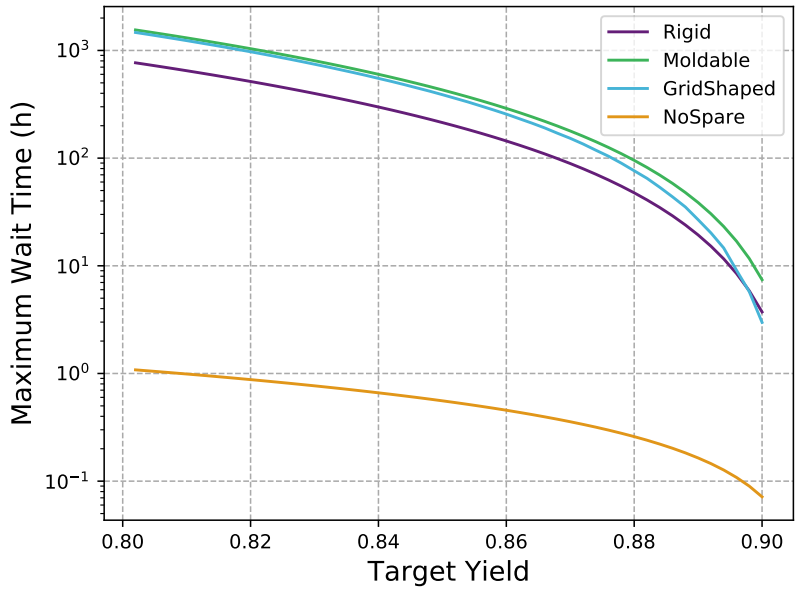
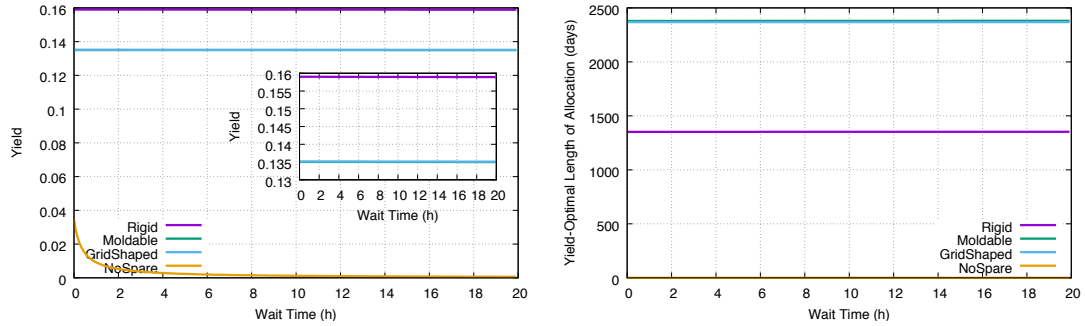


Figure 3.6: Maximum wait time allowed to reach a target yield.

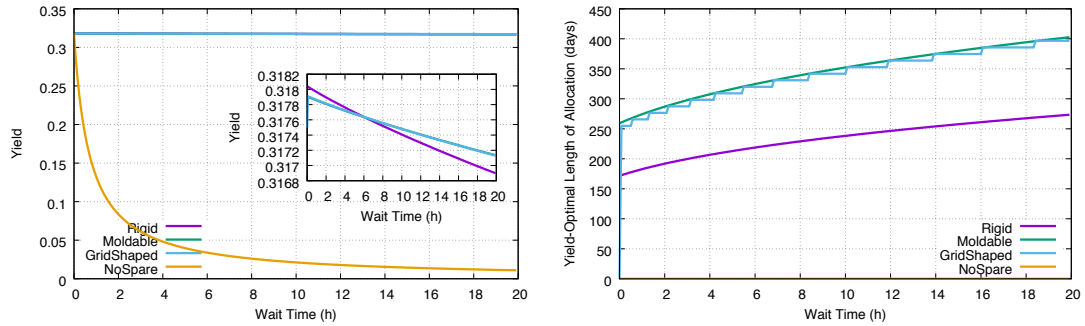
for every type of application stays below or equals 2% of the total application size in all the four cases.

Figure 3.9 aims at showing the impact of the checkpointing cost on the allocation length. The trend is that it does not depend on the checkpointing cost. This can be explained by the fact that the allocation length does not take into account the checkpoint/restart strategy into its computation, only the MTBF and the number of failures allowed. Overall, the impact of the checkpointing cost stays minimal compared to the impact of the wait time or the MTBF.

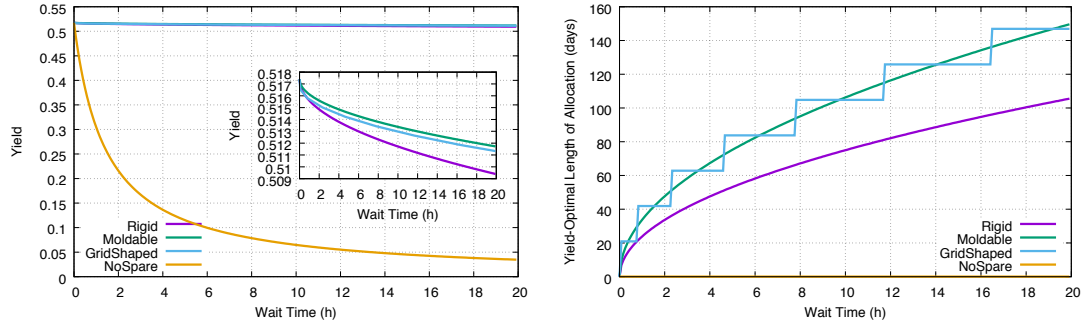
Finally, Figure 3.10 describes the yield obtained when using different models for the checkpointing cost: either the checkpoint is constant (independent of the number of processors: left figure) or it is inversely proportional to the number of processors (right figure). As these plots show, the difference between the two models does not have a noticeable impact on the yield of the applications. This can be explained as follows: as Figure 3.8 showed, only a small number of faults is allowed before resubmission, in comparison to the application size. Changing the number of active processors by a few percentage does not really make a difference for the checkpoint cost, which remains almost the same in both models.



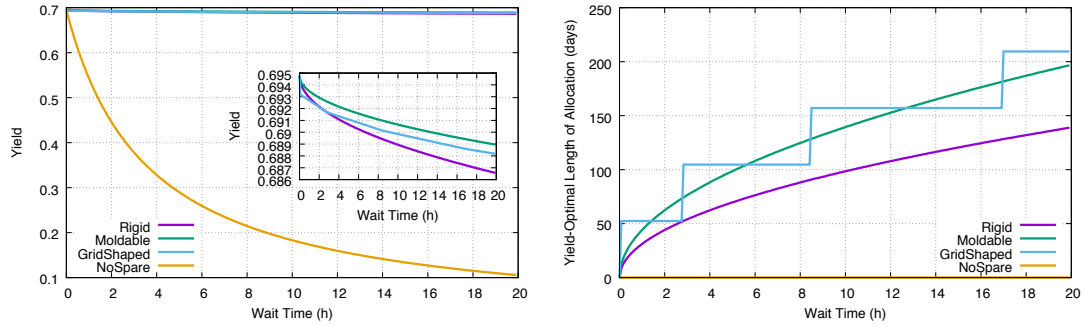
(a) $\mu_{ind} = 5$ years



(b) $\mu_{ind} = 10$ years

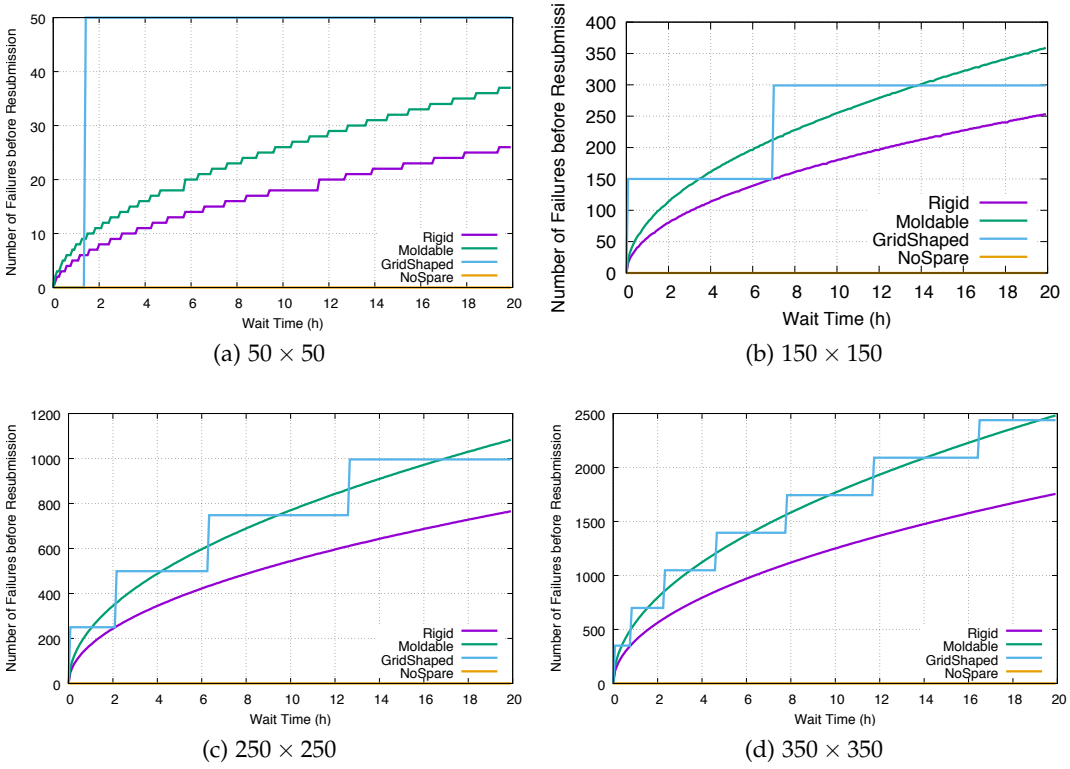


(c) $\mu_{ind} = 20$ years



(d) $\mu_{ind} = 50$ years

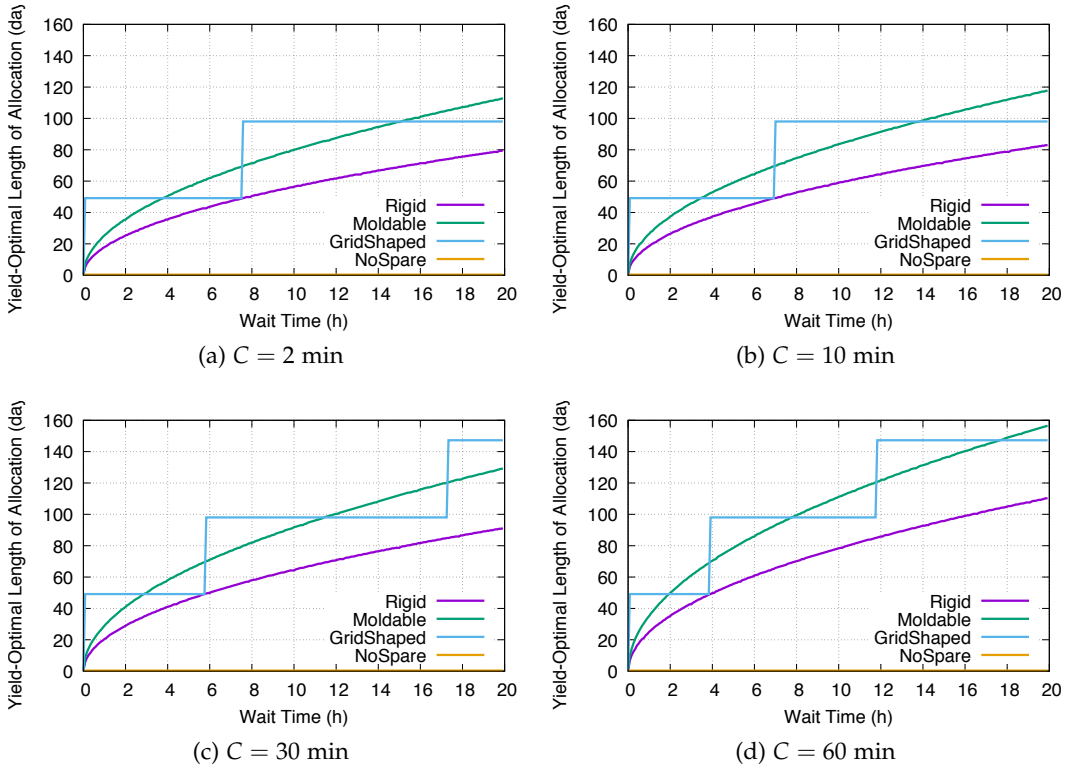
Figure 3.7: Yield and optimal allocation length of as a function of the wait time with $N = 350 \times 350$, and $C = 10$ minutes.



(a) 50×50 (b) 150×150

(c) 250×250 (d) 350×350

Figure 3.8: Optimal number of faults before rescheduling the application for different application sizes.



(a) $C = 2$ min (b) $C = 10$ min

(c) $C = 30$ min (d) $C = 60$ min

Figure 3.9: Optimal number of faults before rescheduling the application for different checkpointing costs.

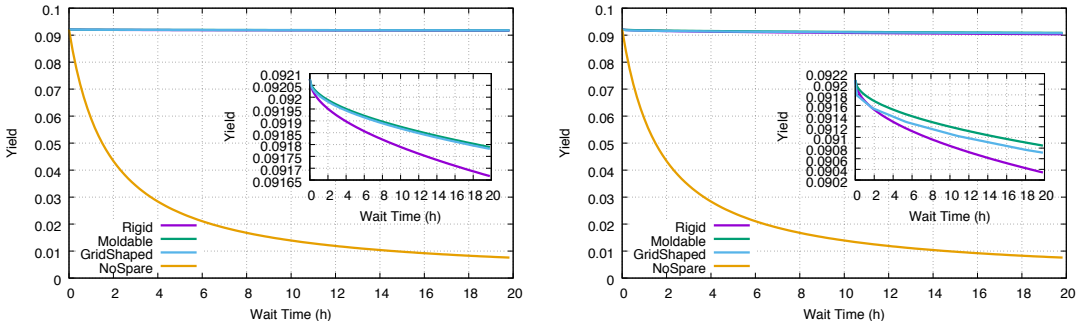


Figure 3.10: Constant checkpoint cost ($C_i = 60$ min) on the left, and increasing checkpoint cost ($C_i = \frac{N}{i} \times 60$ min) on the right, with $\mu_{ind} = 5$ years and $N = 350 \times 350$.

3.4.3 Comparison between C/R and ABFT

In this subsection, we present the results of ABFT and C/R strategies, for a GridShaped application. In order to compare both strategies, we introduce ABFT parameters, and use data from the Titan platform [4]:

- We use tiles of size 180×180, i.e. we let $b = 180$. We set $r = 325$; so that a node holds $325^2 = 105625$ tiles.
- These values give a total of almost 25.5 GB used by each node, which corresponds to 80% of the memory of a node in Titan.
- Overall, the total memory of the application is $8p^2r^2b^2$ bytes, so we set the checkpointing cost to be $\frac{8p^2r^2b^2}{1.4\times1024^4}$, using 1.4 TB/s for the I/O bandwidth of the Titan platform. With r and b set as mentioned, we get $C_i = C \approx \frac{25.5N}{1.4\times1024} \approx \frac{N}{56.3}$.
- Titan has 18,688 cores for a peak performance of 17.59 PFlop/s. We derive a performance per core of 987 GFlop/s, i.e. $\tau_a = \frac{1}{987\times1024^3}$.
- Using the same reasoning, we derive that $\tau_c = \frac{1}{87.2\times1024^3}$.

Figure 3.11 presents the yield obtained by both strategies with either no spare processors (NoSpare and NoSpare-ABFT) or with the optimal number of spare processors (GridShaped for the C/R strategy and ABFT for the ABFT strategy). In Figure 3.11 , we use $N = 150 \times 150$ and $\mu_{ind} = 20$ years. Unsurprisingly, the ABFT strategy grants a better yield than the C/R strategy with a yield very close to 1, compared to ≈ 0.8 for C/R. This is largely due to the fact that the overhead added by the ABFT is $\frac{2}{p}$ and so is negligible compared to the checkpoint overhead. Moreover, the reconstruction of the tiles is done in parallel so it does not induce any significant overhead. This can also be seen when we do not use any spare: C/R and ABFT follow the same trend but ABFT is always more efficient than C/R, which exactly shows that the checkpoint overhead is larger than the ABFT overhead, since it is the only source (along with the wait time) of wasted time if $F = 0$. For a wait time of 10 hours the C/R strategy gives a yield of 0.820 while ABFT grants a better yield of 0.973 (0.364 and 0.426 respectively with no spare processors). We can see on the right figure that the allocation lengths are similar for both strategies. However, for some values, ABFT will have a shorter allocation length, mostly due to the fact that its overhead is small and does not depend on the number of alive processors; hence loosing a few nodes implies a greater slowdown than for the C/R strategy where the checkpointing period is adapted regularly.

The conclusion of this comparative study is that, for a GridShaped application, ABFT uses a very small percentage of spare resources and grants a better yield than classical C/R.

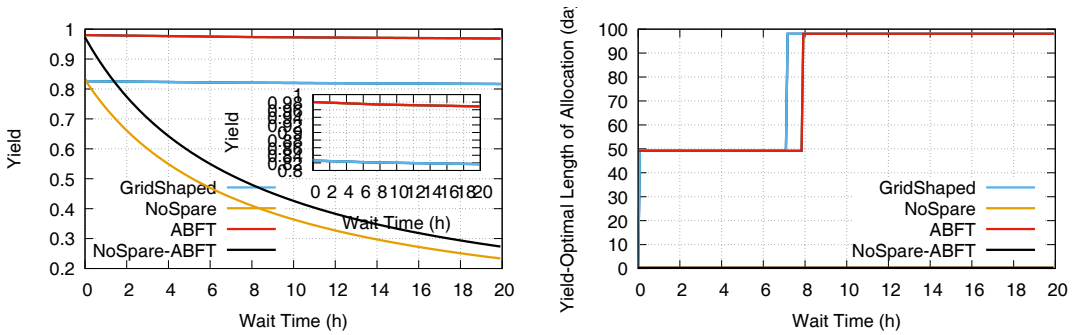


Figure 3.11: Comparison of ABFT and C/R strategies for a GridShaped application, $N = 150 \times 150$ and $\mu_{ind} = 20$ years.

3.5 Related work

We already surveyed related work on checkpoint-restart in Section 2.4. We now discuss previous contributions on Moldable applications in Section 3.5.1. Finally, we provide a few references for ABFT techniques in Section 3.5.2

3.5.1 Moldable and GridShaped applications

Rigid and Moldable applications have been studied for long in the context of scientific applications. A detailed survey on various application types (Rigid, Moldable, malleable) was conducted in [71]. Resizing application to improve performance has been investigated by many authors, including [141, 52, 176, 175] among others. A related recent study is the design of a MPI prototype for enabling tolerance in Moldable MapReduce applications [94].

The TORQUE/Maui scheduler has been extended to support evolving, malleable, and Moldable parallel jobs [153]. In addition, the scheduler may have system-wide spare nodes to replace failed nodes. In contrast, our scheme does not assume a change of behavior from the batch schedulers and resource allocators, but utilizes job-wide spare nodes: a node set including potential spare nodes is allocated and dedicated to a job at the time of scheduling, that can be used by the application to restart within the same job after a failure. At the application level, spare nodes have become common in HPC centers since more than a decade [187]. Recent work aims at sharing spare-nodes across the whole platform to achieve a better global resource utilization [154].

An experimental validation of the feasibility of shrinking application on the fly is provided in [6]. In this paper, the authors used an iterative solver application to compared two recovery strategies, shrinking and spare node substitution. They use ULFM, the fault-tolerant extension of MPI that offers the possibility of dynamically resizing the execution after a failure. Finally, in [79, 107], the authors studied Moldable and GridShaped applications that continue executing after some failures. They focus on the performance degradation incurred after shrinking or spare node substitution,

due to less efficient communications (and in particular collective communications). A major difference with our work is that these studies focus on recovery overhead and do not address overall performance nor yield.

3.5.2 ABFT

ABFT stands for *Algorithm-Based Fault Tolerant* techniques. It is a widely used approach for linear algebra kernels. Since the pioneering paper of Huang and Abraham [109], ABFT protection has been successfully applied to dense LU [69], LU with partial pivoting [197], Cholesky [98] and QR [70] factorizations, and more recently to sparse kernels like SpMxV (matrix-vector product) and triangular solve [162].

In a nutshell, ABFT consists of adding a few checksum vectors as extra columns of each tile, which will be used to reconstruct data lost after a failure. The checksums are maintained by applying the kernel operations to the extra columns, just as if they were matrix elements. The beauty of ABFT is that these checksums can be used to recover from a failure, without any rollback nor re-execution, by reconstructing lost data and proceeding onward. In addition, the failure-free overhead induced by ABFT is usually small, which makes it a good candidate for the design of fault-tolerant linear algebra kernels. We refer to [31, 69] for recent surveys on the approach.

Altogether, we are not aware of any previous study aiming at determining the optimal number of spares as a function of the downtime and resilience parameters, for a general divisible-load application of either type (Rigid, Moldable or GridShaped).

3.6 Conclusion

In this chapter, we have compared the performance of Rigid, Moldable and GridShaped applications when executed on large-scale failure-prone platforms. We have mainly focused on the C/R approach, because it is the most widely used approach for resilience. For each application type, we have computed the optimal number of faults that should be tolerated before requesting a new allocation, as a function of the wait time. Through realistic applicative scenarios inspired by state-of-the-art platforms, we have shown that the three application types experience an optimal yield when requesting a new allocation after experiencing a number of failures that represents a small percentage of the initial number of resources (hence a small percentage of spares for Rigid applications), and this even for large values of the wait time. On the contrary, the NoSpare strategy, where a new allocation is requested after each failure, sees its yield dramatically decrease when the wait time increases. We also observed that Moldable applications enjoy much longer execution periods in between two re-allocations, thereby decreasing the total execution time as compared to Rigid applications (and GridShaped applications lying in between).

GridShaped applications may also be protected using ABFT, and we have compared the efficiency of C/R and ABFT for a typical dense matrix factorization problem. As expected, using ABFT leads to even better yields than C/R for a wide variety

of scenarios, in particular for larger problem sizes for which ABFT scales remarkably well.

Future work will be devoted to exploring more applicative scenarios, and running actual experiments using ULFM [28]. We also intend to extend the model in several directions. On the application side, we aim at dealing with non-perfectly parallel applications but instead with applications whose speedup profile obeys Amdahl's law [3]. On the platform side, we aim at adapting the model to heterogeneous platforms and at doing more experiments with different values for the recovery and checkpoint costs as bandwidths are different when reading or writing data. We will also introduce a more refined speedup profile for GridShaped applications, with an execution speed that depends on the grid shape (a square being usually faster than an elongated rectangle). On the resilience side, we will explore the case with different costs for checkpoint and recovery. More importantly, we will address the combination of ABFT and C/R (instead of dealing with either method individually). Such a combination would allow to tolerate for several failures striking within the same computational step: the idea would be to use ABFT to recover from a single failure and to rollback to the last checkpoint only in the case of multiple failures. Such a combination would enable us to go beyond first-order approximations and single-failure scenarios.

Chapter 4

A generic approach to scheduling and checkpointing workflows

This work deals with scheduling and checkpointing strategies to execute scientific workflows on failure-prone large-scale platforms. To the best of our knowledge, this work is the first to target fail-stop errors for arbitrary workflows. Most previous work addresses soft errors, which corrupt the task being executed by a processor but do not cause the entire memory of that processor to be lost, contrarily to fail-stop errors. We revisit classical mapping heuristics such as HEFT and MinMin and complement them with several checkpointing strategies. The objective is to derive an efficient trade-off between checkpointing every task (CkptAll), which is an overkill when failures are rare events, and checkpointing no task (CkptNone), which induces dramatic re-execution overhead even when only a few failures strike during execution. Contrarily to previous work, our approach applies to arbitrary workflows, not just special classes of dependence graphs such as M-SPGs (Minimal Series-Parallel Graphs). Extensive experiments report significant gain over both CkptAll and CkptNone, for a wide variety of workflows. The work in this chapter is joint work with Li Han, Louis-Claude Canon, Yves Robert and Frédéric Vivien, and has been published in the *International Journal of High Performance Computing Applications* (IJHPCA) [J4].

4.1 Introduction

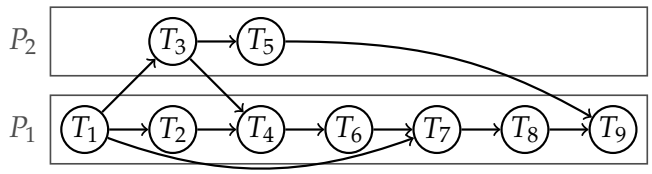


Figure 4.1: Schedule of a workflow with 9 tasks on 2 processors (each edge corresponds to a file dependence between tasks).

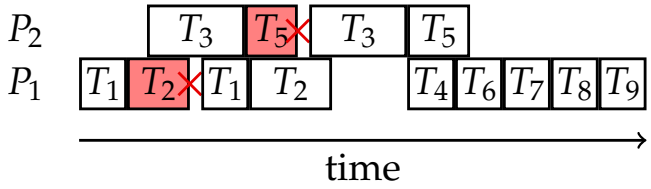


Figure 4.2: Sample execution of the workflow in Figure 4.1 without any checkpoint, with two failures striking during the execution of T_2 on P_1 and during that of T_5 on P_2.

This work deals with scheduling techniques to deploy scientific workflows on large parallel or distributed platforms. Scientific workflows are the archetype of HPC (High Performance Computing) applications, which are naturally partitioned into tasks that represent computational kernels. The tasks are partially ordered because the output of some tasks may be needed as input to some other tasks. Altogether, the application is structured as a DAG (Directed Acyclic Graph) whose nodes are the tasks and whose edges enforce the dependences. Nodes are weighted by the computational requirements (in flops) while edges are weighted by the size of communicated data (in bytes). Given a workflow and a platform, the problem of mapping the tasks onto the processors and to schedule them so as to minimize the total execution time, or makespan, has received considerable attention.

This classical mapping and scheduling problem has recently been revisited to account for the fact that errors and failures can strike during execution. Indeed, platform sizes have become so large that errors and failures are likely to strike at a high rate during application execution ([42]). More precisely, the MTBF (Mean Time Between Failures) μ_P of the platform decreases linearly with the number of processors P, since $\mu_P = \frac{\mu_{\text{ind}}}{P}$, where μ_{ind} is the MTBF of each individual component (see Proposition 1.2 in ([106])). Take $\mu_{\text{ind}} = 10$ years as an example. If $P = 10^5$ then $\mu_P \approx 50$ minutes and if $P = 10^6$ then $\mu_P \approx 5$ minutes: from the point of view of fault-tolerance, scale is the enemy.

Several approaches (see Section 4.6 for a review) have been proposed to mitigate the simplest instance of the problem, that of soft and silent errors. Soft errors cause a task execution to fail but without completely losing the data present in the processor memory. Local checkpointing (or more precisely making a copy of all task input/output data), and task replication, are the most widely used technique to address soft errors. Silent errors represent a different challenge than soft errors, in that they do not interrupt the execution of the task but corrupt its output data. However, their net effect is the same, since a task must be re-executed whenever a silent error is detected. A silent error detector is applied at the end of a task's execution, and the task must be re-executed from scratch in case of an error. Again, local checkpointing (making copies of input/output data) or replicating tasks and comparing outputs, are two common techniques to mitigate the impact of silent errors.

Fail-stop errors, or failures, are much more difficult to deal with. In the case of a fail-stop error (e.g., a crash due to a power loss or some other hardware problem) the

execution of the processor stops, all the content of its memory is lost, and the computations have to be restarted from scratch, either on the same processor once it reboots or on a spare. The de-facto approach to handle such failures is Checkpoint/Restart (C/R), by which application state is saved to stable storage, such as a shared file system, throughout execution. The common strategy used in practice is *checkpoint everything*, or CkptAll: all output data of each task is saved onto stable storage (in which case we say "the task is checkpointed"). For instance, in production Workflow Management Systems (WMSs) ([2, 78, 191, 1, 192, 61]), the default behavior is that all output data is saved to files and all input data is read from files, which is exactly the CkptAll strategy. While this strategy leads to fast restarts in case of failures, its downside is that it maximizes checkpointing overhead. At the other end of the spectrum would be a *checkpoint nothing* strategy, or CkptNone, by which all output data is kept in memory (up to memory capacity constraints) and no task is checkpointed. This corresponds to "in-situ" workflow executions, which has been proposed to reduce I/O overhead ([202]). The downside is that, in case of a failure, a large number of tasks may have to be re-executed, leading to slow restarts. The objective of this work is to achieve a desirable trade-off between these two extremes. To the best of our knowledge, no general solution is available. We build upon previous work ([99]) that was restricted to M-SPGs (Minimal Series-Parallel Graphs) ([186]). In ([99]), the authors took advantage of the recursive structure of M-SPGs and used proportional mapping ([152]) for scheduling and checkpointing M-SPG workflows as sets of superchains. For general graphs, we have to resort to classical scheduling heuristics such as HEFT ([182]) and MinMin ([33]), two reference scheduling algorithms widely used by the community. We provide extensions of HEFT and MinMin that allow for a smaller subset of tasks to be checkpointed and lead to better makespans than the versions where each task (CkptAll) or no task (CkptNone) is checkpointed.

The main contributions of this chapter are the following:

- We deal with arbitrary dependence graphs, and require no graph transformation before applying our scheduling and checkpointing algorithms.
- We compare several mapping strategies and combine them with several checkpointing strategies.
- We design an event-based simulator to evaluate the makespan of the proposed solution. Indeed, computing the expected makespan of a solution is a difficult problem ([99]), and simple Monte-Carlo based simulations cannot be applied to general DAGs unless all tasks are checkpointed: otherwise, sampling the weight distribution for each task independently is not enough to compute the makespan, since a failure may involve re-executing several tasks (as shown in Section 4.2).
- We report extensive experimental evaluation with both real-world and randomly generated workflows to quantify the performance gain achieved by the proposed approach.

The rest of the chapter is organized as follows. First in Section 4.2, we work out an example to help understand the difficulty of the problem. Then we introduce the performance model in Section 6.2. We detail our scheduling and checkpointing

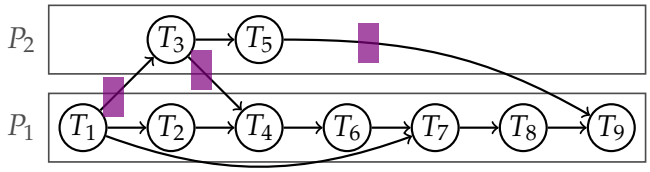


Figure 4.3: A purple *crossover* checkpoint is performed for each file produced by one processor and used by another one.

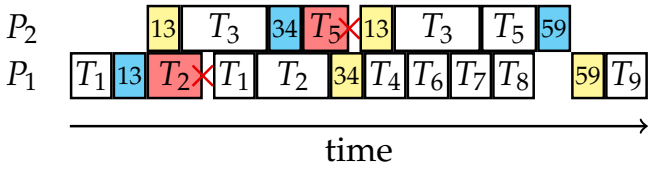


Figure 4.4: Sample execution of the application in Figure 4.3 with two failures striking during the execution of T_2 on P_1 and that of T_5 on P_2, with crossover checkpoints. Label ij indicates the file from T_i to T_j. Now T_4 can start before the re-execution of T_3 since its output was checkpointed.

algorithms in Section 4.4. We give experimental results in Section 4.5. Section 4.6 surveys the related work. Finally, we provide concluding remarks and directions for future work in Section 6.6.

4.2 Example

In this section, we illustrate the difficulty of deciding where to place checkpoints in a workflow. Consider the example of Figure 4.1 with 9 tasks, T_i, $1 \leq i \leq 9$, that have been mapped on 2 processors as shown on the figure. Note that this DAG cannot be reduced to an M-SPG and the previous approach ([99]) cannot be applied for this graph. While most tasks are assigned to processor P_1, some tasks are assigned to the second processor, P_2, to exploit the parallelism of the DAG. Any dependence between two tasks represents a file that is required to start the execution of the successor task; hence, $T_1 \rightarrow T_2$ represents a file produced by task T_1 that is required for the execution of task T_2 to start. Because T_1 and T_2 are both executed on processor P_1, this file is kept in the memory of P_1 after T_1 completes. However, for the dependence $T_1 \rightarrow T_3$, because the tasks T_1 and T_3 are executed on different processors, the corresponding file must be retrieved by P_2. Such a dependence between two tasks assigned to two different processors is called a *crossover dependence*.

In a first scenario, let us suppose that no task is checkpointed as showed in Figure 4.1: then if no failure strikes, the makespan will be the shortest possible, consisting only of the execution time of each task and of retrieving the necessary input files. However, as soon as a failure happens, we may need to restart the whole application from

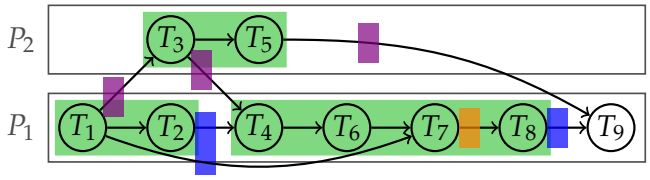


Figure 4.5: Blue *induced* checkpoints are used to isolate task sequences on a processor (labeled in green, such as the sequence T_4, T_6, T_7 and T_8 on P_1). Finally, additional checkpoints can be added inside an idle-free task sequence through a dynamic programming algorithm: the orange checkpoint corresponds to such an addition.

the very beginning. To study such a scenario, we need to explicit the memory management. Let us assume that once a processor has sent a file to another processor, then this file is deleted from the memory of the producing processor. For instance, as soon as P_2 has received from P_1 the file corresponding to the dependence $T_1 \rightarrow T_3$, this file is erased from the memory of P_1. Remember that a failure wipes out the whole content of the memory of the struck processor. Thus, if a failure strikes during the execution of T_5, to be able to re-attempt to execute T_5, T_3 will need to be re-executed before (because the file $T_3 \rightarrow T_5$ is no longer available), which requires T_1 to be re-executed first (because the file $T_1 \rightarrow T_3$ is no longer available). Hence, a single failure in a part of the graph may require the re-execution of most of the workflow. Figure 4.2 shows an example of execution of the DAG when no task is checkpointed. To execute T_4, we need both T_2 and T_3 to finish successfully, and that no fault strikes neither P_1 nor P_2 between the completion of these tasks and the start of T_4. Here, T_2 does not finish so T_1 is re-executed. When P_2 fails, we need to re-execute T_3, which requires input from T_1. Luckily (!), P_1 already suffered from a failure, so T_1 has already been re-executed. Otherwise, we would have had to restart the execution of the whole workflow because of the failure of P_2.

To avoid rolling back to the beginning in case of failures, we can try to place some checkpoints inside the workflow. As commonly assumed in workflow management systems ([2, 78, 191, 1, 192, 61]), we do not rely on direct point-to-point communications between processors but instead assume that task input and output files are exchanged through the file system. Thus, any file produced by one processor and required by another processor is necessarily saved to, and then read from, stable storage. In the second scenario shown in Figure 4.3, we decide to checkpoint every crossover dependence (from T_1 to T_3, T_3 to T_4, and T_5 to T_9). An execution of that schedule is shown in Figure 4.4. Cyan boxes represent checkpoints while yellow boxes represent data being read. The transfer of file $T_1 \rightarrow T_3$ is done through a checkpointing phase on P_1, followed by a reading phase on P_2. We can see that thanks to the crossover checkpoints, T_4 does not need to wait for the completion of the second execution of T_3 anymore, as T_3 output data has already been checkpointed. Moreover, if only a failure on P_2 happened, instead of rolling back to task T_1 to re-execute T_3 as it was the case before, T_3 could have restarted directly (although the entire content of the

processor memory is lost, so all inputs of T_3 must be recovered from stable storage after a downtime before the execution of T_3 can restart). The motivation to checkpoint all files involved in crossover dependences is to isolate the processors. Indeed, if all crossover files are checkpointed, a failure on a processor will never lead to the re-execution of a task successfully executed on another processor. Overall, we will lose less time recomputing tasks or waiting for their second completion. However, reading from stable storage and checkpointing also take time. Finding the right trade-off is the main focus of this chapter: deciding which tasks should be checkpointed, so that the overhead added by the checkpointing and reading of files is not more expensive than the re-execution of tasks.

We conclude by informally introducing examples of checkpointing strategies that achieve desirable trade-offs (see Section 4.4.2 for details). Two additional checkpoints, in blue, called *induced* checkpoints, have been added in Figure 4.5. Their role is to secure the fast re-execution of tasks that are the target of a crossover dependence, namely T_4 and T_9. The blue checkpoint after T_2 isolates the execution of the task sequence $S_1 = \{T_4, T_6, T_7, T_8\}$ on P_1. To this purpose, it is necessary to checkpoint all intermediate results that may be used after the execution of T_2: these are the files generated by previous tasks, namely $T_1 \rightarrow T_7$ and $T_2 \rightarrow T_4$, This way, when a failure strikes, previous tasks do not have to be restarted and the computation may be restarted directly from T_4. This way, tasks in the sequence S_1 may be sequentially executed without idle time. It would not have been possible to include T_1 and T_2 in S_1 because T_4 could have waited for the completion of T_3 leading to idle time in some scenarios. Similarly, the second blue checkpoint isolates the execution of T_9.

Finally, once the four tasks T_4, T_6, T_7, and T_8 of the sequence S_1 have been "isolated" from other tasks, it is possible to use a dynamic programming algorithm similar to that used in ([99]) in order to introduce additional checkpoints. In the example of Figure 4.5, a single additional checkpoint, in orange, is inserted after T_7.

4.3 Model

This section details the execution and fault-tolerance models used to compare scheduling and checkpointing algorithms.

4.3.1 Execution Model

The execution model for a task workflow on a homogeneous system is represented as a Directed Acyclic Graph (DAG), $G = (V, E)$, where V is the set of nodes corresponding to the tasks, and E is the set of edges corresponding to the dependences between tasks. In a DAG, a node without any predecessor is called an *entry* node, while a node without any successor is an *exit* node. For a task T in G, $pred(T)$ and $succ(T)$ represent the set of its immediate predecessors and successors respectively. We say that a task T is *ready* if either it does not have any predecessor in the dependence graph, or if all its predecessors have been executed. In this model, the execution time of a task $T_i \in V$

is w_i, i.e., its execution time in a failure-free execution. Each dependence $(T_i, T_j) \in E$ is associated with the cost $c_{i,j}$ to store/read the data onto/from stable storage. Before the execution of T_j on processor P_k, all input files needed by T_j must be present in the local memory of P_k and absent files must be *read* from the stable storage, which happens as late as possible. We ignore direct communication between processors because each data transfer between two processors (i.e., a *crossover dependence*) consists in writing to and reading from the stable storage. Alternatively, we also say that the file is checkpointed and then recovered.

4.3.2 Fault-Tolerance Model

In this work, each processor is a processing element that is subject to its own individual failures. Failures can strike a processor at any time, during either task execution or waiting time. Failure inter-arrival times are assumed to be Exponentially distributed. These failure-prone processors stop their execution once a failure strikes, i.e., we have fail-stop errors. When a fail-stop error strikes a processor, the whole content of its memory is lost and the computation it was performing must be restarted, either on the same processor after a reboot, or on a spare processor (e.g., taken from a pool of spare processors either specifically requested by the job submitter, or maintained by the resource management infrastructure).

Consider a single task T, with weight w, scheduled on such a processor, and whose input is stored on stable storage. It takes a time r to read that input data from stable storage, either for its first execution or after a failure. The total execution time W of T is a random variable, because several execution attempts may be needed before the task succeeds.

We assume that failures are i.i.d. (independent and identically distributed) across the processors and that the failure inter-arrival times at each processor is Exponentially distributed with Mean Time Between Failures (MTBF) $\mu = 1/\lambda$. Let $\lambda \ll 1$ be the Exponential failure rate of the processor. With probability $e^{-\lambda(r+w)}$, no failure occurs, and W is equal to $r + w$. With probability $(1 - e^{-\lambda(r+w)})$, a failure occurs. For Exponentially distributed failures, the expected time to failure, knowing that a failure occurs during the task execution (i.e., in the next $r + w$ seconds), is $1/\lambda - (r + w)/(e^{\lambda(r+w)} - 1)$ ([106]). After this failure, there is a downtime d, which is (an upper bound of) the time needed to reboot the processor or migrate to a spare. Then we start the execution again, first with the recovery r and then the work w. With a general model where an unbounded number of failures can occur during recovery and work, the expected time W to execute task T is given by $W = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(r+w)} - 1\right)$ ([106]). Now if the output data of task T is checkpointed, with a time c to write all of its output files onto stable storage, the total time becomes:

$$W = \left(\frac{1}{\lambda} + d\right)\left(e^{\lambda(r+w+c)} - 1\right) . \tag{4.1}$$

Equation (4.1) assumes that failures can also occur during checkpoints, which is the

most general model for failures. We also assume that failures may strike during the idle time (i.e., waiting time) of the processor (e.g., the power supply may fail). In the case of a sequence of non-checkpointed tasks to be executed on a processor P, the output data of each task resides in the memory of P for use by subsequent tasks. When a failure strikes P, the entire memory content is lost and the whole task sequence must be re-executed from scratch.

4.3.3 Problem Formulation

Given a DAG and a set of processors on which fail-stop failures strike with Exponentially distributed inter-arrival times, the objective is to schedule the task executions and potential checkpoints such that the expected completion time (or makespan) is minimized. Due to delays resulting from the faults, the schedule of the tasks consists of an assignment to processors and of a task ordering. Each processor executes tasks as soon as possible and resumes their processing when a failure strikes. Finally, the schedule of the checkpoints is the (possibly empty) list of files that must be checkpointed after each task execution.

4.4 Scheduling and checkpointing algorithms

In this section, we first present heuristics to map tasks to processors. Then we propose three different checkpointing strategies that can be used simultaneously.

4.4.1 Scheduling heuristics

We map tasks to processors and schedule them using two classical scheduling heuristics, HEFT ([182])† and MinMin ([33]). We run these heuristics as if the platforms were not subject to failures, that is, without considering checkpoints. Therefore, we decide first on which processor a task will be executed, and the order in which a processor will execute tasks, before deciding when and what to checkpoint (see Section 4.4.2). However, we present variants of HEFT and MinMin, named HEFTC and MinMinC, that are specifically designed for our failure-prone framework.

Heterogeneous Earliest Finish Time first (HEFT) is presented as the HEFTC variant in Algorithm 1. The original HEFT algorithm comprises two phases. In a first *task prioritizing phase*, the bottom-level of all tasks is computed and tasks are ordered by non-increasing bottom-levels. The bottom-level of a task is the maximum length of any path starting at the task and ending in an exit task, considering that all communications take place ([59]). In the second *processor selection phase*, the first unscheduled task is scheduled as early as possible on a processor that minimizes its completion time. In all cases, ties are broken arbitrarily. To these original two phases, we add a third one, the *chain mapping phase* (lines 7 and 8 of Algorithm 1). If the newly mapped task T is the head of a chain in the task graph, then this whole chain is mapped on the same processor as T, and the tasks will be executed consecutively. Ensuring that entire

chain of tasks are scheduled on the same processor decreases the number of crossover dependences and thus, the time to checkpoint them. HEFTC has a complexity of $O(n^2)$ for a workflow with n tasks. During the processor selection phase, the earliest finish time of a task is computed in HEFTC while assuming that the newly mapped task must start *after* all tasks previously scheduled on that processor have completed. On the contrary, the original HEFT heuristic is allowed to perform backfilling following a classical insertion-based policy, as long as the completion time of no task is delayed. Allowing backfilling is more expensive at scheduling time but should lower the execution time (the complexity of HEFT with backfilling is also $O(n^2)$ with homogeneous processors). We do not allow backfilling for HEFTC because it could be antagonistic to the chain mapping phase if it led to backfill the head of the chain, but not the whole chain.

Algorithm 1: HEFTC

```
1 Compute the bottom-level of all tasks by traversing the graph from the exit
    tasks
2 Sort the tasks by non-increasing values of their bottom-levels
3 while there are unscheduled tasks do
4 |   Select the first task T_i
5 |   k ← arg min_{1≤k≤p} EarliestFinishTime(T_i, P_k)
6 |   Schedule task T_i on processor P_k
7 |   if T_i is the head of a chain of tasks then
8 |   |   Schedule the whole chain continuously on P_k
```

The MinMin scheduling algorithm is presented in the MinMinC variant in Algorithm 2. The original MinMin algorithm is a simple loop which, at each step, schedules the task that can finish the earliest among unscheduled tasks. Therefore, at each step it considers all ready tasks and, for each of them, all the processors. We (try to) improve this heuristic by adding a *chain mapping phase* exactly as previously (lines 5 and 6 of Algorithm 2). MinMinC has a complexity of $O(n^2p)$ for a workflow with n tasks and p processors.

4.4.2 Checkpointing strategies

While the previous scheduling algorithms provide mappings of tasks to processors, it remains to decide which files must be checkpointed and when. This section introduces finer strategies than the two extremes solutions that consist of checkpointing no task or all tasks. These two extreme solutions, CkptNone and CkptAll, are denoted with the suffixes None and All, respectively.

In principle, our model forbids direct communications between processors (see Section 4.3.1). However, for the sake of comparison, we make an exception for CkptNone: in the absence of any checkpoint with CkptNone, direct communications must

Algorithm 2: MinMinC

```
1 ReadyTasks ← entry tasks
2 while there are unscheduled tasks do
3     Pick a task T ∈ ReadyTasks and a processor P such that the completion time
        of T on P is minimum among the Earliest Finish Times of all ready tasks
4     Schedule task T on processor P
5     if T is the head of a chain of tasks then
6         Schedule the whole chain continuously on P
7     Update ReadyTasks
```

be performed for each crossover dependence. We assume that, in this special case, transferring a file takes half the time needed to save it to and read it from stable storage. This special case is thus more efficient when files are large.

The minimum strategy that is required to avoid direct communications consists of checkpointing all files that must be transferred between any pair of processors, i.e., exactly the files corresponding to crossover dependences. Moreover, in this case, any failure on a processor will not require any re-execution on other processors. The strategy is denoted with a "C" in the checkpoint suffix.

For the next two additional strategies, we introduce a new type of checkpoints: *task checkpoints*. While a simple file checkpoint consists of writing to stable storage a file that corresponds to a dependence between two tasks, a task checkpoint consists of writing all files that (i) reside in memory on a processor; (ii) will be used later by tasks assigned to the same processor; and (iii) have not already been checkpointed. In the example in Section 4.2, for each crossover dependence we did a simple file checkpoint rather than a full task checkpoint. A task checkpoint after task T_3 would have also checkpointed the file corresponding to the dependence $T_3 \rightarrow T_5$. A non-trivial task checkpoint for the example of Section 4.2 would be a task checkpoint for task T_2. This checkpoint would require checkpointing the files corresponding to the dependences $T_2 \rightarrow T_4$ and $T_1 \rightarrow T_7$.

When a task checkpoint is performed after the execution of a task, multiple files may be checkpointed "at the same time" (either newly created files or previously created ones that will later be used). If several files are checkpointed, they are all checkpointed after the task completion, one after the other (in any order), and they can all be read again only when the last of them has been checkpointed. When absent from memory (following a failure or due to a crossover dependence), input files are read from stable storage as late as possible, just before the execution of the task that needs them. One could imagine optimizations where files (in a task checkpoint) would be checkpointed independently and as soon as possible, or in a carefully designed order. Such optimizations could lead to lower expected makespans in some cases. However, the interplay of file checkpoints and reads that could result from these optimizations may lead to slowdowns. This is the reason why we prefer our simpler scheme.

Checkpointing crossover dependences enable to isolate processors, in that there is no re-execution propagation from a processor to another. However, when a task is the target of a crossover dependence, its starting time is the maximum of the availability times of all its input files, and these files come from different processors. Therefore, its starting time may be delayed by failures occurring on other processors. Because failures can strike during idle time, it may be beneficial to try to use the potential waiting time by performing a task checkpoint of the task preceding the target task. This way, the whole content of the memory will be preserved, the cost of the checkpoint may be offset by some waiting time, and if a failure strikes during the remaining waiting time all input files remain available. Therefore, we propose a new checkpointing strategy denoted with "I" in the checkpoint suffix. This strategy consists of checkpointing all *induced* dependences. A dependence $T_i \rightarrow T_j$ is an *induced* dependence if T_i and T_j are scheduled on the same processor P and there exists a crossover dependence $T_k \rightarrow T_l$ such that T_l is scheduled on P after T_i and before T_j (or $T_l = T_j$). Checkpointing these induced dependences is done by performing a task checkpoint of the task preceding T_l on P. In the example of Section 4.2, the dependences $T_2 \rightarrow T_4$ and $T_1 \rightarrow T_7$ are both induced dependences because of the crossover dependence $T_3 \rightarrow T_4$.

So far, we have only introduced checkpoints to isolate processors, either to avoid failure propagation or to try to minimize the impact of processors having to wait from each other. We further consider checkpoints that more directly optimize expected total execution time. We present an additional strategy, denoted by the suffix "DP", which adds additional checkpoints through a $O(n^2)$ dynamic programming algorithm, which is a transposition of that of ([99]). This dynamic program considers a maximal sequence of consecutive tasks that are all assigned to the same processor, and that are isolated from other tasks: the sequence contains no checkpoint and none of its tasks is the target of a crossover dependence, except for its first task. Let $T_1, ..., T_k$ be such a sequence of tasks. By definition, all input data produced by some previous tasks have been checkpointed. Then, the optimal expected time to execute this sequence is given by $Time(k)$ where $Time$ is defined as follows:

$$Time(j) = \min \left(T(1,j), \min_{1 \leq i < j} Time(i) + T(i+1,j) \right)$$

where $T(i,j)$ is the expected time to execute tasks T_i to T_j provided that two task checkpoints are performed: one right before task T_i and one right after task T_j. Using the same reasoning as in Section 4.3.2, we can provide an upper bound on $T(i,j)$ as follows:

$$T(i,j) = \left(\frac{1}{\lambda} + d \right) \left(e^{\lambda (R_i^j + W_i^j + C_i^j)} - 1 \right)$$

where R_i^j (resp. W_i^j and C_i^j) is the sum of the recovery (resp. execution and checkpointing) costs of tasks T_i to T_j. The recovery costs concern all input files of these tasks that are on the stable storage, while the checkpointing costs concern all files that will be checkpointed when a task checkpoint is done after T_j. This is an upper bound, because

when no failure strikes, some input files of tasks T_i to T_j may already be present in memory and will not be read from stable storage. Because we have no simple mean to know whether some failures had previously struck, we have to resort to this upper bound. This is a necessary condition to be able to reuse, in some way, the dynamic programming approach of ([99]). This algorithm requires, by construction, that induced dependences be checkpointed. However, we heuristically use it even when this condition is not satisfied. In this case, we take a maximal sequence while allowing tasks to be the target of crossover dependences, and behave as if these crossover dependences were not existing: we discard any potential waiting time that may be due to these crossover dependences (because we have no means to estimate them).

4.5 Experiments

In this section, we describe the experiments conducted to assess the efficiency of the checkpointing strategies. In Subsection 4.5.1, we describe the parameters and applications used during our experimental campaign, then in Subsection 4.5.2 we present the simulator used to run the applications and simulate the behavior of large-scale platforms. Finally, we present our results in Subsection 4.5.3.

4.5.1 Experimental methodology

We consider workflows from real-world applications, namely representative workflow applications generated by the Pegasus Workflow Generator (PWG) ([26, 117, 165]), as well as the three most classical matrix decomposition algorithms (LU, QR, and Cholesky) ([51]), and randomly generated DAGs from the Standard Task Graph Set (STG) ([180]).

Pegasus workflows. PWG uses the information gathered from actual executions of scientific workflows as well as domain-specific knowledge of these workflows to generate representative and realistic synthetic workflows (the parameters of which, e.g., the total number of tasks, can be chosen). We consider all of the five workflows ([149]) generated by PWG, including three M-SPGs (Genome, Ligo, and Montage) that are used to compare our new general approach with PropCkpt, the strategy for M-SPGs proposed in ([99]).

- Montage: The NASA/IPAC Montage application stitches together multiple input images to create custom mosaics of the sky. The average weight of a Montage task is 10s. Structurally, Montage is a three-level graph ([60]). The first level (reprojection of input image) consists of a bipartite directed graph. The second level (background rectification) is a bottleneck that consists of a join followed by a fork. Then, the third level (co-addition to form the final mosaic) is simply a join.
- Ligo: LIGO's Inspiral Analysis workflow is used to generate and analyze gravitational waveforms from data collected during the coalescing of compact binary systems. The average weight of a Ligo task is 220s. Structurally, Ligo can be

seen as a succession of Fork-Joins meta-tasks, that each contains either fork-join graphs or bipartite graphs.

- GENOME: The epigenomics workflow created by the USC Epigenome Center and the Pegasus team automates various operations in genome sequence processing. The average weight of a GENOME task depends on the total number of tasks and is greater than 1000s. Structurally, GENOME starts with many parallel fork-join graphs, whose exit tasks are then both joined into a new exit task, which is the root of fork graphs.
- CYBERSHAKE: The CYBERSHAKE workflow is used by the Southern Calfornia Earthquake Center to characterize earthquake hazards in a region. The average weight of a CYBERSHAKE task is 25s. Structurally, the CYBERSHAKE workflow starts with several forks. Then each of the forked tasks has two dependences: one to a single task (join) and one to a specific task for each of the tasks. Finally, all these new tasks are joined without another dependence this time.
- SIPHT: The SIPHT workflow, from the bioinformatics project at Harvard, is used to automate the search for untranslated RNAs (sRNAs) for bacterial replicons in the NCBI database. The average weight of a SIPHT task is 190s. Structurally, the SIPHT workflow is composed of two different parts that are joined at the end: the first one is a series of join/fork/join, while the other is made of a giant join.

We generate these workflows with 50, 300, and 700 tasks (these are the number of tasks given to the generator, the actual number of tasks in the generated workflows depend on the workflow shape). The task weights and file sizes are generated by PWG. In some instances, a single file may be used by more than one task and a dependence may represent multiple files to transfer between two tasks. In the first case, whenever a file is common to multiple dependences, the file is only saved once. In the second case, files are aggregated into a single one.

Matrix factorizations. We consider the three most classical factorizations of a $k \times k$ tiled matrix: LU, QR, and Cholesky factorizations.

- The LU decomposition is the factorization of any matrix into a product of one lower-triangular (L) and one upper-triangular (U) matrices. Structurally, the DAG is made of k steps, with at step i, one task having two sets of $k - i - 1$ children, and each pair of tasks between the two sets having another child.
- The QR decomposition is the decomposition of a matrix into a product of an orthogonal matrix (Q) and upper-triangular matrix (R), i.e., $A = QR$ with $QQ^T = Id$. Structurally, the QR decomposition looks like the LU decomposition but it has more complex dependences between the $k - i - 1$ children at step i.
- Cholesky is a factorization of a positive and definite matrix into the product of a triangular matrix and its transpose, i.e., $A = BB^T$ where B is lower-triangular and has non-zero values of the diagonal. The Cholesky decomposition DAG is the representation of a panel algorithm and can be constructed recursively by removing the first row and the first column of submatrices, to keep factorizing the trailing matrix.

For each factorization, we perform experiments with $k = 6$, 10, and 15, for a total

of $3 \times 3 = 9$ DAGs with up to 1240 tasks. The number of vertices in the DAG depends on k as follows: the Cholesky DAG has $\frac{1}{3}k^3 + O(k^2)$ tasks, while the LU and QR DAGs have $\frac{2}{3}k^3 + O(k^2)$ tasks. There are 4 types of tasks in LU, QR, and Cholesky, which are labeled by the corresponding BLAS kernels ([51]), and their weights are based on actual kernel execution times as reported in ([8]) for an execution on Nvidia Tesla M2070 GPUs with tiles of size $b = 960$.

Random graphs. The STG benchmark ([180]) includes 180 instances for each size of DAGs (from 50 to 5 000). This set is often used in the literature to compare the performance of scheduling strategies. Instead of choosing part of the instances for each size, we did experiments on all instances of size 300 and 750. For each instance, one of the four DAG generators specifies the structure of the dependences (e.g., layer-by-layer) and one of the six cost generators provides the distribution of the processing times (e.g., uniform).

Failure distribution. In the experiments, we consider different exponential processor failure rates. To allow for consistent comparisons of results across different DAGs (with different numbers of tasks and different task weights), we simply fix the probability that a task fails, which we denote as p_{fail}, and then simulate the corresponding failure rate. Formally, for a given DAG $G = (V, E)$ and a given p_{fail} value, we compute the average task weight as $\bar{w} = \sum_{i \in V} w_i / |V|$, where w_i is the weight of the i-th task in V. We then pick the failure rate λ such that $p_{\text{fail}} = 1 - e^{-\lambda \bar{w}}$. We conduct experiments for three p_{fail} values: 0.01, 0.001, and 0.0001.

Checkpointing costs. An important factor that influences the performance of checkpointing strategies, and more precisely of the checkpointing and recovery overheads, is the data-intensiveness of the application. We define the Communication-to-Computation Ratio (CCR) as the time needed to store all the files handled by a workflow (input, output, and intermediate files) divided by the time needed to perform all the computations of that workflow on a single processor. For Pegasus workflows, LU, QR, and Cholesky, we vary the CCR by scaling file sizes by a factor. As STG only provides task weights, we compute the average communication cost as $\bar{c} = \bar{w} \times CCR$. Communication costs are generated with a lognormal distribution with parameters $\mu = \log(\bar{c}) - 2$ and $\sigma = 2$ to ensure an expected value of $\bar{c}$. This distribution with parameter $\sigma = 2$ has been advocated to model file sizes ([67]). This allows considering and quantifying the data-intensiveness of all workflows in a coherent manner across experiments and workflow classes and configurations.

Reference strategies. In the experiments, we compare our strategies to the two extreme approaches CkptAll and CkptNone. We use the simulator described in Subsection 4.5.2. For each parameter setting of each workflow, we run 10,000 random simulations and approximate the makespan by the observed average makespan.

4.5.2 Simulator

In order to evaluate the performance of our strategies, we implemented a discrete event simulator. The C++ code for the simulator is available at `http://github.`

`com/vlefevre/task-graph-simulation`. To simulate the execution of applications on large-scale platforms, we operate in three steps:

1. We first read an input file describing the task-graph and the scheduling/mapping strategy;
2. Then we generate a set of fail-stop error times for each processor during a time `horizon` (that is set by the user);
3. Finally, we execute ready tasks by mapping them to a processor and we keep doing this until all tasks are executed.

The first part is basically reading a file that describes the following important elements for the simulation:

- For each task,
  - its ID,
  - its weight (i.e., duration),
  - the ID of the processor it has been mapped to,
  - several booleans indicating whether the task has to be checkpointed or not, one for each checkpointing strategy.
- For each dependence between two tasks,
  - the ID of the parent,
  - the ID of the child,
  - the list of files with their time to be loaded/written that creates the dependence (i.e., there are some of the output files of the parent and some of the input files of the child).
- For each processor, its schedule: a list of tasks that have been mapped to it and that respects the causal order of the task-graph.

The second part is done by using the inversion sampling method: we generate error times according to a random variable that follows an exponential distribution, and this exponential distribution is generated from an (assumed) uniform distribution between 0 and 1 obtained by calling the C function `rand()`, and dividing its result by the C constant `RAND_MAX`. In our case, if U is a random variable following a uniform distribution between 0 and 1, then $-\frac{\log U}{\lambda}$ follows an exponential distribution of parameter λ. We generate errors one each processor, until the time of one error is greater than the `horizon` parameter. In the experiments, it was set to at least 2 times the expected makespan we have with the CkptAll strategy, which we computed using the Monte-Carlo method. In practice, most of the simulations were done before the horizon was reached except for None with large p_{fail}.

For the last step, we keep a global time t on all the processors, and we generate events happening on each processor (either a failure or the successful completion of a task). Each processor holds the time of its last event in a variable t_i. At each moment of the simulation, we have $t_i \geq t, \forall i$. The algorithm repeats these steps until all tasks are marked executed:

- For each processor p_i,
  - we look at the next task to be executed on p_i (following the list scheduling given as input) if the current task is finished at time t;

  - if it is ready, we compute its full execution time by computing the time of reading the necessary input files, the weight of the task (given as input) and potentially some writing (in case of crossover dependences or if the checkpoint strategy requires this task to be checkpointed);
  - we look at the next error happening after time t: if it is before the end of the task then we set t_i to be the time of that failure, otherwise t_i is set to the time when the task ends and the task is marked executed.
- We set $t = \min_i t_i$.

There are two more things to detail: the computation of reading times and how we rollback when there is a failure. For the first problem, we keep a set of all files loaded on each processor. Before reading an input file, we check if it is already loaded (i.e., belongs to that set). If it is already loaded, we count a cost of 0, otherwise we add the reading time for that file that is given as input. Files are added to the set whenever there are loaded or written (not necessarily a checkpoint). The set is cleared whenever a fail-stop error strikes on the processor or a checkpoint is performed, for simplicity. However, keeping the files needed by tasks after the checkpoint would improve even more the makespan.

When there is a failure, the rollback is easy because we always checkpoint crossover dependences. This implies that a failure on a processor p_i will only impact the tasks that have been executed on p_i since the last checkpointed task that was mapped to p_i. To rollback we explore the list of tasks backward from the current task to the last checkpointed one (we keep two pointers on these two tasks at each time to access them instantaneously), we mark each task unexecuted, we clear the set of loaded files and we can start simulating again from the last checkpointed task as if nothing happened. In the case of CkptNone, the simulation is rolled back from the first task anytime an execution or communication is interrupted.

Finally, the simulator computes the following measures: the number of file checkpoints taken, the number of task checkpoints taken, the number of failures, the total time spent checkpointing data and the execution time of the application.

4.5.3 Results

In this section, we first compare the expected makespan of our proposed checkpointing strategies (CDP and CIDP) over two baseline strategies (All and None) with the same task mapping and scheduling strategy. Then, we compare the solutions (different task mapping and scheduling heuristics combined with several checkpointing strategies) from this work with the method PropCkpt proposed in ([99]) for M-SPGs.

In Figures 4.6-4.10, we compare the four considered task mapping and scheduling strategies: HEFT and MinMin, with their chain-mapping variants HEFTC and MinMinC using boxplots[‡]. On these figures, the lower the better and the baseline at 1 is the performance of HEFT. The chain-mapping variants have the same performance or improve that of their basic counterparts, especially when communications are expensive (rightmost parts of the graphs). The other conclusion is that MinMin (resp. Min-

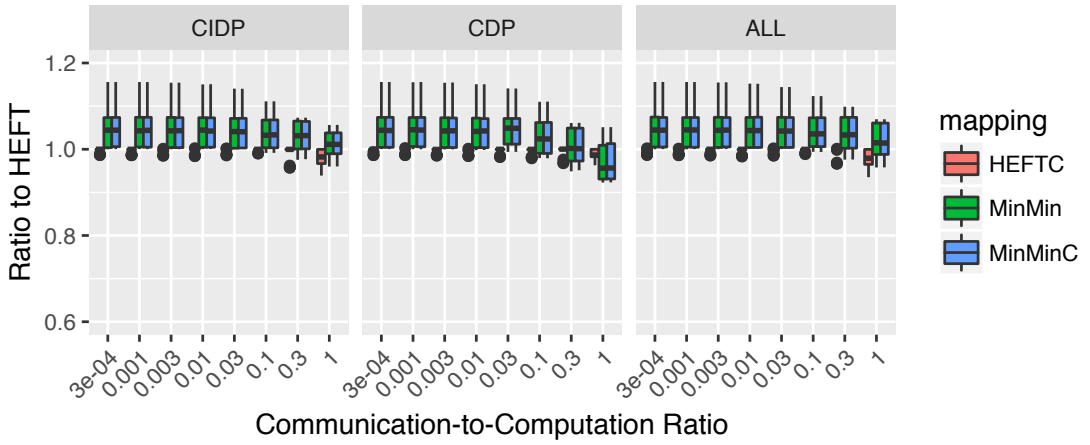


Figure 4.6: Relative performance of the four task mapping and scheduling strategies for Cholesky.

MinC) almost always achieves same or worse performance than HEFT (resp. HEFTC). This is easily explained by the fact that HEFT and HEFTC take into account the critical path of workflows. These trends are representative of the trends that can be observed for all considered graphs and workflows, but suffer from some exceptions. The chain-mapping variants can be superceded by their basic counterparts for workflows that do not include any chains (like LU in Figure 4.7), because the basic variants can use backfilling. However, backfilling sometimes backfires, even in the absence of chains, like for Sipht in Figure 4.9 where HEFTC can decrease the expected makespan by more than 30% with respect to HEFT. Overall, of the four considered task mapping and scheduling heuristics, HEFTC never achieves significantly bad performance, and most of the time achieves the best performance. This is the reason why we focus on it in the remainder of this section.

Figures 4.11 through 4.18 present the expected makespans achieved by CDP, CIDP and None divided by that of All when the Communication-to-Computation Ratio increases. Therefore, the lower the better and data points below the $y = 1$ line denote cases in which these strategies outperform the competitor All (i.e., achieve a lower expected makespan). Each figure shows results for workflows with different number of tasks, ranging from 50 to 1240 tasks (each line of subfigure is for a different size, the number of tasks being reported on the rightmost column), for various number of processors P (different line styles), and for the three p_{fail} values (0.0001, 0.001, 0.01). We report on these figures the average number of failures that occur for the 10,000 random trials for each setting. These numbers are reported in black above the horizontal axis in each figure. The other two lines of numbers are the number of checkpointed tasks for the CDP and CIDP strategies, each number is printed with the same color as the curve of the corresponding strategy.

A clear observation is that CIDP never achieves worse performance than All: ei-

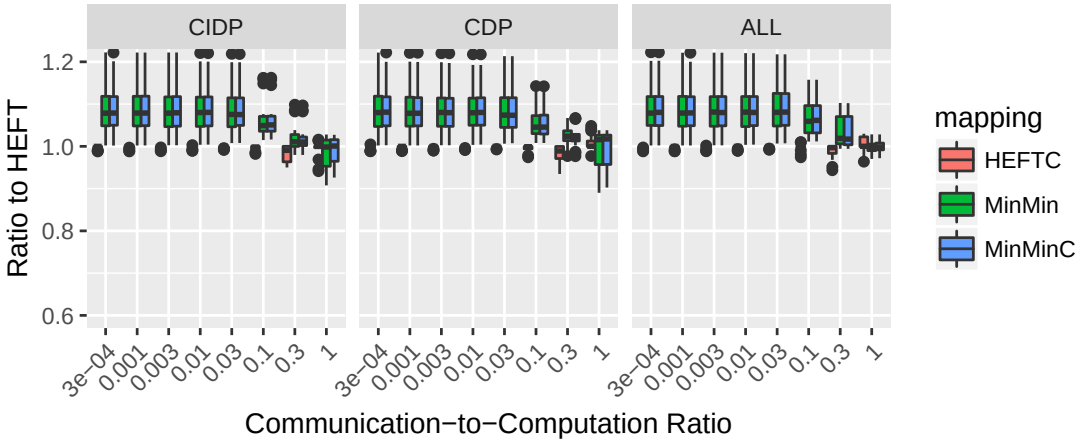


Figure 4.7: Relative performance of the four task mapping and scheduling strategies for LU.

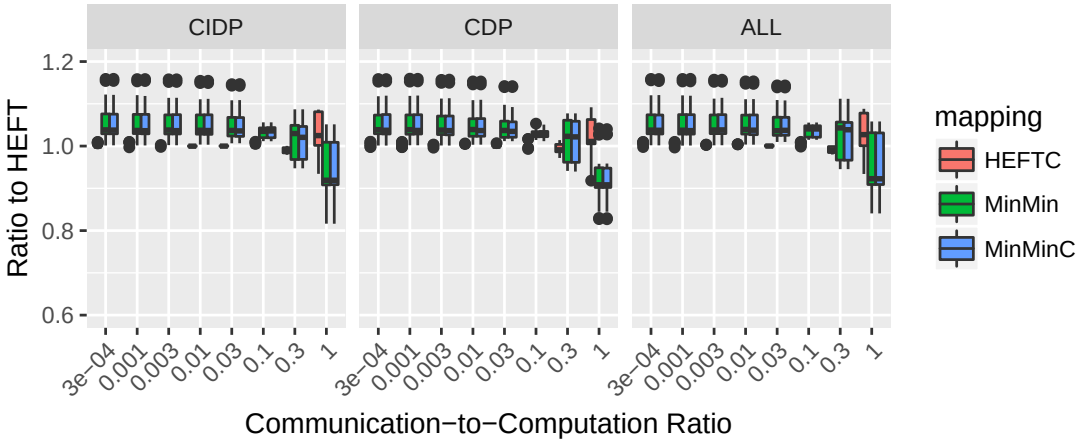


Figure 4.8: Relative performance of the four task mapping and scheduling strategies for QR.

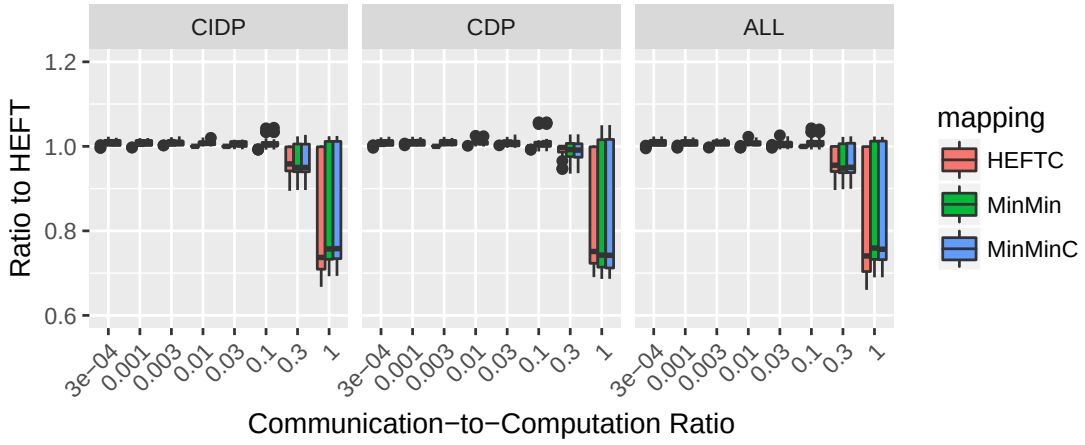


Figure 4.9: Relative performance of the four task mapping and scheduling strategies for Sipht.

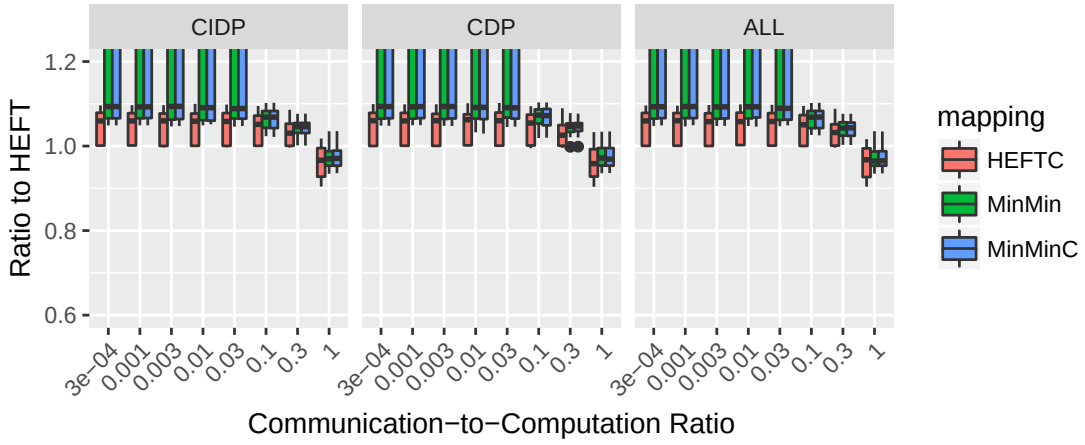


Figure 4.10: Relative performance of the four task mapping and scheduling strategies for CyberShake.

ther it achieves a similar performance or it outperforms ALL, especially when communications, and thus checkpoints, are expensive (in the rightmost parts of graphs). It should be noted that when checkpoints come for free (leftmost parts of graphs), ALL and CIDP have the same performance as they do the same thing: they checkpoint all tasks. When the number of failures rises, the optimal solution is to checkpoint more tasks, potentially all of them, and the gain of CIDP with respect to ALL therefore decreases. This can be seen, for instance, on Figure 4.12 when $p_{\text{fail}} = 0.01$, $n = 385$ and there are 385 tasks checkpointed.

In the majority of cases, CDP also achieves similar or better performance than ALL. As we explained in Section 4.4, the dynamic programming algorithm is well-defined for CIDP, which checkpoints all induced dependences. However, CDP tries to save some checkpointing overhead by not systematically checkpointing induced dependences. As a consequence, the dynamic programming algorithm estimations of expected execution times may be inaccurate, which explains the sometimes bad performance of CDP. There are only a couple of CCR values for CYBERSHAKE for which CDP achieves a significantly worse performance than ALL. On the contrary, CDP often has better performance than CIDP when checkpointing cost is high. In all scenarios, CDP checkpoints less or the same number of tasks than CIDP. Depending on the checkpointing cost and failure rate, CDP can lead to significant improvement over ALL. For workflows as dense as LU, we save more than 10% when $CCR = 1$ for both strategies, and CDP even achieves 35% saving for SIPHT. As the CCR decreases, the ratio converges to 1. As already pointed out, this is because both strategies decide to checkpoint most, if not all, tasks, when checkpointing becomes cheaper.

CDP and CIDP achieve better results than NONE except when (i) checkpoints are expensive (high CCR) and/or (ii) failures are rare (low p_{fail}). In these cases, checkpointing is a losing proposition, and yet our strategies, by design, always checkpoints some files (they checkpoint all crossover files and even induced dependences for CIDP). In practice, in such cases, the optimal approach is to bet that no failure will happen and to restart the whole workflow execution from scratch upon the very rare occurrence of a failure. NONE becomes worse whenever there are more failing tasks, i.e., when the failure rate increases (going from the leftmost column to the rightmost one in the figures), and/or when the number of tasks increases (going from the topmost row to the bottom one in the figures). When the failure rate is high and the workflows are large (the bottom right corner of the figures), the relative expected makespan of NONE is so high that it does not appear in the plots. The above results, and our experimental methodology in general, make it possible to identify these cases so as to select which approach to use in practical situations.

Figure 4.19 presents the aggregated results for the 180 STG random DAGs with boxplots. The trends on these graphs are the same as already reported. This confirms the generality of our conclusions.

Finally, we compare our new general approach with PROPCKPT, the approach specific to M-SPGs that is proposed in ([99]). Figures 4.20-4.22 present this comparison for Montage, Ligo and Genome, which are the three M-SPGs presented in ([99]). Overall,

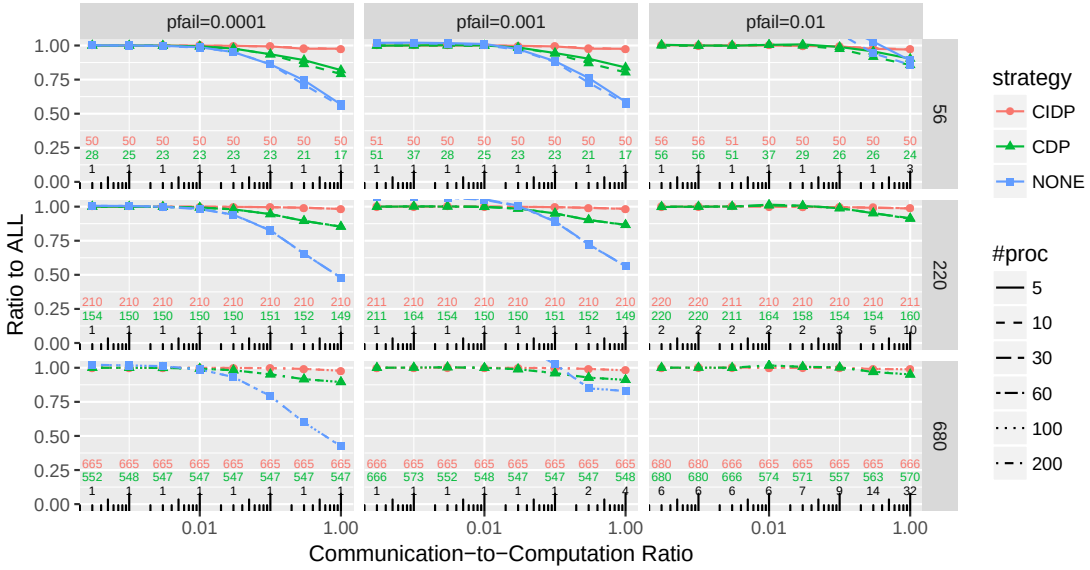


Figure 4.11: Performance of the different checkpointing strategies for Cholesky using HEFTC for task mapping and scheduling.

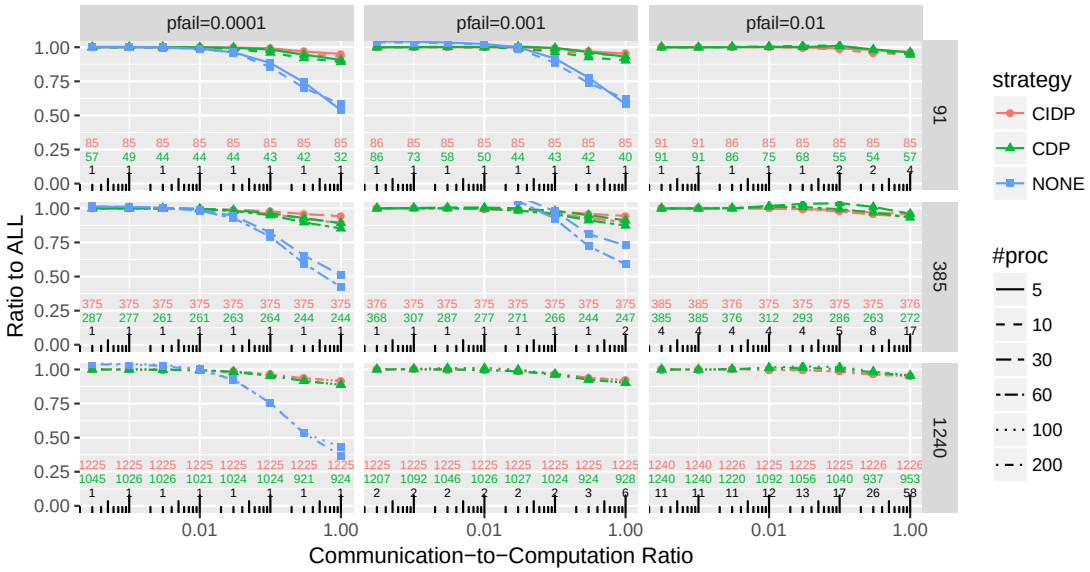


Figure 4.12: Performance of the different checkpointing strategies for LU using HEFTC for task mapping and scheduling.

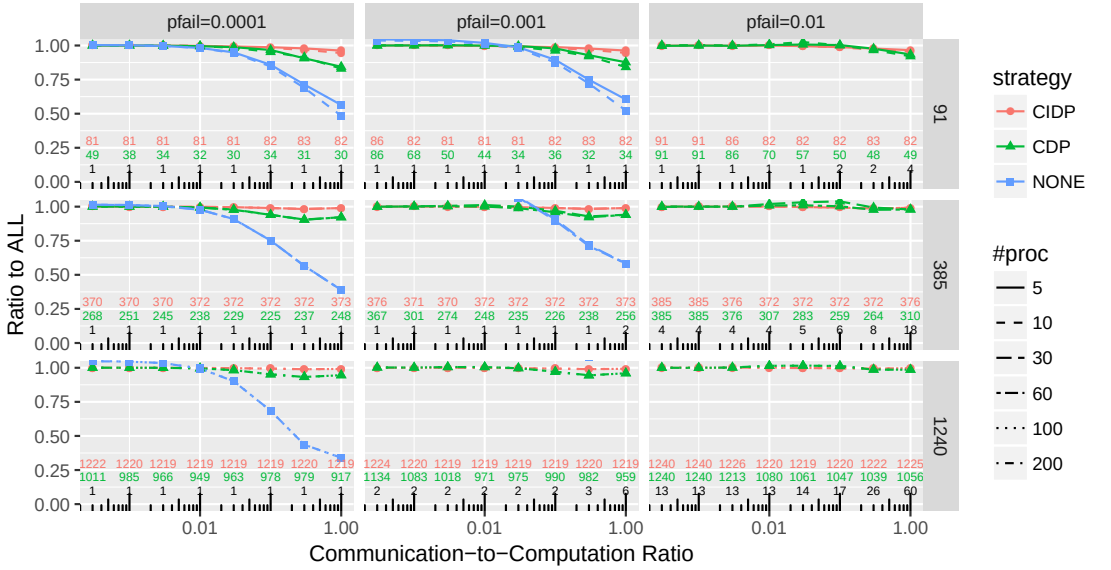


Figure 4.13: Performance of the different checkpointing strategies for QR using HEFTC for task mapping and scheduling.

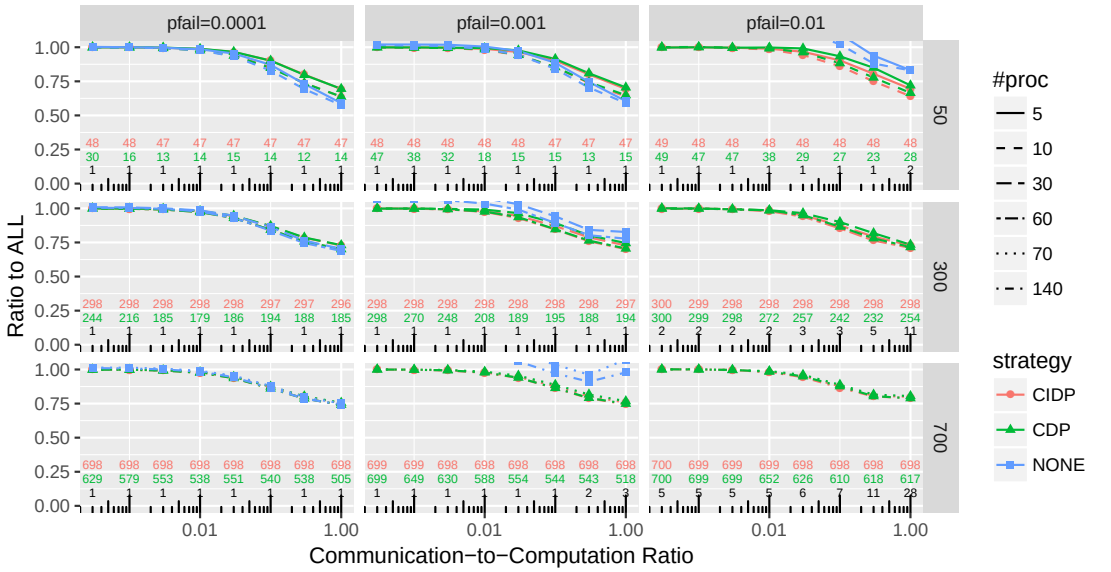


Figure 4.14: Performance of the different checkpointing strategies for Montage using HEFTC for task mapping and scheduling.

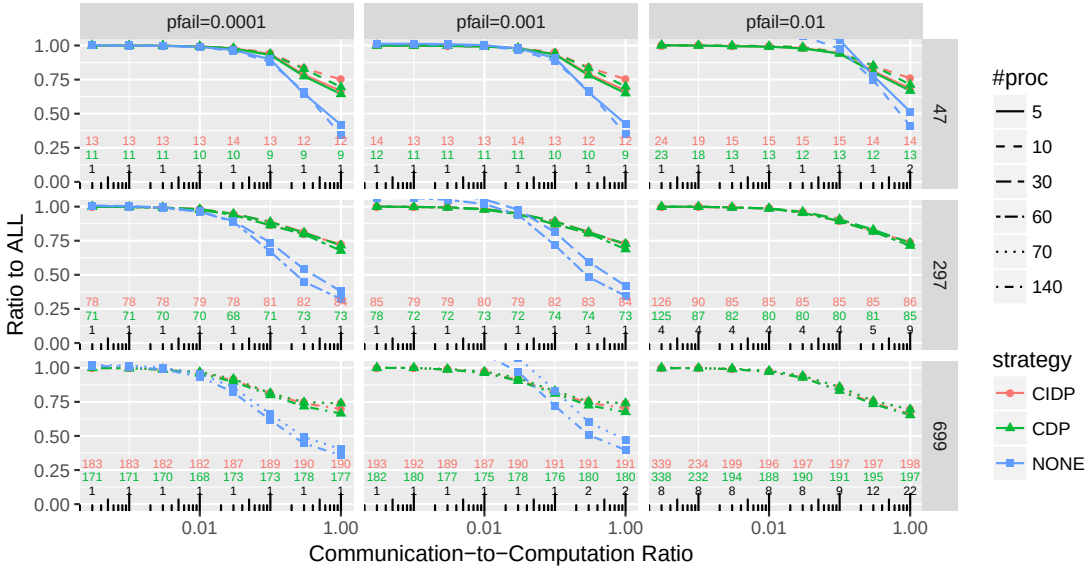


Figure 4.15: Performance of the different checkpointing strategies for Genome using HEFTC for task mapping and scheduling.

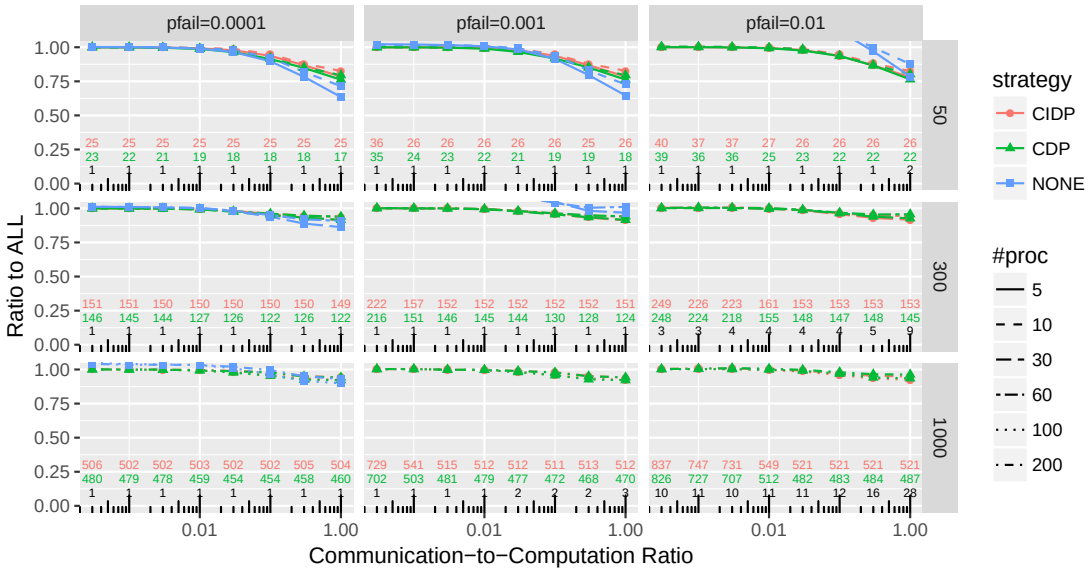


Figure 4.16: Performance of the different checkpointing strategies for Ligo using HEFTC for task mapping and scheduling.

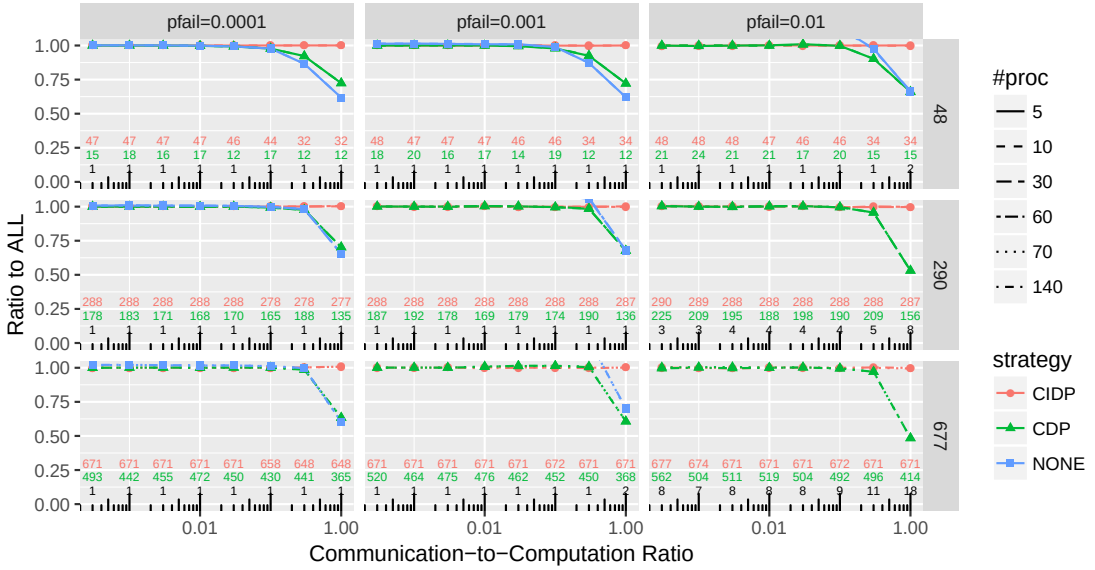


Figure 4.17: Performance of the different checkpointing strategies for Sipht using HEFTC for task mapping and scheduling.

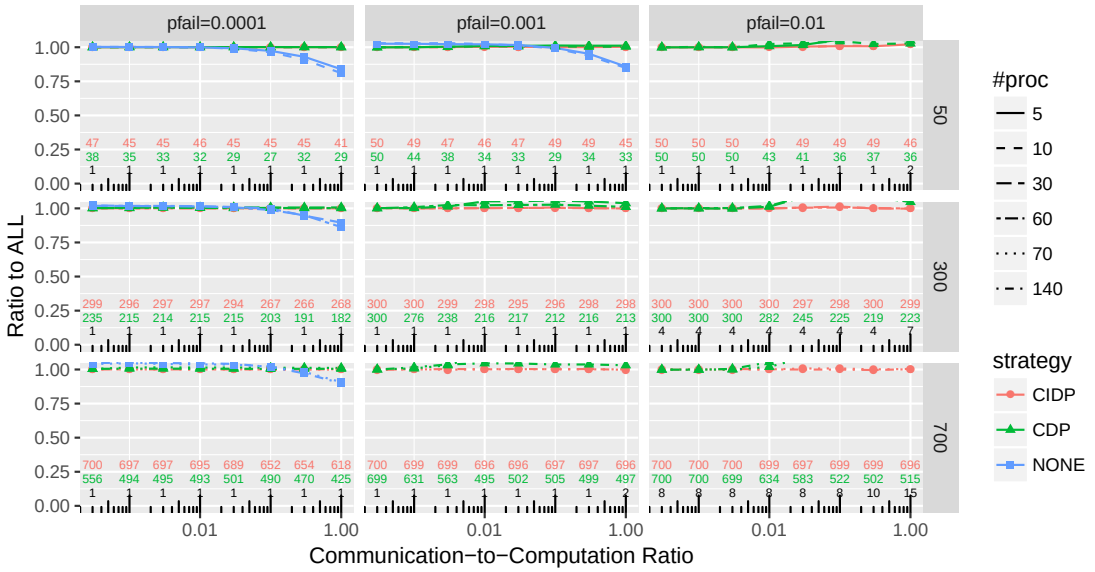


Figure 4.18: Performance of the different checkpointing strategies for CyberShake using HEFTC for task mapping and scheduling.

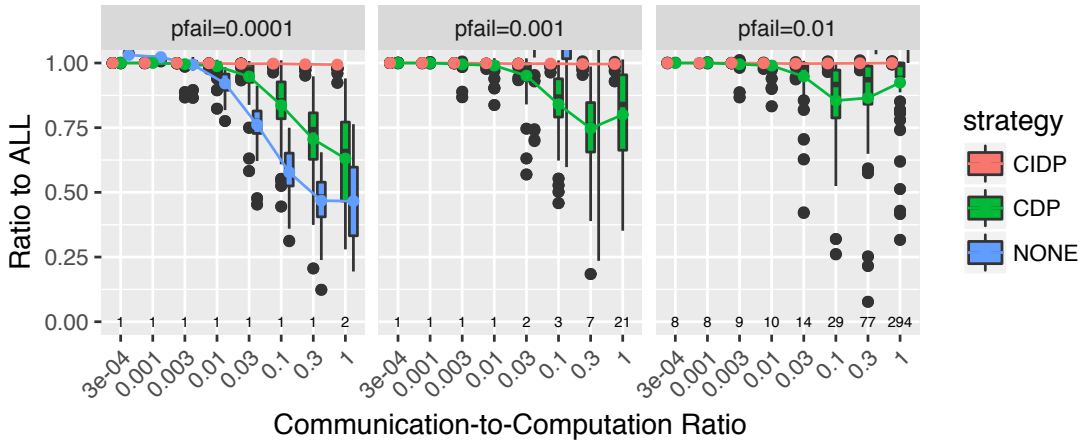


Figure 4.19: Average performance of the different checkpointing strategies for the STG task graphs using HEFTC for task mapping and scheduling.

the new approaches perform better than PROPCKPT.

4.6 Related work

Checkpointing workflows has received considerable attention in the recent years, but no satisfactory solution has yet been proposed for fail-stop failures and general DAGs.

Many authors ([115, 39, 118]) have considered soft errors, by which a task execution fails but does not lead to completely losing the data present in the processor memory. Fail-stop errors have far more drastic consequences than soft errors as they induce the loss of all data present in memory. Therefore they require different solutions.

As discussed in Section 6.1, silent errors represent do not interrupt the execution of the task but corrupt its output data. Their net effect is the same, since a task must be re-executed whenever a silent error is detected. Their detection requires the use of some silent error detectors at the end of a task's execution. Two well-known examples of fault detectors are Algorithm-Based Fault Tolerance (ABFT) (see Section 3.5.2) and silent error detectors based on domain-specific data analytics ([15, 16, 25]). As we only consider fail-stop errors we do not need to use fault detectors.

Relatively few published works have studied fail-stop failures, rather than soft and silent errors, in the context of workflow applications. When the workflow consists of a linear chain of tasks, the problem of finding the optimal checkpoint strategy, i.e., determining which tasks to checkpoint, has been solved by Toueg and Babaoglu ([183]) using a dynamic programming algorithm. The algorithm of ([183]) was later extended in ([23]) to cope with both fail-stop and silent errors simultaneously. When the workflow is general but comprised of parallel tasks that each executes on the whole platform, the problem of placing checkpoints is NP-complete for simple join graphs ([11]) (this is because the original workflow is not a chain but must be linearized). In the

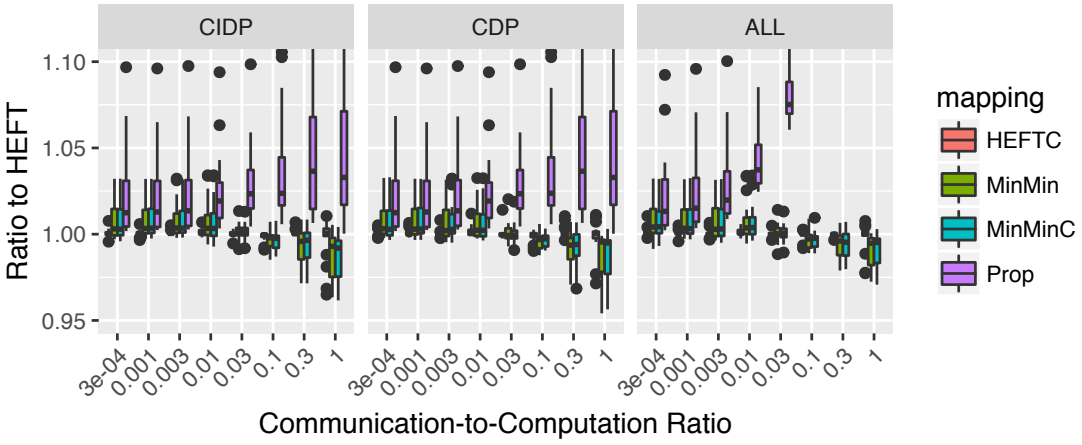


Figure 4.20: Relative performance of the four task mapping and scheduling strategies and of PropCkpt for Montage.

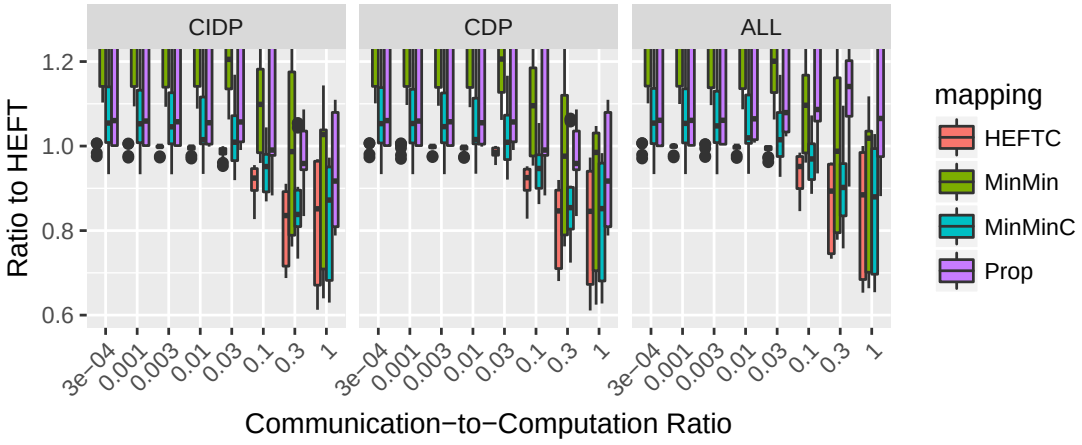


Figure 4.21: Relative performance of the four task mapping and scheduling strategies and of PropCkpt for Ligo.

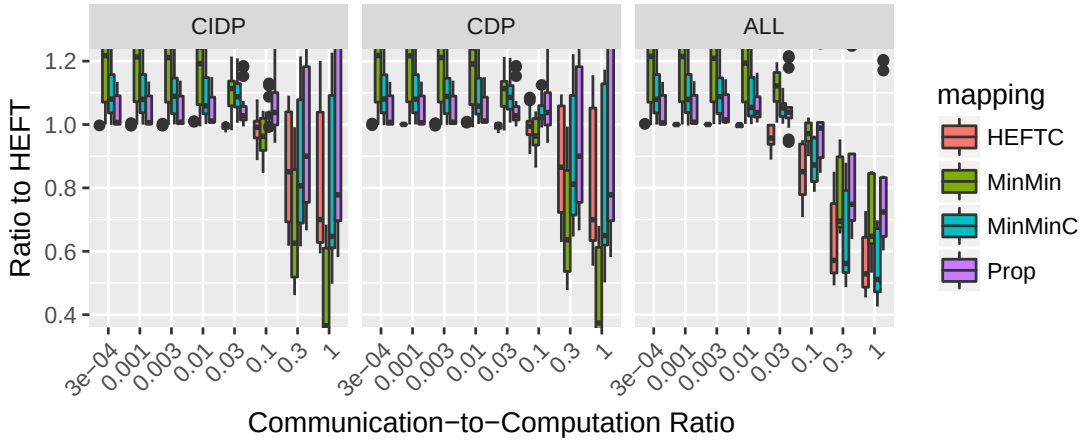


Figure 4.22: Relative performance of the four task mapping and scheduling strategies and of PropCkpt for Genome.

most general case, tasks of a workflow do not necessarily span the whole platform when executing. Existing work in this most general context diverges from ours as follows: either there is a limit to the number of failures that an execution can cope with ([188]), or the optimization objective is reliability ([7]), meaning that the application execution can fail altogether. The only exception that we are aware of is some previous work ([99]). The limitation of that work was different: the proposed solution could only deal with workflows whose structure was a Minimal Series-Parallel Graph (a generalization of Series-Parallel Graph).

To the best of our knowledge, this work is the first approach (beyond application-specific solutions) that (i) does not resort to linearizing the entire workflow as a chain of (parallel) tasks; (ii) can be applied to any workflow; (iii) can cope with an arbitrary number of failures; (iv) always guarantees a successful application execution; and (v) minimizes the (expectation of) the application execution time. As a result, we propose the first DAG scheduling/checkpointing algorithm that allows arbitrary workflows to execute concurrently on multiple failure-prone processors in standard task-parallel fashion.

4.7 Conclusion

This work tackles the challenging problem of executing arbitrary workflows on homogeneous processors, with reasonable performance in presence of failures but without incurring a prohibitive cost when no failure strikes. While CkptAll meets the first objective by expensively checkpointing every task and CkptNone meets the second one by avoiding any checkpoint at all, we propose new strategies that provide different trade-offs between these two extremes. First, all crossover dependences, corresponding to file transfers between processors, are checkpointed, which prevents re-execution

propagation between processors in case of failure. Then, a DP (Dynamic Programming) solution is used to insert additional checkpoints to minimize the expected completion time. Additional (induced) checkpoints may be added prior to the DP execution to provide it with more accurate information. Moreover, different mapping strategies that extend classical ones to reduce the number of checkpoints were also proposed. To the best of our knowledge, these new strategies are the first to be tuned to minimize the need for checkpointing while mapping tasks. Extensive experiments with a discrete event simulator, conducted for both synthetic and realistic instances, show that our approaches significantly outperform CkptAll and CkptNone in most scenarios.

Future work will aim at extending our approach to workflows with parallel moldable tasks ([68]). Such an extension raises yet another significant challenge: now the number of processors assigned to each task becomes a parameter to the proposed solutions, with a dramatic impact on both performance and resilience.

Notes

[†]In fact, because we have homogeneous processors, we use MCP (Modified Critical Path) ([194]) with backfilling, which is exactly HEFT in this context.

[‡]Each boxplot consists of a bold line for the median, a box for the quartiles, whiskers that extend at most to 1.5 times the interquartile range from the box and additional points for outliers.

Part II

Coupling checkpointing with replication

Chapter 5

Using Checkpointing and Replication for Reliable Execution of Linear Workflows with Fail-Stop and Silent Errors

Large-scale platforms currently experience errors from two different sources, namely fail-stop errors (which interrupt the execution) and silent errors (which strike unnoticed and corrupt data). This work combines checkpointing and replication for the reliable execution of linear workflows on platforms subject to these two error types. While checkpointing and replication have been studied separately, their combination has not yet been investigated despite its promising potential to minimize the execution time of linear workflows in error-prone environments. Moreover, combined checkpointing and replication has not yet been studied in the presence of both fail-stop and silent errors. The combination raises new problems: for each task, we have to decide whether to checkpoint and/or replicate it to ensure its reliable execution. We provide an optimal dynamic programming algorithm of quadratic complexity to solve both problems. This dynamic programming algorithm has been validated through extensive simulations that reveal the conditions in which checkpointing only, replication only, or the combination of both techniques, lead to improved performance. The work in this chapter is joint work with Anne Benoit, Aurélien Cavelan, Florina M. Ciorba and Yves Robert, and has been published in the *International Journal of Networking and Computing* (IJNC) [J3].

5.1 Introduction

Several high-performance computing (HPC) applications are designed as a succession of (typically large) tightly-coupled computational kernels, or tasks, that should be executed in sequence [37, 119, 57]. These parallel tasks are executed on the entire platform, and they exchange data at the end of their execution. In other words, the task graph is a linear chain, and each task (except maybe the first one and the last one) reads data from its predecessor and produces data for its successor. Such linear

chains of tasks also appear in image processing applications [137], and are usually called *linear workflows* [177].

The first objective when considering linear workflows is to ensure their *efficient execution*, which amounts to minimizing the total parallel execution time, or makespan. However, a *reliable execution* is also critical to performance. Indeed, large-scale platforms are increasingly subject to errors [41, 42]. Scale is the enemy here: even if each computing resource is very reliable, with, say, a Mean Time Between Errors (MTBE) of ten years, meaning that each resource will experience an error only every 10 years on average, a platform composed of $100,000$ of such resources will experience an error every fifty minutes [106]. Hence, fault-tolerance techniques to mitigate the impact of errors are required to ensure a correct and uninterrupted execution of the application [135]. To further complicate matters, several types of errors need to be considered when computing at scale. In addition to the classical fail-stop errors (such as hardware failures or crashes), silent errors (also known as silent data corruptions) constitute another threat that can no longer be ignored [147, 208, 209, 210, 140]. There are several causes of silent errors, such as cosmic radiation, packaging pollution, among others. Silent errors can strike the cache and memory (bit flips) components as well as the CPU operations; in the latter case they resemble floating-point errors due to improper rounding, but have a dramatically larger impact because any bit of the result, not only low-order mantissa bits, can be corrupted.

The standard approach to cope with fail-stop errors is checkpoint with rollback and recovery [47, 74]: in the context of linear workflow applications, each task can decide to take a checkpoint after it has correctly executed. A checkpoint is simply a file including all intermediate results and associated data that is saved on a storage medium resilient to errors; it can be either the memory of another processor, a local disk, or a remote disk. This file can be recovered if a successor task experiences an error later in the execution. If there is an error while some task is executing, the application has to roll back to the last checkpointed task (or to start recomputing again from scratch if no checkpoint was taken). Then the checkpoint is read from the storage medium (recovery phase), and execution resumes from that task onward. If the checkpoint was taken many tasks before an error strikes, there is a lot of re-execution involved, which calls for more frequent checkpoints. However, checkpointing incurs a significant overhead, and is a mere waste of resources if no error strikes. Altogether, there is a trade-off to be found, and one may want to checkpoint only carefully selected tasks.

While checkpoint/restart [47, 73, 74] is the de-facto recovery technique for addressing fail-stop errors, there is no widely adopted general-purpose technique to cope with silent errors. The challenge with silent errors is *detection latency*: contrarily to a fail-stop error whose detection is immediate, a silent error is identified only when the corrupted data is activated and/or leads to an unusual application behavior. However, checkpoint and rollback recovery assumes instantaneous error detection, and this raises a new difficulty: if the error stroke before the last checkpoint, and is detected after that checkpoint, then the checkpoint is corrupted and cannot be used to restore

the application. To address the problem of silent errors, many application-specific detectors, or verification mechanisms, have been proposed. We apply such a verification mechanism after each task in this chapter. Our approach is agnostic of the nature of the verification mechanism (checksum, error correcting code, coherence test, etc.). In this context, if the verification succeeds, then the output of the task is correct, and one can safely either proceed to the next task directly, or save the result beforehand by taking a checkpoint. Otherwise, if verification fails we have to rollback to the last saved checkpoint and re-execute the work since that point on. However, and contrarily to fail-stop errors, silent errors do not cause the loss of the entire memory content of the affected processor. To account for this difference, we use a two-level checkpointing scheme: the checkpoint file is saved in the main memory of the processor before being transferred to some storage (disk) that is resilient to fail-stop errors. This allows for recovering faster after a silent error than after a fail-stop error.

Replication is a well-known, but costly, method to deal with both, fail-stop errors [159, 75, 205, 76, 82, 72, 172, 46] and silent errors [145, 21]. While both checkpointing and replication have been extensively studied separately, their combination has not yet been investigated in the context of linear workflows, despite its promising potential to minimize the execution time in error-prone environments. The contributions of this work are the following:

- We provide a detailed model for the reliable execution of linear workflows, where each task can be replicated or not, and with a two-level checkpoint/recovery mechanism whose cost depends both on the number of processors executing the task, and on whether the task is replicated or not.
- We address both fail-stop and silent errors. We perform a verification after each task to detect silent errors and recover from the last in-memory checkpoint after detecting one. We recover from the last disk checkpoint after a fail-stop error. If a task is replicated, we do not need to roll back and we can directly proceed to the next task, unless both replicas have been affected (by either error type).
- We design an optimal dynamic programming algorithm that minimizes the makespan of a linear workflow with n tasks, with a quadratic complexity, in the presence of fail-stop and silent errors.
- We conduct extensive experiments to evaluate the impact of using both replication and checkpointing during execution, and compare them to an execution without replication.
- We provide guidelines about when it is beneficial to employ checkpointing only, replication only, or to combine both techniques together.

The chapter is organized as follows. Section 5.2 details the model and formalizes the objective function and the optimization problem. Section 5.3 presents a preliminary result for the dynamic programming algorithm: we explain how to compute the expected time needed to execute a single task (replicated or not), assuming that its predecessor has been checkpointed. The proposed optimal dynamic programming algorithm is outlined in Section 5.4. The experimental validation is provided in Section 5.5. Finally, related work is discussed in Section 5.6, and the work is concluded in

Section 5.7.

5.2 Model and objective

This section details the framework of this study. We start with the application and platform models, then we detail the verification, checkpointing and replication, and finally we state the optimization problem.

5.2.1 Application model

We target applications whose workflows represent linear chains of parallel tasks. More precisely, for one application, consider a chain $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ of n parallel tasks T_i, $1 \leq i \leq n$. Hence, T_1 must be completed before executing T_2, and so on.

Here, each T_i is a parallel task whose speedup profile obeys *Amdahl's law* [3]: the total work, w_i, consists of a sequential fraction $\alpha_i w_i$, $0 \leq \alpha_i \leq 1$, and the remaining fraction $(1-\alpha_i)w_i$ perfectly parallel. The (error-free) execution time, T_i, using q_i processors is thus $w_i \left(\alpha_i + \frac{1-\alpha_i}{q_i}\right)$. Without loss of generality, we assume that processors execute the tasks at unit speed, and we use time units and work units interchangeably. While our study is agnostic of task granularity, it applies primarily to frameworks where tasks represent large computational entities whose execution takes from a few minutes up to tens of minutes. In such frameworks, it may be worthwhile to replicate or to checkpoint tasks to mitigate the impact of errors.

5.2.2 Execution platform

We target a *homogeneous* platform with p processors P_i, $1 \leq i \leq p$. We assume that the platform is subject to fail-stop and silent errors whose inter-arrival times follow an Exponential distribution. More precisely, let λ^F_{ind} be the fail-stop error rate of each individual processor P_i: the probability of having a fail-stop error striking P_i within T time-units is $\mathbb{P}(X \leq T) = 1 - e^{-\lambda^F_{ind} T}$. Similarly, let λ^S_{ind} be the silent error rate of each individual processor P_i: the probability of having a silent error striking P_i within T time-units is $\mathbb{P}(Y \leq T) = 1 - e^{-\lambda^S_{ind} T}$. Then, a computation on $q \leq p$ processors has an error rate $q\lambda^F_{ind}$ for fail-stop errors, and $q\lambda^S_{ind}$ for silent errors. The probability of having a fail-stop error within T time-units and with q processors becomes $1 - e^{-q\lambda^F_{ind} T}$ (and $1 - e^{-q\lambda^S_{ind} T}$ for a silent error) [106].

5.2.3 Verification

To detect silent errors, we add a verification mechanism at the end of each task. This ensures that the error will be detected as soon as possible. The verification following task T_i has a cost V_i. We assume that the verification mechanism has a perfect recall (it detects all errors). This guarantees that all taken checkpoints are correct, because

they are always preceded by a verification. Similarly, we assume that no silent error can strike during the verification.

The cost V_i depends upon the detector and can thereby take a wide range of values. In this work, we adopt a quite general formula and use

$$V_i(q_i) = u_i + \frac{v_i}{q_i} \tag{5.1}$$

to model the cost of verifying task T_i when executed with q_i processors, where u_i and v_i denote the sequential and parallel cost of the verification, respectively. In the experiments (Section 5.5.2), we instantiate the model with two cases:

- We use $u_i = \beta w_i$ and $v_i = 0$, where β is a small parameter (around 1%). This means that the cost of the verification is proportional to the sequential cost w_i of T_i. It corresponds to the case of data-oriented kernels processing large files and checksumming for verification in a centralized location (hence sequentially) [23].

- We use $u_i = 0$ and $v_i = \beta w_i$. This means that the cost of the verification is proportional to the parallel fraction of T_i. It corresponds to the same scenario as above, but where checksumming is performed in parallel on all enrolled processors.

5.2.4 Checkpointing

The output of each task T_i can be checkpointed in time C_i. We use a two-level checkpoint protocol where the checkpoint is first saved locally (memory checkpoint) before being transferred to a slower but reliable storage like a filesystem (disk checkpoint). The memory checkpoint will be lost when a fail-stop error strikes a processor (and its local data), whereas the disk checkpoint will always remain available to restart the application.

When a fail-stop error strikes during the execution of T_i, we first incur a downtime D, and then we must start the execution from the task following the last checkpoint. Hence, if T_j is the last checkpointed task, the execution starts again at task T_{j+1}, and the recovery cost is R^D_{j+1}, which amounts to reading the disk checkpoint of task T_j. When a silent error is detected at the end of T_i by the verification mechanism, we also roll back to the last checkpointed task T_j, but (i) we do not pay the downtime D; and (ii) the recovery cost is R^M_{j+1}, which amounts to reading the memory checkpoint of task T_j (hence at a much smaller cost than for a fail-stop error).

The checkpoint cost C_i, and both recovery costs R^D_{j+1} and R^M_{j+1} clearly depend upon the checkpoint protocol and storage medium, as well as upon the number q_i of enrolled processors. In this work, we adopt a quite general formula for checkpoint times and use

$$C_i(q_i) = a_i + \frac{b_i}{q_i} + c_i q_i \tag{5.2}$$

to model the time to save a checkpoint after T_i executed with q_i processors. Here, $a_i + \frac{b_i}{q_i}$ represents the I/O overhead to write the task output file M_i to the storage medium. For in-memory checkpointing [204], $a_i + \frac{b_i}{q_i}$ is the communication time, in which a_i denotes the latency to access the storage system; then we have $\frac{b_i}{q_i} = \frac{M_i}{\tau_{net} q_i}$, where τ_{net} is the network bandwidth (each processor stores $\frac{M_i}{q_i}$ data items). For coordinated checkpointing to stable storage, there are two cases: if the I/O bottleneck is the storage system's bandwidth, then $a_i = \beta + \frac{M_i}{\tau_{io}}$ and $b_i = 0$, where β is a start-up time and τ_{io} is the I/O bandwidth; otherwise, if the I/O bottleneck is the network latency, we retrieve the same formula as for in-memory checkpointing. Finally, $c_i q_i$ represents the message passing overhead that grows linearly with the number of processors, in order for all processors to reach a global consistent state [74, 206].

For the cost of recovery (from memory or from disk), we assume similar formulas:

$$R_i^M(q_i) = a_i^M + \frac{b_i^M}{q_i} + c_i^M q_i; \quad R_i^D(q_i) = a_i^D + \frac{b_i^D}{q_i} + c_i^D q_i. \tag{5.3}$$

The coefficients depend on the type of recovery: again, a memory recovery is much faster than a disk recovery. If we further assume that reading and writing from/to the same storage medium (memory or disk) have same cost, we have

$$C_i(q_i) = R_{i+1}^D(q_i) + R_{i+1}^M(q_i)$$

since recovering for task T_{i+1} amounts to reading the checkpoint from task T_i.

Finally, we assume that there is a fictitious task T_0 of zero weight ($w_0 = 0$) that is always checkpointed, so that $R_1^D(q_1)$ represents the time for I/O input from the external world. Similarly, we systematically checkpoint the last task T_n, in order to account for the I/O output time $C_n(q_n)$.

5.2.5 Replication

When executing a task, we envision two possibilities: either the task is not replicated, or it is replicated. To explain the impact of replication, we momentarily assume that we consider fail-stop errors only. Then we return to the scenario with both fail-stop and silent errors.

With fail-stop errors only, consider a task T_i, and assume for simplicity that the predecessor T_{i-1} of T_i has been checkpointed. If it is not the case, i.e., if the predecessor T_{i-1} of T_i is not checkpointed, we have to roll back to the last checkpointed task, say T_k where $k < i-1$, whenever an error strikes, and re-execute the entire segment from T_{k+1} to T_i instead of just T_i.

Without replication, a single copy of T_i is executed on the entire platform, hence with $q_i = p$ processors. Then we let $\mathbb{E}^{norep}(i)$ denote the expected execution time of T_i when accounting for errors. We attempt a first execution, which takes $T_i^{norep} = w_i \left(\alpha_i + \frac{1-\alpha_i}{p}\right)$ if no fail-stop error strikes. But if a fail-stop error does strike, we must

account for the time that has been lost (between the beginning of the execution and the fail-stop error), then perform a downtime D, a recovery $R_i(p)$ (since we use the entire platform for T_i), and then re-execute T_i from scratch. Similarly, if we decide to checkpoint after T_i, we need $C_i(p)$ time units. We explain how to compute $\mathbb{E}^{norep}(i)$ in Section 5.3.

With replication, two copies of T_i are executed in parallel, each with $q_i = \frac{p}{2}$ processors. If no fail-stop error strikes, both copies finish execution in time $T_i^{rep} = w_i \left(\alpha_i + \frac{1-\alpha_i}{\frac{p}{2}}\right)$, since each copy uses $\frac{p}{2}$ processors. If a fail-stop error strikes one copy, we proceed as before, account for the downtime D, recover (in time $R_i(\frac{p}{2})$ now), and restart execution with that copy. Then there are two cases: (i) if the second copy successfully completes its first execution, the fail-stop error has no impact and the execution time remains the same as the error-free execution time; (ii) however, if the second copy also fails to execute, we resume its execution, and iterate until one copy successfully completes. Of course, case (ii) is less likely to happen than case (i), which explains why replication can be useful. Finally, if we decide to checkpoint after T_i, the first successful copy will take the checkpoint in time $C_i(\frac{p}{2})$.

Replication raises several complications in terms of checkpoint and recovery costs. When a replicated task T_i is checkpointed, we can enforce that only one copy (the first one to complete execution) would write the output data onto the storage medium, hence with a cost $C_i(\frac{p}{2})$, as stated above. Similarly, when a single copy of a replicated task T_i performs a recovery after a fail-stop error, the cost would be $R_i(\frac{p}{2})$. However, in the unlikely event where both copies are struck by a fail-stop error at close time instances, their recoveries would overlap, and the cost can vary anywhere between $R_i(\frac{p}{2})$ and $2R_i(\frac{p}{2})$, depending upon the amount of contention, the length of the overlap and where the I/O bottleneck lies. We will experimentally evaluate the impact of the recovery cost with replication in Section 5.5.1. For simplicity, in the rest of the chapter, we use C_i^{rep} for the checkpoint cost of T_i when it is replicated, and C_i^{norep} when it is not. Similarly, we use R_i^{Drep} or R_i^{Mrep} for the recovery costs (disk or memory) when T_i is replicated, and R_i^{Dnorep} or R_i^{Mnorep} when it is not. Note that the recovery cost of T_i depends upon whether it is replicated or not, but does not depend upon whether the checkpointed task T_{i-1} was replicated or not, since we need to read the same file from the storage medium in both cases. The values of C_i^{rep} and C_i^{norep} can be instantiated from Equation (5.2) and those of R_i^{Drep}, R_i^{Dnorep}, R_i^{Mrep} and R_i^{Mnorep} can be instantiated from Equation (5.3). We let $\mathbb{E}^{rep}(i)$ denote the expected execution time of T_i with replication and when accounting for fail-stop errors, when T_{i-1} is checkpointed. The derivation of $\mathbb{E}^{rep}(i)$ is significantly more complicated than for $\mathbb{E}^{norep}(i)$ and represents a new contribution of this work, detailed in Section 5.3.2.

We now detail the impact of replication when both fail-stop and silent errors can strike. First, we have to state how the verification cost V_i of task T_i depends upon whether T_i is replicated or not. For the analysis, we keep a general model and let V_i^{rep} be the cost when T_i is replicated, and V_i^{norep} when it is not. However, as explained later in the experimental evaluation (Section 5.5.2), we use two different instantiations

of Equation (5.1), which directly give the two (possibly different) values of V_i^{rep} and V_i^{norep} as a function of parameter β.

Next, consider again a task T_i, and still assume for simplicity that the predecessor T_{i-1} of T_i has been checkpointed. The impact of fail-stop errors is the same as before, and depends upon how many replicas of T_i are executed. The only difference is that the fail-stop error can now strike either during the execution of a replica or during its verification. But if no fail-stop error strikes, we still have to perform the verification to detect a possible silent error, whose probability depends upon the error-free execution time of that replica. Recall that no silent error can strike during the verification (but a fail-stop can strike). If a silent error is detected, we have to re-execute the task, in which case we recover from the memory checkpoint instead of from the disk checkpoint.

Finally, we extend the definition of $\mathbb{E}^{norep}(i)$ and $\mathbb{E}^{rep}(i)$ to account for both fail-stop and silent errors, when T_{i-1} is checkpointed. We explain how to compute both quantities in Section 5.3.2.

5.2.6 Optimization problem

The objective of this work is to *minimize the expected makespan* of the linear workflow in the presence of fail-stop and silent errors. For each task, we have four choices: either we replicate the task or not, and either we checkpoint it or not. More formally, for each task T_i we need to decide: (i) if it is checkpointed or not; and (ii) if it is replicated or not, (meaning that there are 4^n combinations for the whole workflow) with the objective to minimize the total execution time of the workflow. We point out that none of these decisions can be made locally. Instead, we need to account for previous decisions and optimize globally. Our major contribution of this work is to provide an optimal dynamic programming algorithm to solve this problem, which we denote as ChainsRepCkpt.

We point out that ChainsCkpt, the simpler problem without replication, i.e., optimally placing checkpoints for a chain of tasks, has been extensively studied. The first dynamic programming algorithm to solve ChainsCkpt appears in the pioneering paper of Toueg and Babaoğlu [183] back in 1984, for the scenario with fail-stop errors only (see Section 5.6 on related work for further references). Adding replication significantly complicates the solution. Here is an intuitive explanation: When the algorithm recursively considers a segment of tasks from T_i to T_j, where T_{i-1} and T_j are both checkpointed and no intermediate task T_k, $i \leq k < j$ is checkpointed, there are many cases to consider to account for possible different values in: (i) execution time, since some tasks in the segment may be replicated; (ii) checkpoint, whose cost depends upon whether T_j is replicated or not; and (iii) recovery, whose cost depends upon whether T_i is replicated or not. We provide all details in Section 5.4.

5.3 Computing $\mathbb{E}^{norep}(i)$ and $\mathbb{E}^{rep}(i)$

This section details how to compute the expected time needed to execute a task T_i, assuming that the predecessor of T_i has been checkpointed. Hence, we need to re-execute only T_i when an error strikes. We explain how to deal with the general case of re-executing a segment of tasks, some of them replicated, in Section 5.4. Here, we start with the case where T_i is not replicated. It is already known how to compute $\mathbb{E}^{norep}(i)$ [106, 23], but we present this case to help the reader follow the derivation in Section 5.3.2 for the case where T_i is replicated, which is new and much more involved.

5.3.1 Computing $\mathbb{E}^{norep}(i)$

To compute $\mathbb{E}^{norep}(i)$, the average execution time of T_i with p processors without replication, we conduct a case analysis:

- Either a fail-stop error strikes during the execution of the task and its verification (lasting $T_i^{norep} + V_i^{norep}$), and in this case we lose some work and need to re-execute the task, recovering from a disk checkpoint;
- Either there is no fail-stop error, and in this case the verification indicates whether there has been a silent error or not:
  - If a silent error is detected, we need to re-execute the task right after the verification, recovering from a memory checkpoint;
  - Otherwise the execution has been successful.

This leads to the following recursive formula:

$$\begin{aligned}\mathbb{E}^{norep}(i) = \mathbb{P}(X_p \leq T_i^{norep} + V_i^{norep}) \Big(& T_{lost}^{norep}(T_i^{norep} + V_i^{norep}) \\ & + D + R_i^{Dnorep} + \mathbb{E}^{norep}(i) \Big) \\ + (1 - \mathbb{P}(X_p \leq T_i^{norep} + V_i^{norep})) \Big(& T_i^{norep} + V_i^{norep} \\ & + \mathbb{P}(X_p' \leq T_i^{norep})(D + R_i^{Mnorep} + \mathbb{E}^{norep}(i)) \Big),\end{aligned} \tag{5.4}$$

where $\mathbb{P}(X_p \leq t)$ is the probability of having a fail-stop error on one of the p processors before time t, i.e., $\mathbb{P}(X_p \leq t) = 1 - e^{-\lambda_{ind}^F pt}$, and $\mathbb{P}(X_p' \leq t)$ is the probability of having a silent error on one of the p processors before time t, i.e., $\mathbb{P}(X_p' \leq t) = 1 - e^{-\lambda_{ind}^S pt}$. The time lost when an error strikes is the expectation of the random variable X_p, knowing that the error stroke before the end of the task and its verification. We compute it as follows:

$$
\begin{aligned}
T_{lost}^{norep}(T_i^{norep}+V_i^{norep}) &= \int_0^{+\infty} x\mathbb{P}(X_p = x | X_p \leq T_i^{norep}+V_i^{norep})dx \\
&= \frac{1}{\mathbb{P}(X_p \leq T_i^{norep}+V_i^{norep})} \int_0^{T_i^{norep}+V_i^{norep}} x\mathbb{P}(X_p = x)dx \\
&= \frac{1}{\mathbb{P}(X_p \leq T_i^{norep}+V_i^{norep})} \int_0^{T_i^{norep}+V_i^{norep}} x\frac{d\mathbb{P}(X_p \leq x)}{dx}dx
\end{aligned}
$$

After integration, we get the formula:

$$
T_{lost}^{norep}(T_i^{norep}+V_i^{norep}) = \frac{1}{\lambda_{ind}^F p} - \frac{T_i^{norep}+V_i^{norep}}{e^{\lambda_{ind}^F p(T_i^{norep}+V_i^{norep})}-1}. \tag{5.5}
$$

Replacing the left hand side term of Equation (5.5) in Equation (5.4) and solving, we derive:

$$
\begin{aligned}
\mathbb{E}^{norep}(i) = &\left(\frac{1}{\lambda_{ind}^F p} + D + R_i^{Dnorep}\right) e^{p((\lambda_{ind}^F+\lambda_{ind}^S)T_i^{norep}+\lambda_{ind}^F V_i^{norep})} \\
&- \left(\frac{1}{\lambda_{ind}^F p} + (R_i^{Dnorep} - R_i^{Mnorep})\right) e^{\lambda_{ind}^S p T_i^{norep}} - (D + R_i^{Mnorep}).
\end{aligned} \tag{5.6}
$$

Recall that $T_i^{norep} = w_i\left(\alpha_i + \frac{1-\alpha_i}{p}\right)$ in Equation (5.6). Finally, if we decide to checkpoint T_i, we simply add C_i^{norep} to $\mathbb{E}^{norep}(i)$.

5.3.2 Computing $\mathbb{E}^{rep}(i)$

We now discuss the case where T_i is replicated; each copy executes with $\frac{p}{2}$ processors. To compute $\mathbb{E}^{rep}(i)$, the expected execution time of T_i with replication, we conduct a case analysis similar to that of Section 5.3.1:

- Either two fail-stop errors strike before the end of the task and its verification (lasting $T_i^{rep}+V_i^{rep}$), with one fail-stop error striking each copy. Then we have lost some work and need to re-execute the task from a disk checkpoint;
- Or at least one copy is not hit by any fail-stop error. Then we need to account for two different cases in the analysis:
  - Both copies have survived: then we need to re-execute the task (recovering from a memory checkpoint) only if both copies are hit by a silent error.
  - Only one replica survived: then we need to re-execute the task if this replica is hit by a silent error.

This leads to the following formula:

$$\begin{aligned}\mathbb{E}^{rep}(i) = {} & \mathbb{P}(Y_p \leq T_i^{rep} + V_i^{rep})^2 \Big(T_{lost}^{rep}(T_i^{rep} + V_i^{rep}) + D + R_i^{Drep} + \mathbb{E}^{rep}(i) \Big) \\ & + (1 - \mathbb{P}(Y_p \leq T_i^{rep} + V_i^{rep})^2)(T_i^{rep} + V_i^{rep}) \\ & + \Big(2(1 - \mathbb{P}(Y_p \leq T_i^{rep} + V_i^{rep}))\mathbb{P}(Y_p \leq T_i^{rep} + V_i^{rep})\mathbb{P}(Y_p^{'} \leq T_i^{rep}) \\ & + (1 - \mathbb{P}(Y_p \leq T_i^{rep} + V_i^{rep}))^2 \mathbb{P}(Y_p^{'} \leq T_i^{rep})^2 \Big) (D + R_i^{Mrep} + \mathbb{E}^{rep}(i)),\end{aligned} \tag{5.7}$$

where $\mathbb{P}(Y_p \leq t)$ is the probability of having an error on one replica of $\frac{p}{2}$ processors before time t, i.e., $\mathbb{P}(Y_p \leq t) = 1 - e^{-\frac{\lambda_{ind}^F p}{2} t}$, and $\mathbb{P}(Y_p^{'} \leq t)$ is the probability of having a silent error on one replica of $\frac{p}{2}$ processors before time t, i.e., $\mathbb{P}(Y_p^{'} \leq t) = 1 - e^{-\frac{\lambda_{ind}^S p}{2} t}$. The first line of Equation (5.7) corresponds to the case where both replicas are hit by a fail-stop error, the second line accounts for the time spent in case at least one replica survives. The last two lines correspond to the two cases when we need to re-execute the task after the detection of a silent error (one replica alive for line 3, two replicas alive for line 4 of Equation (5.7)).

The time lost when both copies fail can be computed in a similar way as before:

$$T_{lost}^{rep}(T_i^{rep}) = \frac{1}{\mathbb{P}(Y_p \leq T_i^{rep} + V_i^{rep})} \int_0^{T_i^{rep} + V_i^{rep}} x \frac{d\mathbb{P}(Y_p \leq x)}{dx} dx.$$

After computation and verification using a Maple sheet, we obtain the following result:

$$\begin{aligned} T_{lost}^{rep}(T_i^{rep} + V_i^{rep}) = {} & \frac{(-2\lambda_{ind}^F p(T_i^{rep} + V_i^{rep}) - 4) e^{-\frac{\lambda_{ind}^F p(T_i^{rep} + V_i^{rep})}{2}}}{(e^{-\frac{\lambda_{ind}^F p(T_i^{rep} + V_i^{rep})}{2}} - 1)^2 \lambda_{ind}^F p} \\ & + \frac{(\lambda_{ind}^F p(T_i^{rep} + V_i^{rep}) + 1) e^{-\lambda_{ind}^F p(T_i^{rep} + V_i^{rep})} + 3}{(e^{-\frac{\lambda_{ind}^F p(T_i^{rep} + V_i^{rep})}{2}} - 1)^2 \lambda_{ind}^F p}. \end{aligned} \tag{5.8}$$

Replacing the left hand side term of Equation (5.8) in Equation (5.7) and solving,

we get:

$$\begin{aligned}\mathbb{E}^{rep}(i) = & -\frac{(4+2\lambda^F_{ind}p(R_i^{Drep}-R_i^{Mrep}))e^{p(\frac{\lambda^F_{ind}(T_i^{rep}+V_i^{rep})}{2}+\lambda^S_{ind}T_i^{rep})}}{(2e^{p\frac{\lambda^F_{ind}(T_i^{rep}+V_i^{rep})+\lambda^S_{ind}T_i^{rep}}{2}}-1)\cdot\lambda^F_{ind}p} \\ & +\frac{(1+\lambda^F_{ind}p(R_i^{Drep}-R_i^{Mrep}))e^{\lambda^S_{ind}pT_i^{rep}}}{(2e^{p\frac{\lambda^F_{ind}(T_i^{rep}+V_i^{rep})+\lambda^S_{ind}T_i^{rep}}{2}}-1)\cdot\lambda^F_{ind}p} \\ & +\frac{(3+\lambda^F_{ind}p(D+R_i^{Drep}))e^{p(\lambda^F_{ind}(T_i^{rep}+V_i^{rep})+\lambda^S_{ind}T_i^{rep})}}{(2e^{p\frac{\lambda^F_{ind}(T_i^{rep}+V_i^{rep})+\lambda^S_{ind}T_i^{rep}}{2}}-1)\cdot\lambda^F_{ind}p}-(D+R_i^{Mrep})\end{aligned} \tag{5.9}$$

Recall that $T_i^{rep} = w_i\left(\alpha_i + \frac{1-\alpha_i}{\frac{p}{2}}\right)$ in Equation (5.9). Finally, if we decide to checkpoint T_i, we simply add C_i^{rep} to $\mathbb{E}^{rep}(i)$.

5.4 Optimal dynamic programming algorithm

In this section, we provide an optimal dynamic programming (DP) algorithm to solve the ChainsRepCkpt problem for a linear chain of n tasks.

Theorem 4. *The optimal solution to the* ChainsRepCkpt *problem can be obtained using a dynamic programming algorithm in $O(n^2)$ time, where n is the number of tasks in the chain.*

Proof. The algorithm recursively computes the expectation of the optimal time required to execute tasks T_1 to T_i and then checkpoint T_i. As already mentioned, we need to distinguish two cases, according to whether T_i is replicated or not, because the cost of the final checkpoint depends upon this decision. Hence, we recursively compute two different functions:

- $T_{opt}^{rep}(i)$, the expectation of the optimal time required to execute tasks T_1 to T_i, knowing that T_i is replicated;
- $T_{opt}^{norep}(i)$, the expectation of the optimal time required to execute tasks T_1 to T_i, knowing that T_i is not replicated.

Note that checkpoint time is not included in $T_{opt}^{rep}(i)$ nor $T_{opt}^{norep}(i)$. The solution to ChainsRepCkpt will be given by

$$\min\left\{T_{opt}^{rep}(n)+C_n^{rep}, T_{opt}^{norep}(n)+C_n^{norep}\right\}. \tag{5.10}$$

We start with the computation of $T_{opt}^{rep}(j)$ for $1 \le j \le n$, hence assuming that the

last task T_j is replicated. We express $T_{opt}^{rep}(j)$ recursively as follows:

$$T_{opt}^{rep}(j) = \min_{1 \leq i < j} \left\{ \begin{array}{l} T_{opt}^{rep}(i) + C_i^{rep} + T_{NC}^{rep,rep}(i+1,j), \\ T_{opt}^{rep}(i) + C_i^{rep} + T_{NC}^{norep,rep}(i+1,j), \\ T_{opt}^{norep}(i) + C_i^{norep} + T_{NC}^{rep,rep}(i+1,j), \\ T_{opt}^{norep}(i) + C_i^{norep} + T_{NC}^{norep,rep}(i+1,j), \\ R_1^{Drep} + T_{NC}^{rep,rep}(1,j), \\ R_1^{Dnorep} + T_{NC}^{norep,rep}(1,j) \end{array} \right\} \tag{5.11}$$

In Equation (5.11), T_i corresponds to the last checkpointed task before T_j, and we try all possible locations T_i for taking a checkpoint before T_j. The first four lines correspond to the case where there is indeed an intermediate task T_i between T_1 and T_j that is checkpointed, while the last two lines correspond to the case where no checkpoint at all is taken until after T_j.

The first two lines of Equation (5.11) apply to the case where T_i is replicated. Line 1 is for the case when T_{i+1} is replicated, and line 2 when it is not. In the first line of Equation (5.11), $T_{NC}^{rep,rep}(i+1,j)$ denotes the optimal time to execute tasks T_{i+1} to T_j without any intermediate checkpoint, knowing that T_i is checkpointed, and both T_{i+1} and T_j are replicated. If T_{i+1} is not replicated, we use the second line of Equation (5.11), where $T_{NC}^{norep,rep}(i+1,j)$ is the counterpart of $T_{NC}^{rep,rep}(i+1,j)$, except that it assumes that T_{i+1} is not replicated. This information on T_{i+1} (replicated or not) is needed to compute the recovery cost when executing tasks T_{i+1} to T_j and experiencing an error.

Lines 3 and 4 apply to the case where T_i is not replicated, with similar notation as before. In the first four lines, no task between T_{i+1} and T_{j-1} is checkpointed, hence the notation NC for no checkpoint.

If no checkpoint at all is taken before T_j (this corresponds to the case $i = 0$), we use the last two lines of Equation (5.11): we include the cost to read the initial input, which depends whether T_1 is replicated (in line 5) or not (in line 6) of Equation (5.11).

We have a very similar equation to express $T_{opt}^{norep}(j)$ recursively, with intuitive notation:

$$T_{opt}^{norep}(j) = \min_{1 \leq i < j} \left\{ \begin{array}{l} T_{opt}^{rep}(i) + C_i^{rep} + T_{NC}^{rep,norep}(i+1,j), \\ T_{opt}^{rep}(i) + C_i^{rep} + T_{NC}^{norep,norep}(i+1,j), \\ T_{opt}^{norep}(i) + C_i^{norep} + T_{NC}^{rep,norep}(i+1,j), \\ T_{opt}^{norep}(i) + C_i^{norep} + T_{NC}^{norep,norep}(i+1,j), \\ R_1^{Drep} + T_{NC}^{rep,norep}(1,j), \\ R_1^{Dnorep} + T_{NC}^{norep,norep}(1,j) \end{array} \right\} \tag{5.12}$$

To synthesize the notation, we have defined $T_{NC}^{A,B}(i+1,j)$, with $A, B \in \{rep, norep\}$, as the optimal time to execute tasks T_{i+1} to T_j without any intermediate checkpoint, knowing that T_i is checkpointed, T_{i+1} is replicated if and only if $A = rep$, and T_j is

replicated if and only if $B = rep$. In a nutshell, we have to account for the possible replication of the first task T_{i+1} after the last checkpoint, and of the last task T_j, hence the four cases.

There remains to compute $T_{NC}^{A,B}(i,j)$ for all $1 \leq i,j \leq n$ and $A,B \in \{rep, norep\}$. This is still not easy, because there remains to decide which intermediate tasks should be replicated. In addition to the status of T_j (replicated or not, according to the value of B), the only thing we know so far is that the only checkpoint that we can recover from while executing tasks T_i to T_j is the checkpoint taken after task T_{i-1}, hence we need to re-execute from T_i whenever an error strikes. Furthermore, T_i is replicated if and only if $A = rep$, hence we know the corresponding cost for recovery, R_i^A. Letting $T_{NC}^{A,B}(i,j) = 0$ whenever $i > j$, we can express $T_{NC}^{A,B}(i,j)$ for $1 \leq i \leq j \leq n$ as follows:

$$\begin{aligned} T_{NC}^{A,B}(i,j) = \quad & \min\left\{T_{NC}^{A,rep}(i,j-1), T_{NC}^{A,norep}(i,j-1)\right\} \\ & + T^{A,B}(j \mid i). \end{aligned}$$

Here the new (and final) notation $T^{A,B}(j \mid i)$ is simply the time needed to execute task T_j, knowing that an error during T_j implies to recover from T_i. Indeed, to execute tasks T_i to T_j, we account recursively for the time to execute T_i to T_{j-1}; T_{i-1} is still checkpointed; T_i is replicated if and only if $A = rep$, T_j is replicated if and only if $B = rep$, and we consider both cases whether T_{j-1} is replicated or not. The time lost in case of an error during T_j depends whether T_j is replicated or not, and we need to restart from T_i in case of error, hence the notation $T^{A,B}(j \mid i)$, representing the expected execution time for task T_j with or without replication (depending on B), given that we need to restart from T_i if there is an error (and T_i is replicated if and only if $A = rep$).

The last step is hence to express these execution times. We start with the case where T_j is not replicated:

$$\begin{aligned} T^{A,norep}(j \mid i) = & \left(1 - e^{-\lambda_{ind}^F p(T_j^{norep} + V_j^{norep})}\right)\left(T_{lost}^{norep}(T_j^{norep} + V_j^{norep}) + D + R^{D}{}_i^A\right. \\ & + \min\left\{T_{NC}^{A,rep}(i,j-1), T_{NC}^{A,norep}(i,j-1)\right\} + T^{A,norep}(j \mid i)\Big) \\ & + e^{-\lambda_{ind}^F p(T_j^{norep} + V_j^{norep})}\left(T_j^{norep} + V_j^{norep} + \left(1 - e^{-\lambda_{ind}^S p T_j^{norep}}\right)(D + R^{M}{}_i^A\right. \\ & \left. + \min\left\{T_{NC}^{A,rep}(i,j-1), T_{NC}^{A,norep}(i,j-1)\right\} + T^{A,norep}(j \mid i))\right) \end{aligned}$$

.

The term in $e^{-\lambda_{ind}^F p(T_j^{norep} + V_j^{norep})}$ represents the case without fail-stop error, where the execution time is simply $T_j^{norep} + V_j^{norep}$. If a silent error is detected after the verification, we pay a downtime and a memory recovery (with a cost depending on A). Next, we need to re-execute all the tasks since the last checkpoint (T_i to T_{j-1}) and take the minimal value obtained out of the execution where T_{j-1} is replicated or not; finally, we execute T_j again (with a time $T^{A,norep}(j \mid i)$) from last checkpoint. When a fail-stop

error strikes, we account for $T_{lost}^{norep}(T_j^{norep} + V_j^{norep})$, the time lost within T_j, and whose value is given by Equation (5.5). Then we pay a downtime and a disk recovery (with a cost depending on A). Finally, we re-execute all the tasks from last checkpoint and that is similar to the previous case.

The formula is similar with replication of T_j, where the probability of error accounts for the fact that we need to recover only if both replicas fail for the fail-stop errors and accounts for the number of living replicas in the case where a silent error is detected (see Section 5.3.2 for the details):

$$
\begin{aligned}
T^{A,rep}(j \mid i) = & \left(1 - e^{-\frac{\lambda_{ind}^F p(T_j^{rep}+V_j^{rep})}{2}}\right)^2 \Big(T_{lost}^{rep}(T_j^{rep} + V_j^{rep}) + D + R^{D}{}_{i}^{A} \\
& + \min\Big\{T_{NC}^{A,rep}(i, j-1), T_{NC}^{A,norep}(i, j-1)\Big\} + T^{A,rep}(j \mid i)\Big) \\
& + \left(1 - \left(1 - e^{-\frac{\lambda_{ind}^F p(T_j^{rep}+V_j^{rep})}{2}}\right)^2\right)\left(T_j^{rep} + V_j^{rep}\right) \\
& + \Big((1 - e^{-\frac{\lambda_{ind}^F p(T_j^{rep}+V_j^{rep})}{2}})e^{-\frac{\lambda_{ind}^F p(T_j^{rep}+V_j^{rep})}{2}}(1 - e^{-\frac{\lambda_{ind}^S p T_j^{rep}}{2}}) \\
& + e^{-\lambda_{ind}^F p(T_j^{rep}+V_j^{rep})}(1 - e^{-\frac{\lambda_{ind}^S p T_j^{rep}}{2}})^2\Big)\Big(D + R^{M}{}_{i}^{A} \\
& + \min\Big\{T_{NC}^{A,rep}(i, j-1), T_{NC}^{A,norep}(i, j-1)\Big\} + T^{A,rep}(j \mid i)\Big).
\end{aligned}
$$

Note that the value of $T_{lost}^{rep}(T_j^{rep})$ is given by Equation (5.8). Overall, we need to compute the $O(n^2)$ intermediate values $T^{A,B}(j \mid i)$ and $T_{NC}^{A,B}(i,j)$ for $1 \leq i, j \leq n$ and $A, B \in \{rep, norep\}$, and each of these take constant time. There are $O(n)$ values $T_{opt}^{A}(i)$, for $1 \leq i \leq n$ and $A \in \{rep, norep\}$, and these perform a minimum over at most $6n$ elements, hence they can be computed in $O(n)$. The overall complexity is therefore $O(n^2)$. □

5.5 Experiments

In this section, we evaluate the advantages of adding replication to checkpointing in the presence of both, fail-stop and silent errors. We point out that the simulator that implements the proposed DP algorithm is publicly available at `http://graal.ens-lyon.fr/~yrobert/chainsrep.zip` so that interested readers can instantiate their preferred scenarios and repeat the same simulations for reproducibility purpose. The code is written in-house in C++ and does not use any library other than the STL.

We start by assessing scenarios with *fail-stop errors only* in Section 5.5.1. We first describe the evaluation framework in Section 5.5.1.1, then we compare *checkpoint with replication* to *checkpoint only* in Section 5.5.1.2. In Section 5.5.1.3, we assess the impact of the different model parameters on the performance of the optimal strategy. Finally, Section 5.5.1.4 compares the performance of the optimal solution to alternative sub-optimal solutions.

Then we assess scenarios with both fail-stop and silent errors in Section 5.5.2. We first describe the few modifications of the evaluation framework in Section 5.5.2.1, then we compare *checkpoint with replication* to *checkpoint only* in Section 5.5.2.2. Finally, Section 5.5.2.3 assesses the impact of the different model parameters on the performance of the optimal strategy.

5.5.1 Scenarios with fail-stop errors only

5.5.1.1 Experimental setup

We fix the total work in the chain to $W = 10,000$ seconds. The choice of this value is less important than the duration of the tasks compared to the error rate. For this reason, we rely on five different work distributions, where all tasks are fully parallel ($\alpha_i = 0$):

- UNIFORM: every task i is of length $w_i = \frac{W}{n}$, i.e., identical tasks.
- INCREASING: the length of the tasks constantly increases, i.e., task T_i has length $w_i = i\frac{2W}{n(n+1)}$.
- DECREASING: the length of the tasks constantly decreases, i.e., task T_i has length $w_i = (n-i+1)\frac{2W}{n(n+1)}$.
- HIGHLOW: the chain is formed by long tasks followed by short tasks. The long tasks represent 60% of the total work and there are $\lceil \frac{n}{10} \rceil$ such tasks. Short tasks represent the remaining 40% of the total work and consequently there are $n - \lceil \frac{n}{10} \rceil$ small tasks.
- RANDOM: task lengths are uniformly chosen at random between $\frac{W}{2n}$ and $\frac{3W}{2n}$. If the total work of the first i tasks reaches W, the weight of each task is multiplied by $\frac{i}{n}$ so that we can continue adding the remaining tasks.

Experiments with increasing sequential part (α_i) for the tasks are available in the companion research report [22]. Setting $\alpha_i = 0$ amounts to being in the worse possible case for replication, since the tasks will fully benefit of having twice as much processors when not replicated.

For simplicity, we assume that checkpointing costs are equal to the corresponding recovery costs, assuming that read and write operations take approximately the same amount of time, i.e., $R_{i+1}^{Dnorep} = C_i^{norep}$. For replicated tasks, we set $C_i^{rep} = \alpha C_i^{norep}$ and $R_i^{Drep} = \alpha R_i^{Dnorep}$, where $1 \leq \alpha \leq 2$, and we assess the impact of parameter α in Section 5.5.2.3. In the following experiments, we measure the performance of a solution by evaluating the associated normalized expected makespan, i.e., the expected execution time needed to compute all the tasks in the chain, with respect to the execution time without errors, checkpoints, or replicas.

5.5.1.2 Comparison to checkpoint only

We start with an analysis of the solutions obtained by running the optimal dynamic programming (DP) algorithm ChainsRepCkpt on chains of 20 tasks for the five different work distributions described in Section 5.5.1.1. We also run a variant of ChainsRepCkpt that does not perform any replication, hence using a simplified DP algorithm, that is called ChainsCkpt.

We vary the fail-stop error rate $\lambda_{ind}^{F} p$ from 10^{-8} to 10^{-2}. Note that when $\lambda_{ind}^{F} p = 10^{-3}$, we expect an average of 10 errors per execution of the entire chain (neglecting potential errors during checkpoints and recoveries). The checkpoint cost $C_i^{norep} = a_i$ is constant per task (hence $b_i = c_i = 0$) and varies from $10^{-3} T_i^{norep}$ to $10^{3} T_i^{norep}$. For replicated tasks, we set $\alpha = 1$ in this experiment, i.e., $C_i^{rep} = C_i^{norep}$ and $R_i^{Drep} = R_i^{Dnorep}$.

Figure 5.1 presents the results of these experiments for the Uniform distribution. We are interested in the number of checkpoints and replicas in the optimal solution. As the optimal solution may or may not contain checkpoints and replicas, we distinguish 4 cases: *None* means that no task is checkpointed nor replicated, *Checkpointing Only* means that some tasks are checkpointed but no task is replicated, *Replication Only* means that some tasks are replicated, but no task is checkpointed, and *Checkpointing+Replication* means that some tasks are checkpointed and some tasks are replicated. First, we observe that when the checkpointing cost is less than or equal to the length of a task (on the left of the black line), the optimal solution does not use replication, except when the error rate becomes very high. However, if the checkpointing cost exceeds the length of one task (on the right of the black vertical bar), replication proves useful in some cases. In particular, when the fails-stop error rate $\lambda_{ind}^{F} p$ is medium to high (i.e., 10^{-6} to 10^{-4}), we note that only replication is used, meaning that no checkpoint is taken and that replication alone is a better strategy to prevent any error from stopping the application. When the error rate is the highest (i.e., 10^{-4} or higher), replication is added to the checkpointing strategy to ensure maximum reliability. It may seem unusual to use replication alone when checkpointing costs increase. This is because the recovery cost has to be taken into account as well, in addition to re-executing the tasks that have failed. Replication is added to reduce this risk: if successful, there is no recovery cost to pay for, nor any task to re-execute. Finally, note that for low error rates and low checkpointing costs, only checkpoints are used, because their cost is lower than the average re-execution time in case of error. We point out that similar results are obtained when using other work distributions (see the extended version [22]).

In the next experiment, we focus on scenarios where both checkpointing and replication are useful, i.e., we set the checkpointing cost to be twice the length of a task (i.e., $C_i^{norep} = a_i = 2T_i^{norep}$), and we set the fail-stop error rate $\lambda_{ind}^{F} p$ to 10^{-3}, which corresponds to the case highlighted by the red box in Figure 5.1. Figure 5.2 presents the optimal solutions obtained with the ChainsCkpt and ChainsRepCkpt algorithms for the Uniform, Increasing, Decreasing, HighLow and Random work distributions, respectively. First, for the Uniform work distribution, it is clear that the Chains-

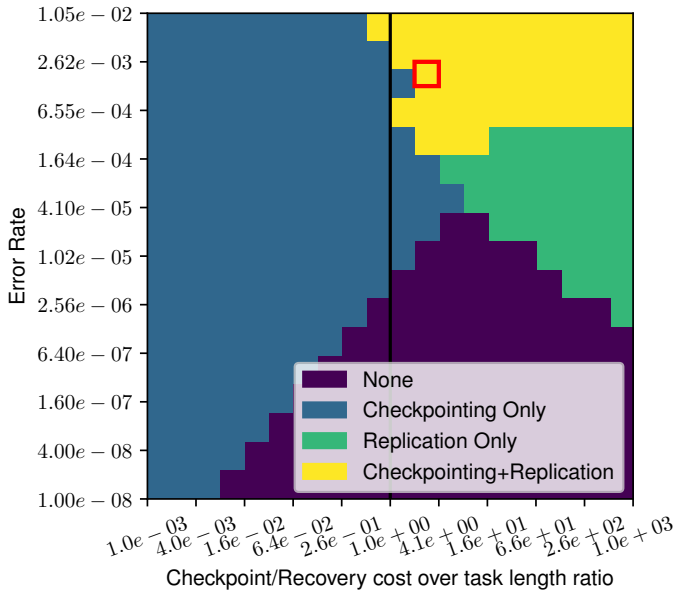
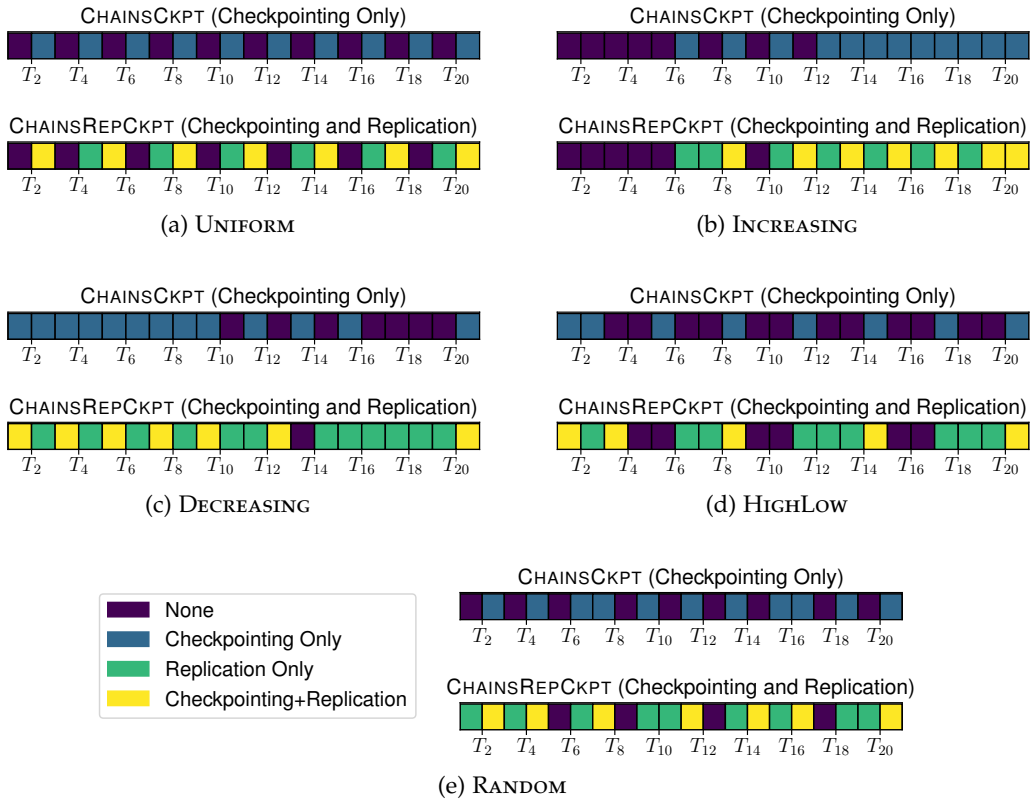


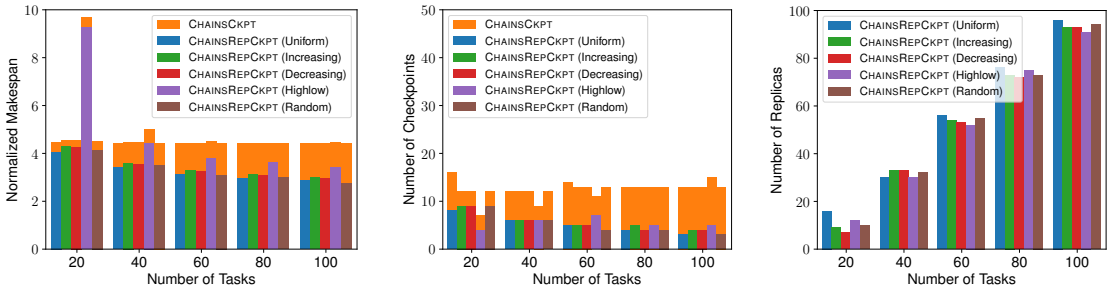
Figure 5.1: Impact of checkpoint/recovery cost and error rate on the usage of checkpointing and replication. Total work is fixed to 10,000s and is distributed uniformly among $n = 20$ tasks (i.e., $T_1 = T_2 = \cdots = T_{20} = 500$s). Each color shows the presence of checkpoints and/or replicas in the optimal solution. Results corresponding to the case highlighted with a red square are presented in Figure 5.2.



(a) Uniform (b) Increasing (c) Decreasing (d) HighLow (e) Random

Figure 5.2: Optimal solutions obtained with the ChainsCkpt algorithm (top) and the ChainsRepCkpt algorithm (bottom) for the five work distributions.

RepCkpt strategy leads to a decrease in the number of checkpoints compared to the ChainsCkpt strategy. Under the ChainsCkpt strategy, a checkpoint is taken every two tasks, while under the ChainsRepCkpt strategy, a checkpoint is instead taken every three tasks, while two out of three tasks are also replicated. Then, for the Increasing and Decreasing work distributions, the results show that most tasks should be replicated, while only the longest tasks are also checkpointed. A general rule of thumb is that *replication only* is preferred for short tasks while *checkpointing and replication* is reserved for longer tasks, where the probability of error and the re-execution cost are the highest. Finally, we observe a similar trend for the HighLow work distribution, where two of the first four longer tasks are checkpointed and replicated.



Uniform, Increasing, Decreasing, HighLow, Random distributions

Figure 5.3: Comparison of the ChainsCkpt and ChainsRepCkpt strategies for different numbers of tasks: impact on the makespan (left), number of checkpoints (middle) and number of replicas (right) with a fail-stop error rate of $\lambda^F_{ind} p = 10^{-3}$ and a constant chekpointing/recovery cost $C_i^{norep} = C_i^{rep} = 1000$s.

Figure 5.3 compares the performance of ChainsRepCkpt to the *checkpoint-only* strategy ChainsCkpt. First, we observe that the expected normalized makespan of ChainsCkpt remains almost constant at ≈ 4.5 for any number of tasks and for any work distribution. Indeed, in our scenario, checkpoints are expensive and the number of checkpoints that can be used is limited to ≈ 17 in the optimal solution, as shown in the middle plot. However, the ChainsRepCkpt strategy can take advantage of the increasing number of shorter tasks by replicating them. In this scenario (high error rate and high checkpoint cost), this is clearly a winning strategy. The normalized expected makespan decreases with increasing n, as the corresponding number of tasks that are replicated increases almost linearly. The ChainsRepCkpt strategy reaches a normalized makespan of ≈ 2.6 for $n = 100$, i.e., a reduction of 35% compared to the normalized expected makespan of the ChainsCkpt strategy. This is because replicated tasks tend to decrease the global probability of having an error, thus reducing even more the number of checkpoints needed as seen previously. Regarding the HighLow work distribution, we observe a higher optimal expected makespan for both the ChainsCkpt and the ChainsRepCkpt strategies. Indeed, in this scenario, the first tasks are very long (60% of the total work), which greatly increases the probability of error and the associated re-execution cost.

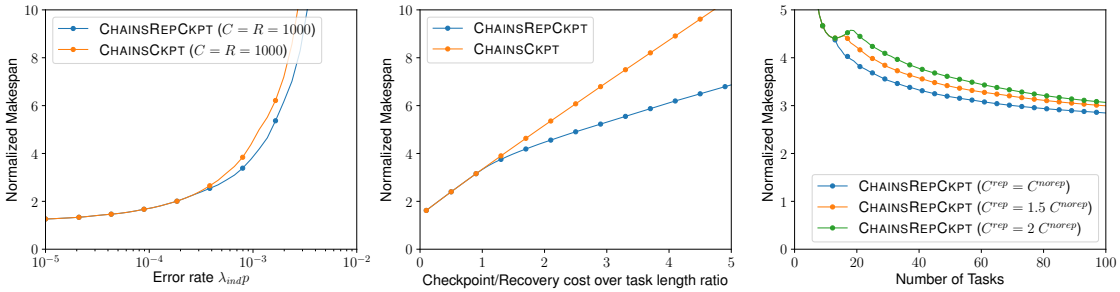
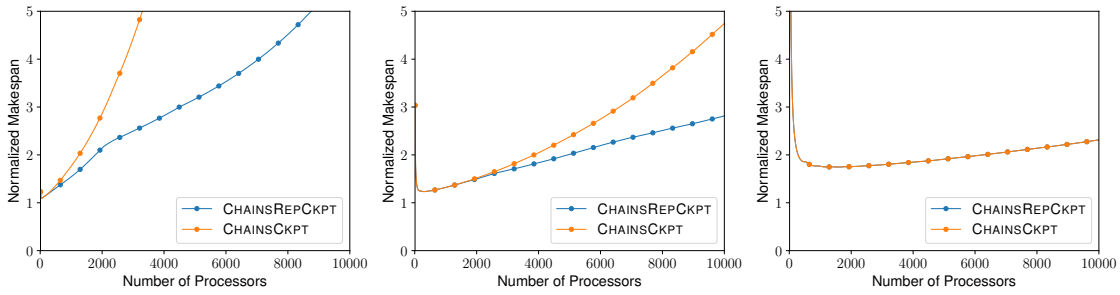


Figure 5.4: Impact of fail-stop error rate $\lambda_{ind}^{F}p$ (left), checkpoint cost (middle), and ratio α between the checkpointing cost for replicated task C_i^{rep} over non-replicated tasks C_i^{norep} (right).



(a) $(100, 10000, 1)$ (b) $(100, 100000, 0.1)$ (c) $(100, 1000000, 0.01)$

Figure 5.5: Comparison of the ChainsCkpt and ChainsRepCkpt strategies for different numbers of processors, with different model parameter values for the checkpointing cost (a_i, b_i, c_i).

5.5.1.3 Impact of error rate and checkpoint cost on the performance

Figure 5.4 shows the impact of three of the model parameters on the optimal expected normalized makespan of both ChainsCkpt and ChainsRepCkpt. First, we show the impact of the fail-stop error rate $\lambda_{ind}^{F}p$ on the performance. The ChainsRepCkpt algorithm improves the ChainsCkpt strategy for large values of $\lambda_{ind}^{F}p$: replication starts to be used for $\lambda_{ind}^{F}p > 2.6 \times 10^{-4}$ and it reduces the makespan by $\approx 16\%$ for $\lambda_{ind}^{F}p = 10^{-3}$ and by up to $\approx 40\%$ when $\lambda_{ind}^{F}p = 10^{-2}$, where all tasks are checkpointed and replicated.

Then, we investigate the impact of the checkpointing cost with respect to the task length. As shown in Figure 5.1, replication is not needed for low checkpointing costs, i.e., when the checkpointing cost is between 0 and 0.8 times the cost of one task: in this scenario, all tasks are checkpointed and both strategies lead to the same makespan. When the checkpointing cost is between 0.9 and 1.6 times the cost of one task, ChainsRepCkpt checkpoints and replicates half of the tasks. Overall, the ChainsRepCkpt strategy improves the optimal normalized expected makespan by $\approx 11\%$ for a checkpointing cost ratio of 1.6, and by as much as $\approx 36\%$ when the checkpointing cost is

five times the length of one task.

We now investigate the impact of the ratio between the checkpointing and recovery cost for replicated tasks and non-replicated tasks α and we present the results for $\alpha = 1$ ($C_i^{rep} = R_i^{Drep} = C_i^{norep} = R_i^{Dnorep}$), $\alpha = 1.5$ ($C_i^{rep} = R_i^{Drep} = 1.5C_i^{norep} = 1.5R_i^{Dnorep}$) and $\alpha = 2$ ($C_i^{rep} = R_i^{Drep} = 2C_i^{norep} = 2R_i^{Dnorep}$). As expected, the makespan increases with α, but it is interesting to note that the makespan converges towards a same lower-bound as the number of (shorter) tasks increases. As shown previously, when tasks are smaller, ChainsRepCkpt favors replication over checkpointing, especially when the checkpointing cost is high, which means less checkpoints, recoveries and re-executions.

Finally, we evaluate the efficiency of both strategies when the number of processors increases. For this experiment, we instantiate the model using variable checkpointing costs, i.e., we do not use $b_i = c_i = 0$ anymore, so that the checkpointing/recovery cost depends on the number of processors. We set $n = 50$, $\lambda_{ind}^F = 10^{-7}$ and we make p vary from 10 to 10,000 (i.e., the global error rate varies between 10^{-6} and 10^{-3}). Figure 5.5 presents the results of the experiment using three different sets of values for a_i, b_i and c_i. We see that when b_i increases while c_i decreases, the replication becomes useless, even for the larger error rate values. However, when the term $c_i p$ becomes large in front of $\frac{b_i}{p}$, we see that ChainsRepCkpt is much better than ChainsCkpt, as the checkpointing costs tend to decrease, in addition to all the other advantages investigated in the previous sections. With $p = 10,000$, the three different experiments show an improvement of 80.5%, 40.7% and 0% (from left to right, respectively).

5.5.1.4 Impact of the number of checkpoints and replicas

Figure 5.6 shows the impact of the number of checkpoints and replicas on the normalized expected makespan for different checkpointing costs and fail-stop error rates $\lambda_{ind}^F p$ under the Uniform work distribution. We show that the optimal solution with ChainsRepCkpt (highlighted by the green box) always matches the minimum value obtained in the simulations, i.e., the optimal number of checkpoints, number of replicas, and expected execution times are consistent. In addition, we show that in scenarios where both the checkpointing cost and the error rate are high, even a small deviation from the optimal solution can quickly lead to a large overhead.

5.5.2 Scenarios with both fail-stop and silent errors

In this section we evaluate the power of replication in addition to checkpointing on platforms subject to both fail-stop and silent errors.

5.5.2.1 Experimental setup

All the model parameters are instantiated as before, with the following changes to account for the presence of silent errors. Unless stated otherwise, the fail-stop error

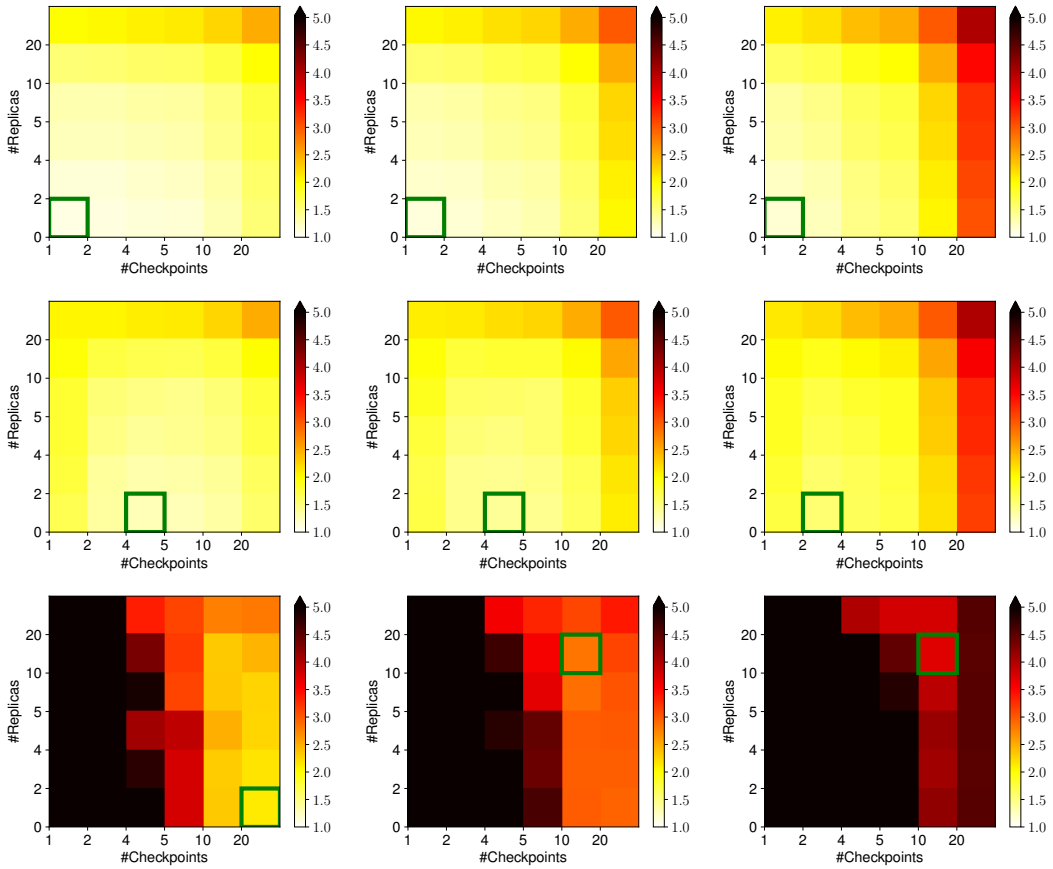


Figure 5.6: Impact of the number of checkpoints and replicas on the normalized expected makespan for fail-stop error rates of $\lambda^F_{ind} = 10^{-4}$ (top), $\lambda = 10^{-3}$ (middle) and $\lambda = 10^{-2}$ (bottom) and for checkpointing costs of $0.5 \times T_i^{norep}$ (left), $1 \times T_i^{norep}$ (middle) and $2 \times T_i^{norep}$ (right), with $C_i^{norep} = C_i^{rep}$ under UNIFORM work distribution. The optimal solution obtained with CHAINSREPCKPT always matches the minimum simulation value and is highlighted by the green box.

rate has been set to 1.28e-3s^{-1} and the silent error rate has been set to 5.48e-3s^{-1}. The silent error rate has been computed from real measures [18]: we derived a non-corrected silent error rate per core of 5.48e-9s^{-1}. Similarly, the fail-stop error rate per core considered was 1.28e-9s^{-1}, which corresponds to a core lifetime of 25 years. Finally, we considered a platform of 1 million cores which tends to be the trend for current Top500 machines [181].

As for other parameters, we considered a verification cost of 1% of the corresponding task length. The cost of memory recovery was set to 5% of that of a disk recovery, considering an average between different measured values from [140].

For simplicity, we assume that checkpointing costs are equal to the sum of the

corresponding recovery costs, assuming that read and write operations take approximately the same amount of time, i.e., $R_{i+1}^{Dnorep} + R_{i+1}^{Mnorep} = C_i^{norep}$. For replicated tasks, we set $C_i^{rep} = \alpha C_i^{norep}$, $R_i^{Drep} = \alpha R_i^{Dnorep}$ and $R_i^{Mrep} = \alpha R_i^{Mnorep}$, where $1 \leq \alpha \leq 2$, and we assess the impact of parameter α in Section 5.5.2.3. As in the previous section, we measure the performance of a solution by evaluating the associated normalized expected makespan, i.e., the expected execution time needed to compute all the tasks in the chain, with respect to the execution time without errors, checkpoints, or replicas.

5.5.2.2 Comparison to checkpoint only

We start with an analysis of the solutions obtained by running the optimal dynamic programming (DP) algorithm ChainsRepCkpt on chains with 20 tasks for the five different work distributions described in Section 5.5.1.1. We also run a variant of ChainsRepCkpt that does not perform any replication, hence using a simplified DP algorithm, that is called ChainsCkpt.

We vary the fail-stop error rate $\lambda_{ind}^F p$ from 10^{-8} to 10^{-2}, without changing the silent error rate λ_{ind}^S. The disk checkpoint/recovery cost is constant per task and varies from $10^{-3} T_i^{norep}$ to $10^3 T_i^{norep}$ (hence, the memory checkpoint/recovery cost varies from $5 \times 10^{-5} T_i^{norep}$ to $50 T_i^{norep}$). Overall, all checkpoints have a cost from $1.05 \times 10^{-3} T_i^{norep}$ to $1.05 \times 10^3 T_i^{norep}$ as we always perform both types of checkpoints. For replicated tasks, we set $\alpha = 1$ in this experiment, i.e., $C_i^{rep} = C_i^{norep}$, $R_i^{Drep} = R_i^{Dnorep}$ and $R_i^{Mrep} = R_i^{Mnorep}$. In another experiment, we also make the silent error rate $\lambda_{ind}^S p$ vary from 10^{-8} to 10^{-2} without changing the fail-stop error rate of 1.28e-3, with the same range for the checkpoint cost.

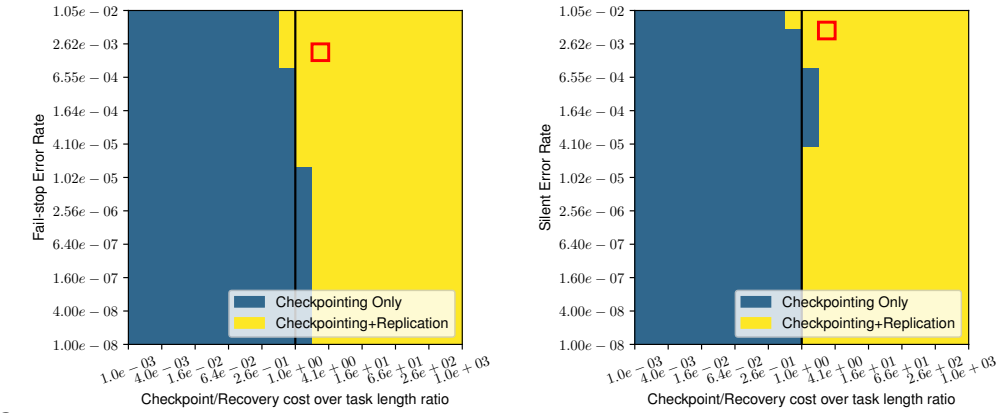


Figure 5.7: Impact of checkpoint/recovery cost and error rates on the usage of checkpointing and replication. Total sequential work is fixed to 10,000s and is distributed uniformly among $n = 20$ tasks (i.e., $T_1 = T_2 = \cdots = T_{20} = 500$s). Each color shows the presence of checkpoints and/or replicas in the optimal solution.

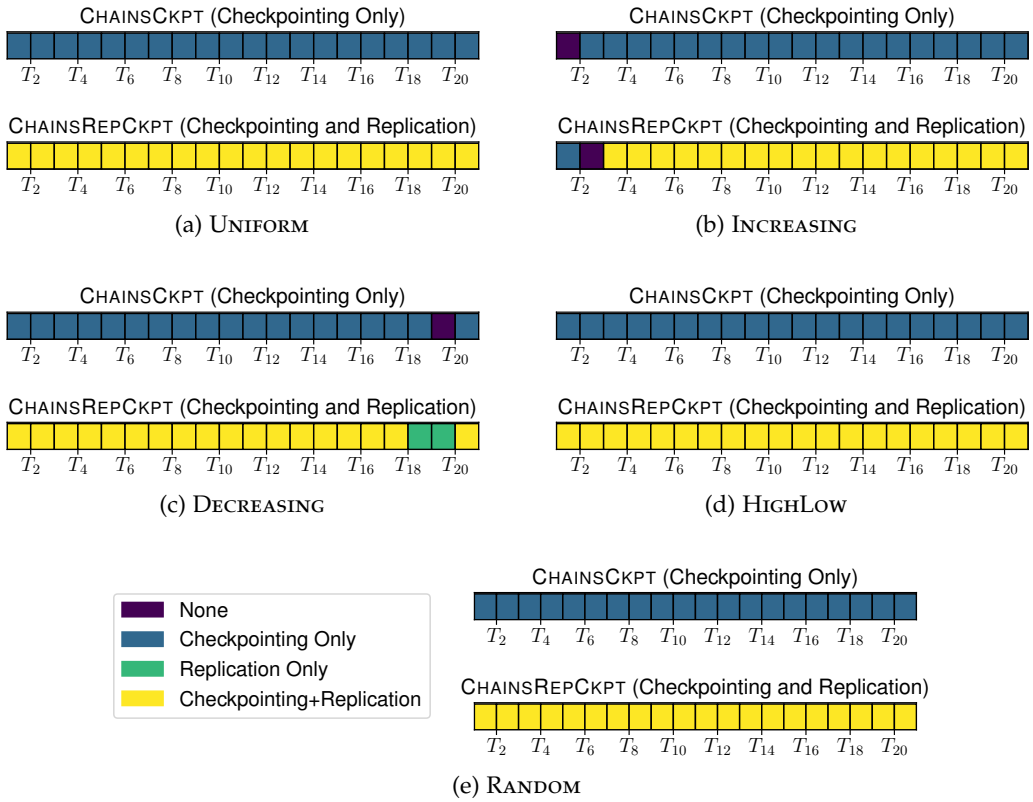
Figure 5.7 presents the results of these experiments for the Uniform distribution.

The colors are the same as in Figure 5.1, with *Checkpointing Only* meaning that some tasks are checkpointed but no task is replicated and *Checkpointing+Replication* meaning that some tasks are checkpointed and some tasks are replicated. The left figure presents the results when the silent error rate is fixed but the fail-stop error rate varies. The right figure presents the results of the other experiment with a fixed fail-stop error rate and different silent error rates.

First, we observe that with silent errors, checkpointing becomes mandatory. Too many failures can strike during the execution, and checkpointing helps reducing the time spent on rollbacks and re-executions. However, as soon as the cost of a checkpoint exceeds the length of a task, replication becomes useful and this remains true even for low error rates. This holds for both fail-stop errors (left) and silent errors (right). There is one exception: when the fail-stop error rate is lower than 10^{-5} and the checkpointing cost is less than twice the length of a task, checkpoints are sufficient and is replication is not needed. Replication is overall not needed under good conditions, however for our real setup, indicated by the red box, using both checkpointing and replication is a better solution. We point out that similar results are obtained when using other work distributions (see the extended version [22]).

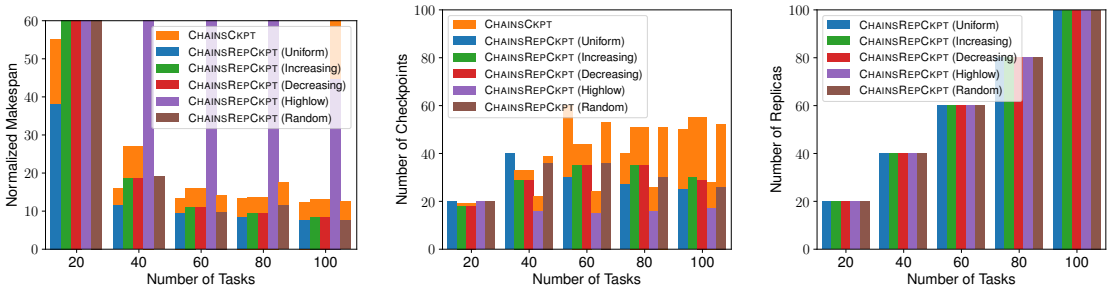
In the next experiment, we focus on scenarios where both checkpointing and replication are useful, i.e., we set the checkpointing cost to be twice the length of a task (i.e., $C_i^{norep} = a_i = 2T_i^{norep}$), keeping $\lambda_{ind}^F p = 1.28\text{e-}3$ and $\lambda_{ind}^S p = 5.48\text{e-}3$, for the fail-stop and silent error rates, respectively, which corresponds to the case highlighted by the red box in Figure 5.7. Figure 5.8 presents the optimal solutions obtained with the ChainsCkpt and ChainsRepCkpt algorithms for the Uniform, Increasing, Decreasing, HighLow and Random work distributions, respectively. With two sources of errors, the solution is straightforward: almost every task must be checkpointed, with the exception of one (short) task for the Decreasing and Increasing distributions. However almost every task is also replicated (20 tasks out of 20 for the Uniform distribution compared to only 13 in the experiments of Section 5.5.1.2), showing once more that replication grants better protection to failures even if it increases the failure-free execution time. Checkpoints are being taken the same way as in our previous experiments: long tasks are systematically checkpointed while shorter tasks are either unprotected or replicated, as can be seen with the first tasks of the Increasing distribution and the last task of the Decreasing distributions.

Figure 5.9 compares the performance of ChainsRepCkpt to the *checkpoint-only* strategy ChainsCkpt with fail-stop and silent errors. First, we observe that long tasks, being more likely to fail than shorter tasks, introduce a high overhead. As a consequence, with 20 tasks, the normalized makespan is too high and the execution of such applications is not possible, independently of the work distribution and the chosen checkpointing strategy. With more tasks however, the ChainsRepCkpt strategy always yield a shorter makespan compared to the ChainsCkpt strategy. For example, with 100 tasks, the normalized makespan obtained with the ChainsRepCkpt strategy is as high as ≈ 8.5 (and much more for the HighLow distribution), compared to ≈ 13 for ChainsCkpt. Indeed, with such high error rates, all tasks are replicated under



(a) Uniform (b) Increasing (c) Decreasing (d) HighLow (e) Random

Figure 5.8: Optimal solutions obtained with the ChainsCkpt algorithm (top) and the ChainsRepCkpt algorithm (bottom) for the five work distributions.



Uniform, Increasing, Decreasing, HighLow, Random distributions

Figure 5.9: Comparison of the ChainsCkpt and ChainsRepCkpt strategies for different numbers of tasks: impact on the makespan (left), number of checkpoints (middle) and number of replicas (right) with a fail-stop error rate of $\lambda^F_{ind}p$ = 1.28e-3, a silent error rate of $\lambda^S_{ind}p$ = 5.48e-3 and a constant chekpointing/recovery cost $C_i^{norep} = C_i^{rep} = 1000$s.

the ChainsRepCkpt strategy, as can be seen on the right plot, but fewer tasks need to be be checkpointed (up to 50% fewer checkpoints with 100 tasks and the Uniform distribution).

The improvement is comparable to the 35% improvement observed with only fail-stop errors. Once again, replicated tasks tend to decrease the global probability of having an error, thus slightly reducing the number of checkpoints needed, while reducing the re-execution costs that can be very important with late-detected silent errors. Regarding the HighLow work distribution, we again observe a higher optimal expected makespan for both the ChainsCkpt and the ChainsRepCkpt strategies. Indeed, in this scenario, the first tasks are very long (60% of the total work), which greatly increases the error probability and the associated re-execution cost. Overall, for such applications on platforms subject to both, fail-stop and silent errors, replication appears to be mandatory and allows a reduction of the makespan of at least 30% if tasks are not too large (i.e. the probability of completing the task is not close to 1).

5.5.2.3 Impact of error rate and checkpoint cost on the performance

Figure 5.10 shows the impact of four of the model parameters on the optimal expected normalized makespan of both ChainsCkpt and ChainsRepCkpt, using the Uniform distribution. First, we show the impact of the fail-stop error rate $\lambda^F_{ind}p$ on the performance. The ChainsRepCkpt strategy always yields shorter makespans compared to the ChainsCkpt strategy. All tasks are always replicated, reducing the probability of having an error for each task, and each task is also checkpointed. The normalized makespan for ChainsCkpt is 19.5 for $\lambda^F_{ind}p = 10^{-5}$, compared to 19.2 for ChainsRepCkpt, i.e. a reduction of only 1.7%, but this goes up to 50.3 for $\lambda^F_{ind}p$ = 1.14e-3 compared to 35.2 when using replication, i.e. a reduction of 30%. The results are similar when we vary the silent error rate: when $\lambda^S_{ind}p = 10^{-5}$, ChainsRepCkpt results in

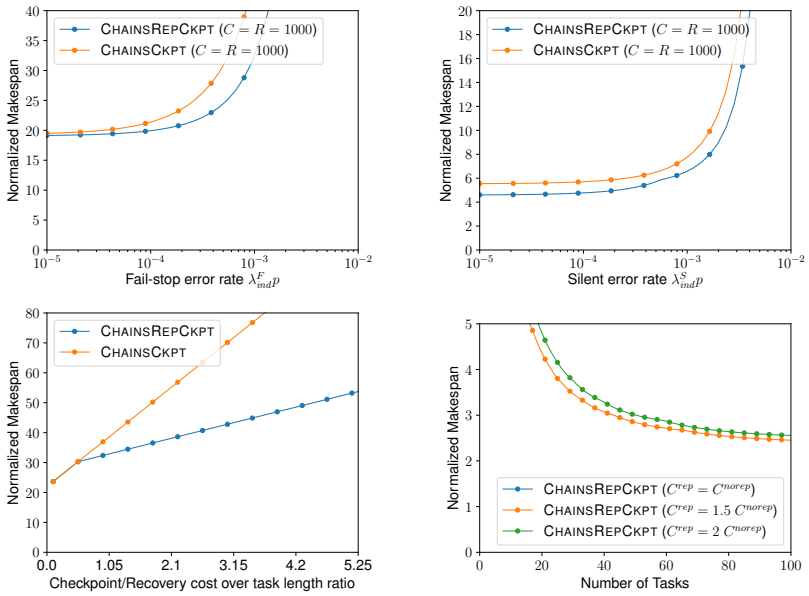
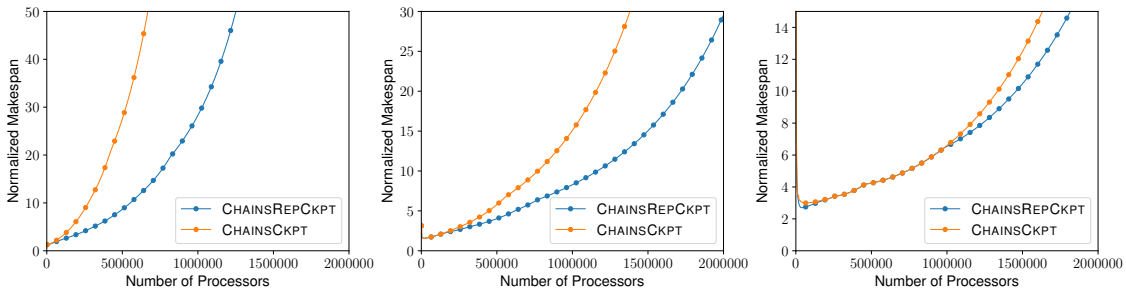


Figure 5.10: Impact of fail-stop error rate $\lambda^F_{ind}p$ (top left), silent error rate $\lambda^S_{ind}p$ (top right), checkpoint cost (bottom left) and ratio α between the checkpointing cost for replicated task C_i^{rep} over non-replicated tasks C_i^{norep} (bottom right) for the Uniform distribution.



(a) $(105, 1050000, 0.0105)$ (b) $(105, 10500000, 0.00105)$ (c) $(105, 105000000, 0.000105)$

Figure 5.11: Comparison of the ChainsCkpt and ChainsRepCkpt strategies for different numbers of processors, with different model parameter values for the checkpointing cost (a_i, b_i, c_i).

a normalized makespan of 4.60 compared to 5.55 with ChainsCkpt, i.e. a reduction of 17%, and this goes up to more than 30% when $\lambda^S_{ind} p > 5 \times 10^{-3}$.

Then, we investigate the impact of the checkpointing cost with respect to the task length. The results are slightly different now that we have silent errors: ChainsCkpt and ChainsRepCkpt behave similarly only for small values of checkpoint cost. ChainsRepCkpt becomes better than ChainsCkpt for $C \geq 0.525$, thus reducing the makespan obtained using only checkpoints. Both strategies yield a makespan that increases linearly with the checkpointing cost, however the ChainsRepCkpt strategy needs less checkpoints, and the makespan increases slower. This means that the costlier the checkpoints the better the improvement thanks to replication. Overall, the execution under the ChainsRepCkpt strategy is 1.17 times faster than ChainsCkpt for a checkpointing cost of $1.05T_i^{norep}$, 1.66 times faster for a checkpoint cost of $3.15T_i^{norep}$, and this goes up to 1.95 times faster when the checkpointing cost is $5.25T_i^{norep}$.

We now investigate the impact of the ratio α between the checkpointing and recovery cost for replicated tasks and non-replicated tasks and we present the results for $\alpha = 1$, $\alpha = 1.5$ and $\alpha = 2$. As expected, the makespan increases with α, but it is interesting to note that the makespan converges towards a same lower-bound as the number of (shorter) tasks increases. As shown previously, when tasks are smaller, ChainsRepCkpt favors replication over checkpointing, especially when the checkpointing cost is high, which means fewer checkpoints, recoveries and re-executions.

Finally, we evaluate the efficiency of both strategies when the number of processors increases. For this experiment, we instantiate the model using variable checkpointing costs, i.e., we do not use $b_i = c_i = 0$ anymore, so that the checkpointing/recovery cost depends on the number of processors. We set $n = 50$, $\lambda^F_{ind} = 1.28 \times 10^{-9}$, $\lambda^S_{ind} = 5.48 \times 10^{-9}$ and we make p vary from 1000 to 2,000,000 (i.e., the error rates vary between 10^{-6} and 10^{-2} approximately). Figure 5.11 presents the results of the experiment using three different sets of values for a_i, b_i and c_i. The trend is the same as previously with fail-stop errors: when b_i increases and c_i decreases, the advantage of using replication becomes less clear. However, on every plot, ChainsCkpt and ChainsRepCkpt grants the same makespan only when using a few cores. Every plot shows that, with the increasing number of cores on nowadays platforms, ChainsRepCkpt will behave better and better compared to ChainsCkpt. In particular, the improvement for each set of parameters (from left to right) is 69%, 30% and 0% for $p = 500000$, and is 76%, 60% and 16% for $p = 1500000$.

5.6 Related work

In this section, we discuss the work related to replication. For related work on checkpointing, refer to Section 2.4. Each of these mechanisms has been studied for coping with fail-stop errors and/or with silent errors. The present work combines checkpointing and replication for linear workflows in the presence of fail-stop and silent errors.

As mentioned earlier, this work only considers *task duplication*. *Triplication* [136]

(three replicas per task) is also possible yet only useful with extremely high error rates, which are unlikely in HPC systems. The use of redundant MPI processes is analyzed in [76, 82, 46]. In particular, Ferreira et al. [82] studied the use of process replication for MPI applications, using two replicas per MPI process. They provide a theoretical analysis of parallel efficiency, an MPI implementation that supports transparent process replication (including error detection, consistent message ordering among replicas, etc.), and a set of experimental and simulation results. Thread-level replication has also been investigated [155, 56, 201]. The present work targets selective task replication as opposed to full task replication in conjunction with selective task checkpointing to cope with fail-stop and silent errors and minimize makespan.

Partial redundancy was also studied (in combination with coordinated checkpointing) to decrease the overhead of full replication [72, 172, 174]. Recently, Hussain et al. [110] have demonstrated the usefulness of partial redundancy for platforms where individual node failure distributions are not identical. They numerically determine the optimal partial replication degree. Adaptive redundancy is introduced in [91], where a subset of processes is dynamically selected for replication. Earlier work [21] considered replication in the context of divisible load applications. In the present work, task replication (including work and data) is studied in the context of linear workflows, which represent a harder case than that of divisible load applications as tasks cannot arbitrarily be divided and are executed non-preemptively.

In contrast to fail-stop errors whose detection is immediate, *silent errors* are identified only when the corrupted data leads to an unusual application behavior, and several works use replication to detect and/or correct silent errors. For instance, thread-level replication has been investigated in [155, 56, 201], which target process-level replication in order to detect (and correct) silent errors striking in all communication-related operations. Ni et al. [145] introduce process duplication to cope both with fail-stop and silent errors. Their pioneering paper contains many interesting results. It differs from this work in that they only consider perfectly parallel applications while we investigate herein per task speedup profiles that obey Amdahl's law. Recently, Subasi et al. [173] proposed a software-based selective replication of task-parallel applications with both, fail-stop and silent errors. In contrast, the present work (i) considers dependent tasks such as found in applications consisting of linear workflows; and (ii) proposes an optimal dynamic programming algorithm to solve the combined selective replication *and* checkpointing problem. More recently, Benoit et al. [21] extended these work to general applications, and compare traditional process replication with *group replication*, where the whole application is replicated as a black box. They analyze several scenarios with duplication or triplication. Combining replication with checkpointing has also been proposed in [159, 75, 205] for HPC platforms, and in [127, 198] for grid computing.

5.7 Conclusion

In this chapter, we studied the combination of checkpointing and replication to minimize the execution time of linear workflows in environments prone to both fail-stop and silent errors. We introduced a sophisticated dynamic programming algorithm that solves the combined problem optimally, by determining which tasks to checkpoint and which tasks to replicate, in order to minimize the total execution time. This dynamic programming algorithm was validated through extensive simulations that reveal the conditions in which checkpointing, replication, or both lead to improved performance. We have observed that the gain over the checkpoint-only approach is quite significant, in particular when checkpointing is costly and error rates are high.

Future work will address workflows whose dependence graphs are more complex than linear chains of tasks. Although an optimal solution seems hard to reach, the design of efficient heuristics that decide where to locate checkpoints and when to use replication, would prove highly beneficial for the efficient and reliable execution of HPC applications on current and future large-scale platforms.

Chapter 6

Optimal Checkpointing Period with Replicated Execution on Heterogeneous Platforms

In this chapter, we design and analyze strategies to replicate the execution of an application on two different platforms subject to failures, using checkpointing on a shared stable storage. We derive the optimal pattern size W for a periodic checkpointing strategy where both platforms concurrently try and execute W units of work before checkpointing. The first platform that completes its pattern takes a checkpoint, and the other platform interrupts its execution to synchronize from that checkpoint. We compare this strategy to a simpler on-failure checkpointing strategy, where a checkpoint is taken by one platform only whenever the other platform encounters a failure. We use first or second-order approximations to compute overheads and optimal pattern sizes, and show through extensive simulations that these models are very accurate. The simulations show the usefulness of a secondary platform to reduce execution time, even when the platforms have relatively different speeds: in average, over a wide range of scenarios, the overhead is reduced by 30%. The simulations also demonstrate that the periodic checkpointing strategy is globally more efficient, unless platform speeds are quite close. The work in this chapter is joint work with Anne Benoit, Aurélien Cavelan and Yves Robert, and has been published in the workshop on *Fault Tolerance for HPC at eXtreme Scale* (FTXS) [W3].

6.1 Introduction

One of the most important challenges faced by large-scale computing systems is the frequent occurence of failures (a.k.a. fails-top errors) [42, 169]. Platform sizes have become so large that failures are likely to strike during the execution of an application. Consider the mean time between failures μ (usually denoted as MTBF) of a platform with p processors: μ decreases linearly with p, since $\mu = \frac{\mu_{\text{ind}}}{p}$, where μ_{ind} is the MTBF of each individual component (see Proposition 1.2 in [106]). For instance, with $\mu_{\text{ind}} =$ 10 years and $p = 10^5$, we have $\mu \approx 50$ minutes, and it goes down to a failure every 5 minutes for $p = 10^6$.

The classical technique to deal with failures is to use a checkpoint-restart mechanism: the state of the application is periodically checkpointed on stable storage, and when a failure occurs, we can recover from the last valid checkpoint and resume the execution, rather than starting again from scratch. Checkpointing policies have been widely studied, see [106] for a survey of various protocols and the derivation of the Young/Daly formula [200, 58] for the optimal checkpointing period. Recent advances include multi-level approaches, or the use of SSD or NVRAM as secondary storage [42].

Another technique that has been advocated to deal with failures is *process replication*, where each process in a parallel MPI (Message Passing Interface) application is duplicated to increase the mean-time to interruption. More precisely, each processor of the platform is paired with a replica so that the execution can continue whenever one of them is struck by a failure. Given the high rate of failures expected in current systems, process replication is usually combined with a periodic checkpointing mechanism, as proposed in [159, 75, 205] for HPC platforms, and in [127, 198] for grid computing. These approaches use *process replication*: each processor of the platform is paired with a replica so that the execution can continue whenever one is struck by a failure.

Another approach introduced in [45] is *group replication*, a technique that can be used whenever process replication is not available. Group replication is agnostic to the parallel programming model, and thus views the application as an unmodified black box. Group replication consists in executing multiple application instances concurrently. For example, two distinct p-process application instances could be executed on a $2p$-processor platform. Once an instance saves a checkpoint, the other instance can use this checkpoint immediately to "jump ahead" in its execution. Hence, group replication is more efficient than the mere independent execution of two instances: each time one instance successfully completes a given "chunk of work", the other instance immediately benefits from this success.

In this work, we extend group replication to the case of two different computing platforms executing concurrently and cooperating to the success of a given application. To the best of our knowledge, this scenario has not been explored yet. The two platforms share a set of remote disks, used as a stable storage for checkpointing. Typically, these platforms would be clusters, which may have different number of processors, and hence different MTBFs and execution speeds. Our goal is to determine the best way to have both platforms cooperate so that the execution time of the application is minimized. We design and analyze two strategies:

1. A periodic checkpointing strategy, where both platforms checkpoint periodically once they have executed a chunk of work of size W. Both platforms synchronize through the shared storage as soon as one of them has completed the execution of a chunk (at the time of the checkpoint). We provide a thorough analysis to express the overhead given the checkpointing period W, and we derive the size of the optimal pattern.

2. An on-failure checkpointing strategy, where each platform progresses at its own

speed, and checkpoints only when a failure occurs on the other platform. Hence, when a failure occurs on one of the platforms (say platform A), the other one (platform B) checkpoints, and platform A gets a copy of this checkpoint to restart its execution at this point. Intuitively, if both platforms have the same speed, we will never roll back with this strategy, unless a failure occurs during checkpoint. We compare both strategies through extensive simulations, and show the gain (or the absence thereof) compared to using a single platform. We also assess the accuracy of the model and of our first or second-order approximations.

The rest of the chapter is organized as follows. We introduce the execution model in Section 6.2, and derive the optimal pattern for the periodic checkpointing strategy in Section 6.3. The analysis for the checkpoint-on-failure strategy is given in Section 6.4. Section 6.5 is devoted to the experimental evaluation. Finally, we provide concluding remarks and directions for future work in Section 6.6.

6.2 Model

We consider a black-box application and replicate its execution on two different computing platforms P_1 and P_2. The platforms may well be heterogeneous, with different processor numbers, different MTBF values and different execution speeds. Both platforms use the same stable storage system. A typical instance is the case of two clusters that share a set of storage disks. We assume that both executions can synchronize through checkpointing. Checkpoint time is C on either platform, and this includes the time to update the state of the application on the other platform. We make no further hypothesis: The checkpointing protocol can be single-level or multi-level, and the update of the application state on the other platform can take place either through the network or via the file system. Recovery time is R, independently of which platform has taken the last checkpoint.

We partition the execution of the application into *periodic patterns*, i.e., computational units that repeat over time. Each pattern includes W units of work (we also say a chunk of size W) and ends with a checkpoint. With a single platform, the optimal pattern length is well-known and obeys the Young/Daly formula [200, 58]. With two platforms executing concurrently, both platforms execute the pattern concurrently, and repeat until success. Once a platform succeeds, the other one stops executing and synchronizes on checkpoint. Computing the optimal pattern length turns out a challenging problem in this case.

We assume that failures independently strike the platforms with an Exponential distribution. Platform P_1 has failure rate λ_1, which means its MTBF (Mean Time Between Failures) is $\mu_1 = \frac{1}{\lambda_1}$. Similarly, P_2 has failure rate λ_2, and MTBF $\mu_2 = \frac{1}{\lambda_2}$. We let σ_1 be the execution speed of the application on platform P_1, and σ_2 be the speed on P_2. We assume that P_1 is the fast platform, so that $\sigma_1 \geq \sigma_2$.

The expected execution time of the pattern is $\mathbb{E}(P)$: we have to take expectations, as the computation time is not deterministic. Letting $T_1 = \frac{W}{\sigma_1}$, we note that $\mathbb{E}(P) > T_1 + C$, the failure-free time on the fast platform. An optimal pattern is defined as

the one minimizing the ratio $\frac{\mathbb{E}(P)}{T_1}$, or equivalently the ratio $\mathbb{H}(P) = \frac{\mathbb{E}(P)}{T_1} - 1$. This latter ratio is the relative overhead paid for executing the pattern. The smaller this overhead, the faster the progress of the execution. For the theoretical analysis, we assume that checkpoint and recovery are failure-free, because this assumption does not modify the dominant terms of the overhead (see Section 6.3.3 for details), but for the simulations, we do account for failures striking anytime. Finally, to be able to write Taylor expansions, we also let λ be the global failure rate and write $\lambda_1 = \alpha_1\lambda$ and $\lambda_2 = \alpha_2\lambda$, with $\alpha_1 + \alpha_2 = 1$.

6.3 Optimal pattern

In this section, we show how to derive the optimal pattern length. The derivation is quite technical, and the reader may want to skip the proofs.

6.3.1 Expected execution time

Consider a pattern P of size W, and let $\mathbb{E}(P)$ denote the expected execution time of the pattern. Because we assume that checkpoints are failure-free, we have $\mathbb{E}(P) = \mathbb{E}(W) + C$, where $\mathbb{E}(W)$ is the expected time to execute a chunk of size W.

We start with some background on well-known results on $\mathbb{E}(W)$ with a single platform P_1, before moving on to our problem with two platforms. With a single platform P_1, let $T_1 = \frac{W}{\sigma_1}$ and $p_1 = 1 - e^{-\lambda_1 T_1}$ be the probability of a failure on P_1 while attempting to execute the chunk of size W. We can write

$$\mathbb{E}(W) = (1 - p_1)T_1 + p_1(\mathbb{E}^{\text{lost}} + R + \mathbb{E}(W)).$$

The first term corresponds to a successful execution, while the second term accounts for a failure striking during execution, with expected time lost $\mathbb{E}^{\text{lost}}$, recovery time R and calling $\mathbb{E}(W)$ recursively to restart from scratch. We know from [106] that $\mathbb{E}^{\text{lost}} = \frac{1}{\lambda_1} - \frac{T_1}{e^{\lambda_1 T_1} - 1}$, and after simplification we get $\mathbb{E}(W) = (\frac{1}{\lambda_1} + R)(e^{\lambda_1 T_1} - 1)$ (see [106]for details). We aim at minimizing the pattern overhead $\mathbb{H}(P) = \frac{E(P)}{T_1} - 1$. To get a first-order approximation, we assume that $\lambda_1 W$ is small so that we can expand $p_1 = 1 - e^{\lambda_1 T_1}$ into

$$p_1 = \lambda_1 \frac{W}{\sigma_1} + \frac{1}{2}\left(\lambda_1 \frac{W}{\sigma_1}\right)^2 + o\left(\left(\lambda_1 \frac{W}{\sigma_1}\right)^2\right).$$

We then derive that $\mathbb{H}(P) = \frac{C\sigma_1}{W} + \frac{\lambda_1 W}{2\sigma_1} + o(\sqrt{\lambda_1})$. The first two-terms show that $W_{opt} = \Theta(\lambda_1^{-1/2})$ and we retrieve the Young/Daly formula $W_{opt} = \sigma_1\sqrt{\frac{2C}{\lambda_1}}$. For the optimal pattern, we have $\mathbb{H}_{opt} = \sqrt{2C\lambda_1} + o(\sqrt{\lambda_1})$.

Equipped with these results for a single platform, we can now tackle the problem

with two platforms. We will need a second-order approximation of the form

$$\mathbb{H}(P) = \frac{C\sigma_1}{W} + \beta\left(\lambda\frac{W}{\sigma_1}\right) + \gamma\left(\lambda\frac{W}{\sigma_1}\right)^2 + \delta\lambda + o\left((\lambda W)^2\right),$$

where $\lambda = \lambda_1 + \lambda_2$ is the total failure rate, and β, γ and δ are constants that we derive below. With a single platform, we had $\beta = \frac{1}{2}$. With two platforms, we obtain a complicated expression for β, whose value will always be nonnegative. If β is strictly positive and above a reasonable threshold, we will proceed as above and be satisfied with the first-order approximation that gives $W_{opt} = \sigma_1\sqrt{\frac{C}{\beta\lambda}} = \Theta(\lambda^{-1/2})$. However, if β is zero or close to zero, we will need to resort to the second-order expansion to derive an accurate approximation of W_{opt}. In particular, when P_1 and P_2 are same-speed platforms, we will find that $\beta = 0$, $\gamma > 0$ and $W_{opt} = \Theta(\lambda^{-2/3})$.

As above, let $\mathbb{E}(W)$ be the expected time to execute a chunk of size W with both platforms. Let $T_1 = \frac{W}{\sigma_1}$ and $p_1 = 1 - e^{-\lambda_1 T_1}$ as before. We write:

$$\mathbb{E}(W) = \sum_{i=0}^{\infty} p_1^i(1-p_1)\mathbb{E}_i\,,$$

where $\mathbb{E}_i$ denotes the expected time to execute W successfully, knowing that there were i failures on P_1 before P_1 executes the chunk successfully. We point out that we condition $\mathbb{E}_i$ on the number of failures on P_1, independently on what is happening on P_2. In other words, we let P_1 execute until success, but we do account for the fact that P_2 may have completed before P_1 when computing $\mathbb{E}_i$. Similarly, letting $T_2 = \frac{W}{\sigma_2}$ and $p_2 = 1 - e^{-\lambda_2 T_2}$ be the probability of a failure on P_2, we write

$$\mathbb{E}_i = \sum_{j=0}^{\infty} p_2^j(1-p_2)\mathbb{E}_{i,j}\,,$$

where $\mathbb{E}_{i,j}$ denotes the expected execution time of the pattern, knowing there were i failures on P_1 and j failures on P_2 before both platforms execute successfully.

Theorem 5. *The expected execution time of a pattern $\mathbb{E}(P)$, whose execution is replicated on two platforms P_1 and P_2, is*

$$\begin{aligned}\mathbb{E}(P) = {} & (1-p_1)T_1 + p_1(1-p_1-p_2)\mathbb{E}_{1,0} \\ & + p_1p_2\mathbb{E}_{1,1} + p_1^2\mathbb{E}_{2,0} + C + o(\lambda^2W^3)\,,\end{aligned}$$

where p_1 and p_2 denote the probability of having a failure during the execution of the pattern on P_1 or P_2, respectively.

Proof. We aim at deriving a second-order approximation of $\mathbb{E}$. When $i = 0$, there is no failure on P_1, and the execution takes time T_1, regardless of the number of failures on

P_2. Therefore:

$$\mathbb{E}(W) = (1-p_1)T_1 + p_1(1-p_1)\mathbb{E}_1 + p_1^2(1-p_1)\mathbb{E}_2 + p_1^3\mathbb{E}_{3+}$$

where $\mathbb{E}_{3+}$ denote the expected time to execute the pattern successfully, knowing that there were at least 3 failures on P_1. The rationale to introduce $\mathbb{E}_{3+}$ is that it represents a lower-order term that can be neglected. Indeed, we have $\mathbb{E}_{3+} \leq \frac{3W}{\sigma_1} + 3R + \mathbb{E}$. To see that: for each failure, we lose at most $T_1 + R$; in the worst case, the three failures strike right before the checkpoint on P_1, and for each of them we lose the entire pattern time T_1 plus the recovery R (and P_2 has not completed execution yet). Then we re-execute the application once more, which is accounted for in re-introducing $\mathbb{E}$. Similarly, we can write $\mathbb{E}_1$ and $\mathbb{E}_2$ as follows:

$$\begin{aligned}\mathbb{E}_1 &= (1-p_2)\mathbb{E}_{1,0} + p_2(1-p_2)\mathbb{E}_{1,1} + p_2^2\mathbb{E}_{1,2+}\\ \mathbb{E}_2 &= (1-p_2)\mathbb{E}_{2,0} + p_2\mathbb{E}_{2,1+}\ ,\end{aligned}$$

where $\mathbb{E}_{1,2+} \leq \frac{W}{\sigma_1} + R + \mathbb{E}_1$ and $\mathbb{E}_{2,1+} \leq \frac{2W}{\sigma_1} + 2R + \mathbb{E}_2$. Then, we use Taylor series to approximate p_1 and p_2 to $\frac{\alpha_1\lambda}{\sigma_1}W + O(\lambda^2W^2)$ and $\frac{\alpha_2\lambda}{\sigma_2}W + o(\lambda^2W^2)$, respectively. Solving for $\mathbb{E}_1$ and $\mathbb{E}_2$, we derive that:

$$\begin{aligned}\mathbb{E}_1 &\leq \frac{1}{1-p_2^2}\left((1-p_2)\mathbb{E}_{1,0} + p_2(1-p_2)\mathbb{E}_{1,1} + p_2^2\left(\frac{1W}{\sigma_1} + 1R\right)\right)\\ \mathbb{E}_2 &\leq \frac{1}{1-p_2}\left((1-p_2)\mathbb{E}_{2,0} + p_2\left(\frac{2W}{\sigma_1} + 2R\right)\right)\ .\end{aligned}$$

Note that $\frac{1}{1-p_2^2} = 1 + O(\lambda^2W^2)$ and that $\frac{1}{1-p_2} = 1 + O(\lambda W)$. Altogether, $p_2^2\left(\frac{W}{\sigma_1} + R\right) = O(\lambda^2W^3)$ and $p_2\left(\frac{2W}{\sigma_1} + 2R\right) = O(\lambda W^2)$. Therefore, we derive that:

$$\begin{aligned}\mathbb{E}_1 &= (1-p_2)\mathbb{E}_{1,0} + p_2\mathbb{E}_{1,1} + O(\lambda^2W^3)\\ \mathbb{E}_2 &= \mathbb{E}_{2,0} + O(\lambda W^2)\ .\end{aligned}$$

Then, putting $\mathbb{E}_1$, $\mathbb{E}_2$, and $\mathbb{E}_{3+}$ back into $\mathbb{E}$ and solving for $\mathbb{E}$, we obtain:

$$\begin{aligned}\mathbb{E}(W) \leq \frac{1}{1-p^3}\Bigg((1-p_1)T_1 &\\ + p_1(1-p_1)\left((1-p_2)\mathbb{E}_{1,0} + p_2\mathbb{E}_{1,1} + O(\lambda^2W^3)\right) &\\ + p_1^2(1-p_1)\left(\mathbb{E}_{2,0} + O(\lambda W^2)\right) &\\ + p_1^3\left(\frac{3W}{\sigma_1} + 3R\right)\Bigg)\ .&\end{aligned}$$

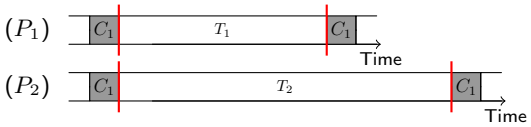


Figure 6.1: I_0 – no failure on P_1 (there can be failures on P_2); P_1 always finishes first.

Finally, note that $\frac{1}{1-p^3} = 1 + O(\lambda^3 W^3)$ and $p_1^3 \left(\frac{3W}{\sigma_1} + 3R\right) = O(\lambda^3 W^4)$.
Therefore, keeping second-order terms only, we obtain

$$\begin{aligned}
\mathbb{E}(W) = {} & (1-p_1)T_1 && (I_0)\\
& + p_1(1-p_1-p_2)\mathbb{E}_{1,0} && (I_1)\\
& + p_1 p_2 \mathbb{E}_{1,1} && (I_2)\\
& + p_1^2 \mathbb{E}_{2,0} && (I_3)\\
& + O(\lambda^3 W^4)\ ,
\end{aligned}$$

where I_0, I_1, I_2 and I_3 denote the four possible outcomes of the execution (up to two failures), with their associated probability. Finally, plugging $\mathbb{E}(W)$ back into $\mathbb{E}(P) = \mathbb{E}(W) + C$, we retrieve the equation of Theorem 5. □

Computing I_0 (Figure 6.1). Let I_0 denote the expected execution time associated with having no failures on P_1. With probability $(1-p_1)$, P_1 finishes faster than P_2 in T_1 time, and we can write:

$$I_0 = (1-p_1)T_1\ .$$

Using Taylor expansions to approximate p_1 to $\lambda_1 T_1 + \frac{\lambda_1^2 T_1^2}{2} + o(\lambda^2 T_1^2)$, we can write:

$$\begin{aligned}
I_0 &= \left(1 - \lambda_1 T_1 - \frac{\lambda_1^2 T_1^2}{2} + o(\lambda^2 T_1^2)\right) T_1\\
&= T_1 - \lambda_1 T_1^2 - \frac{\lambda^2 T_1^3}{2} + o(\lambda^2 T_1^3)\ .
\end{aligned}$$

Computing I_1 (Figure 6.2). Let I_1 denote the expected execution time when having exactly one failure on P_1. Letting $X \sim exp(\lambda_1)$ denote the failure inter-arrival time, we have:

$$\begin{aligned}
I_1 &= p_1(1-p_1-p_2)\int_0^\infty \mathbb{P}(X = t | X \leq T_1)\min(t + R + T_1, T_2)dt\\
&= p_1(1-p_1-p_2)\frac{1}{\mathbb{P}(X \leq T_1)}\int_0^{T_1} \mathbb{P}(X = t)\min(t + R + T_1, T_2)dt\ .
\end{aligned}$$

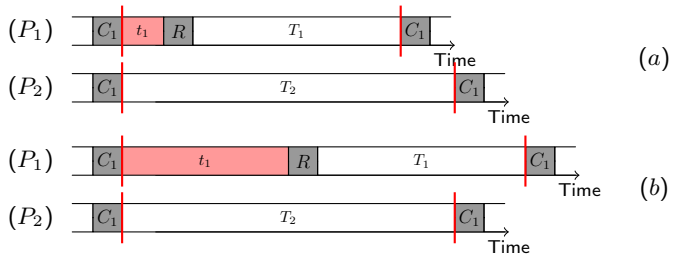


Figure 6.2: I_1 – there is one failure on P_1; depending on the failure arrival time t_1, P_1 finishes either first (a) or last (b).

By definition, $\mathbb{P}(X \leq T_1) = p_1$ and $\mathbb{P}(X = t) = \lambda_1 e^{-\lambda_1 t}$, therefore:

$$I_1 = (1 - p_1 - p_2) \int_0^{T_1} \lambda_1 e^{-\lambda_1 t} \min(t + R + T_1, T_2) dt \ .$$

Note that $\min(t + R + T_1, T_2)$ is in order of $O(W)$. Using Taylor series to approximate p_1 to $\lambda_1 T_1 + o(\lambda W)$, p_2 to $\lambda_2 T_2 + o(\lambda W)$, $e^{-\lambda_1 t}$ to $1 - \lambda_1 t + o(\lambda t)$ and keeping second-order terms only, we can get:

$$\begin{aligned} I_1 = \lambda_1 (1 - \lambda_1 T_1 - \lambda_2 T_2) \int_0^{T_1} (1 - \lambda_1 T_1) \min(t + R + T_1, T_2) dt \\ + o(\lambda^2 W^3) \ . \end{aligned}$$

The minimum depends on which platform finishes first. We know that $t + R + T_1 \leq T_2 \iff t \leq T_2 - T_1 - R$, so that we break the integral into two parts to address both cases, as follows:

$$\begin{aligned} I_1 = \lambda_1 (1 - \lambda_1 T_1 - \lambda_2 T_2) \Bigg(\int_0^{T_2 - T_1 - R} (1 - \lambda_1 t)(t + R + T_1) dt \\ + \int_{T_2 - T_1 - R}^{T_1} (1 - \lambda_1 t) T_2 \Bigg) dt + o(\lambda^2 W^3) \ , \end{aligned}$$

where $T_2 - T_1 - R$ must be both positive and less that T_1. Finally, let $r_1 = \max(\min(T_2 - T_1 - R, T_1), 0)$, and we can write:

$$\begin{aligned} I_1 = \lambda_1 (1 - \lambda_1 T_1 - \lambda_2 T_2) \Bigg(\int_0^{r_1} (1 - \lambda_1 t)(t + R + T_1) dt \\ + \int_{r_1}^{T_1} (1 - \lambda_1 t) T_2 dt \Bigg) + o(\lambda^2 W^3) \ . \end{aligned}$$

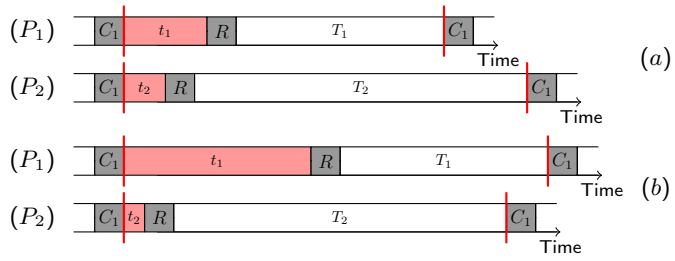


Figure 6.3: I_2 – there is one failure on P_1 and one failure on P_2; depending on the failure arrival times t_1 and t_2, P_1 finishes either first (a) or last (b).

Finally, note that I_1 depends on the value of r_1 as follows:

$$r_1 = \begin{cases} T_2 - T_1 - R, & \text{if } 0 \leq T_2 - T_1 - R \leq T_1 \\ T_1, & \text{if } T_2 - T_1 - R > T_1 \\ 0, & \text{otherwise.} \end{cases}$$

Assuming R is small in front of T_1 and T_2, we derive:

$$r_1 = \begin{cases} T_2 - T_1 - R, & \text{if } 1 \leq \frac{\sigma_1}{\sigma_2} \leq 2 \\ T_1, & \text{if } 2 < \frac{\sigma_1}{\sigma_2}. \end{cases}$$

Computing I_2 (Figure 6.3). Let I_2 denote the expected execution time when having one failure in P_1 and one failure in P_2. Let $X_1 \sim exp(\lambda_1)$ and $X_2 \sim exp(\lambda_2)$ denote the failure arrival time in P_1 and P_2, respectively. We can write:

$$\begin{aligned} I_2 = p_1 p_2 \Bigg(& \int_0^\infty \mathbb{P}(X_1 = t_1 | X_1 \leq T_1) \mathbb{P}(X_2 = t_2 | X_2 \leq T_2) \\ & \min(t_1 + R + T_1, t_2 + R + T_2) dt_1 dt_2 \Bigg) \\ = \frac{p_1 p_2}{\mathbb{P}(X_1 \leq T_1)\mathbb{P}(X_2 \leq T_2)} & \Bigg(\int_0^{T_1} \int_0^{T_2} \mathbb{P}(X_1 = t_1) \mathbb{P}(X_2 = t_2) \\ & \min(t_1 + R + T_1, t_2 + R + T_2) dt_1 dt_2 \Bigg) . \end{aligned}$$

Again, we have $\mathbb{P}(X_1 \leq T_1) = p_1$ and $\mathbb{P}(X_2 \leq T_2) = p_2$, as well as $\mathbb{P}(X_1 = t_1) = \lambda_1 e^{-\lambda_1 t_1}$ and $\mathbb{P}(X_2 = t_2) = \lambda_2 e^{-\lambda_2 t_2}$. Therefore, we can write:

$$I_2 = \int_0^{T_1} \int_0^{T_2} \lambda_1 e^{-\lambda_1 t_1} \lambda_2 e^{-\lambda_2 t_2} \min(t_1 + R + T_1, t_2 + R + T_2) dt_1 dt_2 \, .$$

Using Taylor series to approximate the Exponential terms to $1 + o(1)$ and keeping second-order terms only, we can get:

$$I_2 = \lambda_1\lambda_2 \int_0^{T_1} \int_0^{T_2} \min(t_1+R+T_1, t_2+R+T_2)dt_1dt_2 + o(\lambda^2 W^3) \ .$$

As before, platform P_2 finishes faster $\iff t_2 + R + T_2 \leq t_1 + R + T_1 \iff t_2 \leq t_1 + T_1 - T_2$. Therefore, we can can break the second integral into two parts, and we get:

$$I_2 = \lambda_1\lambda_2 \int_0^{T_1} e^{-\lambda_1 t_1} \Bigg(\int_0^{t_1+T_1-T_2} (t_2 + R + T_2)dt_2 + \int_{t_1+T_1-T_2}^{T_2} (t_1 + R + T_1)dt_2 \Bigg) dt_1 + o(\lambda^2 W^3) \ ,$$

where $t_1 + T_1 - T_2$ must be both positive and less than T_2. We find that $t_1 + T_1 - T_2 \leq T_2 \iff t_1 \leq 2T_2 - T_1$, which is always true, since t_1 is comprised between 0 and T_1. There remains one condition, and we find that $t_1 + T_1 - T_2 \leq 0 \iff t_1 \geq T_2 - T_1$, so that we can break the first integral into parts. Let $r_2 = \min(T_2 - T_1, T_1)$ (note that $T_2 - T_2$ is always positive), and we can write:

$$I_2 = \lambda_1\lambda_2 \Bigg(\int_0^{r_2} \int_0^{T_2} (t_1 + R + T_1)dt_2dt_1 + \int_{r_2}^{T_1} \Bigg(\int_0^{t_1+T_1-T_2} (t_2 + R + T_2)dt_2 + \int_{t_1+T_1-T_2}^{T_2} (t_1 + R + T_1)dt_2 \Bigg) dt_1 \Bigg) + o(\lambda^2 W^3) \ .$$

Finally, note that, similarly to I_1, I_2 depends on the value of r_2, which can be either:

$$r_2 = \begin{cases} T_2 - T_1, & \text{if } T_1 - T_2 \leq T_1 \\ T_1, & \text{otherwise.} \end{cases}$$

Simplifying, we find that:

$$r_2 = \begin{cases} T_2 - T_1, & \text{if } 1 \leq \frac{\sigma_1}{\sigma_2} \leq 2 \\ T_1, & \text{if } 2 < \frac{\sigma_1}{\sigma_2}. \end{cases}$$

Computing I_3 (Figure 6.4). Let I_3 denote the expected execution time when having two failures on P_1. Let $X_1 \sim exp(\lambda_1)$ and $X_2 \sim exp(\lambda_1)$ denote the failure arrival time

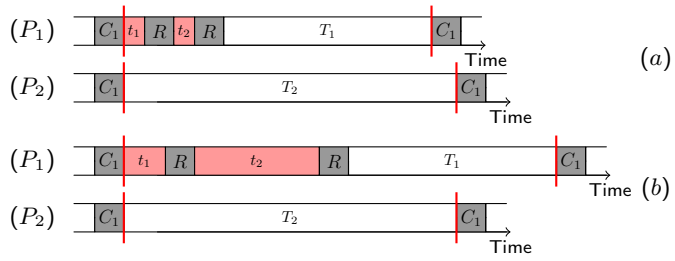


Figure 6.4: I_3 – there are two failures on P_1; depending on the failure arrival times t_1 and t_2, P_1 finishes either first (a) or last (b).

of each failure relative to their execution. We can write:

$$I_3 = p_1 p_1 \left(\int_0^\infty \mathbb{P}(X_1 = t_1 | X_1 \leq T_1) \int_0^\infty \mathbb{P}(X_2 = t_2 | X_2 \leq T_1) \right.$$
$$\left. \min(t_1 + t_2 + 2R + T_1, T_2) dt_1 dt_2 \right)$$
$$= \frac{p_1 p_1}{\mathbb{P}(X_1 \leq T_1)\mathbb{P}(X_2 \leq T_1)} \left(\int_0^{T_1} \mathbb{P}(X_1 = t_1) \int_0^{T_1} \mathbb{P}(X_2 = t_2) \right.$$
$$\left. \min(t_1 + t_2 + 2R + T_1, T_2) dt_1 dt_2 \right) ,$$

where $\mathbb{P}(X_1 \leq T_1) = \mathbb{P}(X_2 \leq T_1) = p_1$, and $\mathbb{P}(X_1 = t_1) = \lambda_1 e^{-\lambda_1 t_1}$ and $\mathbb{P}(X_2 = t_2) = \lambda_1 e^{-\lambda_1 t_2}$. Therefore, we can get:

$$I_3 = \int_0^{T_1} \lambda_1 e^{-\lambda_1 t_1} \int_0^{T_1} \lambda_1 e^{-\lambda_1 t_2} \min(t_1 + t_2 + 2R + T_1, T_2) dt_1 dt_2$$
$$= \int_0^{T_1} \lambda_1 e^{-\lambda_1 t_1} \int_{t_1+R}^{t_1+R+T_1} \lambda_1 e^{-\lambda_1 t_2} \min(t_2 + R + T_1, T_2) dt_1 dt_2 .$$

Using Taylor series to approximate $e^{-\lambda_1 t}$ to $1 + o(1)$ and keeping second-order terms only, we can get:

$$I_3 = \lambda_1^2 \int_0^{T_1} \int_{t_1+R}^{t_1+R+T_1} \min(t_2 + R + T_1, T_2) dt_1 dt_2 + o(\lambda^2 W^3) .$$

We find that $t_2 + R + T_1 \leq T_2 \iff t_2 \leq T_2 - T_1 - R$, and we break the second integral

into two parts, as follows:

$$I_3 = \lambda_1^2 \int_0^{T_1} \left(\int_{t_1+R}^{T_2-T_1-R} (t_2 + R + T_1) dt_2 + \int_{T_2-T_1-R}^{t_1+R+T_1} T_2 dt_2 \right) dt_1 + o(\lambda^2 W^3) \ ,$$

where $T_2 - T_1 - R$ must be both greater than $t_1 + R$ and less than $t_1 + R + T_1$. Again, we find that $T_2 - T_1 - R \leq t_1 + R + T_1 \iff t_1 \leq T_2 - 2T_1 - 2R$, and we can break the first integral into two parts again, as follows:

$$I_3 = \lambda_1^2 \left(\int_0^{T_2-2T_1-2R} \int_{t_1+R}^{t_1+R+T_1} (t_2 + R + T_1) dt_2 dt_1 + \int_{T_2-2T_1-2R}^{T_1} \left(\int_{t_1+R}^{T_2-T_1-R} (t_2 + R + T_1) dt_2 + \int_{T_2-T_1-R}^{t_1+R+T_1} T_2 dt_2 \right) dt_1 \right) + o(\lambda^2 W^3) \ ,$$

where $T_2 - 2T_1 - 2R$ must be both positive and less than T_1. Let $r_{3,1} = \max(\min(T_2 - 2T_1 - 2R, T_1), 0)$. Then, thanks to the last step, we know that the condition $T_2 - T_1 - R \leq t_1 + R + T_1$ is always verified, but there remains $t_1 + R \leq T_2 - T_1 - R \iff t_1 \leq T_2 - T_1 - 2R$. Therefore, we can break the second term into parts, as follows:

$$I_3 = \lambda_1^2 \left(\int_0^{r_{3,1}} \int_{t_1+R}^{t_1+R+T_1} (t_2 + R + T_1) dt_2 dt_1 + \int_{r_{3,1}}^{T_2-T_1-2R} \left(\int_{t_1+R}^{T_2-T_1-R} (t_2 + R + T_1) dt_2 + \int_{T_2-T_1-R}^{t_1+R+T_1} T_2 dt_2 \right) dt_1 + \int_{T_2-T_1-2R}^{T_1} \int_{t_1+R}^{t_1+R+T_1} T_2 dt_2 dt_1 \right) + o(\lambda^2 W^3) \ ,$$

where $T_2 - T_1 - 2R$ must be both greater than $r_{3,1}$, and less than T_1. Let $r_{3,2} =$

$\max(\min(T_2 - T_1 - 2R, T_1), 0)$, and we can write:

$$I_3 = \lambda_1^2 \Bigg(\int_0^{r_{3,1}} \int_{t_1+R}^{t_1+R+T_1} (t_2 + R + T_1) dt_2 dt_1 + \int_{r_{3,1}}^{r_{3,2}} \left(\int_{t_1+R}^{T_2-T_1-R} (t_2 + R + T_1) dt_2 + \int_{T_2-T_1-R}^{t_1+R+T_1} T_2 dt_2 \right) dt_1 + \int_{r_{3,2}}^{T_1} \int_{t_1+R}^{t_1+R+T_1} T_2 dt_2 dt_1 \Bigg) + o(\lambda^2 W^3) \ .$$

Finally, and similarly to I_1 and I_2 before, I_3 depends on the value of $r_{3,1}$ and $r_{3,2}$, which are as follows:

$$r_{3,1} = \begin{cases} T_2 - 2T_1 - 2R, & \text{if } 0 \leq T_2 - 2T_1 - 2R \leq T_1 \\ T_1, & \text{if } T_2 - 2T_1 - 2R > T_1 \\ 0, & \text{otherwise.} \end{cases}$$

$$r_{3,2} = \begin{cases} T_2 - T_1 - 2R, & \text{if } 0 \leq T_2 - T_1 - 2R \leq T_1 \\ T_1, & \text{if } T_2 - T_1 - 2R > T_1 \\ 0, & \text{otherwise.} \end{cases}$$

Asymptotically, the constant R is small in front of T_1 and T_2, which tend to infinity when λ tends to zero, and we derive:

$$r_{3,1} = \begin{cases} 0, & \text{if } 1 \leq \frac{\sigma_1}{\sigma_2} < 2. \\ T_2 - 2T_1 - 2R, & \text{if } 2 \leq \frac{\sigma_1}{\sigma_2} \leq 3 \\ T_1, & \text{if } 3 \leq \frac{\sigma_1}{\sigma_2} \end{cases}$$

$$r_{3,2} = \begin{cases} T_2 - T_1 - 2R, & \text{if } 1 \leq \frac{\sigma_1}{\sigma_2} \leq 2 \\ T_1, & \text{if } 2 < \frac{\sigma_1}{\sigma_2}. \end{cases}$$

6.3.2 Expected overhead

Theorem 6. *The expected overhead of a pattern $\mathbb{H}(P)$, whose execution is replicated on two independent platforms P_1 and P_2 is*

$$\mathbb{H}(P) = \frac{C\sigma_1}{W} + \beta \left(\lambda \frac{W}{\sigma_1} \right) + \gamma \left(\lambda \frac{W}{\sigma_1} \right)^2 + \delta\lambda + o\left((\lambda W)^2 \right) \ , \tag{6.1}$$

where $\lambda_1 = \alpha_1 \lambda$ and $\lambda_2 = \alpha_2 \lambda$ with $\alpha_1 + \alpha_2 = 1$. The values of the constants β, γ and δ are provided by the following case analysis:

Case 1: $1 \leq \frac{\sigma_1}{\sigma_2} \leq 2$.

$$\beta = \frac{\alpha_1}{2}\frac{-\sigma_1^2 + 4\sigma_1\sigma_2 - 3\sigma_2^2}{\sigma_2^2} ,$$
$$\gamma = \frac{\alpha_1^2}{2}\frac{\sigma_1^2 - 3\sigma_1\sigma_2 + 2\sigma_2^2}{\sigma_2^2} + \frac{\alpha_1\alpha_2}{3}\frac{2\sigma_1^3 - 9\sigma_1^2\sigma_2 + 12\sigma_1\sigma_2^2 - 4\sigma_2^3}{\sigma_2^3} ,$$
$$\delta = R\frac{\sigma_1 - \sigma_2}{\sigma_2} .$$

Case 2: $2 \leq \frac{\sigma_1}{\sigma_2} < 3$.

$$\beta = \frac{\alpha_1}{2}$$
$$\gamma = \frac{\alpha_1^2}{6}\frac{\sigma_1^3 - 9\sigma_1^2\sigma_2 + 27\sigma_1\sigma_2^2 - 26\sigma_2^3}{\sigma_2^3}$$
$$\delta = \alpha_1 R .$$

Case 3: $3 \leq \frac{\sigma_1}{\sigma_2}$.

$$\beta = \frac{\alpha_1}{2}$$
$$\gamma = \alpha_1^2$$
$$\delta = \alpha_1 R .$$

The optimal checkpointing period W_{opt} can be obtained by solving the following third-degree equation numerically:

$$\frac{\partial \mathbb{H}(P)}{\partial W} = -\frac{C\sigma_1}{W^2} + \beta\frac{\lambda}{\sigma_1} + 2\gamma\frac{\lambda W}{\sigma_1^2} = 0 . \tag{6.2}$$

Proof. Let $\mathbb{H}(P) = \frac{E(P)}{T_1} - 1$. We can write:

$$\mathbb{H}(P) = \frac{\sigma_1 C}{W} + \sigma_1\frac{I_0 + I_1 + I_2 + I_3}{W} - 1 + o(\lambda^2 W^2) .$$

Then, computing I_0, I_1, I_2 and I_3 according to the values of r_1, r_2, $r_{3,1}$ and $r_{3,2}$ presented in Section 6.3.1, we need to consider three cases, depending upon the ratio $\frac{\sigma_1}{\sigma_2}$. The values of β, γ and δ for each of the three cases are reported above. Finally, in order to get the corresponding optimal period, we need to solve $\frac{\partial \mathbb{H}(P)}{\partial W} = 0$, which amounts to solve Equation (6.2). This can be done numerically. □

For cases 2 and 3 (where $\sigma_1 \geq 2\sigma_2$), we have $\beta = \frac{\alpha_1}{2}$. If we use the first-order approximation, we neglect the last two terms with γ and δ in $\mathbb{H}(P)$. Then we obtain $W_{opt} = \sigma_1\sqrt{\frac{C}{\beta\lambda}}$, a similar formula as with a single platform. We experimentally check

the accuracy of the first-order approximation in Section 6.5.

On the contrary for case 1 (where $\sigma_1 \geq 2\sigma_2$), we have $\beta = \frac{\alpha_1}{2}(\frac{\sigma_1}{\sigma_2} - 1)(3 - \frac{\sigma_1}{\sigma_2}) \geq 0$ but $\beta = 0 \iff \sigma_1 = \sigma_2$. We can still use the first-order approximation when β is not too close to 0. For same-speed platforms, we need to use the second-order approximation:

Theorem 7. *For same-speed platforms ($\sigma_2 = \sigma_1$), the expected overhead is*

$$\mathbb{H}(P) = \frac{C\sigma_1}{W} + \frac{\alpha_1\alpha_2\lambda^2 W^2}{3\sigma_1^2} + o(\lambda^2 W^2) \; . \tag{6.3}$$

and the associated optimal checkpointing period is

$$W_{opt} = \sigma_1 \sqrt[3]{\frac{3C}{2\alpha_1\alpha_2\lambda^2}} \; . \tag{6.4}$$

Proof. With two same-speed platforms, we have $\sigma_2 = \sigma_1$. This corresponds to case 1 with $\beta = \delta = 0$ and $\gamma = \frac{\alpha_1\alpha_2}{3}$, hence we retrieve Equation (6.3). Then, differentiating and solving for W, we obtain Equation (6.4). □

It is striking to note that $W_{opt} = \Theta(\lambda^{-2/3})$ for same speed platforms, instead of $W_{opt} = \Theta(\lambda^{-1/2})$. Finally, with two identical platforms ($\alpha_1 = \alpha_2 = \frac{1}{2}$ and $\lambda = 2\lambda_1$), we obtain $W_{opt} = \sigma_1 \sqrt[3]{\frac{3C}{2\lambda_1^2}}$.

6.3.3 Failures in checkpoints and recoveries

So far, we have assumed that failures do not strike during checkpoints and recoveries. In this section, we show how to handle failures during these operations, and that the approximations derived in the preceding section remain valid as long as the platform MTBF $\mu = 1/\lambda$ is large in front of the other resilience parameters.

Let $\mathbb{E}(R)$ and $\mathbb{E}(C)$ denote the expected time to perform a recovery and a checkpoint, respectively. The probability of a failure occurring during a process of length L on platform P_i is given by $p_i^L = 1 - e^{-\lambda_i \frac{L}{\sigma_i}}$. If a failure strikes during the recovery, we lose $\mathbb{E}_R^{\text{lost}}$ time due to the failure, and we account for the time to try again by calling $\mathbb{E}(R)$ recursively. If a failure strikes during the checkpoint, we lose $\mathbb{E}_C^{\text{lost}}$ time due to the failure, we account for the recovery time $\mathbb{E}(R)$, and we then need to re-execute the entire pattern, which is accounted for by calling $\mathbb{E}(W)$ and $\mathbb{E}(C)$, recursively. Altogether, we have:

$$\begin{aligned}
\mathbb{E}(R) &= p_i^R \left(\mathbb{E}_R^{\text{lost}} + \mathbb{E}(R) \right) + (1 - p_i^R)R \; , \\
\mathbb{E}(C) &= p_i^C \left(\mathbb{E}_C^{\text{lost}} + \mathbb{E}(R) + \mathbb{E}(W) + \mathbb{E}(C) \right) + (1 - p_i^C)C \; .
\end{aligned}$$

We know from [106] that $\mathbb{E}_L^{\text{lost}} = \frac{1}{\lambda_i} - \frac{L}{e^{\lambda_i L}-1}$. Solving the above equations and simplifying, we derive that:

$$\mathbb{E}(R) = \frac{e^{\lambda_i R} - 1}{\lambda_i} , \tag{6.5}$$

$$\mathbb{E}(C) = \frac{e^{\lambda_i C} - 1}{\lambda_1} + (e^{\lambda_i C} - 1)\left(\mathbb{E}(R) + \mathbb{E}(W)\right) . \tag{6.6}$$

Now, recall from our previous analysis in Section 6.3 that the optimal pattern length satisfies $W_{opt} = \Theta(\lambda^{-1/2})$ and that $\mathbb{H}_{opt}(P) = \Theta(\lambda^{1/2})$. Hence, in an optimized pattern, we will have $\mathbb{E}(W) \leq \mathbb{E}(P) = \frac{W_{opt}}{\sigma_i}\left(1 + \mathbb{H}_{opt}(P)\right) = \Theta(\lambda^{-1/2})$. Then, using Taylor expansions to approximate Equations 6.5 and 6.6, we can derive the following results:

$$\mathbb{E}(R) = R + O(\lambda) ,$$
$$\mathbb{E}(C) = C + O(\sqrt{\lambda}) .$$

This suggests that the expected costs to perform checkpoints and recoveries are dominated by their original costs, under the assumption of a large MTBF. Intuitively, this is due to the small probability of encountering a failure during these operations. Thus, in Section 6.3, replacing R and C by their expected values does not affect the expected execution time of the pattern, neither in the first-order nor second-order approximations.

6.4 On-failure checkpointing

In this section, we present another strategy. Contrarily to the approach of Section 6.3, the work is not divided into periodic patterns. We only checkpoint when a failure strikes either platform. More precisely, when a failure f strikes one platform, we use the other platform to checkpoint the work, so that both platforms can resume their execution from this checkpoint, in a synchronized fashion. This scheme is exposed to the risk of having a second failure f' striking the other platform during its checkpoint, which would cause to roll-back and re-execute from the previous checkpoint (which was taken right after the failure preceding f, which may be a long time ago). Such a risk can be neglected in most practical settings. As before, we will assume that failures do not strike during checkpoints.

Intuitively, this checkpoint-on-failure strategy is appealing, because we checkpoint a minimum number of times. And when a failure strikes the slow platform P_2, we do not roll-back. However, when a failure strikes the fast platform P_1, we have to roll-back to the state of P_2. Altogether, we expect this strategy to work better when platform speeds are close. We will experimentally assess the checkpoint-on-failure strategy in Section 6.5.

6.4.1 Expected execution time

Let $\mathbb{E}(A)$ denote the expected time needed to execute the application successfully, and let $T_{base} = \frac{W_{base}}{\sigma_1}$ denote the total execution time of the application on the fast platform P_1, without any resilience mechanism nor failures. Here W_{base} denotes the total amount of work of the application.

Theorem 8. *The expected execution time of the application is*

$$\mathbb{E}(A) = T_{base} + \frac{T_{base}}{\mu}\left(C + \alpha_1\left(\mu\frac{\sigma_1 - \sigma_2}{\sigma_1}\right)\right) . \tag{6.7}$$

where $\mu = \frac{1}{\lambda}$ is the MTBF.

Proof. We first consider the case of two identical platforms, i.e. $\sigma_1 = \sigma_2$ and $\lambda_1 = \lambda_2 = \frac{\lambda}{2}$. In this case, as soon as a failure occurs on either platform, the other one immediately checkpoints, and both platforms synchronize on this checkpoint, before resuming execution. In other words, the execution never rolls back, and no work is ever lost.

Now, in order to compute the expected execution time, we need to account for the time needed to execute the entire application T_{base}, as well as the time lost due to failures. When a failure occurs, we only need to account for the time C to checkpoint and synchronize. In addition, we can estimate the expected number of failures as $\frac{T_{base}}{\mu}$ in average, and we write:

$$\mathbb{E}(A) = T_{base} + \frac{T_{base}}{\mu}C .$$

This is fine for two identical platforms. However, when failure rates and speeds differ, there are two cases: (i) a failure strikes the fast platform P_1. Then platform P_2 checkpoints, but because it is slower than P_1, P_1 needs to rollback and we lose the extra amount of work that P_1 has computed since the last failure and synchronization; (ii) a failure strikes the slow platform P_2. Then platform P_1 checkpoints, and because it is faster, P_2 will roll-forward instead, catching up with the execution of P_1.

Assuming failures are Exponentially distributed, and given that a failure (from either platform) strikes during the execution of the segment, the probability that the failure belongs to a particular platform is proportional to the failure rate of that platform [139], i.e. the probability that the failure belongs to P_1 and P_2 are $\frac{\lambda_1}{\lambda} = \alpha_1$ and $\frac{\lambda_2}{\lambda} = \alpha_2$, respectively.

In order to compute the expected execution time, we first need to account for T_{base}, which is the time to execute the application once, without failures. Then, when a failure strikes, either it strikes P_2, with probability α_2, and we only lose the time to checkpoint C; or it strikes P_1, with probability α_1, and we lose the difference between the amount of work executed on P_1 and P_2 since the last synchronization. In average, the last synchronization was when the last failure occurred, that is μ time-steps ago.

Name	Titan	Cori	K computer	Trinity	Theta
Speed (PFlops)	17.6	14.0	10.5	8.1	5.1
MTBF (s)	50,000	100,000			

Table 6.1: Summary of parameters used for simulations for each platform.

During that time, P_1 and P_2 have executed $\mu\sigma_1$ and $\mu\sigma_2$ units of work, respectively, and we have lost $\mu\frac{\sigma_1-\sigma_2}{\mu}$ due to the failure. Altogether, we can write:

$$\mathbb{E}(A) = T_{base} + \frac{T_{base}}{\mu}\left(C + \alpha_1\left(\mu\frac{\sigma_1-\sigma_2}{\sigma_1}\right)\right) .$$

□

6.4.2 Expected overhead

Theorem 9. *The expected overhead is*

$$\mathbb{H}(A) = \frac{C}{\mu} + \alpha_1\left(\frac{\sigma_1-\sigma_2}{\sigma_1}\right) . \tag{6.8}$$

Proof. Let $\mathbb{H}(A) = \frac{\mathbb{E}(A)}{T_{base}} - 1$. We write:

$$\mathbb{H}(A) = \frac{1}{\mu}\left(C + \alpha_1\left(\mu\frac{\sigma_1-\sigma_2}{\sigma_1}\right)\right) .$$

Then, simplifying, we obtain Equation (6.8). □

6.5 Experimental evaluation

In this section, we conduct a set of simulations, whose goal is three-fold: (i) assess the accuracy of the proposed models; (ii) compare the performance of the two replication strategies in different scenarios; and (iii) evaluate the performance improvement of the approach over classical periodic checkpointing with a single platform.

6.5.1 Simulation setup

This section describes the parameters used for the simulations. First, we set $R = C$ in all cases. Indeed, the recovery time and checkpointing time are equivalent to a read (recovery) and a write (checkpoint) operation, and they take approximately the same amount of time. Then, we set the other parameters according to real behaviors on

today's supercomputers. Because the typical failure rate for the most powerful Top500 platforms [181] is around 1 or 2 failures per day, we choose $\mu_1 = 50,000s \approx 14h$ and $\mu_2 = 100,000s \approx 28h$. The speeds were set using the *Rmax* value (maximum performance achieved when executing LINPACK) in PFlops of Top500 platforms (list of November 2016). We always set $\sigma_1 = 17.6$ (units in Petaflops, corresponding to the Titan platform), and we build four different cases aiming at having different $\frac{\sigma_1}{\sigma_2}$ ratios: σ_2 can be either 14.0 (Cori), 10.5 (K computer), 8.1 (Trinity) or 5.1 (Theta). We also have two possible configurations for the checkpointing (and recovery) time: a small checkpoint of 60 seconds and a large checkpoint of 1800 seconds. Overall, the parameters used by default for each platform are summarized in Table 6.1.

For each experiment, we setup the simulator with the resilience parameters λ_1, λ_2, C and R, and we compute the optimal pattern length W_{opt}, which is obtained by solving Equation 6.1 numerically. The total amount of work in the simulation is fixed to be $1000W_{opt}$, and each simulation is repeated 1000 times. All the figures report the optimal overhead $\mathbb{H}_{opt}$ as a function of some parameter. The solid lines are simulation results: green for the fastest machine alone (with Young/Daly period), blue for the periodic checkpoint strategy, red for the on-failure checkpoint strrtegy. The dashed lines are model predictions: blue for the periodic checkpoint strategy, red for the on-failure checkpoint strategy. The simulator is publicly available at `http://perso.ens-lyon.fr/aurelien.cavelan/replication-ftxs.zip`.

6.5.2 Accuracy of the models

In this section, we study the accuracy of the models and we assess the usefulness of the second-order approximation by comparing the results obtained with both first and second-order formulas. We take the fastest machine Titan and let its speed σ_1 vary, while keeping all other parameters fixed. Hence we always have $\mu_1 = 50,000s$ and four possible second platforms (Cori, K-computer, Trinity, Theta) whose parameters are given in Table 6.1.

Figures 6.5 and 6.6 present the evolution of the overhead as a function of σ_1 varying from σ_2 to $5\sigma_2$, and using a checkpointing time of $60s$ (Figure 6.5), and $1800s$ (Figure 6.6). We observe that the model matches very well the results of the simulations: the maximum relative error is 5% with $C = 1800s$, and is within 0.2% with $C = 60s$. The latter result is expected: we do not account for failures during checkpoints t in the analysis, hence the approximation gets less accurate as checkpoint time increases.

For each value of σ_1 varying from σ_2 to $5\sigma_2$, we set β, γ and δ in Equation 6.1, according to the ratio $\frac{\sigma_1}{\sigma_2}$, which shows the accuracy of the formula in all three cases. Finally, we note that the overhead increases with larger speeds σ_1, but the expected throughput (time per unit of work) keeps decreasing.

Regarding on-failure checkpointing, we observe that the precision of the formula quickly degrades with larger σ_1, because it does not take into account failures that can occur during the re-execution work, which corresponds to the factor $\mu(\frac{\sigma_1-\sigma_2}{\sigma_1})$ in Equation 6.8. Note that this factor grows when σ_1 increases (or when σ_2 decreases),

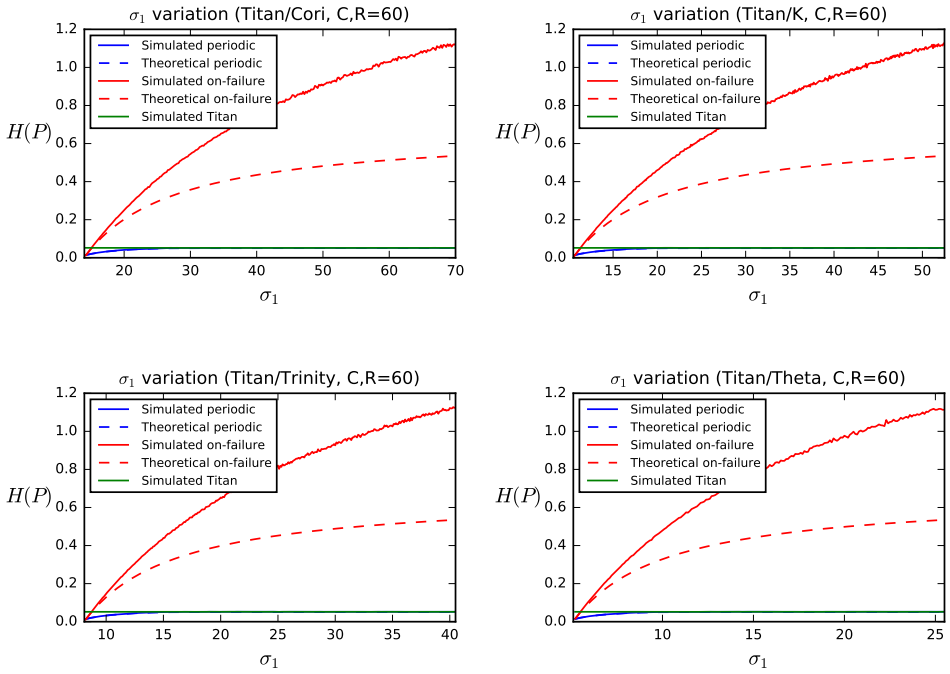


Figure 6.5: Evolution of overhead when σ_1 varies with $C = R = 60s$.

and it is not surprising to find that the overhead is always underestimated when the two speeds are quite different.

Next in Figures 6.7 and 6.8, we compare the simulated and theoretical overheads obtained with the first and second-order approximations. Note that the plot colors have a different meaning in this figure. The difference is small when using small checkpoint time (Figure 6.7), but when the two speeds get close and the checkpoint cost is high (Figure 6.8), the first-order approximation collapses and the theoretical overhead increases dramatically ($\mathbb{H}_{opt} = 0.5$). This is because the coefficient in $O(\lambda W)$ tends to 0, and the first-order approximation used to get W_{opt} is not valid anymore. However, we show that using the second-order approximation, (i.e. considering additional terms in $O(\lambda^2 W^2)$) still yields good results ($\mathbb{H}_{opt} = 0.128$).

6.5.3 Comparison of the two strategies

In this section, we compare the overhead with the two strategies against that with a single platform. Coming back to Figures 6.5 and 6.6, we make two observations. First, when the ratio between σ_1 and σ_2 is large (greater than 2 with a small checkpoint C, somewhat higher when C increases), using a periodic pattern with replication is the same as using the fast platform only: the slow platform is not useful. Second, when this ratio between between σ_1 and σ_2 increases, the on-failure checkpointing strategy becomes worst than using the fast platform alone, especially with small checkpoint costs (left). This can be explained as follows: we wait for a failure on the slow platform to checkpoint the work done by the fast platform. But given the value of μ_2, the slow

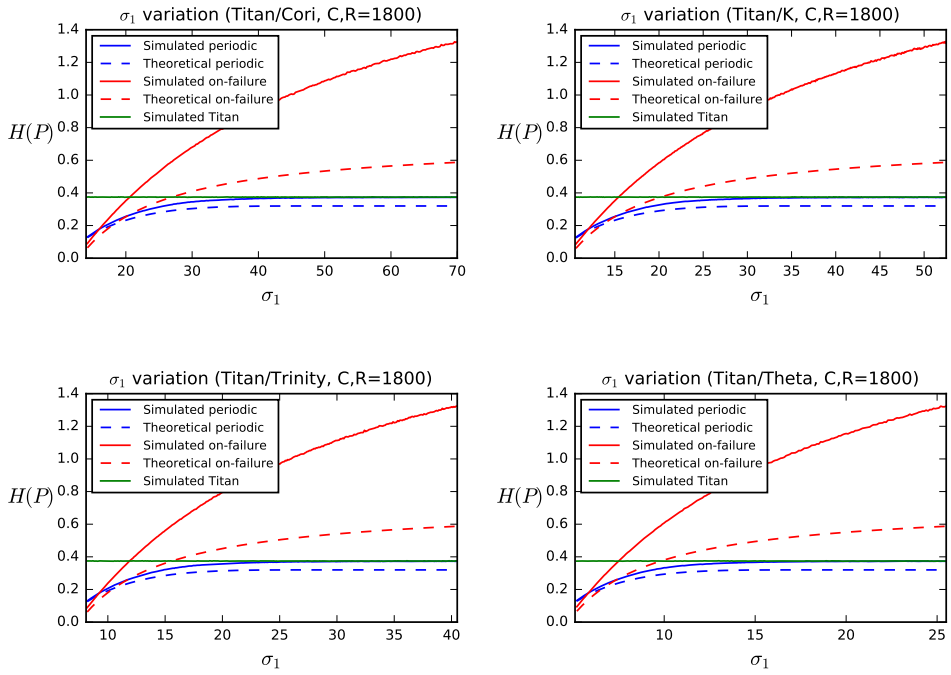


Figure 6.6: Evolution of overhead when σ_1 varies with $C = R = 1800s$.

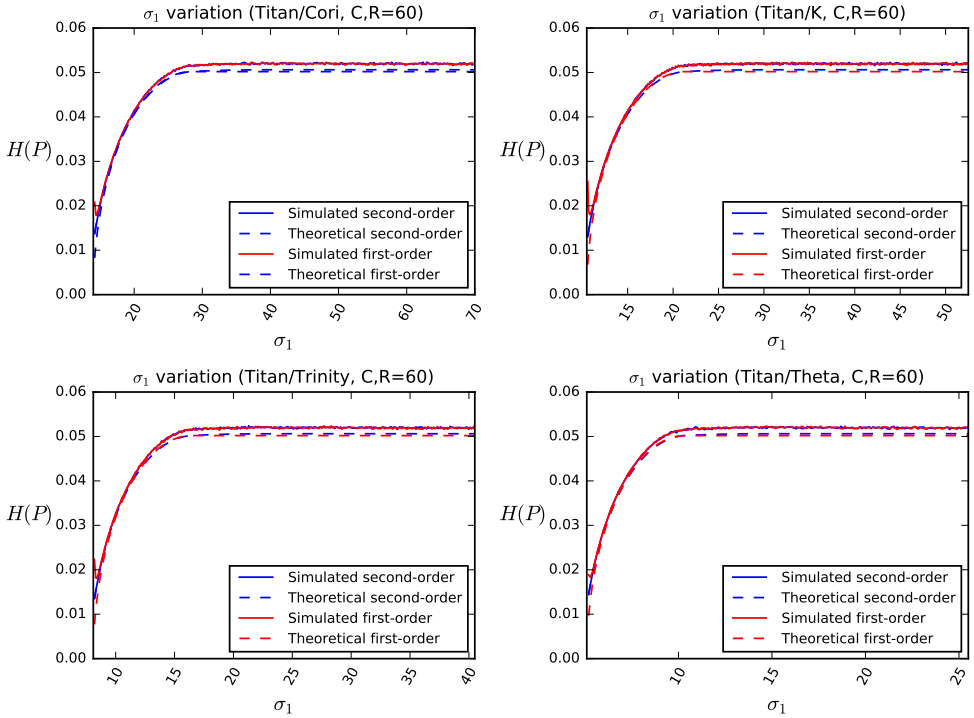


Figure 6.7: Comparison of overhead using first-order approximation and second-order approximation when σ_1 varies, with $C = R = 60s$.

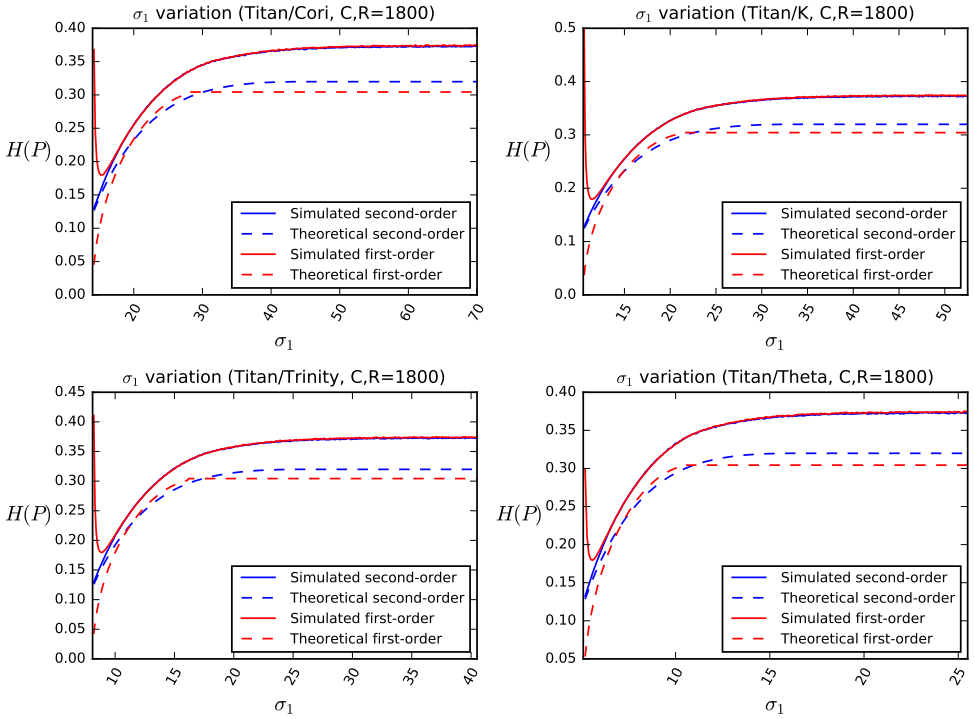


Figure 6.8: Comparison of overhead using first-order approximation and second-order approximation when σ_1 varies, with $C = R = 1800s$.

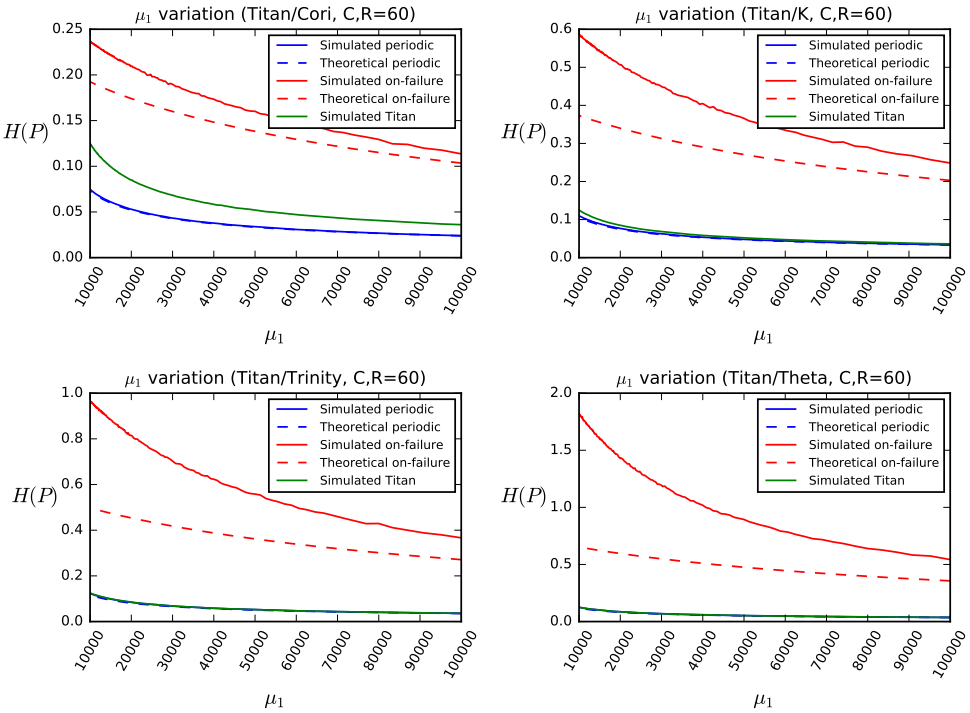


Figure 6.9: Evolution of overhead when μ_1 varies with $C = R = 60s$.

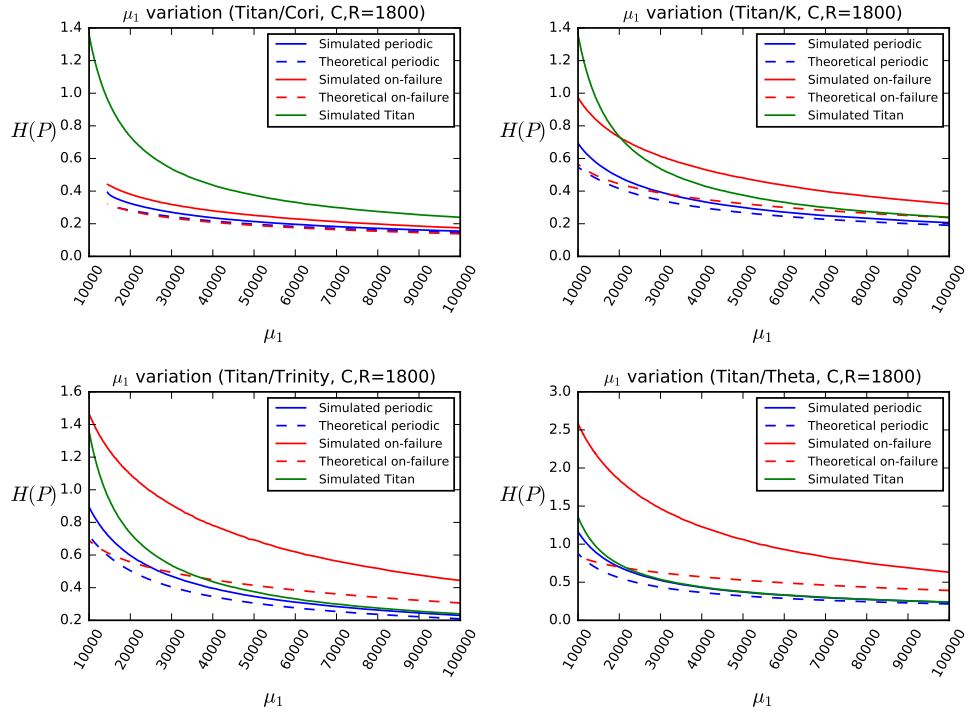


Figure 6.10: Evolution of overhead when μ_1 varies with $C = R = 1800s$.

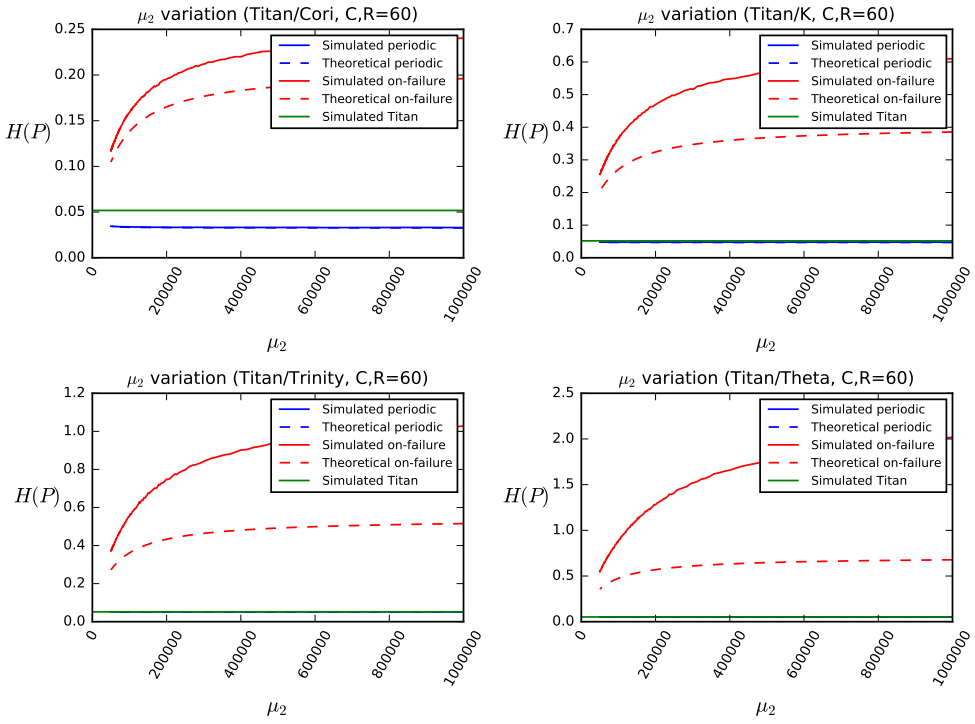


Figure 6.11: Evolution of overhead when μ_2 varies with $C = R = 60s$.

platform is struck less frequently than the fast one, hence we often lose a lot of work in expectation (remember we lose $\mu(\sigma_1 - \sigma_2)$ units of work when a failure strikes on P_1).

Figures 6.9 to 6.13 show the evolution of the overhead when parameters μ_1, μ_2 and C, R vary. Overall, we observe again that the work lost when a failure occurs on P_1 is important with the on-failure checkpointing strategy, whose overhead strongly depends upon on the second platform used. For instance, the overhead for $\mu_1 = 10,000s$ and $C = 60s$ goes from 0.236 (using Cori) to 1.81 (using Theta), whereas the overhead of the periodic checkpointing remains small (between 0.074 and 0.125). This observation is confirmed by Figures 6.11 and 6.12, where the overhead increases when the number of faults actually decreases on the slow platform!

We see the benefits of using replication when looking at Figures 6.9 and 6.10. When μ_1 becomes small ($10,000s$, or 8.6 failures per day), the overhead with a single platform (green) increases a lot, while the overhead with the periodic strategy (blue) increases only a little, even when the second platform is twice slower than the first one. For instance we have an overhead of 1.36 for P_1 alone when $C = 1800s$, whereas we get 0.894 when using P_1 in conjunction with Trinity, i.e. a reduction of 34%. However, when the second platform gets too slow, the improvement brought by the use of P_2 is only meaningful when the checkpointing cost is large: on Figure 6.13, we get 15% of

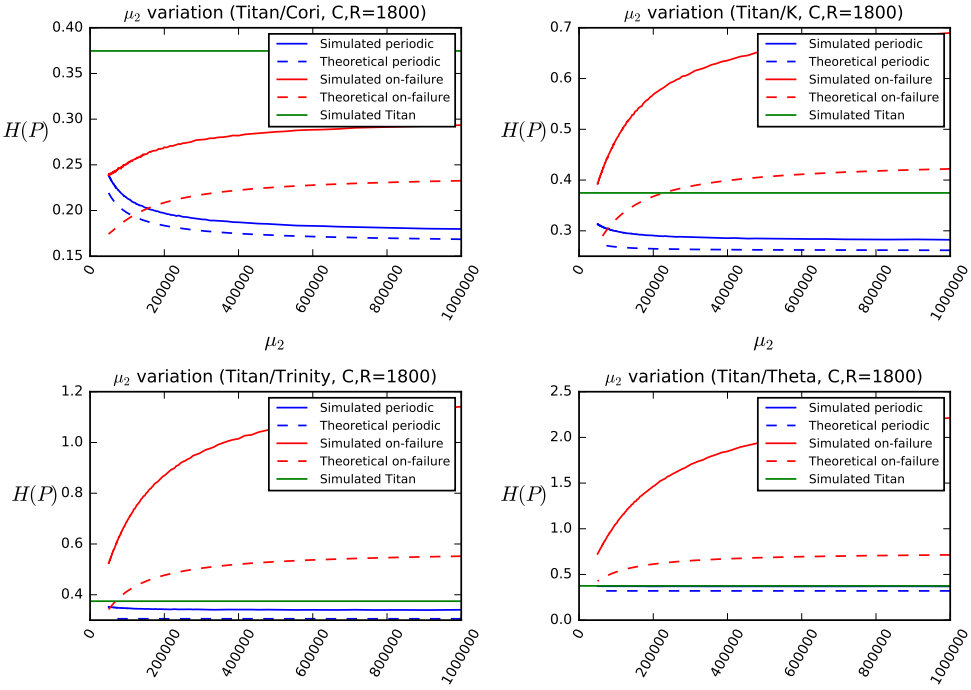


Figure 6.12: Evolution of overhead when μ_2 varies with $C = R = 1800s$.

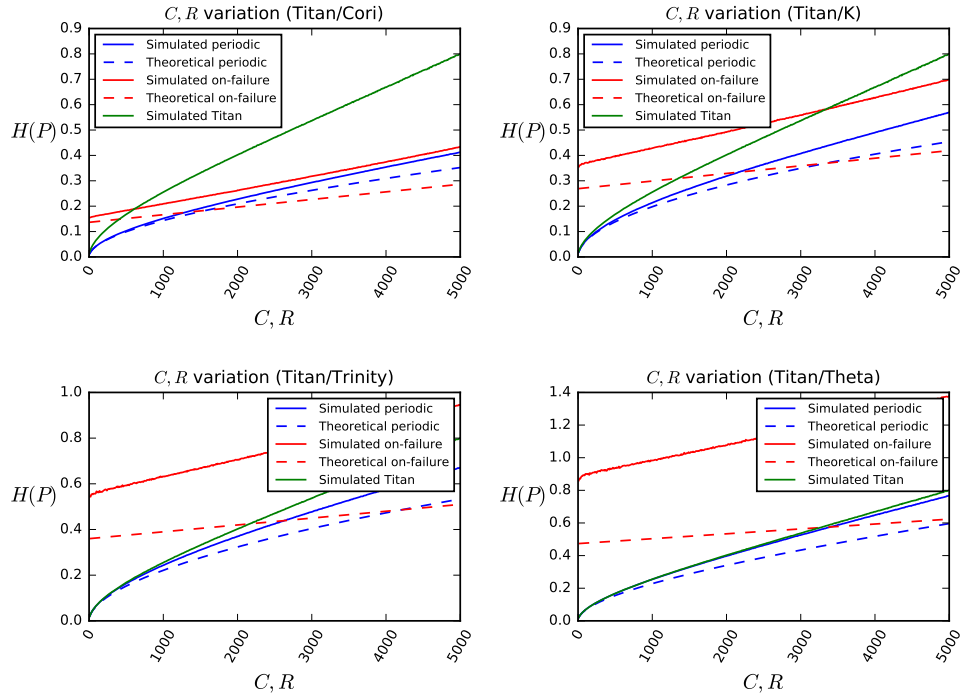


Figure 6.13: Evolution of overhead when C and R vary.

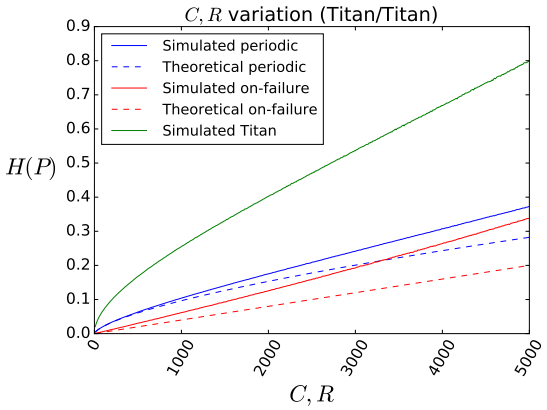


Figure 6.14: Evolution of overhead when C and R vary, using two same-speed platforms.

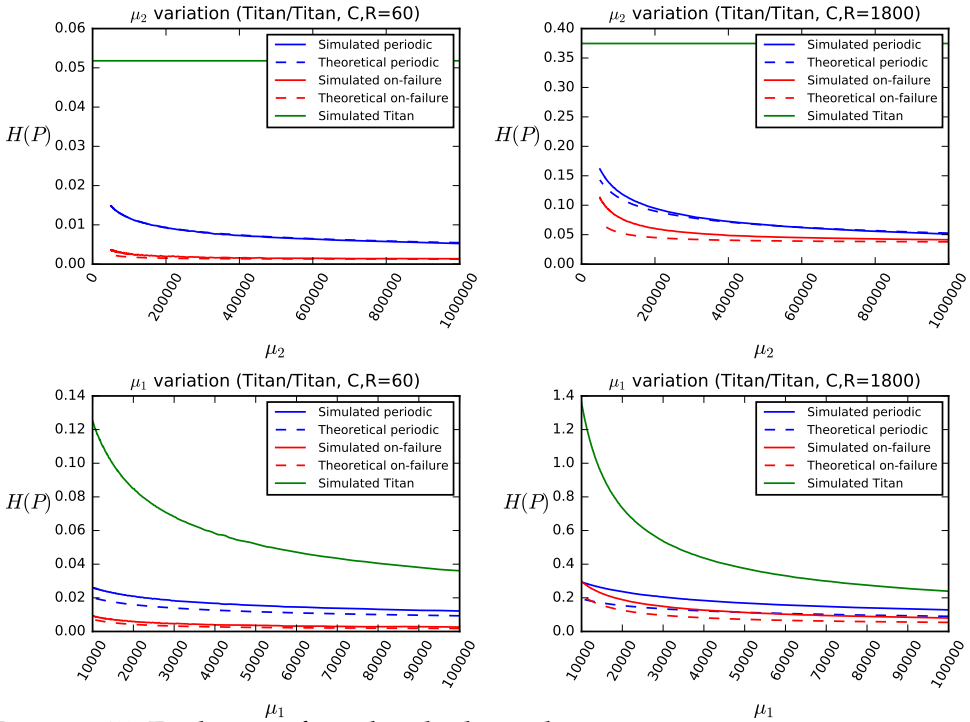


Figure 6.15: Evolution of overhead when other parameters vary, using two same-speed platforms.

improvement if $C \geq 10s$ with Cori, if $C \geq 760s$ with K, if $C \geq 4460s$ with Trinity, and more than $5000s$ with Theta.

Figures 6.14 and 6.15 present the case of same-speed platforms. In this case, for all parameter choices (C, R, μ_1, μ_2), it is interesting to see that on-failure checkpointing is the best strategy, while it was less efficient than periodic checkpointing in almost all the other scenarios that we considered. This can be explained by the fact that there is no work lost at all with this strategy, except when there is a failure during a checkpoint.

6.5.4 Summary

We summarize simulation results as follows:

- The model is very accurate, as long as the resilience parameters remain reasonably small.
- On-failure checkpointing is generally less efficient than periodic checkpointing, except when the speeds of the two platforms are equal ($\sigma_2 = \sigma_1$).
- If P_2 is really too slow compared to P_1 ($\sigma_2 < \frac{\sigma_1}{2}$) or if the checkpointing cost is small, there is little reason to use a second platform.
- In all other cases ($\frac{\sigma_1}{2} \leq \sigma_2 < \sigma_1$), the periodic checkpointing strategy reduces the overhead by 30% in average, and up to 90% in some particular cases.

6.6 Conclusion

This work has addressed group replication for a black-box application executing on two heterogeneous platforms. We designed and thoroughly analyzed two strategies, periodic checkpointing and on-failure checkpointing. For periodic checkpointing, we have been able to analytically derive the best pattern length, using either first-order or second-order approximations. These results nicely extend the Young/Daly formula.

Simulations show that the model is quite accurate. As expected, when the platform speeds have different orders of magnitude, it is better to use only the fast platform. However, periodic checkpointing is useful for a wide range of speeds, and generally more efficient than on-failure checkpointing. The latter strategy is to be preferred only when the platform speeds are close.

Future work will be devoted to extending replication with heterogeneous platforms to deal with more complex applications, such as scientific workflows arranged as linear chains or fork-join graphs. Another interesting direction is to study the bi-criteria problem with energy consumption as a second metric, in addition to total execution time, in order to better assess the cost of replication.

Chapter 7

Replication is more efficient than you think

This chapter revisits replication coupled with checkpointing for fail-stop errors. Replication enables the application to survive many fail-stop errors, thereby allowing for longer checkpointing periods. Previously published works use replication with the *no-restart* strategy, which works as follows: (i) compute the application Mean Time To Interruption (MTTI) M as a function of the number of processor pairs and the individual processor Mean Time Between Failures (MTBF); (ii) use checkpointing period $T^{\text{no}}_{MTTI} = \sqrt{2MC}$ à la Young/Daly, where C is the checkpoint duration; and (iii) never restart failed processors until the application crashes. We introduce the *restart* strategy where failed processors are restarted after each checkpoint. We compute the optimal checkpointing period T^{rs}_{opt} for this strategy, which is much larger than T^{no}_{MTTI}, thereby decreasing I/O pressure. We show through simulations that using T^{rs}_{opt} and the *restart* strategy, instead of T^{no}_{MTTI} and the usual *no-restart* strategy, significantly decreases the overhead induced by replication. The work in this chapter is joint work with Anne Benoit, Thomas Hérault and Yves Robert, and has been published in *Supercomputing* (SC) 2019 [C2].

7.1 Introduction

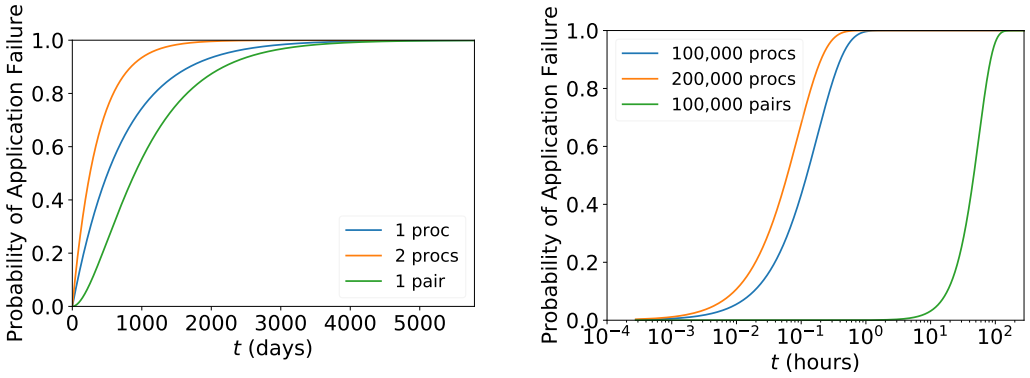
Current computing platforms have millions of cores: the Summit system at the Oak Ridge National Laboratory (ORNL) is listed at number one in the TOP500 ranking [181], and it has more than two million cores. The Chinese Sunway TaihuLight (ranked as number 3) has even more than 10 million cores. These large-scale computing systems are frequently confronted with failures, also called fail-stop errors. Indeed, even if individual cores are reliable, for instance if the *Mean Time Between Failures* (MTBF) for a core is $\mu = 10$ years, then the MTBF for a platform with a million cores ($N = 10^6$) becomes $\mu_N = \frac{\mu}{N} \approx 5.2$ minutes, meaning that a failure strikes the platform every five minutes, as shown in [106].

The classical technique to deal with failures consists of using a checkpoint-restart mechanism: the state of the application is periodically checkpointed, and when a failure occurs, we recover from the last valid checkpoint and resume the execution

from that point on, rather than starting the execution from scratch. The key for an efficient checkpointing policy is to decide how often to checkpoint. Young [200] and Daly [58] derived the well-known Young/Daly formula $T_{YD} = \sqrt{2\mu_N C}$ for the optimal checkpointing period, where μ_N is the platform MTBF, and C is the checkpointing duration.

Another technique that has been advocated for dealing with failures is process replication, where each process in a parallel MPI (Message Passing Interface) application is duplicated to increase the *Mean Time To Interruption* (MTTI). The MTTI is the mean time between two application failures. If a process is struck by a failure, the execution can continue until the replica of this process is also struck by a failure. More precisely, processors are arranged by pairs, i.e., each processor has a replica, and the application fails whenever both processors in a same pair have been struck by a failure. With replication, one considers the MTTI rather than the MTBF, because the application can survive many failures before crashing. Given the high rate of failures on large-scale systems, process replication is combined with periodic checkpoint-restart, as proposed for instance in [159, 75, 205] for high-performance computing (HPC) platforms, and in [127, 198] for grid computing. Then, when the application fails, one can recover from the last valid checkpoint, just as was the case without replication. Intuitively, since many failures are needed to interrupt the application, the checkpointing period should be much larger than without replication. Previous works [82, 46, 110] all use $T^{\text{no}}_{MTTI} = \sqrt{2M_N C}$ for the checkpointing period, where M_N is the MTTI with N processors (instead of the MTBF μ_N).

To illustrate the impact of replication on reliability at scale, Figure 7.1 compares the probability distribution of the time to application failure for: (a) a single processor,



(a) CDFs of the probability distribution of time to app. failure for one processor, two parallel processors and one proc. pair (replication).

(b) CDFs of the proba. distrib. of time to app. failure for 100,000 parallel proc., 200,0000 parallel proc. and 100,000 proc. pairs (replication).

Figure 7.1: Comparison of CDFs with and without replication.

two parallel processors and a pair of replicated processors; and (b) a platform of $N = 100,000$ parallel processors, $N = 200,000$ parallel processors without replication, and $b = 100,000$ processor pairs with replication. In all cases, the individual MTBF of a single processor is $\mu = 5$ years. The time to reach 90% chances of having a fatal failure is: (a) 1688 days for one processor, 844 days for two processors and 2178 days for a processor pair; and (b) 24 minutes for 100,000 processors, 12 minutes for 200,000 processors and 5081 minutes (almost 85 hours) for 100,000 processor pairs. We see that replication is key to safe application progress at scale! Again, the cost is that half of the resources are doing redundant work, hence time-to-solution is increased. We compare time-to-solution with and without replication in Section 7.7.6. We also see that in heavily failure-prone environments (small MTBF values), checkpoint/restart alone cannot ensure full reliability, and must be complemented by replication.

One major contribution of this chapter is to introduce a new approach that minimizes the overhead incurred by the checkpoint-restart mechanism when coupled with replication. Previous works [82, 46, 110] use the *no-restart* strategy: if a processor was struck by a failure (but not its replica), then the processor remains failed (no recovery) until the whole application fails. Hence, there is a recovery only every M_N seconds on average, whenever the application fails. Many periodic checkpoints are taken in between two application crashes, with more and more processors failing on the fly. To the best of our knowledge (i) all related works use the *no-restart* strategy and (ii) analytically computing the optimal period for *no-restart* is an open problem (see Section 7.4.2 for more details, where we also show that non-periodic strategies are more efficient for *no-restart*), but simulations can help assess this approach.

The study of the *no-restart* strategy raises an important question: should failed processors be restarted earlier on in the execution? Instead of waiting for an application crash to rejuvenate the whole platform, a simple approach would be to restart processors immediately after each failure. Let *restart-on-failure* denote this strategy. It ensures that all processor pairs involve two live processors throughout execution, and would even suppress the notion of checkpointing periods. Instead, after each failure striking a processor, its replica would checkpoint immediately, and the spare processor replacing the failed processor would read that checkpoint to resume execution. There is a small risk of fatal crash if a second failure should strike the replica when writing its checkpoint, but (i) the risk is very small because the probability of such a cascade of two narrowly spaced failures is quite low; and (ii) if the checkpoint protocol is scalable, every other processor can checkpoint in parallel with the replica, and there is no additional time overhead. With tightly coupled applications, the other processors would likely have to wait until the spare is able to restart, and they can checkpoint instead of idling during that wait. While intuitively appealing, the *restart-on-failure* strategy may lead to too many checkpoints and restarts, especially in scenarios when failures strike frequently. However, frequent failures were exactly the reason to deploy replication in the first place, precisely to avoid having to restart after each failure.

In this work, we introduce the *restart* strategy, which requires any failed processor to recover each time a checkpoint is taken. This ensures that after any checkpoint at the

end of a successful period, all processors are alive. This is a middle ground between the *no-restart* and *restart-on-failure* strategies, because failed processors are restarted at the end of each period with *restart*. On the one hand, a given period may well include many failures, hence *restart* restarts processors less frequently than *restart-on-failure*. On the other hand, there will be several periods in between two application crashes, hence *restart* restarts processors more frequently than *no-restart*.

Periodic checkpointing is optimal with the *restart* strategy: the next period should have same length as the previous one, because we have the same initial conditions at the beginning of each period. Restarting failed processors when checkpointing can introduce additional overhead, but we show that it is very small, and even non-existent when in-memory (a.k.a. buddy) checkpointing is used as the first-level of a hierarchical multi-level checkpointing protocol (such state-of-the-art protocols are routinely deployed on large-scale platforms [140, 17, 40]). A key contribution of this chapter is a mathematical analysis of the *restart* strategy, with a closed-form formula for its optimal checkpointing period. We show that the optimal checkpointing period for the *restart* strategy has the order $\Theta(\mu^{\frac{2}{3}})$, instead of the $\Theta(\mu^{\frac{1}{2}})$ used in previous works for *no-restart* as an extension of the Young/Daly formula [82, 46, 110]. Hence, as the error rate increases, the optimal period becomes much longer than the value that has been used in all previous works (with *no-restart*). Consequently, checkpoints are much less frequent, thereby dramatically decreasing the pressure on the I/O system.

The main contributions in this chapter are the following:

• We provide the first closed-form expression of the application MTTI M_N with replication;

• We introduce the *restart* strategy for replication, where we recover failed processors during each checkpoint;

• We formally analyze the *restart* strategy, and provide the optimal checkpointing period with this strategy;

• We apply these results to applications following Amdahl's law, i.e., applications that are not fully parallel but have an inherent sequential part, and compare the time-to-solution achieved with and without replication;

• We validate the model through comprehensive simulations, by showing that analytical results, using first-order approximations and making some additional assumptions (no failures during checkpoint and recovery), are quite close to simulation results; for these simulations, we use both randomly generated failures and log traces.

• We compare through simulations the overhead obtained with the optimal strategy introduced in this work (*restart* strategy, optimal checkpointing period) to those used in all previous works (*no-restart* strategy, extension of the Young/Daly checkpointing period), as well as with strategies that use partial replication or that restart only at some of the checkpoints, and demonstrate that we can significantly decrease both total execution time and utilization of the I/O file system.

The chapter is organized as follows. We first describe the model in Section 7.2. We recall how to compute the optimal checkpointing period when no replication is used in Section 7.3. The core contribution is presented in Section 7.4, where we explain how

to compute the MTTI with b ($= \frac{N}{2}$) processor pairs, detail the *restart* strategy, and show how to derive the optimal checkpointing period with this *restart* strategy. Results are applied to applications following Amdahl's law in Section 7.5. An asymptotic analysis of *no-restart* and *restart* is provided in Section 7.6. The experimental evaluation in Section 7.7 presents extensive simulation results, demonstrating that replication is indeed *more efficient than you think*, when enforcing the *restart* strategy instead of the *no-restart* strategy. Finally, we extend the analysis and experiments to energy consumption in Section 7.8, and conclude in Section 7.9.

7.2 Model

This section describes the model, with an emphasis on the cost of a combined checkpoint-restart operation.

Fail-stop errors. Throughout the text, we consider a platform with N identical processors. The platform is subject to fail-stop errors, or failures, that interrupt the application. Similarly to previous work [82, 72, 110], for the mathematical analysis, we assume that errors are independent and identically distributed (IID), and that they strike each processor according to an exponential probability distribution $\exp(\lambda)$ with support $[0, \infty)$, probability density function (PDF) $f(t) = \lambda e^{-\lambda t}$ and cumulative distribution function (CDF) $F(T) = \mathbb{P}(X \leq T) = 1 - e^{-\lambda T}$. We also introduce the reliability function $G(T) = 1 - F(T) = e^{-\lambda T}$. The expected value $\mu = \frac{1}{\lambda}$ of the $\exp(\lambda)$ distribution is the MTBF on one processor. We lift the IID assumption in the performance evaluation section by using trace logs from real platforms.

Checkpointing. To cope with errors, we use periodic coordinated checkpointing. We assume that the divisible application executes for a very long time (asymptotically infinite) and we partition the execution into periods. Each period $\mathcal{P}$ consists of a work segment of duration T followed by a checkpoint of duration C. After an error, there is a downtime of duration D (corresponding to the time needed to migrate to a spare processor), a recovery of size R, and then one needs to re-execute the period from its beginning.

Replication. We use another fault tolerance technique, namely replication. Each process has a replica, which follows the exact same states in its execution. To ensure this, when a process receives a message, its replica also receives the same message, and messages are delivered in the same order to the application (an approach called *active* replication; see [92, 82]). If a crash hits a process at any time, and its replica is still alive, the replica continues the execution alone until a new process can replace the dead one.

We rely on the traditional process allocation strategy that assigns processes and their replicas on remote parts of the system (typically different racks) [34]. This strat-

egy mitigates the risk that a process and its replica would both fail within a short time interval (much shorter than the expected MTTI). As stated in [158], when failure correlations are observed, their correlation diminishes when the processes are far away from each other in the memory hierarchy, and becomes undistinguishable from the null hypothesis (no correlation) when processes belong to different racks.

Combined checkpoint-restart. In this chapter, we propose the *restart* strategy where failed processes are restarted as soon as the next checkpoint wave happens. When that happens, and processes need to be restarted, the cost of a checkpoint and restart wave, C^R, is then increased: one instance of each surviving process must save their state, then processes for the missing instances of the replicas must be allocated; the new processes must load the current state, which has been checkpointed, and join the system to start acting as a replica. The first part of the restart operation, allocating processes to replace the failed ones, can be managed in parallel with the checkpoint of the surviving processes. Using spare processes, this allocation time can be very small and we will consider it negligible compared to the checkpoint saving and loading times. Similarly, integrating the newly spawned process inside the communication system when using spares is negligible when using mechanisms such as the ones described in [29].

There is a large variety of checkpointing libraries and approaches to help applications save their state. [140, 17, 40] are typically used in HPC systems for coordinated checkpointing, and use the entire memory hierarchy to speed up the checkpointing cost: the checkpoint is first saved on local memory, then uploaded onto local storage (SSD, NVRAM if available), and eventually to the shared file system. As soon as a copy of the state is available on the closest memory, the checkpoint is considered as taken. Loading that checkpoint requires that the application state from the closest memory be sent to the memory of the new hosting process.

Another efficient approach to checkpoint is to use in-memory checkpoint replication using the memory of a 'buddy' process (see [204, 146]). To manage the risk of losing the checkpoint in case of failure of two buddy processes, the checkpoint must also be saved on reliable media, as is done in the approaches above. Importantly, in-memory checkpointing is particularly fitted for the *restart* strategy, because the buddy process and the replica are the same process: in that case, the surviving processes upload their checkpoint directly onto the memory of the newly spawned replicas; as soon as this communication is done, the processes can continue working. Contrary to traditional buddy checkpointing, it is not necessary to exchange the checkpoints between a pair of surviving buddies since, per the replication technique, both checkpoints are identical.

In the worst case, if a sequential approach is used, combining checkpointing and restart takes at most twice the time to checkpoint only; in the best case, using buddy checkpointing, the overhead of adding the restart to the checkpoint is negligible. We consider the full spectrum $C \leq C^R \leq 2C$ in the simulations.

As discussed in [148, 82], checkpoint time varies significantly depending upon the

target application and the hardware capabilities. We will consider a time to checkpoint within two reasonable limits: $60s \leq C \leq 600s$, following [110].

First-order approximation. Throughout the chapter, we are interested in first-order approximations, because exact formulas are not analytically tractable. We carefully state the underlying hypotheses that are needed to enforce the validity of first-order results. Basically, the first-order approximation will be the first, and most meaningful, term of the Taylor expansion of the overhead occurring every period when the error rate λ tends to zero.

7.3 Background

In this section, we briefly summarize well-known results on the optimal checkpointing period when replication is not used, starting with a single processor in Section 7.3.1, and then generalizing to the case with N processors in Section 7.3.2.

7.3.1 With a Single Processor

We aim at computing the expected time $\mathbb{E}(T)$ to execute a period of length $\mathcal{P} = T + C$. The optimal period length will be obtained for the value of T, minimizing the overhead

$$\mathbb{H}(T) = \frac{\mathbb{E}(T)}{T} - 1. \tag{7.1}$$

We temporarily assume that fail-stop errors strike only during work T and not during checkpoint C nor recovery R. The following recursive equation is the key to most derivations:

$$\mathbb{E}(T) = (1 - F(T))(T + C) + F(T)(T_{\text{lost}}(T) + D + R + \mathbb{E}(T)). \tag{7.2}$$

Equation (7.2) reads as follows: with probability $1 - F(T)$, the execution is successful and lasts $T + C$ seconds; with probability $F(T)$, an error strikes before completion, and we need to account for time lost $T_{\text{lost}}(T)$, downtime D and recovery R before starting the computation anew. The expression for $T_{\text{lost}}(T)$ is the following:

$$T_{\text{lost}}(T) = \int_0^\infty t\mathbb{P}(X = t | X \leq T)dt = \frac{1}{F(T)} \int_0^T tf(t)dt.$$

Integrating by parts and re-arranging terms in Equation (7.2), we derive

$$\mathbb{E}(T) = T + C + \frac{F(T)}{1 - F(T)}(T_{\text{lost}}(T) + D + R)$$

and $\mathbb{H}(T) = \frac{C}{T} + \frac{F(T)}{T(1-F(T))}(D + R) + \frac{\int_0^T G(t)dt}{T(1-F(T))} - 1$. Now, if we instantiate the value of

$F(T) = 1 - G(T) = 1 - e^{-\lambda T}$, we obtain

$$\mathbb{H}(T) = \frac{C}{T} + \frac{e^{\lambda T} - 1}{T}(D + R + \frac{1}{\lambda}) - 1.$$

We can find the value T_{opt} by differentiating and searching for the zero of the derivative, but the solution is complicated as it involves the Lambert function [58, 106]. Instead, we use the Taylor expansion of $e^{-\lambda T} = \sum_{i=0}^{\infty}(-1)^i \frac{(\lambda T)^i}{i!}$ and the approximation $e^{-\lambda T} = 1 - \lambda T + \frac{(\lambda T)^2}{2} + o(\lambda^2 T^2)$. This makes sense only if λT tends to zero. It is reasonable to make this assumption, since the length of the period $\mathcal{P}$ must be much smaller than the error MTBF $\mu = \frac{1}{\lambda}$. Hence, we look for $T = \Theta(\lambda^{-x})$, where $0 < x < 1$. Note that x represents the order of magnitude of T as a function of the error rate λ. We can then safely write

$$\mathbb{H}(T) = \frac{C}{T} + \frac{\lambda T}{2} + o(\lambda T). \tag{7.3}$$

Now, $\frac{C}{T} = \Theta(\lambda^x)$ and $\frac{\lambda T}{2} = \Theta(\lambda^{1-x})$, hence the order of magnitude of the overhead is $\mathbb{H}(T) = \Theta(\lambda^{\max(x,1-x)})$, which is minimum for $x = \frac{1}{2}$. Differentiating Equation (7.3), we obtain

$$T_{opt} = \sqrt{\frac{2C}{\lambda}} = \Theta(\lambda^{-\frac{1}{2}}), \text{ and } \mathbb{H}_{opt} = \sqrt{2C\lambda} + o(\lambda^{\frac{1}{2}}) = \Theta(\lambda^{\frac{1}{2}}) \tag{7.4}$$

which is the well-known and original Young formula [200].

Variants of Equation (7.4) have been proposed in the literature, such as $T_{opt} = \sqrt{2(\mu + R)C}$ in [58] or $T_{opt} = \sqrt{2(\mu - D - R)C} - C$ in [106]. All variants are approximations that collapse to Equation (7.4). This is because the resilience parameters C, D, and R are constants and thus negligible in front of T_{opt} when λ tends to zero. This also explains that assuming that fail-stop errors may strike during checkpoint or recovery has no impact on the first-order approximation of the period given in Equation (7.4). For instance, assuming that fail-stop errors strike during checkpoints, we would modify Equation (7.2) into

$$\mathbb{E}(T + C) = (1 - F(T + C))(T + C) + F(T + C)(T_{\text{lost}}(T + C) + D + R + \mathbb{E}(T + C))$$

and derive the same result as in Equation (7.4). Similarly, assuming that fail-stop errors strike during recovery, we would replace R with $\mathbb{E}(R)$, which can be computed via an equation similar to that for $\mathbb{E}(T)$, again without modifying the final result.

Finally, a very intuitive way to retrieve Equation (7.4) is the following: consider a period of length $\mathcal{P} = T + C$. There is a failure-free overhead $\frac{C}{T}$, and a failure-induced overhead $\frac{1}{\mu} \times \frac{T}{2}$, because with frequency $\frac{1}{\mu}$ an error strikes, and on average it strikes in the middle of the period and we lose half of it. Adding up both overhead sources gives

$$\frac{C}{T} + \frac{T}{2\mu}, \tag{7.5}$$

which is minimum when $T = \sqrt{2\mu C}$. While not fully rigorous, this derivation helps understand the tradeoff related to the optimal checkpointing frequency.

7.3.2 With N Processors

The previous analysis can be directly extended to multiple processors. Indeed, if fail-stop errors strike each processor according to an $\exp(\lambda)$ probability distribution, then these errors strike the whole platform made of N identical processors according to an $\exp(N\lambda)$ probability distribution [106]. In other words, the platform MTBF is $\mu_N = \frac{\mu}{N}$, which is intuitive: the number of failures increases linearly with the number of processors N, hence the mean time between two failures is divided by N. All previous derivations apply, and we obtain the optimal checkpointing period and overhead:

$$T_{opt} = \sqrt{\frac{2C}{N\lambda}} = \Theta(\lambda^{-\frac{1}{2}}), \text{ and } \mathbb{H}_{opt} = \sqrt{2CN\lambda} + o(\lambda^{\frac{1}{2}}) = \Theta(\lambda^{\frac{1}{2}}) \tag{7.6}$$

This value of T_{opt} can be intuitively retrieved with the same (not fully rigorous) reasoning as before (Equation (7.5)): in a period of length $\mathcal{P} = T + C$, the failure-free overhead is $\frac{C}{T}$, and the failure-induced overhead becomes $\frac{1}{\mu_N} \times \frac{T}{2}$: we factor in an updated value of the failure frequency, using $\frac{1}{\mu_N} = \frac{N}{\mu}$ instead of $\frac{1}{\mu}$. Both overhead sources add up to

$$\frac{C}{T} + \frac{T}{2\mu_N} = \frac{C}{T} + \frac{NT}{2\mu}, \tag{7.7}$$

which is minimum when $T = \sqrt{\frac{2\mu C}{N}}$.

7.4 Replication

This section deals with process replication for fail-stop errors, as introduced in [82] and recently revisited by [110]. We consider a platform with $N = 2b$ processors. Exactly as in Section 7.3, each processor fails according to a probability distribution $\exp(\lambda)$, and the platform MTBF is $\mu_N = \frac{\mu}{N}$. We still assume that checkpoint and recovery are error-free: it simplifies the analysis without modifying the first-order approximation of the optimal checkpointing period.

Processors are arranged by pairs, meaning that each processor has a replica. The application executes as if there were only b available processors, hence with a reduced throughput. However, a single failure does not interrupt the application, because the replica of the failed processor can continue the execution. The application can thus survive many failures, until both replicas of a given pair are struck by a failure. How many failures are needed, in expectation, to interrupt the application? We compute this value in Section 7.4.1. Then, we proceed to deriving the optimal checkpointing period, first with one processor pair in Section 7.4.2, before dealing with the general case in Section 7.4.3.

7.4.1 Computing the Mean Time To Interruption

Let $n_{\text{fail}}(2b)$ be the expected number of failures to interrupt the application, with b processor pairs. Then, the application MTTI M_{2b} with b processor pairs (hence $N = 2b$ processors) is given by

$$M_{2b} = n_{\text{fail}}(2b)\,\mu_{2b} = n_{\text{fail}}(2b)\frac{\mu}{2b} = \frac{n_{\text{fail}}(2b)}{2\lambda b}, \tag{7.8}$$

because each failure strikes every μ_{2b} seconds in expectation. Computing the value of $n_{\text{fail}}(2b)$ has received considerable attention in previous work. In [156, 82], the authors made an analogy with the birthday problem and use the Ramanujan function [84] to derive the formula $n_{\text{fail}}(2b) = 1 + \sum_{k=0}^{b} \frac{b!}{(b-k)!b;k} \approx \sqrt{\frac{\pi b}{2}}$. The analogy is not fully correct, because failures can strike either replica of a pair. A correct recursive formula is provided in [46], albeit without a closed-form expression. Recently, the authors in [110] showed that

$$n_{\text{fail}}(2b) = 2b4^b \int_0^{\frac{1}{2}} x^{b-1}(1-x)^b dx \tag{7.9}$$

but did not give a closed-form expression either. We provide such an expression below:

Theorem 10.

$$n_{fail}(2b) = 1 + 4^b \;/\; \binom{2b}{b}. \tag{7.10}$$

Proof. The integral in Equation (7.9) is known as the incomplete Beta function $B(\frac{1}{2}, b, b+1)$, where $B(z,u,v) = \int_0^z x^{u-1}(1-x)^{v-1}dx$. This incomplete Beta function is also known [190] as the hypergeometric function $B(z,u,v) = \frac{z^u}{u} \times {}_2F_1\left[\begin{smallmatrix} u, & 1-v \\ u+1 & \end{smallmatrix}; z\right]$, where

$${}_2F_1\left[\begin{smallmatrix} u, & v \\ w & \end{smallmatrix}; z\right] = \sum_{n=0}^{\infty} \frac{\langle u\rangle_n \langle v\rangle_n}{\langle w\rangle_n}\frac{z^n}{n!} = 1 + \frac{uv}{1!w}z + \frac{u(u+1)v(v+1)}{2!w}z^2 + \dots$$

We need to compute $B(\frac{1}{2}, b, b+1) = \frac{1}{b2^b} \times {}_2F_1\left[\begin{smallmatrix} b, & -b \\ b+1 & \end{smallmatrix}; \frac{1}{2}\right]$, and according to [189], we have

$${}_2F_1\begin{bmatrix} b, & -b \\ b+1 & \end{bmatrix}; \frac{1}{2}\Big] = \frac{\sqrt{\pi}\,\Gamma(b+1)}{2^{b+1}}\Big[\frac{1}{\Gamma(b+1)\Gamma(\frac{1}{2})} + \frac{1}{\Gamma(b+\frac{1}{2}))\Gamma(1)}\Big].$$

Here, Γ is the well-known Gamma function extending the factorial over real numbers: $\Gamma(z) = \int_0^\infty x^{z-1}e^{-x}dx$. We have $\Gamma(1) = 1$, $\Gamma(b+1) = b!$, $\Gamma(\frac{1}{2}) = \sqrt{\pi}$, and $\Gamma(b+\frac{1}{2}) = \frac{\sqrt{\pi}(2b)!}{4^b b!}$. Hence,

$${}_2F_1\begin{bmatrix} b, & -b \\ b+1 & \end{bmatrix}; \frac{1}{2}\Big] = \frac{1}{2^{b+1}}\Big[1 + \frac{4^b(b!)^2}{(2b)!}\Big] = \frac{1}{2^{b+1}}\Big[1 + \frac{4^b}{\binom{2b}{b}}\Big].$$

For the last equality, we observe that $\binom{2b}{b} = \frac{(2b)!}{(b!)^2}$. We derive $B(\frac{1}{2}, b, b+1) = \frac{1}{2b4^b}\left[1 + \frac{4^b}{\binom{2b}{b}}\right]$, and finally $n_{\text{fail}}(2b) = 1 + \frac{4^b}{\binom{2b}{b}}$, which concludes the proof. □

Using Sterling's formula, we easily derive that $n_{\text{fail}}(2b) \approx \sqrt{\pi b}$, which is 40% more than the value $\sqrt{\frac{\pi b}{2}}$ used in [156, 82].

Plugging the value of $n_{\text{fail}}(2b)$ back in Equation (7.8) gives the value of the MTTI M_{2b}. As already mentioned, previous works [82, 46, 110] all use the checkpointing period

$$T_{MTTI}^{\text{no}} = \sqrt{2M_{2b}C} \tag{7.11}$$

to minimize execution time overhead. This value follows from the same derivation as in Equations (7.5) and (7.7). Consider a period of length $\mathcal{P} = T + C$. The failure-free overhead is still $\frac{C}{T}$, and the failure-induced overhead becomes $\frac{1}{M_{2b}} \times \frac{T}{2}$: we factor in an updated value of the failure frequency, which now becomes the fatal failure frequency, namely $\frac{1}{M_{2b}}$. Both overhead sources add up to

$$\frac{C}{T} + \frac{T}{2M_{2b}}, \tag{7.12}$$

which is minimum when $T = \sqrt{2M_{2b}C}$.

In the following, we analyze the *restart* strategy. We start with one processor pair ($b = 1$) in Section 7.4.2, before dealing with the general case in Section 7.4.3.

7.4.2 With One Processor Pair

We consider two processors working together as replicas. The failure rate is $\lambda = \frac{1}{\mu}$ for each processor, and the pair MTBF is $\mu_2 = \frac{\mu}{2}$, while the pair MTTI is $M_2 = \frac{3\mu}{2}$ because $n_{\text{fail}}(2) = 3$. We analyze the *restart* strategy, which restarts a (potentially) failed processor at every checkpoint. Hence, the checkpoint has duration C^R and not C. Consider a period of length $\mathcal{P} = T + C^R$. If one processor fails before the checkpoint but the other survives until reaching it, the period is executed successfully. The period is re-executed only when both processors fail within T seconds. Let $p_1(T)$ denote the probability that both processors fail during T seconds: $p_1(T) = (1 - e^{-\lambda T})^2$. We compute the expected time $\mathbb{E}(T)$ for period of duration $\mathcal{P} = T + C^R$ using the following recursive equation:

$$\mathbb{E}(T) = (1 - p_1(T))(T + C^R) + p_1(T)(T_{\text{lost}}(T) + D + R + \mathbb{E}(T)). \tag{7.13}$$

Here, C^R denotes the time to checkpoint, and in addition, to recover whenever one of the two processors had failed during the period. As discussed in Section 7.2, we have $C \leq C^R \leq C + R$: the value of C^R depends upon the amount of overlap between the checkpoint and the possible recovery of one processor.

Consider the scenario where one processor fails before reaching the end of the period, while the other succeeds and takes the checkpoint. The *no-restart* strategy

continues execution, hence pays only for a regular checkpoint of cost C, and when the live processor is struck by a failure (every M_2 seconds on average), we roll back and recover for both processors [82, 46, 110]. However, the new *restart* strategy requires any failed processor to recover whenever a checkpoint is taken, hence at a cost C^R. This ensures that after any checkpoint at the end of a successful period, we have two live processors, and thus the same initial conditions. Hence, periodic checkpointing is optimal with this strategy. We compare the *restart* and *no-restart* strategies through simulations in Section 7.7.

As before, in Equation (7.13), $T_{\text{lost}}(T)$ is the average time lost, knowing that both processors have failed before T seconds. While $T_{\text{lost}}(T) \sim \frac{T}{2}$ when considering a single processor, it is no longer the case with a pair of replicas. Indeed, we compute $T_{\text{lost}}(T)$ as follows:

$$\begin{aligned} T_{\text{lost}}(T) &= \int_0^\infty t\mathbb{P}(X = t | t \leq T)dt = \frac{1}{p_1(T)} \int_0^T t\frac{d\mathbb{P}(X \leq t)}{dt}dt \\ &= \frac{2\lambda}{(1-e^{-\lambda T})^2} \int_0^T t(e^{-\lambda t} - e^{-2\lambda t})dt. \end{aligned}$$

After integration, we find that

$$T_{\text{lost}}(T) = \frac{(2e^{-2\lambda T} - 4e^{-\lambda T})\lambda T + e^{-2\lambda T} - 4e^{-\lambda T} + 3}{2\lambda(1-e^{-\lambda T})^2} = \frac{1}{2\lambda}\frac{u(\lambda T)}{v(\lambda T)},$$

with $u(y) = (2e^{-2y} - 4e^{-y})y + e^{-2y} - 4e^{-y} + 3$ and $v(y) = (1-e^{-y})^2$.

Assuming that $T = \Theta(\lambda^{-x})$ with $0 < x < 1$ as in Section 7.3.1, then Taylor expansions lead to $u(y) = \frac{4}{3}y^3 + o(y^3)$ and $v(y) = y^2 + y^3 + o(y^3)$ for $y = \lambda T = o(1)$, meaning that $T_{\text{lost}}(T) = \frac{1}{2\lambda}\frac{\frac{4\lambda T}{3} + o(\lambda T)}{1 + \lambda T + o(\lambda T)}$. Using the division rule, we obtain $T_{\text{lost}}(T) = \frac{1}{2\lambda}(\frac{4\lambda T}{3} + o(\lambda T)) = \frac{2T}{3} + o(T)$. Note that we lose two thirds of the period with a processor pair rather than one half with a single processor. Plugging back the value of $T_{\text{lost}}(T)$ and solving, we obtain:

$$\mathbb{E}(T) = T + C^R + (D + R + \frac{(2e^{-2\lambda T} - 4e^{-\lambda T})\lambda T + e^{-2\lambda T} - 4e^{-\lambda T} + 3}{2\lambda(1-e^{-\lambda T})^2}) \cdot \frac{(e^{\lambda T} - 1)^2}{2e^{\lambda T} - 1}. \tag{7.14}$$

We then compute the waste $\mathbb{H}^{\text{rs}}(T)$ of the *restart* strategy as follows:

$$\mathbb{H}^{\text{rs}}(T) = \frac{\mathbb{E}(T)}{T} - 1 = \frac{C^R}{T} + \frac{2}{3}\lambda^2 T^2 + o(\lambda^2 T^2). \tag{7.15}$$

Moreover, with $T = \Theta(\lambda^{-x})$, we have $\frac{C^R}{T} = \Theta(\lambda^x)$ and $\frac{2}{3}\lambda^2 T^2 = \Theta(\lambda^{2-2x})$, hence $\mathbb{H}^{\text{rs}}(T) = \Theta(\lambda^{\max(x, 2-2x)})$, which is minimum for $x = \frac{2}{3}$. Differentiating, we readily obtain:

$$T_{opt} = \left(\frac{3C^R}{4\lambda^2}\right)^{\frac{1}{3}} = \Theta(\lambda^{-\frac{2}{3}}), \tag{7.16}$$

$$\mathbb{H}^{\text{rs}}(T_{opt}) = \left(\frac{3C^R\lambda}{\sqrt{2}}\right)^{\frac{2}{3}} + o(\lambda^{\frac{2}{3}}) = \Theta(\lambda^{\frac{2}{3}}). \tag{7.17}$$

Note that the optimal period has the order $T_{opt} = \Theta(\lambda^{-\frac{2}{3}}) = \Theta(\mu^{\frac{2}{3}})$, while the extension $\sqrt{2M_2C}$ of the Young/Daly formula has the order $\Theta(\lambda^{-\frac{1}{2}}) = \Theta(\mu^{\frac{1}{2}})$. This means that the optimal period is much longer than the value that has been used in all previous works. This result generalizes to several processor pairs, as shown in Section 7.4.3. We further discuss asymptotic results in Section 7.6.

For an intuitive way to retrieve Equation (7.16), the derivation is similar to that used for Equations (7.5), (7.7) and (7.12). Consider a period of length $\mathcal{P} = T + C^R$. The failure-free overhead is still $\frac{C^R}{T}$, and the failure-induced overhead becomes $\frac{1}{\mu}\frac{T}{\mu} \times \frac{2T}{3}$: we factor in an updated value of the fatal failure frequency $\frac{1}{\mu}\frac{T}{\mu}$: the first failure strikes with frequency $\frac{1}{\mu}$, and then with frequency $\frac{T}{\mu}$, there is another failure before the end of the period. As for the time lost, it becomes $\frac{2T}{3}$, because in average the first error strikes at one third of the period and the second error strikes at two-third of the period: indeed, we know that there are two errors in the period, and they are equally spaced in average. Altogether, both overhead sources add up to

$$\frac{C^R}{T} + \frac{2T^2}{3\mu^2}, \tag{7.18}$$

which is exactly Equation (7.15).

We conclude this section with a comment on the *no-restart* strategy. The intuitive derivation in Equation (7.12) leads to $\mathbb{H}^{\text{no}}(T) = \frac{C}{T} + \frac{T}{2M_{2b}}$. We now understand that this derivation is accurate if we have $T_{\text{lost}}(T) = \frac{T}{2} + o(T)$. While this latter equality is proven true without replication [58], it is unknown whether it still holds with replication. Hence, computing the optimal period for *no-restart* remains an open problem, even with a single processor pair.

Going further, Figure 7.2 shows that periodic checkpointing is not optimal for *no-restart* with a single processor pair, which provides another hint of the difficulty of the problem. In the figure, we compare four approaches: in addition to Restart(T^{rs}_{opt}) and NoRestart(T^{no}_{MTTI}), we use two non-periodic variants of *no-restart*, Non-Periodic(T_1, T_2). In both variants, we use a first checkpointing period T_1 while both processors are alive, and then a shorter period T_2 as soon as one processor has been struck by a failure. When an application failure occurs, we start anew with periods of length T_1. For both variants, we only restart processors after an application failure, just as *no-restart* does. The first variant uses $T_1 = T^{\text{no}}_{MTTI} = \sqrt{3\mu C}$ (the MTTI is $M_2 = 3\frac{\mu}{2}$) and the second variant uses $T_1 = T^{\text{rs}}_{opt} = \left(\frac{3}{4}C\mu^2\right)^{\frac{1}{3}}$. We use the Young/Daly period $T_2 = \sqrt{2\mu C}$ for both variants, because there remains a single live processor when period T_2 is enforced. The figure shows the ratio of the time-to-solution for the two non-periodic approaches over that of periodic *no-restart* (with period T^{no}_{MTTI}). Note that the application is perfectly parallel, and that the only overhead is for checkpoints

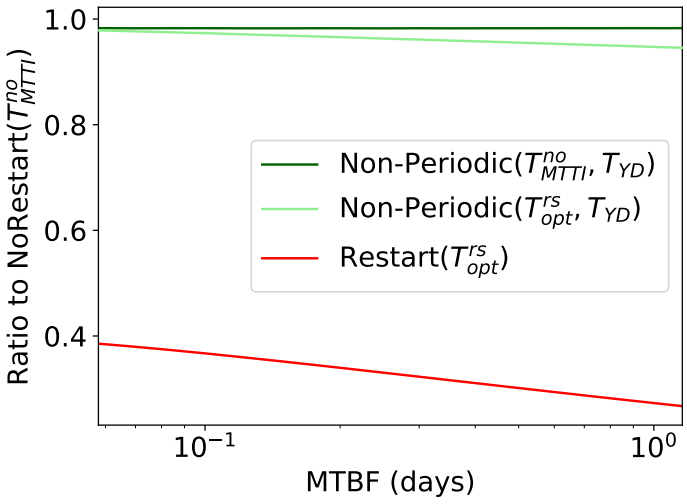


Figure 7.2: Ratio of time-to-solution of two non-periodic strategies and *restart* over time-to-solution of *no-restart* (one processor pair, $C = C^R = 60$).

and re-executions after failures. Both non-periodic variants are better than *no-restart*, the first one is within 98.3% of *no-restart*, and the second one is even better (95% of *no-restart*) when the MTBF increases. We also see that *restart* is more than twice better than *no-restart* with a single processor pair. Note that results are averaged over 100,000 simulations, each lasting for 10,000 periods, so that they are statistically guaranteed to be accurate.

7.4.3 With b Processor Pairs

For b pairs, the reasoning is the same as with one pair, but the probability of having a fatal error (both processors of a same pair failing) before the end of the period changes. Letting $p_b(T)$ be the probability of failure before time T with b pairs, we have $p_b(T) = 1 - (1 - (1 - e^{-\lambda T})^2)^b$. As a consequence, computing the exact value of $T_{\text{lost}}(T)$ becomes complicated: obtaining a compact closed-form is not easy, because we would need to expand terms using the binomial formula. Instead, we directly use the Taylor expansion of $p_b(T)$ for λT close to 0. Again, this is valid only if $T = \Theta(\lambda^{-x})$ with $x < 1$. We have $p_b(T) = 1 - (1 - (\lambda T + o(\lambda T))^2)^b = b\lambda^2 T^2 + o(\lambda^2 T^2)$ and compute $T_{\text{lost}}(T)$ with b pairs as $T_{\text{lost}}(T) = \frac{1}{p_b(T)} \int_0^T t \frac{d\mathbb{P}(X \leq t)}{dt} dt = \frac{2T}{3} + o(T)$. As before, $T_{\text{lost}}(T) \sim \frac{2T}{3}$. Also, as in Section 7.4.2, we analyze the *restart* strategy, which requires any failed processor to recover whenever a checkpoint is taken. We come back to the difference with the *no-restart* strategy after deriving the period for the *restart* strategy. We compute the expected execution time of one period: $\mathbb{E}(T) = p_b(T)\big(T_{\text{lost}}(T) + D + R + \mathbb{E}(T)\big) + (1 - p_b(T))\big(T + C^R\big) = T + \frac{2b\lambda^2 T^3}{3} + o(\lambda^2 T^3)$, and

$$\mathbb{H}^{\text{rs}}(T) = \frac{\mathbb{E}(T)}{T} - 1 = \frac{C^R}{T} + \frac{2b\lambda^2 T^2}{3} + o(\lambda^2 T^2). \tag{7.19}$$

We finally derive the expression of the optimal checkpointing period with b pairs:

$$T_{opt}^{\text{rs}} = \left(\frac{3C^R}{4b\lambda^2}\right)^{\frac{1}{3}} = \Theta(\lambda^{-\frac{2}{3}}). \tag{7.20}$$

When plugging it back in Equation (7.19), we get

$$\mathbb{H}^{\text{rs}}(T_{opt}^{\text{rs}}) = \left(\frac{3C^R\sqrt{b}\lambda}{\sqrt{2}}\right)^{\frac{2}{3}} + o(\lambda^{\frac{2}{3}}) = \Theta(\lambda^{\frac{2}{3}}). \tag{7.21}$$

for the optimal overhead when using b pairs of processors.

The derivation is very similar to the case with a single pair, and the result is essentially the same, up to factoring in the number of pairs to account for a higher failure rate. However, the difference between the *no-restart* and the *restart* strategies gets more important. Indeed, with the *no-restart* strategy, several pairs can be struck once (and even several times if the failures always strike the failed processor) before a pair finally gets both its processors killed. While the *no-restart* strategy spares the cost of several restarts, it runs at risk with periods whose length has been estimated à la Young/Daly, thereby assuming an identical setting at the beginning of each period.

Finally, for the intuitive way to retrieve Equation (7.20), it goes as for Equation (7.18), multiplying the frequency of fatal failures $\frac{1}{\mu}\frac{T}{\mu}$ by a factor b to account for each of the b pairs possibly experiencing a fatal failure.

7.5 Time-To-Solution

So far, we have focused on period length. In this section, we move to actual work achieved by the application. Following [110], we account for two sources of overhead for the application. First, the application is not perfectly parallel and obeys Amdahl's law [3], which limits its parallel speedup. Second, there is an intrinsic slowdown due to active replication related to duplicating every application message [82, 110].

First, for applications following Amdahl's law, the total time spent to compute W units of computation with N processors is $T_{Amdahl} = \gamma W + (1-\gamma)\frac{W}{N} = (\gamma + \frac{1-\gamma}{N})W$, where γ is the proportion of inherently sequential tasks. When replication is used, this time becomes $T_{Amdahl} = (\gamma + \frac{2(1-\gamma)}{N})W$. Following [110], we use $\gamma = 10^{-5}$ in Section 7.7. Second, as stated in [82, 110], another slowdown related to active replication and its incurred increase of communications writes $T_{rep} = (1+\alpha)T_{Amdahl}$, where α is some parameter depending upon the application and the replication library. Following [110], we use either $\alpha = 0$ or $\alpha = 0.2$ in Section 7.7.

All in all, once we have derived T_{opt}, the optimal period between two checkpoints without replication (see Equation (7.6)), and T_{opt}^{rs}, the optimal period between two checkpoints with replication and *restart* (see Equation (7.20)), we are able to compute the optimal number of operations to be executed by an application be-

tween two checkpoints as $W_{opt} = \frac{T_{opt}}{\left(\gamma+\frac{1-\gamma}{N}\right)}$ for an application without replication, and $W^{\text{rs}}_{opt} = \frac{T^{\text{rs}}_{opt}}{(1+\alpha)\left(\gamma+\frac{1-\gamma}{b}\right)} = \frac{T^{\text{rs}}_{opt}}{(1+\alpha)\left(\gamma+\frac{2(1-\gamma)}{N}\right)}$ for an application with replication and the *restart* strategy. Finally, for the *no-restart* strategy, using T^{no}_{MTTI} (see Equation (7.11)), the number of operations becomes $W^{\text{no}}_{MTTI} = \frac{T^{\text{no}}_{MTTI}}{(1+\alpha)\left(\gamma+\frac{1-\gamma}{b}\right)} = \frac{T^{\text{no}}_{MTTI}}{(1+\alpha)\left(\gamma+\frac{2(1-\gamma)}{N}\right)}$.

To compute the actual time-to-solution, assume that we have a total of W_{seq} operations to do. With one processor, the execution time is $T_{seq} = W_{seq}$ (assuming unit execution speed). With N processors working in parallel (no replication), the failure-free execution time is $T_{par} = (\gamma + \frac{1-\gamma}{N})T_{seq}$. Since we partition the execution into periods of length T, meaning that we have $\frac{T_{par}}{T}$ periods overall, the time-to-solution is $T_{final} = \frac{T_{par}}{T}\mathbb{E}(T) = T_{par}(\mathbb{H}(T) + 1)$, hence

$$T_{final} = \left(\gamma + \frac{1-\gamma}{N}\right)(\mathbb{H}(T) + 1)T_{seq}. \tag{7.22}$$

If we use replication with b pairs of processors (i.e., $\frac{N}{2}$ pairs) instead, the difference is that $T_{par} = (1+\alpha)\left(\gamma + \frac{1-\gamma}{b}\right)T_{seq}$, hence

$$T_{final} = (1+\alpha)\left(\gamma + \frac{2(1-\gamma)}{N}\right)\left(\gamma + \frac{2(1-\gamma)}{N}\right)(\mathbb{H}(T) + 1)T_{seq}. \tag{7.23}$$

Without replication, we use the optimal period $T = T_{opt}$. For the *restart* strategy, we use the optimal period $T = T^{\text{rs}}_{opt}$, and for *no-restart*, we use $T = T^{\text{no}}_{MTTI}$, as stated above.

7.6 Asymptotic Behavior

In this section, we compare the *restart* and *no-restart* strategies asymptotically. Both approaches (and, as far as we know, all coordinated rollback-recovery approaches) are subject to a design constraint: if the time between two restarts becomes of same magnitude as the time to take a checkpoint, the application cannot progress. Therefore, when evaluating the asymptotic behavior (i.e., when the number of nodes tends to infinity, and hence the MTTI tends to 0), a first consideration is to state that none of these techniques can support infinite growth, under the assumption that the checkpoint time remains constant and that the MTTI decreases with scale. Still, in that case, because the *restart* approach has a much longer checkpointing period than *no-restart*, it will provide progress for lower MTTIs (and same checkpointing cost).

However, we can (optimistically) assume that checkpointing technology will evolve, and that rollback-recovery protocols will be allowed to scale infinitely, because the checkpoint time will remain a fraction of the MTTI. In that case, assume that with any number N of processors, we have $C = xM_N$ for some small constant $x < 1$ (where M_N is the MTTI with N processors). Consider a parallel and replicated application

that would take a time T_{app} to complete without failures (and with no fault-tolerance overheads). We compute the ratio $\mathcal{R}$, which is the expected time-to-solution using the *restart* strategy divided by the expected time-to-solution using the *no-restart* strategy:

$$\mathcal{R} = \frac{(\mathbb{H}^{\text{rs}}(T^{\text{rs}}_{opt}) + 1)T_{app}}{(\mathbb{H}^{\text{no}}(T^{\text{no}}_{MTTI}) + 1)T_{app}} = \frac{\sqrt[3]{\frac{9}{8}\pi x^2} + 1}{\sqrt{2x} + 1}.$$

Because of the assumption $C = xM_N$, both the number of nodes N and the MTBF μ simplify out in the above ratio. Under this assumption, the *restart* strategy is up to 8.4% faster than the *no-restart* strategy if x is within the range $[0, 0.64]$, i.e., as long as the checkpoint time takes less than 2/3 of the MTTI.

In the next section, we consider realistic parameters to evaluate the performance of various strategies through simulations, and we also provide results when increasing the number of processors N or reducing the MTBF.

7.7 Experimental Evaluation

In this section, we evaluate the performance of the *no-restart* and *restart* strategies through simulations. Our simulator is publicly available [24] so that interested readers can instantiate their preferred scenarios and repeat the same simulations for reproducibility purpose. The code is written in-house in C++ and does not use any library other than the Standard Template Library (STL).

We compare different instances of the models presented above. We let *Restart*(T) denote the *restart* strategy with checkpointing period T, and *NoRestart*(T) denote the *no-restart* strategy with checkpointing period T. In most figures, we present the overhead as given by Equation (7.1): it is a *relative* time overhead, that represents the time spent tolerating failures divided by the duration of the protected application. Recall previously introduced notations:

• For *Restart*(T), the overhead $\mathbb{H}^{\text{rs}}(T)$ is predicted by the model according to Equation (7.19);

• For *NoRestart*(T), the overhead $\mathbb{H}^{\text{no}}(T)$ is estimated in the literature according to Equation (7.12);

• T^{rs}_{opt} denotes the optimal period for minimizing the time overhead for the *restart* strategy, as computed in Equation (7.20);

• T^{no}_{MTTI} from Equation (7.11) is the standard period used in the literature for the *no-restart* strategy, after an analogy with the Young/Daly formula.

The *no-restart* strategy with overhead $\mathbb{H}^{\text{no}}(T^{\text{no}}_{MTTI})$ represents the state of the art for full replication [82]. For completeness, we also compare the *no-restart* and *restart* strategies with several levels of partial replication [72, 110].

We describe the simulation setup in Section 7.7.1. We assess the accuracy of our model and of first-order approximations in Section 7.7.2. We compare the performance of *restart* with *restart-on-failure* in Section 7.7.3. In Section 7.7.4, we show the impact of key parameters on the difference between the checkpointing periods of the *no-restart* and *restart* strategies, and on the associated time overheads. Section 7.7.5 discusses the

impact of the different strategies on I/O pressure. Section 7.7.6 investigates in which scenarios a smaller time-to-solution can be achieved with full or partial replication. Section 7.7.7 explores strategies that restart after a given number of failures.

7.7.1 Simulation Setup

To evaluate the performance of the *no-restart* and *restart* strategies, we use a publicly available simulator [24] that generates random failures following an exponential probability distribution with a given mean time between individual node failures and number of processor pairs. Then, we set the checkpointing period, and checkpointing cost. Default values are chosen to correspond to the values used in [110], and are defined as follows. For the checkpointing cost, we consider two default values: $C = 60$ seconds corresponds to buddy checkpointing, and $C = 600$ seconds corresponds to checkpointing on remote storage. We let the MTBF of an individual node be $\mu = 5$ years, and we use $N = 200,000$, hence having $b = 100,000$ pairs when replication is used. We then simulate the execution of an application lasting for 100 periods (total execution time $100T$) and we average the results on 1000 runs. We measure two main quantities: time overhead and optimal period length. For simplicity, we always assume that $R = C$, i.e., read and write operations take (approximately) the same time. We cover the whole range of possible values for C^R, using either $C, 1.5C$ or $2C$. This will show the impact of overlapping checkpoint and processor restart.

7.7.2 Model Accuracy

Figure 7.3 compares three different ways of estimating the time overhead of an application running on $b = 10^5$ processor pairs. Solid lines are measurements from the simulations, while dashed lines are theoretical values. The red color is for $Restart(T^{\text{rs}}_{opt})$, the blue color is for $Restart(T^{\text{no}}_{MTTI})$ and the green color is for $NoRestart(T^{\text{no}}_{MTTI})$. For the *restart* strategy, $C^R = C$ in this figure.

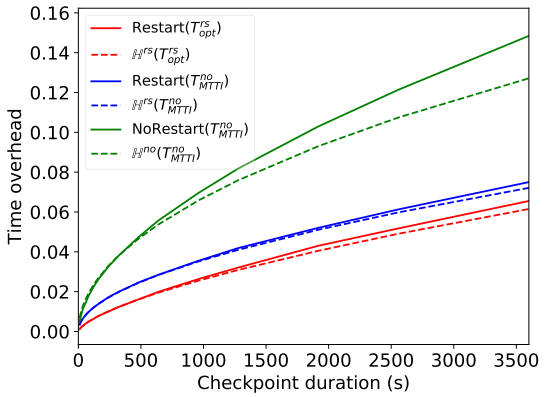


Figure 7.3: Evaluation of model accuracy for time overhead. $\mu = 5$ years, $b = 10^5$.

For the *restart* strategy, the results from simulation match the results from the theory quite accurately. Because our formula is an approximation valid when $T \gg C$, the difference between simulated time overhead and $\mathbb{H}^{\text{rs}}(T^{\text{rs}}_{opt})$ slightly increases when the checkpointing cost becomes greater than 1500 seconds. We also verify that $Restart(T^{\text{rs}}_{opt})$ has smaller overhead than $Restart(T^{\text{no}}_{MTTI})$ in the simulations, which nicely corroborates the model.

We also see that $\mathbb{H}^{\text{no}}(T^{\text{no}}_{MTTI})$ is a good estimate of the actual simulated overhead of *NoRestart*(T^{no}_{MTTI}) only for $C < 500$. Larger values of C induce a significant deviation between the prediction and the simulation. Values given by $\mathbb{H}^{\text{no}}(T)$ underestimate the overheads for lower values of C more than $\mathbb{H}^{\text{rs}}(T)$, even when using the same T^{no}_{MTTI} period to checkpoint. As described at the end of Section 7.4.1, the $\mathbb{H}^{\text{no}}(T)$ formula is an approximation whose accuracy is unknown, and when C scales up, some elements that were neglected by the approximation become significant. The formula for T^{rs}_{opt}, on the contrary, remains accurate for higher values of C.

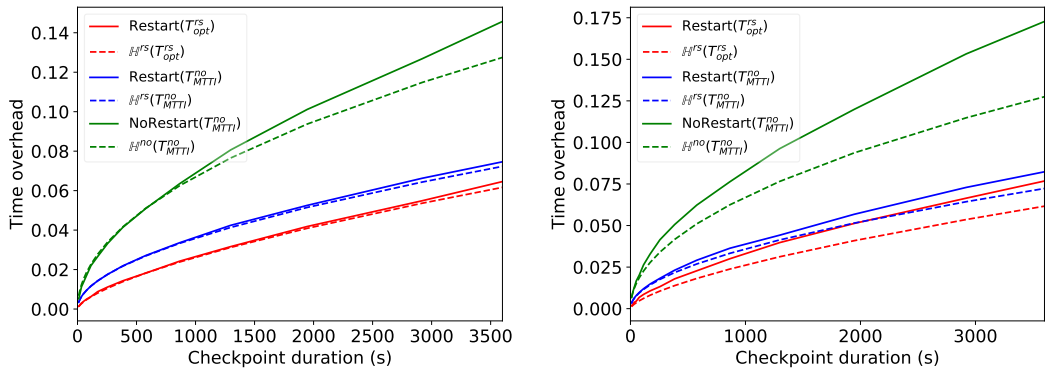


Figure 7.4: Evaluation of model accuracy for time overhead with two trace logs (LANL#18 on the left, and LANL#2 on the right).

Figure 7.4 is the exact counterpart of Figure 7.3 when using log traces from real platforms instead of randomly generated failures with an exponential distribution. We use the two traces featuring the largest number of failures from the LANL archive [125, 120], namely LANL#2 and LANL#18. According to the detailed study in [12], failures in LANL#18 are not correlated while those in LANL#2 are correlated, providing perfect candidates to experimentally study the impact of failure distributions. LANL#2 has an MTBF of 14.1 hours and is composed of 5350 failures, while LANL#18 has an MTBF of 7.5 hours and is composed of 3899 failures. For the sake of comparing with Figure 7.3 that used a processor MTBF of 5 years (and an exponential distribution), we scale both traces as follows:

- We target a platform of 200,000 processors with an individual MTBF of 5 years. Thus the global platform MTBF needs to be 64 times smaller than the MTBF of LANL#2, and 32 times smaller than the MTBF of LANL#18. Hence we partition the global platform into 64 groups (of 3,125 processors) for LANL#2, and into 32 groups (of 6,250 processors) for LANL#18;
- Within each group, the trace is rotated around a randomly chosen date, so that each trace starts independently;
- We generate 200 sets of failures for each experiment and report the average time overhead.

We observe similar results in Figure 7.3 and Figure 7.4. For LANL#18, the exper-

imental results are quite close to the model. For LANL#2, the model is slightly less accurate because of some severely degraded intervals with failure cascades. However, the *restart* strategy still grants lower time overheads than the *no-restart* strategy. For an exponential distribution, only 15% of the runs where an application failure was experienced did experience two or more failures. This ratio increases to 20% for LANL#18 and reaches 50% for LANL#2; this leads to a higher overhead than estimated for IID failures, but this is true for all strategies, and *restart* remains the best one.

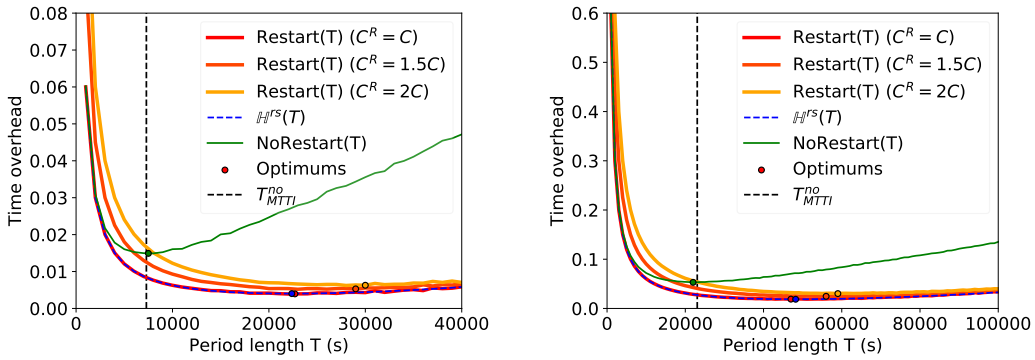


Figure 7.5: Time overhead as a function of the checkpointing period T for $C = 60$ seconds (left) or $C = 600$ seconds (right), MTBF of 5 years, IID failures and $b = 10^5$ processor pairs.

Next, on both graphs in Figure 7.5, we present the details of the evolution of the time overhead as a function of the period length for $C = 60$s and $C = 600$s. Here, we compare the overhead of the *restart* strategy obtained through simulations (solid red, orange and yellow lines for different values of C^R), the overhead of the *restart* strategy obtained through the theoretical model with C^R =C (dashed blue line), and the overhead of the *no-restart* strategy obtained through simulations (solid green line). In each case, a circle denotes the optimal period, while T^{no}_{MTTI} (the MTTI extension of the Young/Daly formula for *no-restart*) is shown with a vertical bar.

$\mathbb{H}^{\text{rs}}(T^{\text{rs}}_{opt})$ perfectly matches the behavior of the simulations, and the optimal value is very close to the one found through simulations. The simulated overhead of $NoRestart(T)$ is always larger than for $Restart(T)$, with a significant difference as T increases. Surprisingly, the optimal value for the simulated overhead of $NoRestart(T)$ is obtained for a value of T close to T^{no}_{MTTI}, which shows *a posteriori* that the approximation worked out pretty well in this scenario. The figure also shows that the *restart* strategy is much more robust than the *no-restart* one: in all cases, $Restart(T)$ provides a lower overhead than $NoRestart(T)$ throughout the spectrum, even when $C^R = 2C$. More importantly, this overhead remains close to the minimum for a large range of values of T: when $C^R = C = 60s$, for values of T between 21,000s and 25,000s, the overhead remains between 0.39% (the optimal), and 0.41%. If we take the same tolerance (overhead increased by 5%), the checkpointing period must be between 6,000s

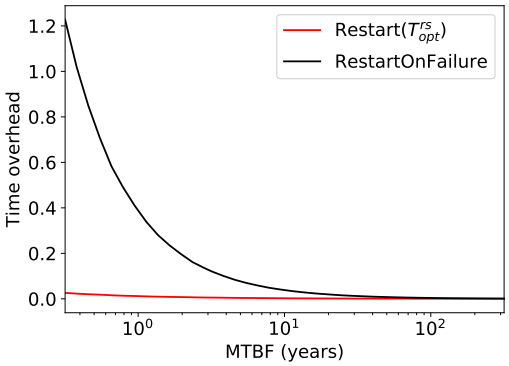


Figure 7.6: Comparison with *restart-on-failure*.

and 9,000s, thus a range that is 1/3rd larger than for the *restart* strategy. When considering $C^R = C = 600s$, this range is 18,000s (40,000s to 58,000s) for the *restart* strategy, and 7,000s (22,000s to 29,000s) for the *no-restart* one. This means that a user has a much higher chance of obtaining close-to-optimum performance by using the *restart* strategy than if she was relying on the *no-restart* one, even if some key parameters that are used to derive T^{rs}_{opt} are mis-evaluated. If $C^R = 1.5C$ or $C^R = 2C$, the same trends are observed: the optimal values are obtained for longer periods, but they remain similar in all cases, and significantly lower than for the *no-restart* strategy. Moreover, the figures show the same plateau effect around the optimal, which makes the *restart* strategy robust.

7.7.3 *Restart-on-failure*

Figures 7.3 to 7.5 showed that the *restart* strategy is more efficient than the *no-restart* one. Intuitively, this is due to the rejuvenation introduced by the periodical restarts: when reaching the end of a period, failed processes are restarted, even if the application could continue progressing in a more risky configuration. A natural extension would be to consider the *restart-on-failure* strategy described in Section 7.1. This is the scenario evaluated in Figure 7.6: we compare the time overhead of $\mathit{Restart}(T^{\text{rs}}_{opt})$ with *restart-on-failure*, which restarts each processor after each failure.

Compared to $\mathit{Restart}(T^{\text{no}}_{MTTI})$, the *restart-on-failure* strategy grants a significantly higher overhead that quickly grows to high values as the MTBF decreases. The *restart-on-failure* strategy works as designed: no rollback was ever needed, for any of the simulations (i.e., failures never hit a pair of replicated processors within the time needed to checkpoint). However, the time spent checkpointing after each failure quickly dominates the execution. This reflects the issue with this strategy, and the benefit of combined replication and checkpointing: as failures hit the system, it is necessary for performance to let processors fail and the system absorb most of the failures

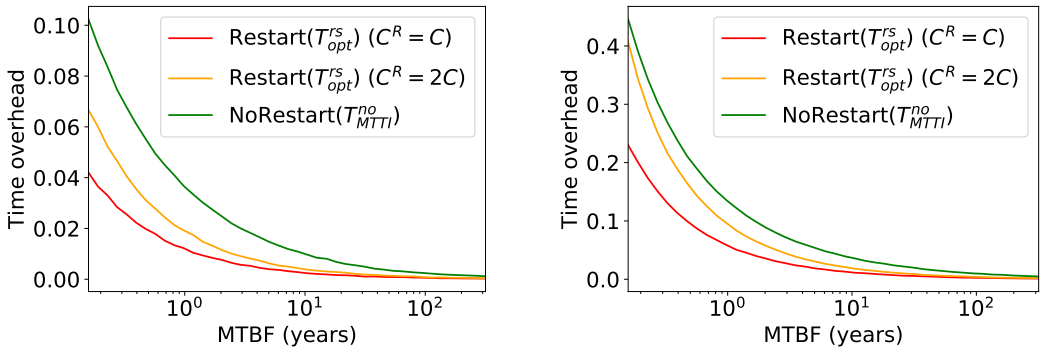


Figure 7.7: Time overhead as a function of MTBF, with $C = 60$s (left) or $C = 600$s (right), $b = 10^5$ processor pairs.

using the replicates. Combining this result with Figure 7.5, we see that it is critical for performance to find the optimal rejuvenation period: restarting failed processes too frequently is detrimental to performance, as is restarting them too infrequently.

7.7.4 Impact of Parameters

The graphs in Figure 7.7 describe the impact of the individual MTBF of the processors on the time overhead. We compare $Restart(T^{\text{rs}}_{opt})$, $Restart(T^{\text{no}}_{MTTI})$ (both in the most optimistic case when $C^R = C$ and in the least optimistic case when $C^R = 2C$) and $NoRestart(T^{\text{no}}_{MTTI})$. As expected, when C^R increases, the time overhead increases. However, even in the case $C^R = 2C$, both *restart* strategies outperform the *no-restart* strategy. As the MTBF increases, the overhead of all strategies tends to be negligible, since a long MTBF has the cumulated effect that the checkpointing period increases and the risk of needing to re-execute decreases. The longer the checkpoint time C, the higher the overheads, which is to be expected; more interestingly, with higher C, the *restart* strategy needs C^R to remain close to C to keep its advantage against the *no-restart* strategy. This advocates for a buddy checkpointing approach with *restart* strategy when considering replication and checkpointing over unreliable platforms.

7.7.5 I/O Pressure

Figure 7.8 reports the difference between T^{rs}_{opt} and T^{no}_{MTTI}. We see that T^{rs}_{opt} increases faster than T^{no}_{MTTI} when the MTBF decreases. This is due to the fact that the processors are restarted at each checkpoint, hence reducing the probability of failure for each period; it mainly means that using the *restart* strategy (i) decreases the total application time, and (ii) decreases the I/O congestion in the machine, since checkpoints are less frequent. This second property is critical for machines where a large number of applications are running concurrently, and for which, with high probability, the checkpoint

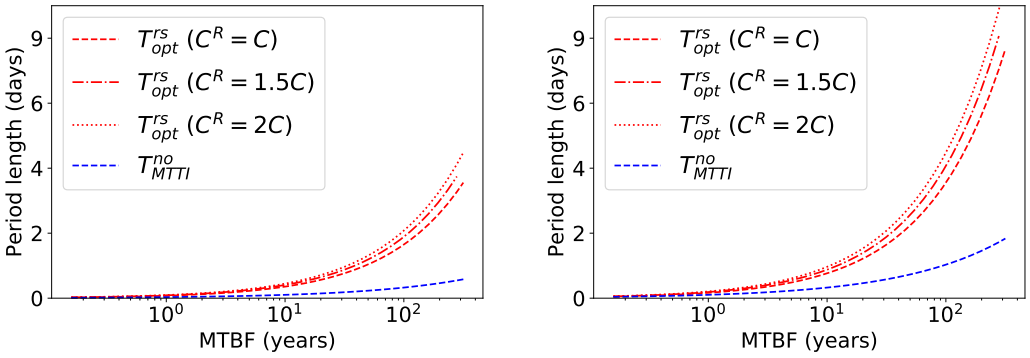


Figure 7.8: Period length T as function of MTBF, with $C = 60$s (left) or $C = 600$s (right), $b = 10^5$ processor pairs.

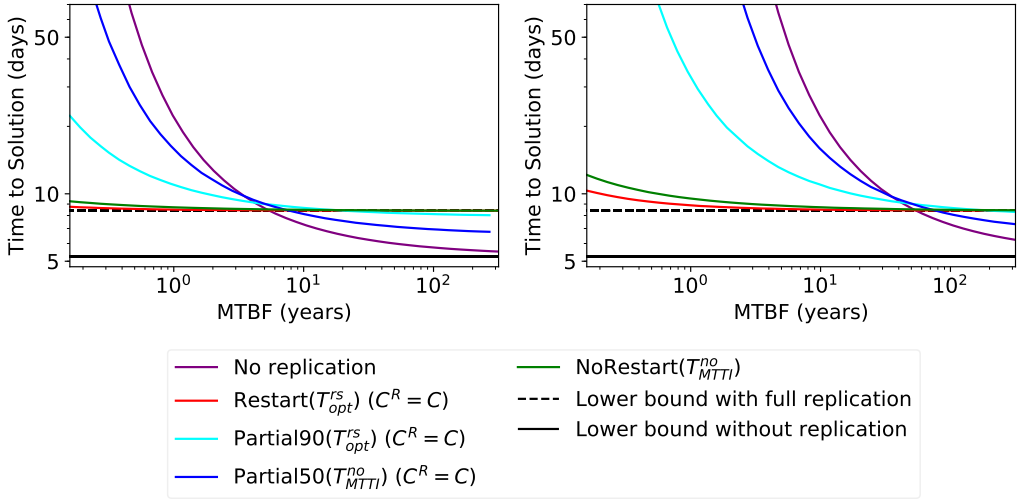


Figure 7.9: Time-to-solution for $N = 2 \times 10^5$ standalone proc. against full and partial replication approaches, as a function of MTBF, with $C^R = C = 60$s (left) or $C^R = C = 600$s (right), $\gamma = 10^{-5}, \alpha = 0.2$.

times are longer than expected because of I/O congestion.

7.7.6 Time-To-Solution

Looking at the time overhead is not sufficient to evaluate the efficiency of replication. So far, we only compared different strategies that all use full process replication. We now compare the *restart* and *no-restart* strategies to the approach without replication, and also to the approach with partial replication [72, 110]. Figure 7.9 shows the corresponding time-to-solution for $\gamma = 10^{-5}$ and $\alpha = 0.2$ (values used in [110]), and $C^R = C$

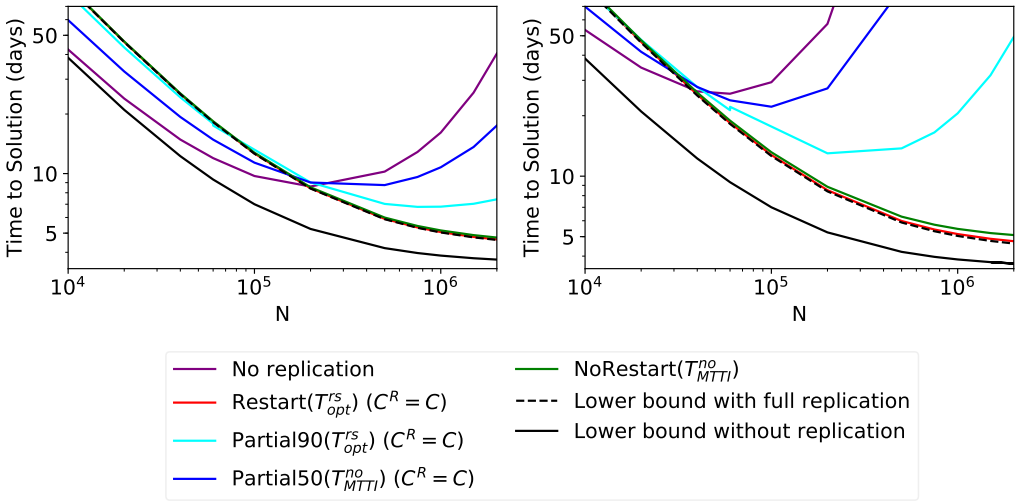


Figure 7.10: Time-to-solution with MTBF of 5 years against full and partial replication approaches, as a function of N, with $C^R = C = 60$s (left) or $C^R = C = 600$s (right), $\gamma = 10^{-5}, \alpha = 0.2$.

when the individual MTBF varies. Recall that the time-to-solution is computed using Equation (7.22) without replication (where $\mathbb{H}(T)$ is given by Equation (7.7)), and using Equation (7.23) with replication (where $\mathbb{H}(T)$ is given by Equation (7.12) for *no-restart*, and by Equation (7.19) for *restart*). In the simulations, T_{seq} is set so that the application lasts one week with 100,000 processors (and no replication).

In addition to the previously introduced approaches, we evaluate *Partial*90(T^{rs}_{opt}) and *Partial*50(T^{no}_{MTTI}). *Partial*90 represents a partial replication approach where 90% of the platform is replicated (there are 90,000 processor pairs and 20,000 standalone processors). Similarly, 50% of the platform is replicated for *Partial*50 (there are 50,000 processor pairs and 100,000 standalone processors). Figure 7.9 illustrates the benefit of full replication: when the MTBF becomes too short, replication becomes mandatory. Indeed, in some cases, simulations without replication or with partial replication would not complete, because one fault was (almost) always striking before a checkpoint, preventing progress. For $C = 60$s and $N = 2 \times 10^5$, $\gamma = 10^{-5}$ and $\alpha = 0.2$, full replication grants the best time-to-solution for an MTBF shorter than 1.8×10^8. However, when the checkpointing cost increases, this value climbs up to 1.9×10^9, i.e., roughly 10 times higher than with 60 seconds. As stated before, T^{rs}_{opt} gives a better overhead, thus a better execution time than T^{no}_{MTTI}. If machines become more unreliable, the *restart* strategy allows us to maintain the best execution time. Different values of γ and α give the same trend as in our example, with large values of γ making replication more efficient, while large values of α reduce the performance. Similarly to what was observed in [110], for a homogeneous platform (i.e., if all processors have a similar risk of failure), partial replication (at 50% or 90%) exhibits lower performance

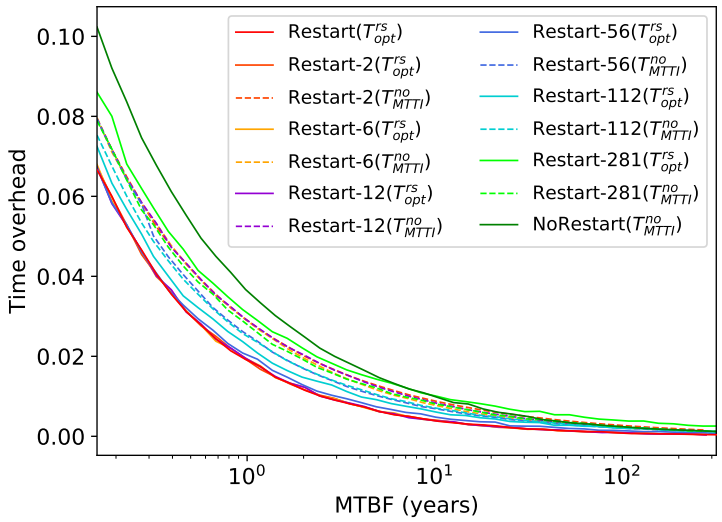


Figure 7.11: Comparison of *restart* strategy with *restart* only every 2, 6, 12, 56, 112, or 281 dead proc., with T^{rs}_{opt} and T^{no}_{MTTI}.

than no replication for long MTBF, and lower performance than the *no-restart* strategy (hence even lower performance than the *restart* strategy) for short MTBF. This confirms that partial replication has potential benefit only for heterogeneous platforms, which is outside the scope of this study.

We now further focus on discussing when replication should be used. Figure 7.10 shows the execution time of an application when the number of processors N varies. Each processor has an individual MTBF of 5 years. The same general comments can be made: $Restart(T^{\text{rs}}_{opt})$ always grants a slightly lower time-to-solution than $NoRestart(T^{\text{no}}_{MTTI})$, because it has a smaller overhead. As before, when N is large, the platform is less reliable and the difference between $Restart(T^{\text{rs}}_{opt})$ and $NoRestart(T^{\text{no}}_{MTTI})$ is higher compared to small values of N. We see that replication becomes mandatory for large platforms: without replication, or even with 50% of the platform replicated, the time-to-solution is more than 10 times higher than the execution time without failures. With $\gamma = 10^{-5}$ and $\alpha = 0.2$, replication becomes more efficient than no replication for $N \geq 2 \times 10^5$ processors when $C = 60$s. However, when $C = 600$s, it starts being more efficient when $N \geq 2.5 \times 10^4$, i.e., roughly 10 times less processors when C is 10 times longer. This study further confirms that partial replication never proved to be useful throughout our experiments.

7.7.7 When to Restart

In this section, we consider a natural extension of the *restart* approach: instead of restarting failed processors at each checkpoint, the restart can be delayed until the

next checkpoint where the number of accumulated failures reaches or exceeds a given bound n_{bound}, thereby reducing the frequency of the restarts.

The *restart* strategy assumes that after a checkpoint, the risk of any processor failing is the same as in the initial configuration. For the extension, there is no guarantee that T_{opt}^{rs} remains the optimal interval between checkpoints; worse, there is no guarantee that periodic checkpointing remains optimal. To evaluate the potential gain of reducing the restart frequency, we consider the two proposed intervals: T_{opt}^{rs} and T_{MTTI}^{no}. And, since most checkpoints will not incur a restart, we assume $C^R = C$ when computing T_{opt}^{rs}. However, checkpoints where processes are restarted have a cost of twice the cost of a simple checkpoint in the simulation: this is the worst case for the *restart* strategy. We then simulate the execution, including restarts due to reaching n_{bound} failures and due to application crashes. With $b = 100,000$ processor pairs, we expect $n_{\text{fail}}(2b) = 561$ failures before the application is interrupted; so we will consider a large range of values for n_{bound}: from 2, 6, 12, to cover cases where few failures are left to accumulate, to 56, 112, or 281, that represent respectively 10%, 20% and 50% of $n_{\text{fail}}(2b)$, to cover cases where many failures can accumulate.

The results are presented in Figure 7.11, for a variable node MTBF. The time overhead of the extended versions is higher than the time overhead of the *restart* approach using T_{opt}^{rs} as a checkpointing (and restarting) interval. The latter is also lower than the overhead of the *no-restart* strategy, which on average corresponds to restarting after $n_{\text{bound}} = n_{\text{fail}}(2b) = 561$ failures. This shows that restarting the processes after each checkpoint consistently decreases the time overhead. Using the optimal checkpointing period for *restart* T_{opt}^{rs}, increasing n_{bound} also increases the overhead. Moreover, when using small values (such as 2 and 6) for n_{bound}, we obtain exactly the same results as for the *restart* strategy. This is due to the fact that between two checkpoints, the *restart* strategy usually looses around 6 processors, meaning that *restart* is already the same strategy as accumulating errors up to 6 (or less) before restarting. With n_{bound} =12, on average the restart happens every two checkpoints, and the performance is close, but slightly slower than the *restart* strategy.

Finally, an open problem is to determine the optimal checkpointing strategy for the extension of *restart* tolerating n_{bound} failures before restarting failed processors. This optimal strategy could render the extension more efficient than the baseline *restart* strategy. Given the results of the simulations, we conjecture this optimal number to be 0, i.e., *restart* would be the optimal strategy.

Summary. Overall, we have shown that the *restart* strategy with period T_{opt}^{rs} is indeed optimal and that our model is realistic. We showed that *restart* decreases time overhead, hence time-to-solution, compared to using *no-restart* with period T_{MTTI}^{no}. The next section shows similar gains in energy overheads. The main decision is still to decide whether the application should be replicated or not. However, whenever it should be (which is favored by a large ratio of sequential tasks γ, a large checkpointing cost C, or a short MTBF), we are now able to determine the best strategy: use full replication, restart dead processors at each checkpoint (overlapped if possible), and use T_{opt}^{rs} for

the checkpointing period.

7.8 Energy consumption

In this last section, we extend the approach to a different objective function: the goal is now to minimize the energy overhead. If $\mathcal{E}(T)$ is the expected energy consumption of a period of length $\mathcal{P} = T + C$, the energy overhead with a single processor is expressed as:

$$\mathbb{H}^{\text{energy}}(T) = \frac{\mathcal{E}(T)}{T(P_{\text{comp}} + P_{\text{static}})} - 1, \tag{7.24}$$

where P_{comp} is the dynamic power consumption of a processor when computing, and P_{static} denotes the static power, which is paid when the processor is kept idle, but still turned on.

We also denote by $P_{\text{I/O}}$ the dynamic power when performing I/O operations, which has to be accounted for when checkpointing, hence in the expression of $\mathcal{E}(T)$. We express below $\mathcal{E}(T)$ and $\mathbb{H}^{\text{energy}}(T)$ in the cases without replication (single processor or N processors, Section 7.8.1) and with replication (one pair or b pairs, Section 7.8.2), and derive in each case the optimal period, and the optimal energy overhead. Finally, we present comprehensive simulation results in Section 7.8.3.

7.8.1 Without replication

7.8.1.1 With a single processor

In this case, we use the same approach as in Section 7.3.1 and we write a recursive formula similar to Equation (7.2):

$$\begin{aligned}\mathcal{E}(T) = (1 - F(T))&(T(P_{\text{comp}} + P_{\text{static}}) + C(P_{\text{I/O}} + P_{\text{static}})) \\ &+ F(T)(\mathbb{E}^{\text{lost}}T(P_{\text{comp}} + P_{\text{static}}) + DP_{\text{static}} + R(P_{\text{I/O}} + P_{\text{static}}) \\ &\qquad + \mathcal{E}(T))\end{aligned} \tag{7.25}$$

If the computation is successful, then we compute at power $P_{\text{comp}} + P_{\text{static}}$ during a time T and use the power $P_{\text{I/O}} + P_{\text{static}}$ during the checkpoint. However if a failure strikes, the machine is used at power $P_{\text{comp}} + P_{\text{static}}$ for $\mathbb{E}^{\text{lost}}T$ seconds, then used at power P_{static} for the downtime, and used at power $P_{\text{I/O}} + P_{\text{static}}$ during the recovery before starting the period anew. Finally, we obtain:

$$\begin{aligned}\mathcal{E}(T) = &\left(C + (e^{\lambda T} - 1)R\right)(P_{\text{I/O}} + P_{\text{static}}) \\ &+ (e^{\lambda T} - 1)DP_{\text{static}} + \frac{e^{\lambda T} - 1}{\lambda}(P_{\text{comp}} + P_{\text{static}}).\end{aligned}$$

Using the Taylor expansion as previously, we obtain the overhead

$$\mathbb{H}^{\text{energy}}(T) = \frac{C(P_{\text{I/O}} + P_{\text{static}})}{T(P_{\text{comp}} + P_{\text{static}})} + \frac{\lambda T}{2} + o(\lambda T). \tag{7.26}$$

Again, this overhead is minimized for $T = \Theta(\lambda^{-\frac{1}{2}})$. By differentiating Equation (7.26), we get the optimal period minimizing energy consumption:

$$T_{opt}^{\text{energy}} = \sqrt{\frac{2C(P_{\text{I/O}} + P_{\text{static}})}{\lambda(P_{\text{comp}} + P_{\text{static}})}} = \Theta(\lambda^{-\frac{1}{2}}). \tag{7.27}$$

Plugging it back into Equation (7.26), we get the optimal energy overhead:

$$\mathbb{H}_{opt}^{\text{energy}} = \sqrt{\frac{2C\lambda(P_{\text{I/O}} + P_{\text{static}})}{P_{\text{comp}} + P_{\text{static}}}} + o(\lambda^{\frac{1}{2}}) = \Theta(\lambda^{\frac{1}{2}}). \tag{7.28}$$

7.8.1.2 With N processors

We can generalize the previous result for the case with N processors, as done in Section 7.3.2 for the time overhead. We obtain another similar formula:

$$T_{opt}^{\text{energy}} = \sqrt{\frac{2C(P_{\text{I/O}}^{(N)} + NP_{\text{static}})}{N^2\lambda(P_{\text{comp}} + P_{\text{static}})}} = \Theta(\lambda^{-\frac{1}{2}}) \tag{7.29}$$

for the optimal checkpointing period, while the overhead becomes:

$$\mathbb{H}_{opt}^{\text{energy}} = \sqrt{\frac{2C\lambda(P_{\text{I/O}}^{(N)} + NP_{\text{static}})}{P_{\text{comp}} + P_{\text{static}}}} + o(\lambda^{\frac{1}{2}}) = \Theta(\lambda^{\frac{1}{2}}). \tag{7.30}$$

The main difference between Equations (7.27), (7.28) and Equations (7.29), (7.30) is that the dynamic power and the static power is multiplied by N, the number of processors, as more processors consume more energy. Similarly, $P_{\text{I/O}}$ becomes $P_{\text{I/O}}^{(N)} = P_{\text{I/O,static}} + NP_{\text{I/O,comm}}$ to take into account that more nodes are sending data to the external storage.

7.8.2 With replication

7.8.2.1 With one processor pair

We now compute the expected energy consumption $\mathcal{E}(T)$ of a period of length $\mathcal{P} = T + C^R$. We use the same approach as in Section 7.8.1.1 and aim at minimizing the

energy overhead

$$\mathbb{H}^{\text{energy}}(T) = \frac{\mathcal{E}(T)}{2T(P_{\text{comp}} + P_{\text{static}})} - 1 \tag{7.31}$$

We write a recursive formula similar to Equation (7.13):

$$\begin{aligned}\mathcal{E}(T) = (1 - p_1(T))\Big(T(2P_{\text{comp}} + 2P_{\text{static}}) + C^R(P_{\text{I/O}}^{(2)} + 2P_{\text{static}})\Big) \\ + p_1(T)\Big(\mathbb{E}^{\text{lost}}T(2P_{\text{comp}} + 2P_{\text{static}}) + 2DP_{\text{static}} \\ + R(P_{\text{I/O}}^{(2)} + 2P_{\text{static}}) + \mathcal{E}(T)\Big).\end{aligned} \tag{7.32}$$

After solving, we obtain:

$$\begin{aligned}\mathcal{E}(T) = T(2P_{\text{comp}} + 2P_{\text{static}}) + C^R(P_{\text{I/O}}^{(2)} + 2P_{\text{static}}) \\ + \frac{(e^{\lambda T} - 1)^2}{2e^{\lambda T} - 1}\Big(\frac{(2e^{-2\lambda T} - 4e^{-\lambda T})\lambda T + e^{-2\lambda T} - 4e^{-\lambda T} + 3}{2\lambda(1 - e^{-\lambda T})^2}(2P_{\text{comp}} + 2P_{\text{static}}) \\ + 2DP_{\text{static}} + R(P_{\text{I/O}}^{(2)} + 2P_{\text{static}})\Big).\end{aligned}$$

After Taylor expansion, we derive the overhead $\mathbb{H}^{\text{energy}}(T)$:

$$\mathbb{H}^{\text{energy}}(T) = \frac{C^R(P_{\text{I/O}}^{(2)} + 2P_{\text{static}})}{2T(P_{\text{comp}} + P_{\text{static}})} + \frac{2\lambda^2 T^2}{3} + o(\lambda^2 T^2) \tag{7.33}$$

The computations are similar to Section 7.4.2 and we find the following optimal values for T_{opt}^{energy} and $\mathbb{H}_{opt}^{\text{energy}}$:

$$T_{opt}^{\text{energy}} = \left(\frac{3C^R(P_{\text{I/O}}^{(2)} + 2P_{\text{static}})}{8\lambda^2(P_{\text{comp}} + P_{\text{static}})}\right)^{\frac{1}{3}} = \Theta(\lambda^{\frac{-2}{3}}). \tag{7.34}$$

$$\mathbb{H}_{opt}^{\text{energy}} = \left(\frac{3C^R\lambda(P_{\text{I/O}}^{(2)} + 2P_{\text{static}})}{2\sqrt{2}(P_{\text{comp}} + P_{\text{static}})}\right)^{\frac{2}{3}} + o(\lambda^{\frac{2}{3}}) = \Theta(\lambda^{\frac{2}{3}}). \tag{7.35}$$

7.8.2.2 With b processor pairs

As previously, we compute the energy consumption for the execution of one period of duration $\mathcal{P} = T + C^R$ using the following recursion:

$$\begin{aligned}\mathcal{E}(T) = p_b(T)\big(\mathbb{E}^{\text{lost}}T(2bP_{\text{comp}} + 2bP_{\text{static}}) + D2bP_{\text{static}} \\ + R(P_{\text{I/O}}^{(N)} + 2bP_{\text{static}}) + \mathcal{E}(T)\big) \\ + (1 - p_b(T))\big(T(2bP_{\text{comp}} + 2bP_{\text{static}}) + C^R(P_{\text{I/O}}^{(N)} + 2bP_{\text{static}})\big).\end{aligned}$$

With probability $p_b(T)$, the application fails so we account for the energy consumed until the failure $\mathbb{E}^{\text{lost}}T(2bP_{\text{comp}} + 2bP_{\text{static}})$, followed by a downtime and a restart (power consumption of $P_{\text{I/O}}^{(N)} + 2bP_{\text{static}}$). Otherwise, the application is successful, meaning that we computed at power $2b(P_{\text{comp}} + P_{\text{static}})$ during T seconds and we stored a checkpoint (overlapped with a restart) at power $P_{\text{I/O}}^{(N)} + 2bP_{\text{static}}$. We already computed $\mathbb{E}^{\text{lost}}T$ in the previous subsection so we can directly derive, using a Taylor expansion of the exponential function and solving the previous equation that:

$$\begin{aligned}\mathbb{H}^{\text{energy}}(T) &= \frac{\mathcal{E}(T)}{T \cdot 2b(P_{\text{comp}} + P_{\text{static}})} - 1 \\ &= \frac{C^R(P_{\text{I/O}}^{(N)} + 2bP_{\text{static}})}{2bT(P_{\text{comp}} + P_{\text{static}})} + \frac{2b\lambda^2T^2}{3} + o(\lambda^2T^2),\end{aligned} \tag{7.36}$$

which is very similar to Equation (7.33), with the only difference being a factor b on the second term and power consumption factors. We then derive a similar optimal period time T_{opt}^{energy} as well as the optimal energy overhead $\mathbb{H}_{opt}^{\text{energy}}$:

$$T_{opt}^{\text{energy}} = \left(\frac{3C^R(P_{\text{I/O}}^{(N)} + 2bP_{\text{static}})}{8b^2\lambda^2(P_{\text{comp}} + P_{\text{static}})}\right)^{\frac{1}{3}} = \Theta(\lambda^{\frac{-2}{3}}). \tag{7.37}$$

$$\mathbb{H}_{opt}^{\text{energy}} = \left(\frac{3C^R\lambda(P_{\text{I/O}}^{(N)} + 2bP_{\text{static}})}{2\sqrt{2b}(P_{\text{comp}} + P_{\text{static}})}\right)^{\frac{2}{3}} + o(\lambda^{\frac{2}{3}}) = \Theta(\lambda^{\frac{2}{3}}). \tag{7.38}$$

7.8.3 Experiments

For the power consumption, we chose $P_{\text{static}} = 10\text{W/node}$ and $P_{\text{comp}} = P_{\text{static}}$, so that the non-idle power consumption of a node is 20W (i.e., an exascale machine with 10^6 nodes would reach the proposed bound of 20MW). For $P_{\text{I/O}}$, as measured in [64], we set it to 15% of the static power, i.e., $P_{\text{I/O}} = 0.15P_{\text{static}} = 1.5\text{W/node}$. With these values, we have $\frac{P_{\text{I/O}}+P_{\text{static}}}{P_{\text{comp}}+P_{\text{static}}} = 0.575$, meaning that optimizing energy overhead will result in a shorter period than when optimizing time overhead.

Graphs in Figure 7.12 describe the impact of the individual MTBF of the processors on the energy overhead: they are the counterpart of Figure 7.7 that focused on execution time. The energy overheads reduce by a factor ranging from 62% to 80%, with the average being 72%.

Figure 7.13 shows the difference between the two optimal periods T_{opt}^{rs} and $T_{opt,en}^{\text{rs}}$. As we can see, optimizing the time overhead or the energy overhead has a negligible impact on their values. When we optimize the energy overhead, our worst increase for the time overhead is around 15% for a MTBF ranging from 5×10^6 to 10^10, $C^R = C =$ 60 seconds and $b = 10^5$. The average increase however is of 3.1% over the whole range.

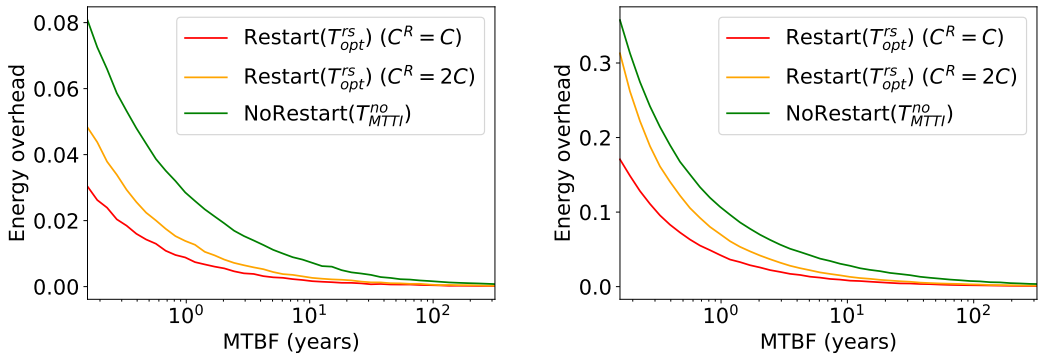


Figure 7.12: Influence of the MTBF on the energy overhead for the R_a nd *no-restart* strategies. Checkpointing time set to 60 seconds (left) or 600 seconds (right), with 10^5 pairs of processors.

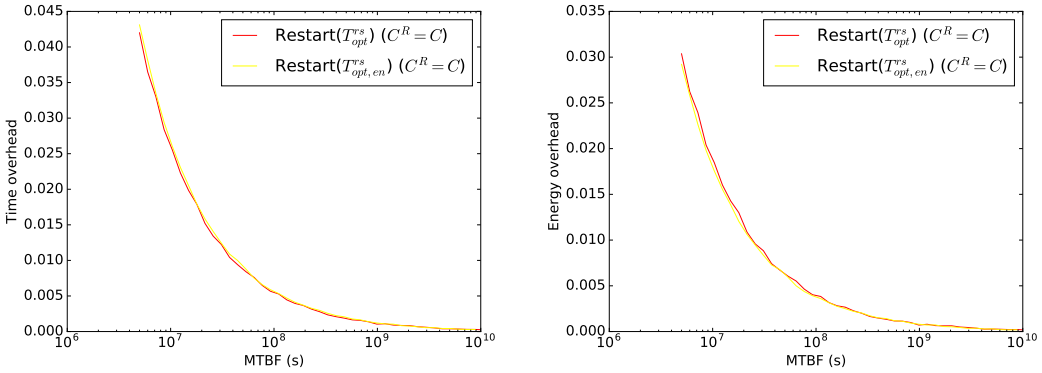


Figure 7.13: Impact of optimizing the time overhead or the energy overhead on the time overhead (left) or the energy overhead (right) as a function of the MTBF ($C = 60$s, 10^5 pairs of processors).

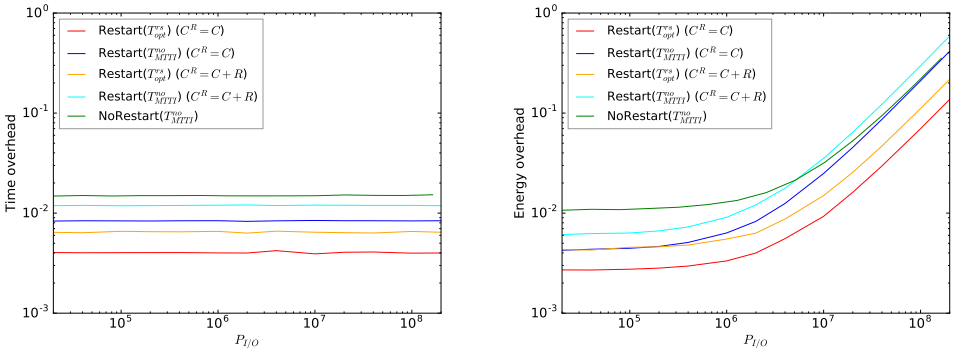


Figure 7.14: Time and energy overheads when varying $P_{\text{I/O}}$ (MTBF 5 years, $C = 60$s, $b = 10^5$) when optimizing the time overhead.

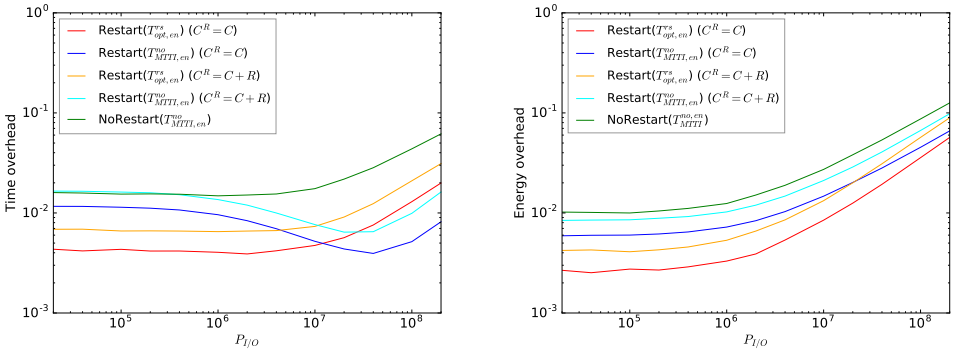


Figure 7.15: Time and energy overheads when varying $P_{\text{I/O}}$ (MTBF 5 years, $C = 60$s, $b = 10^5$) when optimizing the energy overhead.

When optimizing the time overhead, we measured a maximum of 23% improvement under the same conditions with the average increase being 4.2%. Overall, with our values we do not need to specifically optimize the energy overhead, except if the ratio between $P_{\text{I/O}} + P_{\text{static}}$ and $P_{\text{comp}} + P_{\text{static}}$ is much greater or much smaller than 1, where the difference between the two optimal periods might differ more than that.

Finally, Figures 7.14 and 7.15 show the impact of varying $P_{\text{I/O}}$ on both time and energy overheads, the former when optimizing the time overhead, the latter when optimizing the energy overhead. We once again show that the R_s trategy with the corresponding optimal period is the best one, with the only exception being when $P_{\text{I/O}}$ is high (more than 10^7W, i.e. 10 times the value set in the previous experiments). If we want to optimize the energy consumption, then using $T^{no}_{MTTI,en}$ can lower the time overhead (and only increases a bit the energy overhead).

7.9 Conclusion

In this work, we have revisited process replication combined with checkpointing, an approach that has received considerable attention from the HPC community in recent years. Opinion is divided about replication. By definition, its main drawback is that 50% of platform resources will not contribute to execution progress, and such a reduced throughput does not seem acceptable in many scenarios. However, checkpoint/restart alone cannot ensure full reliability in heavily failure-prone environments, and must be complemented by replication in such unreliable environments. Previous approaches all used the *no-restart* strategy. In this work, we have introduced a new rollback/recovery strategy, the *restart* strategy, which consists of restarting all failed processes at the beginning of each period. Thanks to this rejuvenation, the system remains in the same conditions at the beginning of each checkpointing period, which allowed us to build an accurate performance model and to derive the optimal checkpointing period for this strategy. This period turns out to be much longer than the one used with the *no-restart* strategy, hence reducing significantly the I/O pressure introduced by checkpoints, and improving the overall time-to-solution. To validate this approach, we have simulated the behavior of realistic large-scale systems, with failures either IID or from log traces. We have compared the performance of *restart* with the state-of-the-art strategies. Another key advantage of the *restart* strategy is its robustness: the range of periods in which its performance is close to optimal is much larger than for the *no-restart* strategy, making it a better practical choice to target unreliable platforms where the key elements (MTBF and checkpoint duration) are hard to estimate. In the future, we plan to evaluate, at least experimentally, non-periodic checkpointing strategies that rejuvenate failed processors after a given number of failures is reached or after a given time interval is exceeded.

Part III

Scheduling problems

Chapter 8

Design and Comparison of Resilient Scheduling Heuristics for Parallel Jobs

This chapter focuses on the resilient scheduling of parallel jobs on high-performance computing (HPC) platforms to minimize the overall completion time, or makespan. We revisit the classical problem while assuming that jobs are subject to transient or silent errors, and hence may need to be re-executed each time they fail to complete successfully. This work generalizes the classical framework where jobs are known offline and do not fail: in the classical framework, list scheduling that gives priority to longest jobs is known to be a 3-approximation when imposing to use shelves, and a 2-approximation without this restriction. We show that when jobs can fail, using shelves can be arbitrarily bad, but unrestricted list scheduling remains a 2-approximation. The chapter focuses on the design of several heuristics, some list-based and some shelf-based, along with different priority rules and backfilling strategies. We assess and compare their performance through an extensive set of simulations, using both synthetic jobs and log traces from the Mira supercomputer. The work in this chapter is joint work with Anne Benoit, Hongyang Sun and Yves Robert, and will be published in the workshop on *Advances in Parallel and Distributed Computational Models* (APDCM) [W7].

8.1 Introduction

One of the main challenges faced by today's HPC platforms is resilience, since such platforms are confronted with many failures or errors due to their large scale [169]. Indeed, the number of failures is known to grow proportionally with the number of nodes on a platform [106], and the largest supercomputers today experience several failures per day. There are two main classes of errors that can cause failures in an application's execution, namely, fail-stop and silent errors. While fail-stop errors cause the execution to terminate (e.g., due to hardware fault), large-scale platforms are also confronted with *silent errors*, or *silent data corruptions (SDCs)*. Such errors are caused by cosmic radiation or packaging pollution, striking either the cache or memory units

(bit flips), or the CPU operations [147, 210]. Even though any bit can be corrupted, the execution continues (unlike fail-stop errors), hence the error is transient, but it may dramatically impact the result of a running application. Many silent errors can be accurately detected by verifying the data using dedicated, lightweight detectors (e.g., [195, 50, 93, 49]). In this work, we focus on job failures caused by silent errors, and we aim to design resilient scheduling heuristics while assuming the availability of ad-hoc detectors to detect such errors.

The problem of scheduling a set of independent jobs on parallel platforms with the goal of minimizing the total completion time, or *makespan*, has been extensively studied (see Section 8.5). Jobs may be parallel and should be executed on a given number of processors for a certain duration; both the processor requirement and the execution time of each job are known at the beginning. Such jobs are called *rigid* jobs, contrarily to moldable or malleable jobs, whose processor allocations can vary at launch time or during execution [80]. While moldable or malleable jobs offer more flexibility in the execution, rigid jobs remain the most prevalent form of parallel jobs submitted on today's HPC systems, and we focus on rigid jobs in this chapter.

Unlike the classical scheduling problem without job failures, we consider *failure-prone platforms* subject to silent errors. Hence, at the end of each job's execution, an SDC detector will flag if a silent error has occurred during its execution. In this case, the job must be re-executed until it has been successfully completed without errors. For a set of jobs, each execution may lead to a different failure scenario, depending upon the jobs that have experienced failures as well as the number of such failures. The objective is to minimize the makespan under any failure scenario, as well as the *expected makespan*, averaged over all possible failure scenarios, where each scenario is weighted by a probability that governs its occurrence under certain failure assumptions. Since a failure scenario is unknown a priori, the scheduling decisions must be made *dynamically* on-the-fly, whenever an error has been detected. As a result, even for the same set of jobs, different schedules may be produced, depending on the failure scenario that occurred in a particular execution.

Building upon the existing framework for scheduling parallel jobs without failures, we propose two scheduling strategies, namely, a *list-based* strategy and a *shelf-based* strategy. While list-based schedules have no restrictions on the starting times of the jobs, shelf-based schedules group all jobs into subsets of jobs having the same starting time (called shelves); a shelf of jobs can start its execution once the longest job from the previous shelf has completed. For list-based scheduling, practical systems also employ a combination of reservation and backfilling strategies with different job priority rules to increase the system utilization. On platforms with no failures, variants for all of these strategies exist that could achieve constant approximations for the makespan (see Section 8.5 for details). The main focus of this chapter is to extend these existing heuristics to execution scenarios with job failures, and to experimentally compare their performance using a variety of job and platform configurations.

Our main contributions are the following:

- We propose a formal model for the problem of resilient scheduling of parallel

jobs on failure-prone platforms. The model formulates the performance of an algorithm under both worst-case and expected executions.
- We design a resilient list-based strategy, and prove that its greedy variant achieves $(2 - \frac{1}{P})$-approximation, and its reservation variant is $(3 - \frac{4}{P+1})$-approximation, where P is the total number of processors. These results apply to both worst-case and expected makespans.
- We design a resilient shelf-based strategy, but we show that, under some failure scenarios, any shelf-based algorithm has an unbounded approximation ratio, thus having a makespan that is arbitrarily higher than the optimal makespan in the worst case.
- We conduct an extensive set of simulations to evaluate and compare different variants of these heuristics using both synthetic jobs and log traces from the Mira supercomputer. The results show that the performance of these resilient scheduling heuristics is close to the optimal in practice, even when confronted with failures.

The rest of this chapter is organized as follows: The formal models and the problem statement are presented in Section 8.2. The key contributions of the chapter are presented in Section 8.3, where we describe both list-based and shelf-based strategies, and analyze their performance. Section 8.4 presents an extensive set of simulation results and highlights the main findings. Section 8.5 describes the background of parallel job scheduling and presents some related work. Finally, Section 8.6 concludes the chapter and discusses future directions.

8.2 Models

In this section, we formally present the models, the problem statement, and the main assumptions we make in the chapter.

8.2.1 Job model

We consider a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of n parallel jobs to be executed on a platform consisting of P identical processors. All jobs are released at the same time, corresponding to the batch scheduling scenario in an HPC environment. We focus on *rigid* jobs, which must be executed with a fixed number of processors set by the user when the job is submitted[1]. For each job $J_j \in \mathcal{J}$, let $p_j \in \{1, 2, \ldots, P\}$ denote its fixed (integral) processor allocation, and let t_j denote its error-free execution time. The *area* of the job is defined as $a_j = p_j \times t_j$.

[1]Other parallel job models include *moldable* and *malleable* models, which allow the processor allocation of a job to vary at launch time or during execution [80]. Considering alternative job models will be part of our future work.

8.2.2 Error model

We consider failures that manifest as *silent errors* or *silent data corruptions (SDCs)* [169] that could corrupt a job during execution. A silent error detector is assumed to be available for each job, which is triggered at the end of the job's execution. If an error is detected, the job needs to be re-executed, followed by another error detection. This process repeats until the job completes successfully without errors. Current state-of-the-art SDC detectors are typically lightweighted (e.g., ABFT for matrix computations [195, 50], data analytics for scientific applications [93, 49]), and hence incur a negligible cost compared to the job's overall execution time.

All the list-based and shelf-based scheduling heuristics introduced and compared in this chapter are agnostic of the probability of each job to fail any given number of times. Specifically, for a job J_j, consider a particular run where it fails f_j times before succeeding on the (f_j+1)-th execution. The probability that this happens is denoted as $q_j(f_j)$. Let $\mathbf{f} = (f_1, f_2, \ldots, f_n)$ denote a *failure scenario*, i.e., a vector of the number of failed execution attempts for all jobs, during a particular run. Assuming that errors occur independently for different jobs, the probability that this combined failure scenario happens can be computed as $Q(\mathbf{f}) = \prod_{j=1\ldots n} q_j(f_j)$. The failure scenario $\mathbf{f}$, as well as the associated probabilities $q_j(f_j)$ and $Q(\mathbf{f})$ may be unknown to the scheduler.

8.2.3 Problem statement

We study the following resilient scheduling problem: Given a set $\mathcal{J}$ of parallel jobs, find a schedule for $\mathcal{J}$ on P identical processors under any failure scenario $\mathbf{f} = (f_1, f_2, \ldots, f_n)$. Here, a *schedule* for $\mathbf{f}$ is defined by a collection $\mathbf{s} = (\vec{s}_1, \vec{s}_2, \ldots, \vec{s}_n)$ of starting time vectors for all jobs, where vector $\vec{s}_j = (s_j^{(1)}, s_j^{(2)}, \ldots, s_j^{(f_j+1)})$ specifies the starting times for job J_j at different execution attempts until success.

The objective is to minimize the overall completion time of all jobs, or the *makespan*. Suppose an algorithm ALG makes scheduling decision $\mathbf{s}$ during a failure scenario $\mathbf{f}$, then the makespan of the algorithm for this scenario is defined as:

$$T_{\text{ALG}}(\mathbf{f}, \mathbf{s}) = \max_{j=1\ldots n} \left(s_j^{(f_j+1)} + t_j\right) . \tag{8.1}$$

All scheduling decisions should be made while satisfying the following two constraints:

1. *Processor constraint*: The number of processors used at any time t by the set $\mathcal{J}_t$ of running jobs should not exceed the total number P of available processors on the platform, i.e., $\sum_{J_j \in \mathcal{J}_t} p_j \leq P, \forall t$.

2. *Re-execution constraint*: A job cannot be re-executed if its previous execution attempt has not yet been completed, i.e., $s_j^{(i+1)} \geq s_j^{(i)} + t_j, \forall j = 1 \ldots n, \forall i \geq 1$.

This scheduling problem, encompassing the failure-free problem as a special case, is clearly NP-hard. A scheduling algorithm ALG is said to be *c-approximation* if its

makespan is at most c times that of an optimal scheduler for all possible sets of jobs under all possible failure scenarios, i.e.,

$$T_{\text{ALG}}(\mathbf{f}, \mathbf{s}) \leq c \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \ , \tag{8.2}$$

where $T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*)$ denotes the optimal makespan with scheduling decision $\mathbf{s}^*$ under failure scenario $\mathbf{f}$. Clearly, this optimal makespan admits the following two lower bounds:

$$T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \geq t_{\max}(\mathbf{f}) \ , \tag{8.3}$$

$$T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \geq \frac{A(\mathbf{f})}{P} \ , \tag{8.4}$$

where $t_{\max}(\mathbf{f}) = \max_{j=1\ldots n}(f_j + 1) \cdot t_j$ is the maximum cumulative execution time of any job under $\mathbf{f}$, and $A(\mathbf{f}) = \sum_{j=1}^{n}(f_j + 1) \cdot a_j$ is the total cumulative area.

In Section 8.3, we establish several approximation results, which are valid for any failure scenario regardless of its individual probability. This is the strongest result that can be obtained from a theoretical perspective. However, from a practical perspective, given a set of jobs, it is not easy to assess the performance of a scheduling heuristic if the probability $Q(\mathbf{f}) = \prod_{j=1\ldots n} q_j(f_j)$ of each failure scenario $\mathbf{f}$ is not known. Thus, for the experiments in Section 8.4, we report the expected makespan of each heuristic under the standard Exponential probability distribution, as explained below.

8.2.4 Expected makespan

Suppose the occurrence of silent errors striking the jobs follows an Exponential probability distribution, and that the mean time between error (MTBE) of an individual processor is μ, so the error rate of the processor is given by $\lambda = 1/\mu$. For a job J_j executed on p_j processors, the probability that the job is struck by a silent error during execution is then given by $q_j = 1 - e^{-\lambda p_j \cdot t_j} = 1 - e^{-\lambda a_j}$ [106]. Then, the probability for job J_i to fail f_j times before succeeding on the $(f_j + 1)$-th execution is $q_j(f_j) = q_j^{f_j}(1 - q_j)$.

Given a set $\mathcal{J}$ of jobs, we can now define the *expected makespan* of an algorithm ALG, taken over all possible failure scenarios weighted by their probabilities, as:

$$\mathbb{E}(T_{\text{ALG}}) = \sum_{\mathbf{f}} Q(\mathbf{f}) \cdot T_{\text{ALG}}(\mathbf{f}, \mathbf{s}) \ . \tag{8.5}$$

In this case, an algorithm is a c-approximation if we have:

$$\mathbb{E}(T_{\text{ALG}}) \leq c \cdot \mathbb{E}(T_{\text{OPT}}) \ , \tag{8.6}$$

for all possible sets of jobs, where $\mathbb{E}(T_{\text{OPT}})$ denotes the optimal expected makespan. This is because the inequality is true for each failure scenario, hence for the weighted sum. Obviously, the converse is not true: an algorithm could satisfy Equation (8.6) (thus achieving c-approximation in expectation) but be arbitrarily worse than the op-

timal on some (low probability) failure scenarios. Still, expected makespan provides a synthetic indicator on the performance of an algorithm under study, enabling easy and quantitative comparisons. Thus, we use it for the experimental evaluations in Section 8.4.

8.2.5 Static vs. dynamic scheduling

As all the information regarding the set of jobs (except the failure scenario $\mathbf{f}$) is available, one approach would be to make all scheduling decisions (i.e., starting times $\mathbf{s}$) *statically* at the beginning, and then execute the jobs according to this static schedule. While this approach works for failure-free executions, it is problematic when jobs can fail and re-execute. In particular, a static schedule needs to pre-compute a (possibly infinite) sequence of starting times for all jobs to account for every possible failure scenario, while ensuring the satisfaction of the constraints. Pre-computing such a static schedule would be computationally intractable, especially when there turn out to be only a few failures in a run.

In contrast, another more flexible approach is to make scheduling decisions *dynamically* depending on the particular failure scenario that is unveiled from an execution. For example, a scheduling algorithm may decide the starting time for the next execution attempt of a job depending on the failure scenario and schedule so far. As a result, even for the same set of jobs, the algorithm may produce different schedules in response to the different failure scenarios that could arise at runtime. In this chapter, we adopt this dynamic approach.

8.3 Resilient Scheduling Heuristics

In this section, we present a resilient list-based heuristic (R-List) and a resilient shelf-based heuristic (R-Shelf) for scheduling rigid parallel jobs that could fail due to silent errors. We show that the greedy variant of R-List without reservation is 2-approximation, and a variant with reservations is 3-approximation. For R-Shelf, even though it achieves 3-approximation in the failure-free case, we show through an example that any resilient shelf-based algorithm may have an approximation ratio of $\Omega(\ln P)$ compared to the optimal in some failure scenario.

8.3.1 R-List scheduling heuristic

We first present a resilient list-based scheduling heuristic, called R-List, that schedules any set of jobs with the capability to handle failures. Algorithm 3 shows the pseudocode of R-List. It extends the classical batch scheduler that combines reservation and backfilling strategies. The algorithm first organizes all jobs in a list (or a queue) based on some priority rule. Then, whenever an existing job J_k completes and hence releases processors (at time 0, a virtual job J_0 can be considered to complete), the algorithm schedules the remaining jobs in the queue. First, it checks if job J_k completes

Algorithm 3: R-LIST

Input: a set $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ of rigid jobs, with processor allocation p_j and error-free execution time t_j for each job $J_j \in \mathcal{J}$, a platform with P identical processors, parameter m;
Output: a list schedule with starting times for all jobs in $\mathcal{J}$ till they complete successfully.
begin
  Insert all jobs into a queue Q according to some priority rule;
  whenever an existing job J_k completes **do**
    if error detected for J_k **then**
      $Q.insert\_with\_priority(J_k)$;
    `// schedule high-priority jobs using reservation`
    for $j = 1, 2 \ldots, \min(m, |Q|)$ **do**
      $J_j \leftarrow Q(j)$;
      Give job J_j an earliest possible reservation without delaying the reservation of job $J_{j'}, \forall j' = 1, \ldots, j-1$;
    `// schedule low-priority jobs using backfilling`
    for $j = m+1, \ldots, |Q|$ **do**
      $J_j \leftarrow Q(j)$;
      if Job J_j can be scheduled at the current time without delaying the reservation of job $J_{j'}, \forall j' = 1 \ldots m$ **then**
        execute job J_j at the current time;

with error. If so, the job will be inserted back into the queue, based on its priority, to be rescheduled later. All jobs in the queue are divided into two groups: the first m jobs with the highest priorities are each given a reservation at the earliest possible time, provided that any reservation made should not delay the starting times of the higher-priority jobs; the subsequent jobs in the queue (if any) are then examined one by one and backfilled to start at the current time, again if such backfilling does not affect any reservations for the higher-priority jobs.[2]

The R-LIST heuristic takes a parameter m, and depending on the value of m chosen, it resembles several scheduling strategies known in theory and practice:

- $m = |Q|$ (Conservative backfilling [142]): this strategy makes reservations for all pending jobs in the queue;
- $m = 1$ (Aggressive or EASY backfilling [129, 168]): this strategy makes a reservation only for the job at the head of the queue, and uses backfilling to schedule all remaining jobs in the queue;
- $m = 0$ (Greedy scheduler [88, 184, 81]): this strategy does not make any reservation, and uses backfilling to schedule all jobs in the queue.

Note that, when $m > 0$ and when a job J_k with high priority fails, it may be reinserted back into the first part of the queue (i.e., among the top m jobs). This may require recomputing the existing reservations (made previously) for some jobs that have lower priority than J_k. From an analysis point of view, we can think of each job completion as a trigger, which deletes all previous reservations and makes a fresh

[2]For practical schedulers, this is typically implemented using two separate job queues, one for reservation and one for backfilling.

round of reservation and backfilling decisions based on the updated queue.

In the following, we denote by RESERVATION this variant of R-LIST with reservations ($m > 0$), and by GREEDY the variant with $m = 0$.

8.3.2 Approximation ratios of R-List

We show that, under any failure scenario, RESERVATION with a particular priority rule is a $(3 - \frac{4}{P+1})$-approximation, and that GREEDY with any priority rule is a $(2 - \frac{1}{P})$-approximation. According to Equation (8.6), these results directly imply the same approximation ratios for the respective heuristic variants in terms of the expected makespan.

8.3.2.1 Result for Reservation

We first consider the RESERVATION variant, while applying a priority rule that favors large jobs and uses any priority for small jobs. We call this rule *Large Job First (LJF)*. Specifically, a job is said to be *large* if its processor allocation is at least $\frac{P+1}{2}$, and *small* otherwise. The LJF rule specifies that: (1) all large jobs have higher priority than all small jobs; (2) the priorities for large jobs are based on decreasing processor allocation; and (3) the priorities for small jobs are defined arbitrarily.

The following proposition shows the performance of RESERVATION in any failure scenario using the above LJF rule. The result matches the 3-approximation ratio [13, 184] known for failure-free jobs.

Proposition 5. *For any set of rigid parallel jobs under any failure scenario* $\mathbf{f}$*, the makespan of* RESERVATION *with the LJF priority rule satisfies:*

$$T_{\mathrm{R}}(\mathbf{f}, \mathbf{s}) \leq (3 - \frac{4}{P+1}) \cdot T_{\mathrm{OPT}}(\mathbf{f}, \mathbf{s}^*) \; . \tag{8.7}$$

Proof. Let J_j be a last successfully completed job in the schedule. We divide the set $\mathcal{I} = \{I_1, I_2, \ldots, I_v\}$ of all intervals into two disjoint subsets $\mathcal{I}_1$ and $\mathcal{I}_2$, where $\mathcal{I}_1$ contains the intervals in which job J_j is executing (including all of its execution attempts), and $\mathcal{I}_2 = \mathcal{I} \backslash \mathcal{I}_1$. Let $T_1 = \sum_{I \in \mathcal{I}_1} |I|$ and $T_2 = \sum_{I \in \mathcal{I}_2} |I|$ denote the total lengths of all intervals in $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. Based on Equation (8.3), we have $T_1 = (f_j + 1) \cdot t_j(p_j) \leq t_{\max}(\mathbf{f}) \leq T_{\mathrm{opt}}(\mathbf{f}, \mathbf{s}^*)$.

We will show that the processor utilization in any interval $I \in \mathcal{I}_2$ satisfies $p(I) \geq \frac{P+1}{2}$. First, we observe that all large jobs are completed sequentially (in decreasing order of processor allocation) at the beginning of the entire schedule, since no two large jobs can be scheduled at the same time, and no small (backfilling) jobs can delay their executions because large jobs have higher priority based on the LJF rule. Thus, if an interval $I \in \mathcal{I}_2$ contains a large job, its processor allocation must satisfy $p(I) \geq \frac{P+1}{2}$.

Now, consider any interval $I \in \mathcal{I}_2$ after all the large jobs have completed, and suppose I lies in between the i-th execution attempt and the $(i+1)$-th execution attempt of J_j, where $0 \leq i \leq f_j$. Hence, if such an interval exists, it means that J_j is a small job

(with $p_j \leq \frac{P+1}{2}$), as well as all remaining jobs that are to be executed. Let t be the time at the beginning of this interval I. Recall that we can consider RESERVATION to make a fresh round of reservations and backfillings based on the current job queue Q at time t. Let J_k be the first job in Q that cannot be scheduled (either reserved or backfilled) to run at t. We know that such a job always exists because of the $(i+1)$-th execution attempt of J_j, which is scheduled to run at a later time. Let $\mathcal{J}_t$ be the set of jobs already running at time t or just scheduled to run at time t before job J_k, and let $p(\mathcal{J}_t)$ be the total processor allocation of all jobs in $\mathcal{J}_t$. As J_k cannot be scheduled to run at time t, it must be due to $p(\mathcal{J}_t) + p_k \geq P + 1$. Since J_k is a small job, i.e., $p_k \leq \frac{P+1}{2}$, it implies that $p(I) \geq p(\mathcal{J}_t) \geq \frac{P+1}{2}$.

Thus, based on Equation (8.4) and since $p_j \geq 1$, we have $P \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \geq A(\mathbf{f}) \geq \frac{P+1}{2} \cdot T_2 + p_j \cdot T_1 \geq \frac{P+1}{2} \cdot T_2 + T_1$. The overall execution time of RESERVATION with the LJF priority rule therefore satisfies:

$$\begin{aligned} T_{\text{R}}(\mathbf{f}, \mathbf{s}) &= T_1 + T_2 \\ &\leq T_1 + 2 \cdot \frac{P \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) - T_1}{P+1} \\ &= \frac{2P}{P+1} \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) + \left(1 - \frac{2}{P+1}\right) \cdot T_1 \\ &\leq (3 - \frac{4}{P+1}) \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \ .\square \end{aligned}$$

8.3.2.2 Result for Greedy

We now consider the GREEDY variant. The following proposition shows the performance of GREEDY in any failure scenario regardless of the priority rule. The result generalizes the 2-approximation ratio [88, 184, 81] known for failure-free jobs.

Proposition 6. *For any set of rigid parallel jobs under any failure scenario* $\mathbf{f}$*, the makespan of* GREEDY *regardless of the priority rule satisfies:*

$$T_{\text{G}}(\mathbf{f}, \mathbf{s}) \leq (2 - \frac{1}{P}) \cdot T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) \ . \tag{8.8}$$

Proof. Given the set $\mathcal{I} = \{I_1, I_2, \ldots, I_v\}$ of all intervals in the schedule, let $p_{\min} = \min_{1 \leq \ell \leq v} p(I_\ell)$ denote the minimum processor utilization among them. Since the algorithm never idles all processors unless all jobs complete successfully, we have $p_{\min} \geq 1$. We consider two cases:

Case 1: $p_{\min} \geq \frac{P+1}{2}$. In this case, we have $p(I_\ell) \geq p_{\min} \geq \frac{P+1}{2}$ for all $1 \leq \ell \leq v$. Hence, based on Equation (8.4), we get $P \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \geq A(\mathbf{f}) = \sum_{\ell=1,\ldots,v} |I_\ell| \cdot p(I_\ell) \geq \frac{P+1}{2} \cdot T_{\text{G}}(\mathbf{f}, \mathbf{s})$. This implies:

$$T_{\text{G}}(\mathbf{f}, \mathbf{s}) \leq \frac{2P}{P+1} \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \leq (2 - \frac{1}{P}) \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \ .$$

Case 2: $p_{\min} < \frac{P+1}{2}$. In this case, let $I_{\min}$ denote the last-executed interval that has processor utilization $p_{\min}$. Consider a job J_j that is running during interval $I_{\min}$. Necessarily, we have $p_j \leq p_{\min}$. We divide the set $\mathcal{I}$ of intervals into two disjoint subsets $\mathcal{I}_1$ and $\mathcal{I}_2$, where $\mathcal{I}_1$ contains the intervals in which job J_j is executing (including all of its execution attempts), and $\mathcal{I}_2 = \mathcal{I} \backslash \mathcal{I}_1$. Let $T_1 = \sum_{I \in \mathcal{I}_1} |I|$ and $T_2 = \sum_{I \in \mathcal{I}_2} |I|$ denote the total lengths of all intervals in $\mathcal{I}_1$ and $\mathcal{I}_2$, respectively. Based on Equation (8.3), we have $T_1 = (f_j + 1) \cdot t_j(p_j) \leq t_{\max}(\mathbf{f}) \leq T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*)$.

For any interval $I \in \mathcal{I}_2$ that lies between the i-th execution attempt and the $(i+1)$-th execution attempt of J_j in the schedule, where $0 \leq i \leq f_j$, the processor utilization of I must satisfy $p(I) \geq P - p_{\min} + 1$, since otherwise there are at least $p_{\min} \geq p_j$ available processors during interval I and hence the $(i+1)$-th execution attempt of J_j would have been scheduled at the beginning of I.

For any interval $I \in \mathcal{I}_2$ that lies after the $(f_j + 1)$-th (last) execution attempt of J_j, there must be a job J_k running during I and that was not running during $I_{\min}$ (meaning no attempt of executing J_k was made during $I_{\min}$). This is because $p(I) > p_{\min}$, hence the job configuration must differ between I and $I_{\min}$. The processor utilization during interval I must also satisfy $p(I) \geq P - p_{\min} + 1$, since otherwise the processor allocation of J_k will be $p_k \leq p(I) \leq P - p_{\min}$, implying that the first execution attempt of J_k after interval $I_{\min}$ would have been scheduled at the beginning of $I_{\min}$.

Thus, for all $I \in \mathcal{I}_2$, we have $p(I) \geq P - p_{\min} + 1$. Based on Equation (8.4), we have $P \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \geq A(\mathbf{f}) \geq (P - p_{\min} + 1) \cdot T_2 + p_{\min} \cdot T_1$. Since $p_{\min} \geq 1$, the overall execution time of GREEDY therefore satisfies:

$$
\begin{aligned}
T_{\text{G}}(\mathbf{f}, \mathbf{s}) &= T_1 + T_2 \\
&\leq T_1 + \frac{P \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) - p_{\min} \cdot T_1}{P - p_{\min} + 1} \\
&= \frac{P}{P - p_{\min} + 1} \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) + \frac{P - 2p_{\min} + 1}{P - p_{\min} + 1} \cdot T_1 \\
&\leq \frac{2P - 2p_{\min} + 1}{P - p_{\min} + 1} \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \\
&\leq (2 - \frac{1}{P - p_{\min} + 1}) \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \\
&\leq (2 - \frac{1}{P}) \cdot T_{\text{opt}}(\mathbf{f}, \mathbf{s}^*) \ .\square
\end{aligned}
$$

8.3.3 R-Shelf scheduling heuristic

We now present a shelf-based scheduling heuristic, called R-SHELF, that schedules any set of parallel jobs onto a series of shelves while handling job failures.

Heuristic description Algorithm 4 shows the pseudocode of R-SHELF. As in R-LIST, the algorithm starts by organizing all jobs in a queue based on some priority rule. Whenever the jobs in the preceding shelf all complete (at time 0, a virtual shelf S_0 with

Algorithm 4: R-SHELF

Input: a set $\mathcal{J} = \{J_1, J_2, \cdots, J_n\}$ of rigid jobs, with processor allocation p_j and error-free execution time t_j for each job $J_j \in \mathcal{J}$, a platform with P identical processors, parameter b;
Output: a shelf schedule with starting times for all jobs in $\mathcal{J}$ till they complete successfully.

```
begin
    Insert all jobs into a queue Q according to some priority rule;
    i ← 0, S_i ← ∅;
    whenever all jobs in S_i complete do
        if error detected for J_k ∈ S_i then
            Q.insert_with_priority(J_k);
        i ← i + 1 and S_i ← ∅; // start a new shelf
        for j = 1, 2 ..., |Q| do
            J_j ← Q(j);
            if Job J_j can fit in shelf S_i then
                S_i ← S_i ∪ {J_j};
            else if b = 0 then
                break ; // no backfilling
        execute all jobs in S_i at the current time;
```

no job in it can be considered to complete), the algorithm builds a new shelf and adds the remaining jobs to it. First, any job in the preceding shelf that completes with error will be inserted back into the queue based on its priority. Then, the algorithm scans the queue and adds a job to the new shelf if the job can fit in without violating the processor constraint. R-SHELF takes a binary parameter b that determines if backfilling is used in the process:

- $b = 0$ (No backfilling): the heuristic closes the new shelf upon encountering the first job in the queue that does not fit in the shelf. This resembles the Next-Fit (NF) strategy for bin-packing.
- $b = 1$ (Backfilling): the heuristic scans all the jobs in the queue until no more job can be added to the new shelf. This resembles the First-Fit (FF) strategy for bin-packing.

Once the jobs in the new shelf have been selected, they will simultaneously start their executions.

Inapproximability result For failure-free jobs, the variant of R-SHELF without backfilling and considering jobs in the non-increasing execution time order is equivalent to the Next-Fit Decreasing Height (NFDH) [53] algorithm for strip packing. The algorithm starts with the longest job J_1, which is put on the first shelf, whose height is t_1. Then, the next job J_2 is put on the same shelf if it fits in, meaning that $p_1 + p_2 \leq P$, otherwise a new shelf is started for J_2, whose height is t_2. The algorithm proceeds like this, either putting the next job on the last shelf if it fits in, or creating a new shelf otherwise. Despite its simplicity, the algorithm is shown to be a 3-approximation for failure-free jobs [53, 184].

Now, when jobs can fail, we show that there exists a job instance $\mathcal{J}$ and a failure scenario $\mathbf{f}$ such that any shelf-based algorithm has a makespan $T_S(\mathbf{f}, \mathbf{s})$ that is arbitrar-

ily higher than the optimal makespan $T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*)$ regardless of the job priority used. This is in clear contrast with the 3-approximation result for the failure-free case. Note that $T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*)$ is not necessarily the optimal makespan of a shelf-based schedule.

Proposition 7. *There exists a job instance and a failure scenario such that any shelf-based algorithm has an approximation ratio of* $\Omega(\ln P)$.

Proof. Consider a set $\mathcal{J} = \{J_1, \ldots, J_P\}$ of P uniprocessor jobs, where $t_j = P/j$ and $p_j = 1$ for $1 \leq j \leq P$. For the failure scenario $\mathbf{f}$, we let $f_j = j - 1$ for $1 \leq j \leq P$; hence job J_1 does not fail, job J_2 fails once before success, and job J_P fails $f_P = P - 1$ times before success.

We first consider the R-SHELF algorithm. Because the problem instance above has only P uniprocessor jobs, R-SHELF has no freedom at all: it schedules the first execution of all P jobs in the first shelf of height t_1, then the second execution of jobs J_2 to J_P in the second shelf of height t_2, and so on until the last shelf of height t_P, which includes only the P-th execution of job J_P. Therefore, the makespan of R-SHELF is $T_{\text{S}}(\mathbf{f}, \mathbf{s}) = P + \frac{P}{2} + \cdots + 1 = P \sum_{j=1}^{P} \frac{1}{j}$, while the optimal algorithm schedules the different executions of all jobs right after each other, thus having a makespan of $T_{\text{OPT}}(\mathbf{f}, \mathbf{s}^*) = P$. The ratio $\frac{T_{\text{S}}(\mathbf{f},\mathbf{s})}{T_{\text{OPT}}(\mathbf{f},\mathbf{s}^*)}$ tends to $\ln(P)$ when P tends to infinity, hence it is not bounded.

Furthermore, since the P jobs have decreasing execution time and increasing number of failures, any shelf-based algorithm will have at least one shelf of height t_j, for all $1 \leq j \leq P$, thus having a makespan that is at least $T_{\text{S}}(\mathbf{f}, \mathbf{s})$. Therefore, the same ratio applies to any shelf-based algorithm. □

We conclude this section with an open problem. Instead of a single failure scenario, consider an Exponential probability distribution and the expected makespan as defined in Section 8.2.4. Will R-SHELF or any shelf-based algorithm admit a constant approximation ratio in expectation? To answer this question seems difficult, because computing the expected makespan seems out of reach analytically. Given $P = 10$ in the above example, we find numerically (using a computer program) that the expected makespan ratio of R-SHELF is 1.00005 for $\lambda = 10^{-7}$ and 1.07 for $\lambda = 10^{-3}$. We have not been able to build an example where this ratio (computed numerically) is greater than 3.

8.4 Performance Evaluation

We now evaluate and compare the performance of all heuristics presented in Section 8.3, using different job priority rules and backfilling strategies. The evaluation is performed by simulation using both synthetic jobs and jobs extracted from the log traces of the Mira supercomputer.

8.4.1 Simulation setup

We compare five different heuristics combined with seven different priority rules. The five heuristics are:

- R-List-0: The list-based algorithm with $m = 0$;
- R-List-1: The list-based algorithm with $m = 1$;
- R-List-Q: The list-based algorithm with $m = |Q|$;
- R-Shelf-B: The shelf-based algorithm with $b = 1$.
- R-Shelf-NB: The shelf-based algorithm with $b = 0$.

For each of these five heuristics, we consider seven different job priority rules:

- Lpt/Spt (Longest/Shortest Processing Time): a job with a longer/shorter processing time will have higher priority;
- Hpa/Lpa (Highest/Lowest Processor Allocation): a job with a higher/lower number of requested processors will have higher priority;
- La/Sa (Largest/Smallest Area): a job with a larger/smaller area will have higher priority;
- Random (Random): the priorities are determined randomly for all jobs.

We simulate two different settings, one using synthetic jobs and the other using real job traces from the Mira logs.

- *Synthetic jobs*: We generate 30 different job sets, each containing 100 jobs. For each job, the processor allocation is generated uniformly at random between 50 and 2000, while the execution time is generated uniformly at random between 100 and 20000 seconds. The total number of processors is set to be $P = 10000$. In the experiments, we also vary P to study its impact.
- *Jobs from Mira logs*: We generate jobs by extracting from the log traces [5] (of June 2019) of the Mira supercomputer, which has $P = 49152$ compute nodes. There were 4699 jobs submitted in June 2019, and we group the ones submitted each day as a set to form 30 sets of jobs. Figure 8.1(a) plots the number of jobs in each day of the month, varying between 66 and 277. The processor allocations of the jobs vary between 512 and 49152, and the execution times vary between 37 and 86494 seconds. Figure 8.1(b) plots these two parameters for all jobs in the month (with each point representing a job).

In both settings, silent errors are injected to the jobs based on the Exponential distribution as described in Section 8.2.4. To study the impact of error rate, we further define the average failure probability for a set of jobs to be $\bar{q} = 1 - e^{-\lambda \bar{a}}$, where $\bar{a} = \sum_{j=1}^{n} a_j/n$ is the average area of all jobs in the set. Intuitively, $\bar{q}$ represents the probability that a job with the average area over all jobs would fail due to silent errors. For a given value of $\bar{q}$, we can compute the error rate as $\lambda = -\ln(1-\bar{q})/\bar{a}$, and hence the failure probability of any job J_j with area a_j to be $q_j = 1 - e^{-\lambda a_j} = 1 - (1-\bar{q})^{a_j/\bar{a}}$. Based on this $\bar{q}$, we then randomly generate 1000 failure scenarios for the set of jobs following the probabilities. For each failure scenario $\mathbf{f}$, we evaluate the makespans of the heuristics, normalized by the lower bound $L(\mathbf{f}) = \max(t_{\max}(\mathbf{f}), A(\mathbf{f})/P)$ as defined in Equations (8.3) and (8.4). The normalized makespans are then averaged over the

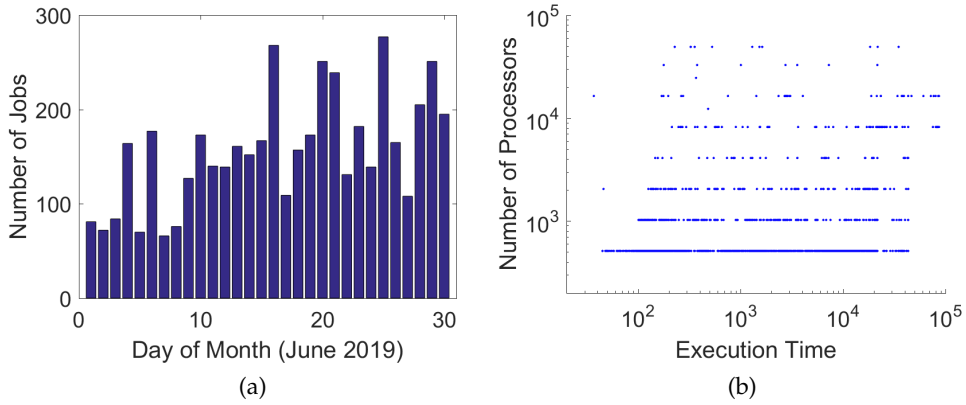


Figure 8.1: Data from the trace logs of the Mira supercomputer.

1000 failure scenarios for comparison.

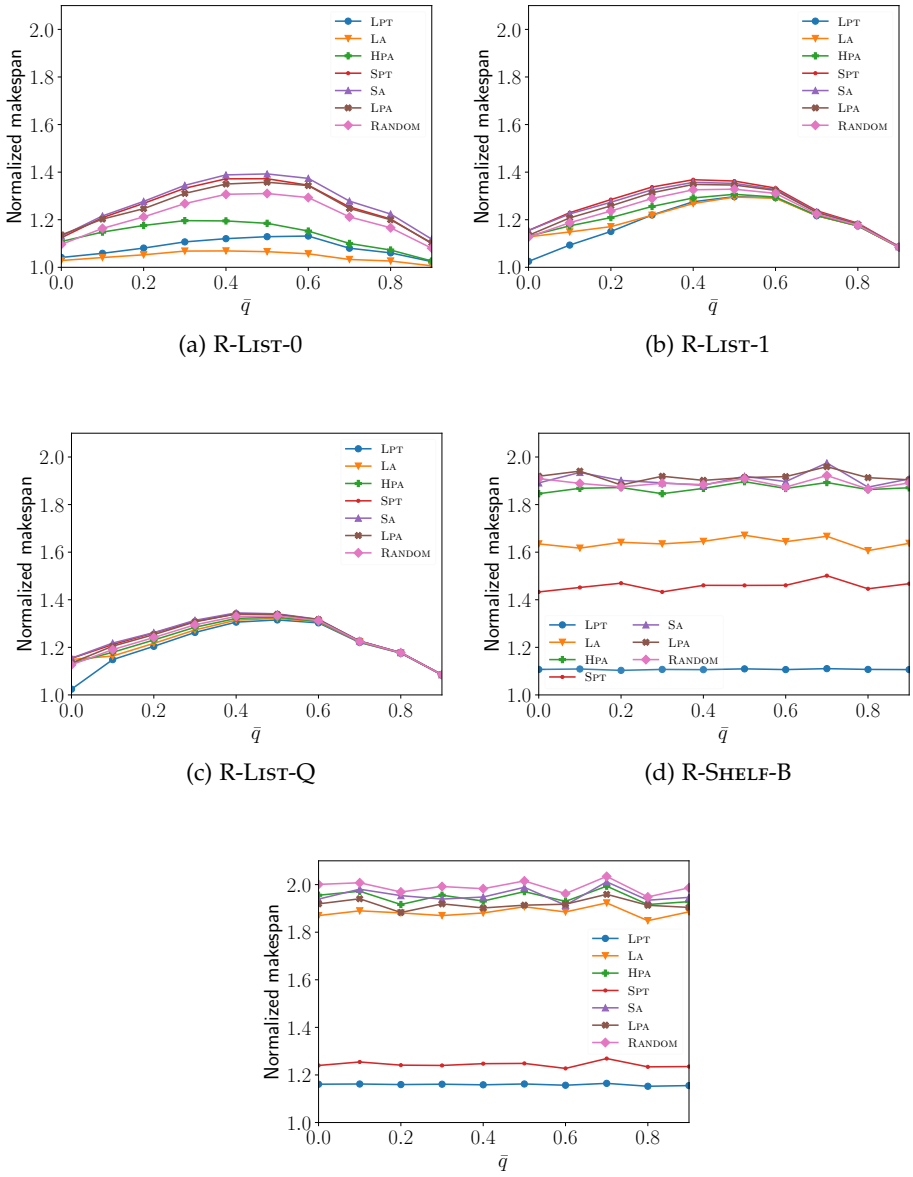
The simulation code for all experiments is publicly available at `http://www.github.com/vlefevre/job-scheduling`.

8.4.2 Results for synthetic jobs

We first compare the performance of different heuristics using synthetic jobs. Here, we focus on assessing the impact of two parameters: the average failure probability $\bar{q}$, and the total number of processors P. The results are averaged over the 30 different sets of jobs.

Figure 8.2 shows the performance of different heuristics when $\bar{q}$ varies from 0 to 0.9. First, we can see that, for all list-based heuristics, the normalized makespans first increase with $\bar{q}$ and then decrease. Indeed, a higher failure probability will result in a larger number of errors, thus increasing the difficulty of scheduling and hence the makespan. However, when the probability is too high, an excessive number of errors will occur, making the optimal scheduler perform equally worse and thus reducing the makespan ratios for the heuristics. For the shelf-based heuristics, the performance appears to be independent of the failure probability. Here, tasks that fail need to wait for the completion of the current shelf to be re-executed, so the number of shelves is mainly determined by the number of re-executions, which influences both the makespan and an optimal scheduler. The normalized makespan is thus mainly decided by the efficiency of the heuristic to fill one shelf, which does not depend on the failure probabilities. Second, the Lpt and La priorities lead to the best performance for all list-based heuristics, with Lpt performing better when $\bar{q}$ is low for R-List-1 and R-List-Q, and La performing better for R-List-0 under any $\bar{q}$. For the shelf-based heuristics, Lpt and Spt are the two best priorities, which is not surprising as the performance of these algorithms is mainly determined by the duration of each shelf.

Figure 8.4(a) further compares the performance of the five heuristics using some of the best priorities. While most list-based heuristics behave similarly when there is



(a) R-LIST-0

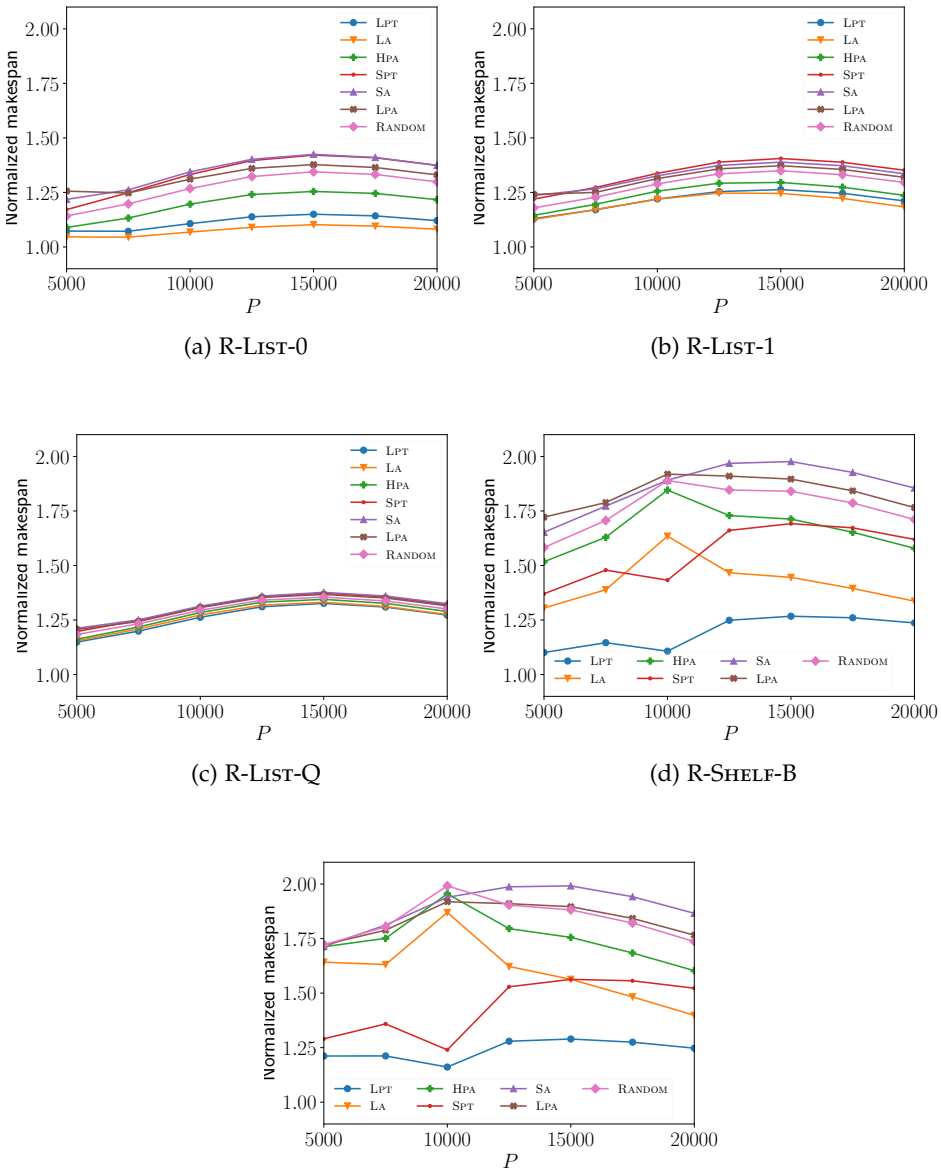
(b) R-LIST-1

(c) R-LIST-Q

(d) R-SHELF-B

(e) R-SHELF-NB

Figure 8.2: Normalized makespans of different heuristics and priority rules over 30 sets of jobs when $\bar{q}$ varies between 0 and 0.9, and $P = 10000$.



(a) R-LIST-0

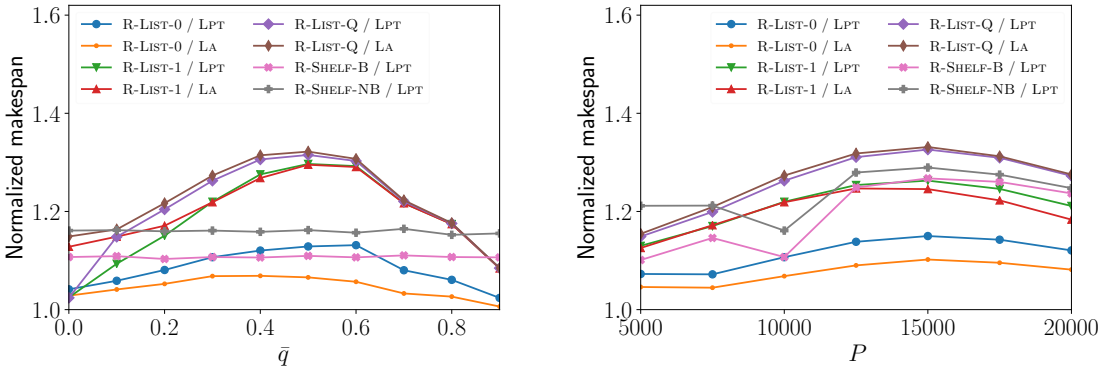
(b) R-LIST-1

(c) R-LIST-Q

(d) R-SHELF-B

(e) R-SHELF-NB

Figure 8.3: Normalized makespans of different heuristics and priority rules over 30 sets of jobs when P varies between 5000 and 20000, and $\bar{q} = 0.3$.



(a) (b)

Figure 8.4: Comparison of different heuristics with the best priority rules Lpt and/or La when: (a) $\bar{q}$ varies between 0 and 0.9, and $P = 10000$; and (b) P varies between 5000 and 20000, and $\bar{q} = 0.3$.

no failure (i.e., $\bar{q} = 0$), R-List-0 clearly outperforms the rest when jobs can fail. This corroborates the theoretical result that R-List-0 (i.e., Greedy) has the lowest approximation ratio regardless of the priority rule and failure scenario. Moreover, R-List-0 is also the heuristic that is least affected by job failures, with an increase in normalized makespan by less than 10% compared to the case of $\bar{q} = 0$, while the other heuristics experience 20-30% increase in normalized makespan. Finally, R-Shelf-NB appears to be the worst heuristic for small and high probabilities of failure with a makespan that is up to 15% higher than that of R-List-0 (when $\bar{q} = 0.9$), while R-List-Q is the worst for medium probabilities (e.g., 26% higher than that of R-List-0 for $\bar{q} = 0.5$). The results are likely due to: (i) the restriction of R-Shelf-NB for building shelves in a schedule, which leads to poor performance for some failure scenarios (such as the one discussed in Section 8.3.3), and hence an increase in the expected makespan, and (ii) the fact that R-List-Q is more affected by the increasing failure probability.

Figure 8.3 shows the performance of different heuristics when the number of processors P varies from 5000 to 20000 while the failure probability is fixed at $\bar{q} = 0.3$. Again, we can see that La and Lpt are the two best priority rules for all heuristics, with La performing better for R-List-0 and R-List-1, and Lpt performing better for other heuristics under all P. Also, the normalized makespans of the heuristics first increase with the number of processors and then tend to decrease. This is because when P is either too small (i.e., total resource is constrained) or too big (i.e., total resource is almost unconstrained), the optimal scheduler tends to have very similar performance as the heuristics.

We further compare the performance of the five heuristics using some of the best priorities in Figure 8.4(b). As in the previous experiment, the best heuristic is R-List-0 with the La priority, which is the least impacted by the total number of processors (with $< 10\%$ variations in normalized makespan). Also, R-List-Q gives the worst performance (with a 23% increase in makespan compared to R-List-0 with La when

Table 8.1: Performance of different heuristics using LPT priority for all 30 days (sets) of jobs from June 2019 on the Mira supercomputer.

$\bar{q}$	Average #failures	Average makespan ratio					Standard deviation					Maximum makespan ratio				
		R-LIST			R-SHELF		R-LIST			R-SHELF		R-LIST			R-SHELF	
		0	1	Q	B	NB	0	1	Q	B	NB	0	1	Q	B	NB
0	0	1.067	1.051	1.051	1.407	1.441	8.78×10^{-2}	8.19×10^{-2}	8.23×10^{-2}	1.29×10^{-1}	1.45×10^{-1}	1.425	1.425	1.425	1.633	1.760
0.05	15.2913	1.031	1.049	1.061	1.129	1.141	6.72×10^{-2}	6.87×10^{-2}	7.76×10^{-2}	1.30×10^{-1}	1.40×10^{-1}	1.278	1.292	1.292	1.489	1.510
0.1	254.453	1.016	1.025	1.028	1.071	1.073	4.66×10^{-2}	4.54×10^{-2}	4.97×10^{-2}	1.03×10^{-1}	1.06×10^{-1}	1.249	1.224	1.245	1.398	1.413

$P = 15000$) and has the largest variation ($\sim$20%) in normalized makespan as the number of processors changes.

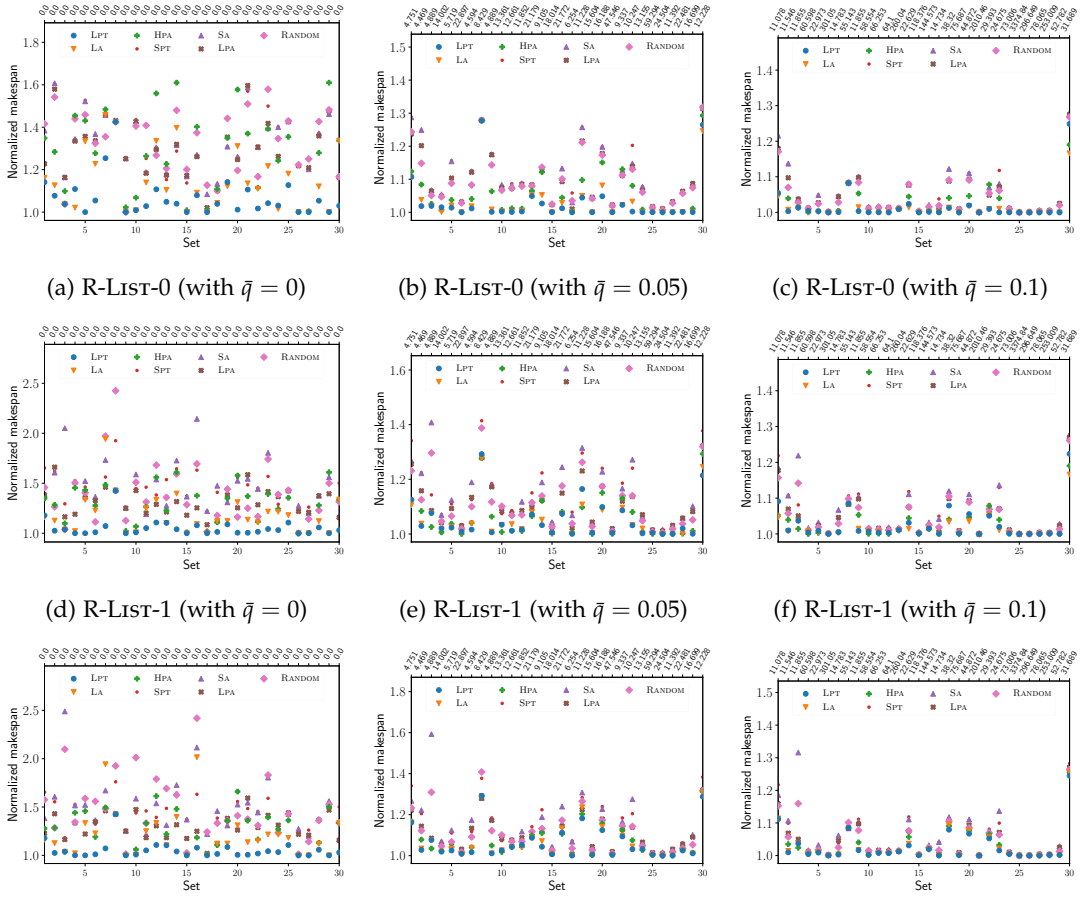
From these experiments, we can see that job failures and processor variations do have an impact on the relative performance of different heuristics. Nevertheless, the makespans of all the heuristics (with good priorities) are never more than 40% worse than the theoretical lower bound, which can be much less than the optimal makespan. The results suggest the robustness of these heuristics, and that they should actually perform really well in practice, even with job failures.

8.4.3 Results for jobs from Mira

We now evaluate the performance of different heuristics using real jobs from the Mira trace logs. Figures 8.5 and 8.6 show the normalized makespans of all heuristics and priority rules under all 30 days (sets) of jobs with and without failures. We observe that the LPT and LA priorities again offer the best performance, with LPT performing better this time for most job sets. This holds for every heuristic on average, especially when there is no failure (i.e., $\bar{q} = 0$). As the failure probability increases, both LPT and LA (and even HPA) give similar performance. The reason is that the processor allocations and execution times of the jobs in Mira are more skewed than those of the synthetic ones. Here, some jobs use a very large number of processors and have long execution times, which make them fail more often even with small values of $\bar{q}$. As a result, the makespan lower bound is largely determined by the total execution times of these jobs, thus any priority rule that favors these jobs will achieve similar performance. Comparing different heuristics, we can see that R-LIST-0 again performs the best and R-SHELF-B the worse, especially with higher failure probability ($\bar{q} = 0.1$). This is consistent with the previous findings and corroborates the analysis.

Table 8.1 summarizes the results of the five heuristics using the LPT priority (which is overall the best one) over 30 days (sets) of jobs, which have an average of 157.63 jobs per day (set). As $\bar{q}$ increases to 0.05 and 0.1, the average number of failures rises to around 15 and 254, respectively. All list-based heuristics have good average makespan ratios that are very close to 1 (with low standard deviations), as well as good maximum makespan ratios that are lower than 1.5, while the two shelf-based heuristics have worse performance in comparison, even when failures are not present. The maximum makespans, however, are never more than 80% of the theoretical lower bound. This again corroborates the results in Section 8.4.2.

Overall, these results confirm the efficacy and robustness of the resilient scheduling

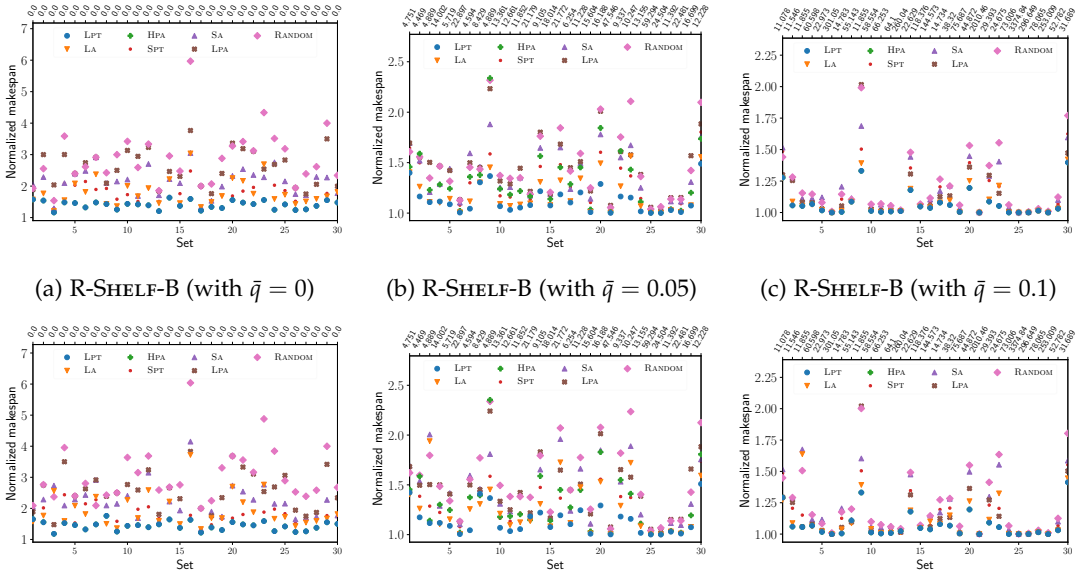


(a) R-LIST-0 (with $\bar{q} = 0$) (b) R-LIST-0 (with $\bar{q} = 0.05$) (c) R-LIST-0 (with $\bar{q} = 0.1$)

(d) R-LIST-1 (with $\bar{q} = 0$) (e) R-LIST-1 (with $\bar{q} = 0.05$) (f) R-LIST-1 (with $\bar{q} = 0.1$)

(g) R-LIST-Q (with $\bar{q} = 0$) (h) R-LIST-Q (with $\bar{q} = 0.05$) (i) R-LIST-Q (with $\bar{q} = 0.1$)

Figure 8.5: Performance of list-based heuristics for 30 job sets using the Mira trace logs (June 2019) with and without failures. Each row represents a different heuristic (R-LIST-0, R-LIST-1 and R-LIST-Q), and each column represents a different failure probability ($\bar{q} = 0$, $\bar{q} = 0.05$ and $\bar{q} = 0.1$). The average number of failures for each job set is indicated on top of each plot.



(a) R-SHELF-B (with $\bar{q} = 0$) (b) R-SHELF-B (with $\bar{q} = 0.05$) (c) R-SHELF-B (with $\bar{q} = 0.1$)

(d) R-SHELF-NB (with $\bar{q} = 0$) (e) R-SHELF-NB (with $\bar{q} = 0.05$) (f) R-SHELF-NB (with $\bar{q} = 0.1$)

Figure 8.6: Performance of shelf-based heuristics for 30 job sets using the Mira trace logs (June 2019) with and without failures. Each row represents a different heuristic (R-SHELF-B and R-SHELF-NB), and each column represents a different failure probability ($\bar{q} = 0$, $\bar{q} = 0.05$ and $\bar{q} = 0.1$). The average number of failures for each job set is indicated on top of each plot.

heuristics, not only for synthetic jobs, but also for real sets of jobs. In particular, both theory and practice have suggested that R-LIST-0 is the best heuristic when silent errors are present, and LPT and LA are the two best priorities for most cases. In all experiments we have conducted, this heuristic achieves a makespan that is within a few percent of the lower bound on average, and never more than 50% in the worst case.

8.5 Background and Related Work

This section describes the background of scheduling rigid parallel jobs and reviews some related work.

8.5.1 Different scheduling flavors and strategies

Historically, scheduling parallel jobs comes in two flavors: if a job requests p processors, either any subset of p processors can be assigned, or only subsets of p *contiguous* processors can be chosen. In the latter case, processors are organized as a linear array and labeled from 1 to P, where P is the total number of processors; then only

neighboring processors (whose labels differ by one) can be assigned to a job. The *contiguous* variant is equivalent to the *rectangle strip packing* problem, where rectangles are to be stacked (without rotation) within a strip of width P: rectangle widths represent processor numbers, and rectangle heights represent execution times.

Most scheduling strategies also come in two flavors: either the schedule is restricted to building *shelves* (also referred to as *levels* in some literature), or it is unrestricted, in which case the jobs are often scheduled based on an ordered *list*. Shelves are subsets of jobs with the same starting time, and for which each of the P processors is used at most once: the height of a shelf is the length of its longest job; when the shorter jobs complete, their processors become idle, but these processors are not reassigned to other jobs until the completion of the longest job of the shelf. Thus, a shelf resembles a bookshelf, hence the name. Shelf-based schedules play an important role in HPC, because they correspond to batched execution scenarios, where jobs are grouped into batches that are scheduled one after another. Note that for shelf-based algorithms, the contiguous and non-contiguous variants collapse.

8.5.2 Offline scheduling of rigid jobs

To minimize the makespan for a set of rigid jobs that are known statically and available initially (i.e., offline), the problem is obviously NP-complete, as it generalizes the problem of scheduling independent jobs on two processors, a variant of the 2-PARTITION problem [89]. Coffman et al. [53] showed that the Next-Fit Decreasing Height (NFDH) algorithm is 3-approximation, and the First-Fit Decreasing-Height (FFDH) algorithm is 2.7-approximation. Both algorithms are shelf-based. See the survey by Lodi et al. [132] for more results and lower bounds on the best possible approximation ratio for shelf-based algorithms, and see Han et al. [100] for the intricate relationship between strip packing and bin packing.

For list-based scheduling, Baker et al. [13] showed that the Bottom-up Left-justified (BL) heuristic while ordering the jobs in decreasing processor requirement achieves 3-approximation. Turek et al. [184] showed that ordering jobs in decreasing execution time is also 3-approximation. Moreover, both algorithms guarantee contiguous processor allocations for all jobs. Without the contiguous processor constraint, several works [88, 184, 81] showed that the greedy list-scheduling heuristic achieves 2-approximation. Finally, Jansen [114] presented a $(3/2 + \epsilon)$-approximation algorithm for any fixed $\epsilon > 0$. This is the best result possible, since a lower bound on the approximation ratio is $3/2$, which holds even when considering asymptotic performance [116].

8.5.3 Online scheduling of rigid jobs

In the online problem, a set of rigid jobs arrive dynamically over time and information of a job is not known until the job has arrived. In this case, the list-based greedy algorithm maintains a competitive ratio of 2 [144, 116]. Chen and Vestjens [48] showed a 1.3473 lower bound on the competitive ratio of any deterministic online algorithm

even when all jobs are sequential. Shmoys et al. [163] showed that by collecting all jobs that arrive during a batch and then scheduling them together in the next batch, one can transform any c-approximation offline algorithm into a $2c$-competitive online algorithm. We point out that this technique, however, does not apply to the model considered in this chapter, because it relies on jobs having fixed, although unknown, release times, whereas the "new job arrivals" in our model (corresponding to failed jobs restarting) depend on the decisions made on-the-fly by the schedulers.

8.5.4 Batch schedulers in practical systems

In practical systems, parallel jobs are often scheduled by batch schedulers [113, 199, 171] that use a combination of *reservation* and *backfilling* strategies: while high-priority jobs are scheduled by reserving processors in advance, low-priority ones are used to fill in the "holes" to improve system utilization. Two popular backfilling strategies are *conservative* [142] and *aggressive* (a.k.a. *EASY*) [129, 168]. The former gives a reservation for every job in the queue, and a lower-priority job is moved forward as long as it does not delay the reservation for any higher-priority job. The latter only gives reservation to the job at the head of the queue (i.e., the one with the highest priority), and backfilling is allowed without delaying this highest-priority job. As jobs arrive over time, most practical schedulers use First-Come First-Serve (FCFS) in conjunction with these strategies to prevent job starvation, but no worst-case performance guarantee is known. Various priority rules have been empirically evaluated to characterize and tune their performance for different metrics (e.g., [170, 193, 90]).

8.6 Conclusion

In this chapter, we have investigated the problem of scheduling rigid jobs onto a parallel platform subject to silent errors. We have revisited the classical scheduling algorithms in this new framework, where jobs that have been struck by errors must be re-executed (possibly many times) until success. We designed resilient list-based and shelf-based scheduling heuristics, along with different priority rules and backfilling strategies. On the theoretical side, we proved that variants of the list-based heuristic achieve a constant approximation ratio (2 or 3 depending whether reservation is used or not). We also showed that any shelf-based heuristic is no longer a constant-factor approximation, while a failure-free variant was known to be a 3-approximation. Extensive simulations conducted using both synthetic jobs and real traces from the Mira supercomputer demonstrate that these heuristics are quite robust, and achieve makespans close to the optimal. As highlighted by the theoretical analysis, the best strategy remains the unrestricted greedy list-based scheduling with no reservations, and good results are obtained in practice when job priorities are assigned by processing times (favoring jobs with long execution times) or by areas (favoring jobs with many processors and/or long execution times).

Some problems remain open, in particular for the study of shelf-based algorithms, whose expected makespan under the Exponential probability distribution is not known to be bounded by a constant factor of the optimal or not. A natural extension of this work would be to consider moldable jobs, whose processor allocations can be decided at launch time. However, for jobs with nonlinear speedup curves, changing the number of processors assigned to a job also changes its error probability under the Exponential probability distribution, thereby severely complicating the problem, and thus calling for the design of novel heuristics.

Chapter 9

I/O scheduling strategy for periodic applications

With the ever-growing need of data in HPC applications, the congestion at the I/O level becomes critical in supercomputers. Architectural enhancement such as burst buffers and pre-fetching are added to machines, but are not sufficient to prevent congestion. Recent online I/O scheduling strategies have been put in place, but they add an additional congestion point and overheads in the computation of applications.

In this work, we show how to take advantage of the periodic nature of HPC applications in order to develop efficient periodic scheduling strategies for their I/O transfers. Our strategy computes once during the job scheduling phase a pattern which defines the I/O behavior for each application, after which the applications run independently, performing their I/O at the specified times. Our strategy limits the amount of congestion at the I/O node level and can be easily integrated into current job schedulers. We validate this model through extensive simulations and experiments on an HPC cluster by comparing it to state-of-the-art online solutions, showing that not only does our scheduler have the advantage of being de-centralized and thus overcoming the overhead of online schedulers, but also that it performs better than the other solutions, improving the application dilation up to 16% and the maximum system efficiency up to 18%. The work in this chapter is joint work with Guillaume Pallez and Ana Gainaru, and has been published in *Transactions on Parallel Computing* (TOPC) [J2].

9.1 Introduction

Nowadays, supercomputing applications create or process TeraBytes of data. This is true in all fields: for example LIGO (gravitational wave detection) generates 1500TB/year [126], the Large Hadron Collider generates 15PB/year, light source projects deal with 300TB of data per day and climate modeling are expected to have to deal with 100EB of data [102].

Management of I/O operations is critical at scale. However, observations on the

Intrepid machine at Argonne National Lab show that I/O transfer can be slowed down up to 70% due to congestion [85]. For instance, when Los Alamos National Laboratory moved from Cielo to Trinity, the peak performance moved from 1.4 Petaflops to 40 Petaflops ($\times$28) while the I/O bandwidth moved to 160 GB/s to 1.45TB/s (only $\times$9) [124]. The same kind of results can be observed at Argonne National Laboratory when moving from Intrepid (0.6 PF, 88 GB/s) and to Mira (10PF, 240 GB/s). While both peak performance and peak I/O improve, the reality is that I/O throughput scales worse than linearly compared to performance, and hence, what should be noticed is a downgrade from 160 GB/PFlop (Intrepid) to 24 GB/PFlop (Mira).

With this in mind, to be able to scale, the conception of new algorithms has to change paradigm: going from a compute-centric model to a data-centric model.

To help with the ever growing amount of data created, architectural improvements such as burst buffers [130, 9] have been added to the system. Work is being done to transform the data before sending it to the disks in the hope of reducing the I/O [65]. However, even with the current I/O footprint burst buffers are not able to completely hide I/O congestion. Moreover, the data used is always expected to grow. Recent works [85] have started working on novel online, centralized I/O scheduling strategies at the I/O node level. However, one of the risks noted on these strategies is the scalability issue caused by potentially high overheads (between 1 and 5% depending on the number of nodes used in the experiments) [85]. Moreover, it is expected that this overhead will increase at larger scale since it need centralized information about all applications running in the system.

In this chapter, we present a decentralized I/O scheduling strategy for supercomputers. We show how to take known HPC application behaviors (namely their periodicity) into account to derive novel static scheduling algorithms. This chapter is an extended version of our previous work [W2]. We improve the main algorithm with a new loop aiming at correcting the size of the period at the end. We also added a detailed complexity analysis and more simulations on synthetic applications to show the wide applicability of our solution. Overall we consider that close to 50% of the technical content is new material.

Periodic Applications Many recent HPC studies have observed independent patterns in the I/O behavior of HPC applications. The periodicity of HPC applications has been well observed and documented [43, 66, 85, 108]: HPC applications alternate between computation and I/O transfer, this pattern being repeated over-time. Carns et al. [43] observed with Darshan [43] the periodicity of four different applications (MADBench2 [44], Chombo I/O benchmark [54], S3D IO [157] and HOMME [143]). Furthermore, in a previous work [85], the authors were able to verify the periodicity of gyrokinetic toroidal code (GTC) [77], Enzo [36], HACC application [96] and CM1 [35]. Furthermore, fault-tolerance techniques (such as periodic checkpointing [58, 105]) also add to this periodic behavior.

The key idea in this project is to take into account those known structural behaviors of HPC applications and to include them in scheduling strategies.

Using this periodicity property, we compute a static periodic scheduling strategy (introduced in our previous work [W2]), which provides a way for each application to know when it should start transferring its I/O (i) hence reducing potential bottlenecks due to I/O congestion, and (ii) without having to consult with I/O nodes every time I/O should be done and hence adding an extra overhead. The main contributions of this chapter are:

- A novel light-weight I/O algorithm that looks at optimizing both application-oriented and platform-oriented objectives;
- The full details of this algorithm and its implementation along with the full complexity analysis;
- A set of extensive simulations and experiments that show that this algorithm performs as well or better than current state of the art heavy-weight online algorithms.
- More simulations to show the performance at scale and a full evaluation to understand how each parameter of the algorithm impacts its performance

Of course, not all applications exhibit a perfect periodic behavior, but based on our experience, many of the HPC scientific applications have this property. This work is preliminary in the sense that we are offering a proof of concept in this chapter and we plan to tackle more complex patterns in the future. In addition, future research will be done for including dynamic schedules instead of only relying on static schedules. This work aims at being the basis of a new class of data-centric scheduling algorithms based on well-know characteristics of HPC applications.

The algorithm presented here is done as a proof of concept to show the efficiency of these light-weight techniques. We believe our scheduler can be implemented naturally into a data scheduler (such as Clarisse [112]) and we provide experimental results backing this claim. We also give hints of how this could be naturally coupled with non-periodic applications. However, this integration is beyond the scope of this chapter. For the purpose of this chapter the applications are already scheduled on the system and are able to receive information about their I/O scheduling. The goal of our I/O scheduler is to eliminate congestion points caused by application interference while keeping the overhead seen by all applications to the minimum. Computing a full I/O schedule over all iterations of all applications is not realistic at today's scale. The process would be too expensive both in time and space. Our scheduler overcomes this by computing a period of I/O scheduling that includes different number of iterations for each application.

The rest of the chapter is organized as follows: in Section 9.2 we present the application model and optimization problem. In Section 9.3 we present our novel algorithm technique as well as a brief proof of concept for a future implementation. In Section 9.4 we present extensive simulations based on the model to show the performance of our algorithm compared to state of the art. We then confirm the performance by performing experiments on a supercomputer to validate the model. We give some background and related work in Section 9.5. We provide concluding remarks and ideas for future research directions in Section 9.6.

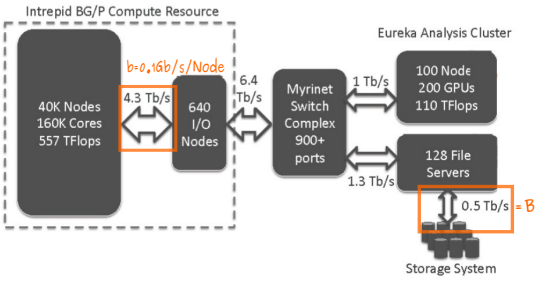


Figure 9.1: Model instantiation for Intrepid [85].

9.2 Model

In this section we use the model introduced in the previous work [85] that has been verified experimentally to be consistent with the behavior of Intrepid and Mira, supercomputers at Argonne.

We consider scientific applications running at the same time on a parallel platform. The applications consist of series of computations followed by I/O operations. On a supercomputer, the computations are done independently because each application uses its own nodes. However, the applications are concurrently sending and receiving data during their I/O phase on a dedicated I/O network. The consequence of this I/O concurrency is congestion between an I/O node of the platform and the file storage.

9.2.1 Parameters

We assume that we have a parallel platform made up of N identical unit-speed nodes, composed of the same number of identical processors, each equipped with an I/O card of bandwidth b (expressed in bytes per second). We further assume having a centralized I/O system with a total bandwidth B (also expressed in bytes per second). This means that the total bandwidth between the computation nodes and an I/O node is $N \cdot b$ while the bandwidth between an I/O node and the file storage is B, with usually $N \cdot b \gg B$. We have instantiated this model for the Intrepid platform on Figure 9.1.

We have K applications, all assigned to independent and dedicated computational resources, but competing for I/O. For each application $\text{App}^{(k)}$ we define:

- Its size: $\text{App}^{(k)}$ executes with $\beta^{(k)}$ dedicated processors;
- Its pattern: $\text{App}^{(k)}$ obeys a pattern that repeats over time. There are $n_{\text{tot}}^{(k)}$ *instances* of $\text{App}^{(k)}$ that are executed one after the other. Each instance consists of two disjoint phases: computations that take a time $w^{(k)}$, followed by I/O transfers for a total volume $\text{vol}_{\text{io}}^{(k)}$. The next instance cannot start before I/O operations for the current instance are terminated.

We further denote by r_k the time when $\text{App}^{(k)}$ is executed on the platform and d_k the

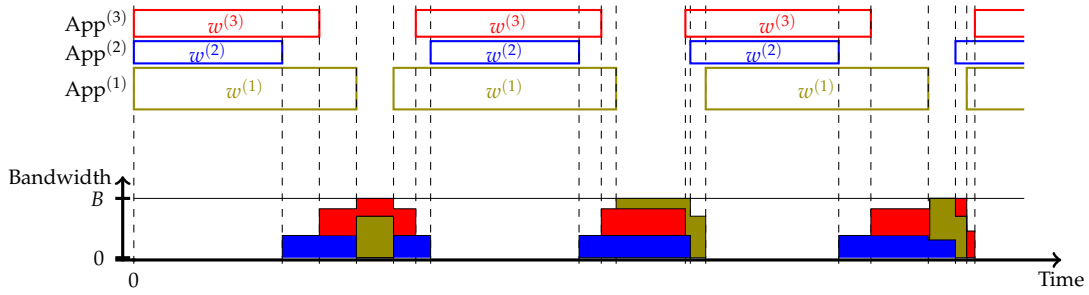


Figure 9.2: Scheduling the I/O of three periodic applications (top: computation, bottom: I/O).

time when the last instance is completed. Finally, we denote by $\gamma^{(k)}(t)$, the bandwidth used by a node on which application App$^{(k)}$ is running, at instant t. For simplicity we assume just one I/O transfer in each loop. However, our model can be extended to work with multiple I/O patterns as long as these are periodic in nature or as long as they are known in advance. In addition, our scheduler can complement I/O prefetching mechanisms like [38, 103] that use the regular patterns within each data access (contiguous/non-contiguous, read or write, parallel or sequential, etc) to avoid congestion.

9.2.2 Execution Model

As the computation resources are dedicated, we can always assume without loss of generality that the next computation chunk starts immediately after completion of the previous I/O transfers, and is executed at full (unit) speed. On the contrary, all applications compete for I/O, and congestion will likely occur. The simplest case is that of a single periodic application App$^{(k)}$ using the I/O system in dedicated mode during a time-interval of duration D. In that case, let γ be the I/O bandwidth used by each processor of App$^{(k)}$ during that time-interval. We derive the condition $\beta^{(k)}\gamma D = \text{vol}_{\text{io}}^{(k)}$ to express that the entire I/O data volume is transferred. We must also enforce the constraints that (i) $\gamma \leq b$ (output capacity of each processor); and (ii) $\beta^{(k)}\gamma \leq B$ (total capacity of I/O system). Therefore, the minimum time to perform the I/O transfers for an instance of App$^{(k)}$ is $\text{time}_{\text{io}}^{(k)} = \frac{\text{vol}_{\text{io}}^{(k)}}{\min(\beta^{(k)}b, B)}$. However, in general many applications will use the I/O system simultaneously, and the bandwidth capacity B will be shared among all applications (see Figure 9.2). Scheduling application I/O will guarantee that the I/O network will not be loaded with more than its designed capacity. Figure 9.2 presents the view of the machine when 3 applications are sharing the I/O system. This translates at the application level to delays inserted before I/O bursts (see Figure 9.3 for application 2's point of view).

This model is very flexible, and the only assumption is that at any instant, all processors assigned to a given application are assigned the same bandwidth. This

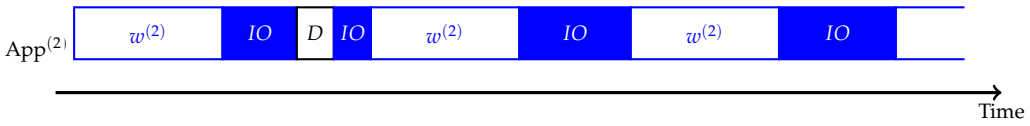


Figure 9.3: Application 2 execution view (D represents the delay in I/O operations)

assumption is transparent for the I/O system and simplifies the problem statement without being restrictive. Again, in the end, the total volume of I/O transfers for an instance of $\text{App}^{(k)}$ must be $\text{vol}_{\text{io}}^{(k)}$, and at any instant, the rules of the game are simple: never exceed the individual bandwidth b of each processor ($\gamma^{(k)}(t) \leq b$ for any k and t), and never exceed the total bandwidth B of the I/O system ($\sum_{k=1}^{K} \beta^{(k)}\gamma^{(k)}(t) \leq B$ for any t).

9.2.3 Objectives

We now focus on the optimization objectives at hand. We use the objectives introduced in [85].

First, the *application efficiency* achieved for each application $\text{App}^{(k)}$ at time t is defined as

$$\tilde{\rho}^{(k)}(t) = \frac{\sum_{i \leq n^{(k)}(t)} w^{(k,i)}}{t - r_k},$$

where $n^{(k)}(t) \leq n_{\text{tot}}^{(k)}$ is the number of instances of application $\text{App}^{(k)}$ that have been executed at time t, since the release of $\text{App}^{(k)}$ at time r_k. Because we execute $w^{(k,i)}$ units of computation followed by $\text{vol}_{\text{io}}^{(k,i)}$ units of I/O operations on instance $\mathcal{I}_i^{(k)}$ of $\text{App}^{(k)}$, we have $t - r_k \geq \sum_{i \leq n^{(k)}(t)} \left(w^{(k,i)} + \text{time}_{\text{io}}^{(k,i)} \right)$. Due to I/O congestion, $\tilde{\rho}^{(k)}$ never exceeds the optimal efficiency that can be achieved for $\text{App}^{(k)}$, namely

$$\rho^{(k)} = \frac{w^{(k)}}{w^{(k)} + \text{time}_{\text{io}}^{(k)}}$$

The two key optimization objectives, together with a rationale for each of them, are:

- SysEff: maximize the peak performance of the platform, namely maximizing the amount of operations per time unit:

$$\text{maximize } \frac{1}{N} \sum_{k=1}^{K} \beta^{(k)} \tilde{\rho}^{(k)}(d_k). \tag{9.1}$$

  The rationale is to squeeze the most flops out of the platform aggregated computational power. We say that this objective is CPU-oriented, as the schedule

will give priority to compute-intensive applications with large $w^{(k)}$and small $\text{vol}_{\text{io}}^{(k)}$values.

- DILATION: minimize the largest slowdown imposed to each application (hence optimizing fairness across applications):

$$\text{minimize} \max_{k=1..K} \frac{\rho^{(k)}}{\tilde{\rho}^{(k)}(d_k)}. \tag{9.2}$$

The rationale is to provide more fairness across applications and corresponds to the stretch in classical scheduling: each application incurs a slowdown factor due to I/O congestion, and we want the largest slowdown factor to be kept minimal. We say that this objective is user-oriented, as it gives each application a guarantee on the relative rate at which the application will progress.

We can now define the optimization problem:

Definition 1 (PERIODIC [85])**.** *We consider a platform of N processors, a set of applications* $\cup_{k=1}^{K}(App^{(k)}, \beta^{(k)}, w^{(k)}, vol_{io}^{(k)})$*, a maximum period* $T_{\max}$*, we want to find a periodic schedule* $\mathcal{P}$ *of period* $T \leq T_{\max}$*, in order to optimize one of the following objectives:*

1. SYSEFF
2. DILATION

Note that it is known that both problems are NP-complete, even in an (easier) offline setting [85].

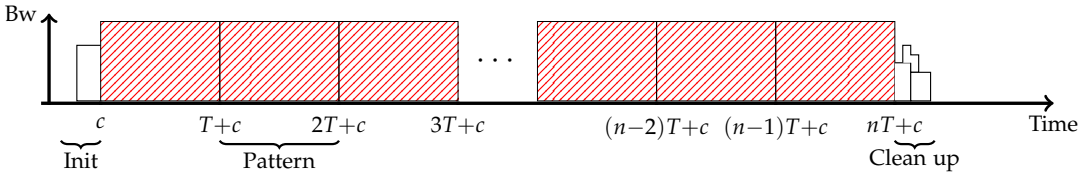
9.3 Periodic scheduling strategy

In general, for an application $\text{App}^{(k)}$, $n_{\text{tot}}^{(k)}$ the number of instances of $\text{App}^{(k)}$ is very large and not polynomial in the size of the problem. For this reason, online schedules have been preferred until now. The key novelty of this chapter is to introduce *periodic schedules* for the K applications. Intuitively, we are looking for a computation and I/O *pattern* of duration T that will be repeated over time (except for *initialization* and *clean up* phases), as shown on Figure 9.4a. In this section, we start by introducing the notion of periodic schedule and a way to compute the application efficiency differently. We then provide the algorithms that are at the core of this work.

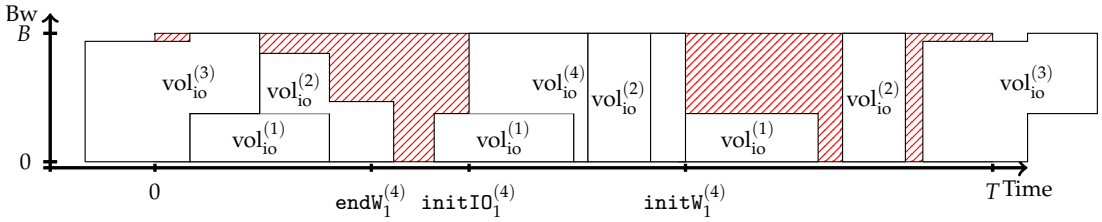
Because there is no competition on computation (no shared resources), we can consider that a chunk of computation directly follows the end of the I/O transfer, hence we need only to represent I/O transfers in this pattern. The bandwidth used by each application during the I/O operations is represented over time, as shown in Figure 9.4b. We can see that an I/O operation can overlap with the previous pattern or the next pattern, but overall, the pattern will just repeat.

To describe a pattern, we use the following notations:

- $n_{\text{per}}^{(k)}$: the number of instances of $\text{App}^{(k)}$ during a pattern.



(a) Periodic schedule (phases)



(b) Detail of I/O in a period/pattern

Figure 9.4: A schedule (above), and the detail of one of its regular pattern (below), where $(w^{(1)} = 3.5; \text{vol}_{\text{io}}^{(1)} = 240; n_{\text{per}}^{(1)} = 3)$, $(w^{(2)} = 27.5; \text{vol}_{\text{io}}^{(2)} = 288; n_{\text{per}}^{(2)} = 3)$, $(w^{(3)} = 90; \text{vol}_{\text{io}}^{(3)} = 350; n_{\text{per}}^{(3)} = 1)$, $(w^{(4)} = 75; \text{vol}_{\text{io}}^{(4)} = 524; n_{\text{per}}^{(4)} = 1)$, and c is the duration of the initilization phase.

- $\mathcal{I}_i^{(k)}$: the i-th instance of App$^{(k)}$ during a pattern.
- $\texttt{initW}_i^{(k)}$: the time of the beginning of $\mathcal{I}_i^{(k)}$. So, $\mathcal{I}_i^{(k)}$ has a computation interval going from $\texttt{initW}_i^{(k)}$ to $\texttt{endW}_i^{(k)} = \texttt{initW}_i^{(k)} + w^{(k)} \bmod T$.
- $\texttt{initIO}_i^{(k)}$: the time when the I/O transfer from the i-th instance of App$^{(k)}$ starts (between $\texttt{endW}_i^{(k)}$ and $\texttt{initIO}_i^{(k)}$, App$^{(k)}$ is idle). Therefore, we have

$$\int_{\texttt{initIO}_i^{(k)}}^{\texttt{initW}_{(i+1)\%n_{\text{per}}^{(k)}}^{(k)}} \beta^{(k)} \gamma^{(k)}(t) dt = \text{vol}_{\text{io}}^{(k)}.$$

Globally, if we consider the two instants per instance $\texttt{initW}_i^{(k)}$ and $\texttt{initIO}_i^{(k)}$, that define the change between computation and I/O phases, we have a total of $S \leq \sum_{k=1}^{K} 2n_{\text{per}}^{(k)}$ distinct instants, that are called the *events* of the pattern.
We define the periodic efficiency of a pattern of size T:

$$\tilde{\rho}_{\text{per}}^{(k)} = \frac{n_{\text{per}}^{(k)} w^{(k)}}{T}. \tag{9.3}$$

For periodic schedules, we use it to approximate the actual efficiency achieved for each application. The rationale behind this can be seen in Figure 9.4. If App$^{(k)}$ is released at time r_k, and the first pattern starts at time $r_k + c$, that is after an initialization phase of duration c, then the main pattern is repeated n times (until time $n \cdot T + r_k + c$),

and finally App$^{(k)}$ ends its execution after a clean-up phase of duration c' at time $d_k = r_k + c + n \cdot T + c'$. If we assume that $n \cdot T \gg c + c'$, then $d_k - r_k \approx n \cdot T$. Then the value of the $\tilde{\rho}^{(k)}(d_k)$ for App$^{(k)}$ is:

$$\begin{aligned}\tilde{\rho}^{(k)}(d_k) &= \frac{\left(n \cdot n_{\text{per}}^{(k)} + \delta\right) w^{(k)}}{d_k - r_k} = \frac{\left(n \cdot n_{\text{per}}^{(k)} + \delta\right) w^{(k)}}{c + n \cdot T + c'} \\ &\approx \frac{n_{\text{per}}^{(k)} w^{(k)}}{T} = \tilde{\rho}_{\text{per}}^{(k)}\end{aligned}$$

where δ can be 1 or 0 depending whether App$^{(k)}$ was executed or not during the clean-up or init phase.

9.3.1 PerSched: a periodic scheduling algorithm

For details in the implementation, we refer the interested reader to the source code available at [83].

The difficulties of finding an efficient periodic schedule are three-fold:

- The right pattern size has to be determined;
- For a given pattern size, the number of instances of each application that should be included in this pattern need to be determined;
- The time constraint between two consecutive I/O transfers of a given application, due to the computation in-between makes naive scheduling strategies harder to implement.

Finding the right pattern size A solution is to find schedules with different pattern sizes between a minimum pattern size T_{min} and a maximum pattern size T_{max}.

Because we want a pattern to have at least one instance of each application, we can trivially set up $T_{\text{min}} = \max_k(w^{(k)} + \text{time}_{\text{io}}^{(k)})$. Intuitively, the larger T_{max} is, the more possibilities we can have to find a good solution. However this also increases the complexity of the algorithm. We want to limit the number of instances of all applications in a schedule. For this reason we chose to have $T_{\text{max}} = O(\max_k(w^{(k)} + \text{time}_{\text{io}}^{(k)}))$. We discuss this hypothesis in Section 9.4, where we give better experimental intuition on finding the right value for T_{max}. Experimentally we observe (Section 9.4, Figure 9.11) that $T_{\text{max}} = 10T_{\text{min}}$ seems to be sufficient.

We then decided on an iterative search where the pattern size increases exponentially at each iteration from T_{min} to T_{max}. In particular, we use a precision ε as input and we iteratively increase the pattern size from T_{min} to T_{max} by a factor $(1 + \varepsilon)$. This allows us to have a polynomial number of iterations. The rationale behind the exponential increase is that when the pattern size gets large, we expect performance to converge to an optimal value, hence needing less the precision of a precise pattern size. Furthermore while we could try only large pattern sizes, it seems important to find a good small pattern size as it simplifies the scheduling step. Hence a more precise search for smaller pattern sizes. Finally, we expect the best performance to cycle with

the pattern size. We verify these statements experimentally in Section 9.4 (Figure 9.10).

Determining the number of instances of each application By choosing $T_{\max} = O(\max_k(w^{(k)} + \text{time}_{\text{io}}^{(k)}))$, we guarantee the maximum number of instances of each application that fit into a pattern is $O\left(\frac{\max_k(w^{(k)}+\text{time}_{\text{io}}^{(k)})}{\min_k(w^{(k)}+\text{time}_{\text{io}}^{(k)})}\right)$.

Instance scheduling Finally, our last item is, given a pattern of size T, how to schedule instances of applications into a periodic schedule.

To do this, we decided on a strategy where we insert instances of applications in a pattern, without modifying dates and bandwidth of already scheduled instances. Formally, we call an application schedulable:

Definition 2 (Schedulable)**.** *Given an existing pattern* $\mathcal{P} = \cup_{k=1}^{K}\left(n_{per}^{(k)}, \cup_{i=1}^{n_{per}^{(k)}}\{\mathtt{initW}_i^{(k)}, \mathtt{initIO}_i^{(k)}, \gamma^{(k)}()\}\right)$*, we say that an application* $App^{(k)}$ *is schedulable if there exists* $1 \leq i \leq n_{per}^{(k)}$*, such that:*

$$\int_{\mathtt{initW}_i^{(k)}+w^{(k)}}^{\mathtt{initIO}_i^{(k)}-w^{(k)}} \min\left(\beta^{(k)}b, B - \sum_l \beta^{(l)}\gamma^{(l)}(t)\right) dt \geq vol_{io}^{(k)} \tag{9.4}$$

To understand Equation (9.4): we are checking that during the end of the computation of the i^{th} instance ($\mathtt{initW}_i^{(k)} + w^{(k)}$), and the beginning of the computation of the $i+1^{\text{th}}$ instance to be, there is enough bandwidth to perform at least a volume of I/O of $\text{vol}_{\text{io}}^{(k)}$. Indeed if a new instance is inserted, $\mathtt{initIO}_i^{(k)} - w^{(k)}$ would then become the beginning of computation of the $i+1^{\text{th}}$ instance. Currently it is just some time before the I/O transfer of the i^{th} instance. We represent it graphically on Figure 9.5.

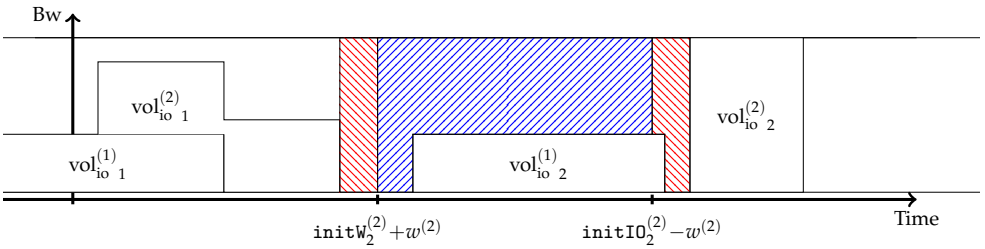


Figure 9.5: Graphical description of Definition 2: two instances of $\text{App}^{(1)}$ and $\text{App}^{(2)}$ are already scheduled. To insert a third instance of $\text{App}^{(2)}$, we need to check that the blue area is greater than $\text{vol}_{\text{io}}^{(2)}$ with the bandwidth constraint (because an instance of $\text{App}^{(1)}$ is already scheduled, the bandwidth is reduced for the new instance of $\text{App}^{(2)}$). The red area is off limit for I/O (used for computations).

With Definition 2, we can now explain the core idea of the instance scheduling part

of our algorithm. Starting from an existing pattern, while there exist applications that are schedulable:

- Amongst the applications that are schedulable, we choose the application that has the worst DILATION. The rationale is that even though we want to increase SYSEFF, we do it in a way that ensures that all applications are treated fairly;
- We insert the instance into an existing scheduling using a procedure INSERT-IN-PATTERN such that (i) the first instance of each application is inserted using procedure INSERT-FIRST-INSTANCE which minimizes the time of the I/O transfer of this new instance, (ii) the other instances are inserted just after the last inserted one.

Note that INSERT-FIRST-INSTANCE is implemented using a water-filling algorithm [86] and INSERT-IN-PATTERN is implemented as described in Algorithm 5 below. We use a different function for the first instance of each application because we do not have any previous instance to use the INSERT-IN-PATTERN function. Thus, the basic idea would be to put them at the beginning of the pattern, but it will be more likely to create congestion if all applications are "synchronized" (for example if all the applications are the same, they will all start their I/O phase at the same time). By using INSERT-FIRST-INSTANCE, every first instance will be at a place where the congestion for it is minimized. This creates a starting point for the subsequent instances.

The function `addInstance` updates the pattern with the new instance, given a list of the intervals $(\mathcal{E}_l, \mathcal{E}_{l'}, b_l)$ during which $\text{App}^{(k)}$ transfers I/O between $\mathcal{E}_l$ and $\mathcal{E}_{l'}$ using a bandwidth b_l.

Correcting the period size In Algorithm 6, the pattern sizes evaluated are determined by $T_{\min}$ and ε. There is no reason why this would be the right pattern size, and one might be interested in reducing it to fit precisely the instances that are included in the solutions that we found.

In order to do so, once a periodic pattern has been computed, we try to improve the best pattern size we found in the first loop of the algorithm, by trying new pattern sizes, close to the previous best one, T_{curr}. To do this, we add a second loop which tries $1/\varepsilon$ uniformly distributed pattern sizes from T_{curr} to $T_{curr}/(1+\varepsilon)$.

With all of this in mind, we can now write PERSCHED (Algorithm 6), our algorithm to construct a periodic pattern. For all pattern sizes tried between $T_{\min}$ and $T_{\max}$, we return the pattern with maximal SYSEFF.

9.3.2 Complexity analysis

In this section we show that our algorithm runs in reasonable execution time. We detail theoretical results that allowed us to reduce the complexity. We want to show the following result:

Theorem 11. *Let* $n_{\max} = \left(\frac{\max_k(w^{(k)} + time_{io}^{(k)})}{\min_k(w^{(k)} + time_{io}^{(k)})} \right)$,

Algorithm 5: INSERT-IN-PATTERN

```
1  procedure INSERT-IN-PATTERN(𝒫, App^(k))
2  begin
3      if App^(k) has 0 instance then
4          return INSERT-FIRST-INSTANCE (𝒫,App^(k));
5      else
6          T_min := +∞ ;
7          Let I_k^{i} be the last inserted instance of App^(k);
8          Let ℰ_0, ℰ_1, ··· , ℰ_{j_i} the times of the events between the end of I_k^{i} + w^(k) and
             the beginning of I_k^{(i+1) mod l_T(k)};
9          For l = 0 ··· j_i − 1, let B_l be the minimum between β^(k) b and the available
             bandwidth during [ℰ_l, ℰ_{l+1}];
10         DataLeft = vol_io^(k);
11         l = 0;
12         sol = [];
13         while DataLeft > 0 and l < j_i do
14             if B_l > 0 then
15                 TimeAdded = min(ℰ_{l+1} − ℰ_l, DataLeft/B_l);
16                 DataLeft -= TimeAdded·B_l;
17                 sol = [(ℰ_l, ℰ_l + TimeAdded, B_l)] + sol;
18             l++;
19         if DataLeft> 0 then
20             return 𝒫
21         else
22             return 𝒫.addInstance(App^(k),sol)
```

PERSCHED$(K', \varepsilon, \{App^{(k)}\}_{1 \le k \le K})$ *runs in*

$$O\left(\left(\left\lceil \frac{1}{\varepsilon} \right\rceil + \left\lceil \frac{\log K'}{\log(1+\varepsilon)} \right\rceil\right) \cdot K^2 \left(n_{\max} + \log K'\right)\right).$$

Algorithm 6: Periodic Scheduling heuristic: PERSCHED

```
1  procedure PERSCHED(K', ε, {App^(k)}_{1≤k≤K})
2  begin
3      T_min ← max_k(w^(k) + time_io^(k));
4      T_max ← K' · T_min;
5      T = T_min;
6      SE ← 0;
7      T_opt ← 0;
8      P_opt ← {};
9      while T ≤ T_max do
10         P = {};
11         while exists a schedulable application do
12             A = {App^(k) | App^(k) is schedulable};
13             Let App^(k) be the element of A minimal with respect to the lexicographic
                 order ( ρ^(k) / ρ̃_per^(k) , w^(k) / time_io^(k) );
14             P ← INSERT-IN-PATTERN(P, App^(k));
15         if SE < SYSEFF(P) then
16             SE ← SYSEFF(P);
17             T_opt ← T;
18             P_opt ← P
19         T ← T · (1 + ε);
20     T ← T_opt;
21     while true do
22         P = {};
23         while exists a schedulable application do
24             A = {App^(k) | App^(k) is schedulable};
25             Let App^(k) be the element of A minimal with respect to the lexicographic
                 order ( ρ^(k) / ρ̃_per^(k) , w^(k) / time_io^(k) );
26             P ← INSERT-IN-PATTERN(P, App^(k));
27         if SYSEFF(P) = (T_opt / T) · SE then
28             P_opt ← P;
29             T ← T − (T_opt − T_opt/(1+ε)) / ⌊1/ε⌋
30         else
31             return P_opt
```

We estimate SYSEFF of a periodic pattern, by replacing $\tilde{\rho}^{(k)}(d_k)$ by $\tilde{\rho}_{per}^{(k)}$ in Equation (9.1)

Some of the complexity results are straightforward. The key results to show are:

- The complexity of the tests "`while` *exists a schedulable application*" on lines 11 and 23
- The complexity of computing $\mathcal{A}$ and finding its minimum element on line 13 and 25.
- The complexity of Insert-In-Pattern

To reduce the execution time, we proceed as follows: instead of implementing the set $\mathcal{A}$, we implement a heap $\tilde{\mathcal{A}}$ that could be summarized as

$$\{\text{App}^{(k)} | \text{App}^{(k)} \text{ is not yet known to } \textbf{not} \text{ be schedulable}\}$$

sorted following the lexicographic order: $\left(\frac{\rho^{(k)}}{\tilde{\rho}^{(k)}_{\text{per}}}, \frac{w^{(k)}}{\text{time}^{(k)}_{\text{io}}}\right)$. Hence, we replace the while loops on lines 11 and 23 by the algorithm snippet described in Algorithm 7. The idea is to avoid calling Insert-In-Pattern after each new inserted instance to know which applications are schedulable.

Algorithm 7: Schedulability snippet

```
11 Ã = ∪_k{App^(k)} (sorted by (ρ^(k)/ρ̃^(k)_per, w^(k)/time^(k)_io));
12 while Ã ≠ ∅ do
13     Let App^(k) be the minimum element of Ã;
14     Ã ← Ã \ {App^(k)};
15     Let P' = Insert-In-Pattern(P, App^(k));
16     if P' ≠ P then
17         P ← P';
18         Insert App^(k) in Ã following (ρ^(k)/ρ̃^(k)_per, w^(k)/time^(k)_io);
```

We then need to show that they are equivalent:

- At all time, the minimum element of $\tilde{\mathcal{A}}$ is minimal amongst the schedulable applications with respect to the order $\left(\frac{\rho^{(k)}}{\tilde{\rho}^{(k)}_{\text{per}}}, \frac{w^{(k)}}{\text{time}^{(k)}_{\text{io}}}\right)$ (shown in Lemma 4);
- If $\tilde{\mathcal{A}} = \emptyset$ then there are no more schedulable applications (shown in Corollary 3).

To show this, it is sufficient to show that (i) at all time, $\mathcal{A} \subset \tilde{\mathcal{A}}$, and (ii) $\tilde{\mathcal{A}}$ is always sorted according to $\left(\frac{\rho^{(k)}}{\tilde{\rho}^{(k)}_{\text{per}}}, \frac{w^{(k)}}{\text{time}^{(k)}_{\text{io}}}\right)$.

Definition 3 (Compact pattern)**.**
We say that a pattern $\mathcal{P} = \cup_{k=1}^{K} \left(n_{per}^{(k)}, \cup_{i=1}^{n_{per}^{(k)}} \{ \mathtt{initW}_i^{(k)}, \mathtt{initIO}_i^{(k)}, \gamma^{(k)}() \} \right)$ *is compact if for all* $1 \leq i < n_{per}^{(k)}$*, either* $\mathtt{initW}_i^{(k)} + w^{(k)} = \mathtt{initIO}_i^{(k)}$*, or for all* $t \in [\mathtt{initW}_i^{(k)}, \mathtt{initIO}_i^{(k)}]$*,* $\sum_l \beta^{(l)} \gamma^{(l)}(t) = B$*.*

Intuitively, this means that, for all applications $\text{App}^{(k)}$, we can only schedule a new instance between $\mathcal{I}_{n_{\text{per}}^{(k)}}^{(k)}$ and $\mathcal{I}_1^{(k)}$.

Lemma 1. *At any time during* PerSched*,* $\mathcal{P}$ *is compact.*

Proof. For each application, either we use Insert-First-Instance to insert the first instance (so $\mathcal{P}$ is compact as there is only one instance of an application at this step), or we use Insert-In-Pattern which inserts an instance just after the last inserted one, which is the definition of being compact. Hence, $\mathcal{P}$ is compact at any time during PerSched. □

Lemma 2. Insert-In-Pattern$(\mathcal{P}, App^{(k)})$ *returns* $\mathcal{P}$*, if and only if* $App^{(k)}$ *is not schedulable.*

Proof. One can easily check that Insert-In-Pattern checks the schedulability of $\text{App}^{(k)}$ only between the last inserted instance of $\text{App}^{(k)}$ and the first instance of $\text{App}^{(k)}$. Furthermore, because of the compactness of $\mathcal{P}$ (Lemma 1), this is sufficient to test the overall schedulability.

Then the test is provided by the last condition $\texttt{Dataleft} > 0$.

- If the condition is false, then the algorithm actually inserts a new instance, so it means that one more instance of $\text{App}^{(k)}$ is schedulable.
- If the condition is true, it means that we cannot insert a new instance after the last inserted one. Because $\mathcal{P}$ is compact, we cannot insert an instance at another place. So if the condition is true, we cannot add one more instance of $\text{App}^{(k)}$ in the pattern.

□

Corollary 2. *In Algorithm 7, an application* $App^{(k)}$ *is removed from* $\tilde{\mathcal{A}}$ *if and only if it is not schedulable.*

Lemma 3. *If an application is not schedulable at some step, it will not be either in the future.*

Proof. Let us suppose that $\text{App}^{(k)}$ is not schedulable at some step. In the future, new instances of other applications can be added, thus possibly increasing the total bandwidth used at each instant. The total I/O load is non-decreasing during the execution of the algorithm. Thus if for all i, we had

$$\int_{\mathtt{initW}_i^{(k)} + w^{(k)}}^{\mathtt{initIO}_i^{(k)} - w^{(k)}} \min \left(\beta^{(k)} b, B - \sum_l \beta^{(l)} \gamma^{(l)}(t) \right) dt < \text{vol}_{\text{io}}^{(k)},$$

then in the future, with new bandwidths used $\gamma'^{(l)}(t) > \gamma^{(l)}(t)$, we will still have that for all i,

$$\int_{\texttt{initW}_i^{(k)}+w^{(k)}}^{\texttt{initIO}_i^{(k)}-w^{(k)}} \min\left(\beta^{(k)}b, B - \sum_l \beta^{(l)}\gamma'^{(l)}(t)\right) dt < \text{vol}_{\text{io}}^{(k)}.$$

□

Corollary 3. *At all time,*

$$\mathcal{A} = \{App^{(k)} | App^{(k)} \text{ is schedulable}\} \subset \tilde{\mathcal{A}}.$$

This is a direct corollary of Corollary 2 and Lemma 3

Lemma 4. *At all time, the minimum element of $\tilde{\mathcal{A}}$ is minimal amongst the schedulable applications with respect to the order $\left(\frac{\rho^{(k)}}{\tilde{\rho}_{per}^{(k)}}, \frac{w^{(k)}}{time_{io}^{(k)}}\right)$ (but not necessarily schedulable).*

Proof. First see that $\{\text{App}^{(k)} | \text{App}^{(k)} \text{ is schedulable}\} \subset \tilde{\mathcal{A}}$. Furthermore, initially the minimality property is true. Then the set $\tilde{\mathcal{A}}$ is modified only when a new instance of an application is added to the pattern. More specifically, only the application that was modified has its position in $\tilde{\mathcal{A}}$ modified. One can easily verify that for all other applications, their order with respect to $\left(\frac{\rho^{(k)}}{\tilde{\rho}_{\text{per}}^{(k)}}, \frac{w^{(k)}}{\text{time}_{\text{io}}^{(k)}}\right)$ has not changed, hence the set is still sorted. □

This concludes the proof that the snippet is equivalent to the while loops. With all this we are now able to show timing results for the version of Algorithm 6 that uses Algorithm 7.

Lemma 5. *The loop on line 21 of Algorithm 6 terminates in at most $\lceil 1/\varepsilon \rceil$ steps.*

Proof. The stopping criteria on line 27 checks that the number of instances did not change when reducing the pattern size. Indeed, by definition for a pattern $\mathcal{P}$,

$$\textsc{SysEff}(\mathcal{P}) = \sum_k \beta^{(k)}\tilde{\rho}_{\text{per}}^{(k)} = \frac{\sum_k \beta^{(k)} n_{\text{per}}^{(k)} w^{(k)}}{T}.$$

Denote SE the SysEff reached in T_{opt} at the end of the while loop on line 9 of Algorithm 6. Let $\textsc{SysEff}(\mathcal{P})$ be the SysEff obtained in $T_{\text{opt}}/(1+\varepsilon)$. By definition,

$$\textsc{SysEff}(\mathcal{P}) < \texttt{SE} \qquad \text{and}$$
$$\frac{T_{\text{opt}}}{1+\varepsilon}\textsc{SysEff}(\mathcal{P}) < T_{\text{opt}}\texttt{SE}.$$

Necessarily, after at most $\lceil 1/\varepsilon \rceil$ iterations, Algorithm 6 exits the loop on line 21.

□

Proof of Theorem 11. There are $\lfloor m \rfloor$ pattern sizes tried where $T_{\min} \cdot (1+\varepsilon)^m = T_{\max}$ in the main "while" loop (line 9), that is

$$m = \frac{\log T_{\max} - \log T_{\min}}{\log(1+\varepsilon)} = \frac{\log K'}{\log(1+\varepsilon)}.$$

Furthermore, we have seen (Lemma 5) that there are a maximum of $\lceil 1/\varepsilon \rceil$ pattern sizes tried of the second loop (line 21).

For each pattern size tried, the cost is dominated by the complexity of Algorithm 7. Let us compute this complexity.

- The construction of $\tilde{\mathcal{A}}$ is done in $O(K \log K)$.
- In sum, each application can be inserted a maximum of $n_{\max}$times in $\tilde{\mathcal{A}}$ (maximum number of instances in any pattern), that is the total of all insertions has a complexity of $O(K \log K n_{\max})$.

We are now interested in the complexity of the different calls to Insert-In-Pattern.

First one can see that we only call Insert-First-Instance K times, and in particular they correspond to the first K calls of Insert-In-Pattern. Indeed, we always choose to insert a new instance of the application that has the largest current slowdown. The slowdown is infinite for all applications at the beginning, until their first instance is inserted (or they are removed from $\tilde{\mathcal{A}}$) when it becomes finite, meaning that the K first insertions will be the first instance of all applications.

During the k-th call, for $1 \leq k \leq K$, there will be $n = 2(k-1)+2$ events (2 for each previously inserted instances and the two bounds on the pattern), meaning that the complexity of Insert-First-Instance will be $O(n \log n)$ (because of the sorting of the bandwidths available by non-increasing order to choose the intervals to use). So overall, the K first calls have a complexity of $O(K^2 \log K)$.

Furthermore, to understand the complexity of the remaining calls to Insert-In-Pattern we are going to look at the end result. In the end there is a maximum of $n_{\max}$instance of each applications, that is a maximum of $2n_{\max}K$ events. For all application $\text{App}^{(k)}$, for all instance $\mathcal{I}_i^{(k)}$k, $1 < i \leq n^{(k)}$, the only events considered in Insert-In-Pattern when scheduling $\mathcal{I}_i^{(k)}$k were the ones between the end of $\texttt{initW}_k^{(i)} + w^{(k)}$ and $\texttt{initW}_{k+1}^{(i)}$. Indeed, since the schedule has been able to schedule $\text{vol}_{\text{io}}^{(k)}$, Insert-In-Pattern will exit the while loop on line 13. Finally, one can see that the events considered for all instances of an application partition the pattern without overlapping. Furthermore, Insert-In-Pattern has a linear complexity in the number of events considered. Hence a total complexity by application of $O(n_{\max}K)$. Finally, we have K applications, the overall time spent in Insert-In-Pattern for inserting new instances is $O(K^2 n_{\max})$.

Hence, with the number of different pattern tried, we obtain a complexity of

$$O\left(\left(\lceil m \rceil + \left\lceil \frac{1}{\varepsilon} \right\rceil\right)\left(K^2 \log K + K^2 n_{\max}\right)\right).$$

□

In practice, both K' and K are small (≈ 10), and ε is close to 0, hence making the complexity $O\left(\frac{n_{\max}}{\varepsilon}\right)$.

9.3.3 High-level implementation, proof of concept

We envision the implementation of this periodic scheduler to take place at two levels:

1) The job scheduler would know the applications profiles (using solutions such as Omnisc'IO [66]). Using profiles it would be in charge of computing a periodic pattern every time an application enters or leaves the system.

2) Application-side I/O management strategies (such as [133, 203, 112, 179]) then would be responsible to ensure the correct I/O transfer at the right time by limiting the bandwidth used by nodes that transfer I/O. The start and end time for each I/O as well as the used bandwidth are described in input files.

To deal with the fact that some applications may not be fully-periodic, several directions could be encompassed:

- Dedicating some part of the IO bandwidth to non-periodic applications depending on the respective IO load of periodic and non-periodic applications;
- Coupling a dynamic I/O scheduler to the periodic scheduler;
- Using burst buffers to protect from the interference caused by non-predictable I/O.

Note that these directions are out of scope for this chapter, as the goal of this chapter aims to show a proof-of-concept. Although future work will be devoted to the study of those directions.

9.4 Evaluation and model validation

Note that the data used for this section and the scripts to generate the figures are available at `https://github.com/vlefevre/IO-scheduling-simu`.

In this section, (i) we assess the efficiency of our algorithm by comparing it to a recent dynamic framework [85], and (ii) we validate our model by comparing theoretical performance (as obtained by the simulations) to actual performance on a real system.

We perform the evaluation in three steps: first we simulate behavior of applications and input them into our model to estimate both Dilation and SysEff of our algorithm (Section 9.4.4) and evaluate these cases on an actual machine to confirm the validity of our model. Finally, in Section 9.4.5 we confirm the intuitions introduced in Section 9.3 to determine the parameters used by PerSched.

9.4.1 Experimental Setup

The platform available for experimentation is Jupiter at Mellanox, Inc. To be able to verify our model, we use it to instantiate our platform model. Jupiter is a Dell PowerEdge R720xd/R720 32-node cluster using Intel Sandy Bridge CPUs. Each node has dual Intel Xeon 10-core CPUs running at 2.80 GHz, 25 MB of L3, 256 KB unified L2 and a separate L1 cache for data and instructions, each 32 KB in size. The system has a total of 64GB DDR3 RDIMMs running at 1.6 GHz per node. Jupiter uses Mellanox ConnectX-3 FDR 56Gb/s InfiniBand and Ethernet VPI adapters and Mellanox SwitchX SX6036 36-Port 56Gb/s FDR VPI InfiniBand switches.

We measured the different bandwidths of the machine and obtained $b = 0.01$GB/s and $B = 3$GB/s. Therefore, when 300 cores transfer at full speed (less than half of the 640 available cores), congestion occurs.

Implementation of scheduler on Jupiter We simulate the existence of such a scheduler by computing beforehand the I/O pattern for each application and providing it as an input file. The experiments require a way to control for how long each application uses the CPU or stays idle waiting to start its I/O in addition to the amount of I/O it is writing to the disk. For this purpose, we modified the IOR benchmark [161] to read the input files that provide the start and end time for each I/O transfer as well as the bandwidth used. Our scheduler generates one such file for each application. The IOR benchmark is split in different sets of processes running independently on different nodes, where each set represents a different application. One separate process acts as the scheduler and receives I/O requests for all groups in IOR. Since we are interested in modeling the I/O delays due to congestion or scheduler imposed delays, the modified IOR benchmarks do not use inter-processor communications. Our modified version of the benchmark reads the I/O scheduling file and adapts the bandwidth used for I/O transfers for each application as well as delaying the beginning of I/O transfers accordingly.

We made experiments on our IOR benchmark and compared the results between periodic and online schedulers as well as with the performance of the original IOR benchmark without any extra scheduler.

9.4.2 Applications and scenarios

In the literature, there are many examples of periodic applications. Carns et al. [43] observed with Darshan [43] the periodicity of four different applications (MADBench2 [44], Chombo I/O benchmark [54], S3D IO [157] and HOMME [143]). Furthermore, in a previous work [85], the authors were able to verify the periodicity of Enzo [36], HACC application [96] and CM1 [35].

Unfortunately, few documents give the actual values for $w^{(k)}$, $\text{vol}_{\text{io}}^{(k)}$and $\beta^{(k)}$. Liu et al. [130] provide different periodic patterns of four scientific applications: Plasma-Physics, Turbulence1, Astrophysics and Turbulence2. They were also the top four

App$^{(k)}$		$w^{(k)}$(s)	$\text{vol}_{\text{io}}^{(k)}$(GB)	$\beta^{(k)}$
Turbulence1	(T1)	70	128.2	32,768
Turbulence2	(T2)	1.2	235.8	4,096
AstroPhysics	(AP)	240	423.4	8,192
PlasmaPhysics	(PP)	7554	34304	32,768

Table 9.1: Details of each application.

Set #	T1	T2	AP	PP
1	0	**10**	0	0
2	0	**8**	**1**	0
3	0	**6**	**2**	0
4	0	**4**	**3**	0
5	0	**2**	0	**1**
6	0	**2**	**4**	0
7	**1**	**2**	0	0
8	0	0	**1**	**1**
9	0	0	**5**	0
10	**1**	0	**1**	0

Table 9.2: Number of applications of each type launched at the same time for each experiment scenario.

write-intensive jobs run on Intrepid in 2011. We chose the most I/O intensive patterns for all applications (as they are the most likely to create I/O congestion). We present these results in Table 9.1. Note that to scale those values to our system, we divided the number of processors $\beta^{(k)}$ by 64, hence increasing $w^{(k)}$by 64. The I/O volume stays constant.

To compare our strategy, we tried all possible combinations of those applications such that the number of nodes used equals 640. That is a total of ten different scenarios that we report in Table 9.2.

9.4.3 Baseline and evaluation of existing degradation

We ran all scenarios on Jupiter without any additional scheduler. In all tested scenarios congestion occurred and decreased the visible bandwidth used by each applications as well as significantly increased the total execution time. We present in Table 9.3 the average I/O bandwidth slowdown due to congestion for the most representative scenarios together with the corresponding values for SysEff. Depending on the I/O transfers per computation ratio of each application as well as how the transfers of multiple applications overlap, the slowdown in the perceived bandwidth ranges between 25% to 65%.

Interestingly, set 1 presents the worst degradation. This scenario is running concurrently ten times the same application, which means that the I/O for all applications are executed almost at the same time (depending on the small differences in CPU execution time between nodes). This scenario could correspond to coordinated checkpoints for an application running on the entire system. The degradation in the perceived bandwidth can be as high as 65% which considerably increases the time to save a checkpoint. The use of I/O schedulers can decrease this cost, making the entire process more efficient.

9.4.4 Comparison to online algorithms

In this subsection, we present the results obtained by running PerSched and the online heuristics from recent work [85]. Because in [85], the authors had different heuristics to optimize either Dilation or SysEff, in this work, the Dilation and SysEff presented

Set #	Application	BW slowdown	SysEff
1	Turbulence 2	65.72%	0.064561
2	Turbulence 2	63.93%	0.250105
	AstroPhysics	38.12%	
3	Turbulence 2	56.92%	0.439038
	AstroPhysics	30.21%	
4	Turbulence 2	34.9%	0.610826
	AstroPhysics	24.92%	
6	Turbulence 2	34.67%	0.621977
	AstroPhysics	52.06%	
10	Turbulence 1	11.79%	0.98547
	AstroPhysics	21.08%	

Table 9.3: Bandwidth slowdown, performance and application slowdown for each set of experiments

are the best reached by *any* of those heuristics. This means that *there are no online solution able to reach them both at the same time*! We show that even in this scenario, our algorithm outperforms simultaneously these heuristics *for both optimization objectives*!

The results presented in [85] represent the state of the art in what can be achieved with online schedulers. Other solutions show comparable results, with [207] presenting similar algorithms but focusing on dilation and [65] having the extra limitation of allowing the scheduling of only two applications.

PerSched takes as input a list of applications, as well as the parameters, presented in Section 9.3, $K' = \frac{T_{\max}}{T_{\min}}$, ε. All scenarios were tested with $K' = 10$ and $\varepsilon = 0.01$.

Simulation results We present in Table 9.4 all evaluation results. The results obtained by running Algorithm 6 are called PerSched. To go further in our evaluation, we also look for the best Dilation obtainable with our pattern (we do so by changing line 15 of PerSched). We call this result *min* Dilation in Table 9.4. This allows us to estimate how far the Dilation that we obtain is from what we can do. Furthermore, we can compute an upper bound to SysEff by replacing $\tilde{\rho}^{(k)}$ by $\rho^{(k)}$ in Equation (9.1):

$$\text{Upper bound} = \frac{1}{N} \sum_{k=1}^{K} \frac{\beta^{(k)} w^{(k)}}{w^{(k)} + \text{time}_{\text{io}}^{(k)}}. \tag{9.5}$$

The first noticeable result is that PerSched almost always outperforms (when it does not, it matches) both the Dilation and SysEff attainable by the online scheduling

Set	Min	Upper bound	PerSched		Online	
	Dilation	SysEff	Dilation	SysEff	Dilation	SysEff
1	1.777	0.172	1.896	0.0973	2.091	0.0825
2	1.422	0.334	1.429	0.290	1.658	0.271
3	1.079	0.495	1.087	0.480	1.291	0.442
4	1.014	0.656	1.014	0.647	1.029	0.640
5	1.010	0.816	1.024	0.815	1.039	0.810
6	1.005	0.818	1.005	0.814	1.035	0.761
7	1.007	0.827	1.007	0.824	1.012	0.818
8	1.005	0.977	1.005	0.976	1.005	0.976
9	1.000	0.979	1.000	0.979	1.004	0.978
10	1.009	0.988	1.009	0.986	1.015	0.985

Table 9.4: Best Dilation and SysEff for our periodic heuristic and online heuristics.

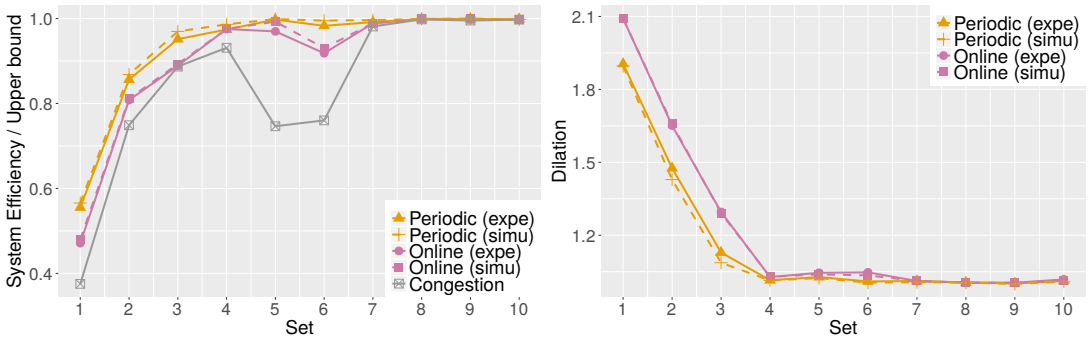
algorithms! This is particularly impressive as these objectives are not obtained by the same online algorithms (hence conjointly), contrarily to the PerSched result.

While the gain is minimal (from 0 to 3%, except SysEff increased by 7% for case 6) when little congestion occurs (cases 4 to 10), the gain is between 9% and 16% for Dilation and between 7% and 18% for SysEff when congestion occurs (cases 1, 2, 3)!

The value of ε has been chosen so that the computation stays short. It seems to be a good compromise as the results are good and the execution times vary from 4 ms (case 10) to 1.8s (case 5) using a Intel Core I7-6700Q. Note that the algorithm is easily parallelizable, as each iteration of the loop is independent. Thus it may be worth considering a smaller value of ε, but we expect no big improvement on the results.

Model validation through experimental evaluation We used the modified IOR benchmark to reproduce the behavior of applications running on HPC systems and analyze the benefits of I/O schedulers. We made experiments on the 640 cores of the Jupiter system. Additionally to the results from both periodic and online heuristics, we present the performance of the system with no additional I/O scheduler.

Figure 9.6 shows the SysEff (normalized using the upper bound in Table 9.4) and Dilation when using the periodic scheduler in comparison with the online scheduler. The results when applications are running without any scheduler are also shown. As observed in the previous section, the periodic scheduler gives better or similar results to the best solutions that can be returned by the online ones, in some cases increasing the system performance by 18% and the dilation by 13%. When we compare to the current strategy on Jupiter, the SysEff reach 48%! In addition, the periodic scheduler has the benefit of not requiring a global view of the execution of the applications at



(a) SysEff/ Upper bound SysEff (b) Dilation

Figure 9.6: Performance for both experimental evaluation and theoretical (simulated) results. The performance estimated by our model is accurate within 3.8% for periodic schedules and 2.3% for online schedules.

every moment of time (by opposition to the online scheduler).

Finally, a key information from those results is the precision of our model introduced in Section 9.2. The theoretical results (based on the model) are within 3% of the experimental results!

This observation is key in launching more thorough evaluation via extensive simulations and is critical in the experimentation of novel periodic scheduling strategies.

Synthetic applications The previous experiments showed that our model can be used to simulate real life machines (that was already observed for Intrepid and Mira in [85]). In this next step, we now rely on synthetic applications and simulation to test extensively the efficiency of our solution.

We considered two platforms (Intrepid and Mira) to run the simulations with concrete values of bandwidths (B, b) and number of nodes (N). The values are reported in Table 9.5.

Platform	B (GB/s)	b (GB/s)	N	$\frac{N \cdot b}{B}$	GFlops/node
Intrepid	64	0.0125	40,960	8	2.87
Mira	240	0.03125	49,152	6	11.18

Table 9.5: Bandwidth and number of processors of each platform used for simulations.

The parameters of the synthetic applications are generated as followed:

- $w^{(k)}$is chosen uniformly at random between 2 and 7500 seconds for Intrepid (and between 0.5 and 1875s for Mira whose nodes are about 4 times faster than Intrepid's nodes),

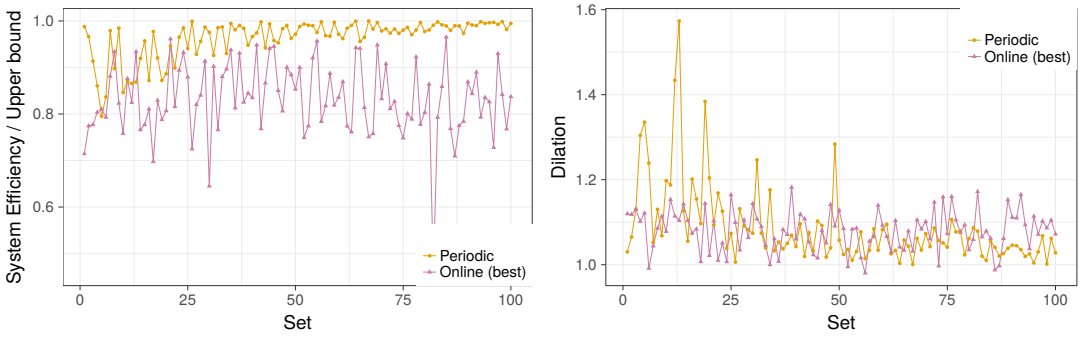
- the volume of I/O data $\text{vol}_{\text{io}}^{(k)}$ is chosen uniformly at random between 100 GB and 35 TB.

These values where based on the applications we previously studied.

We generate the different sets of applications using the following method: let n be the number of unused nodes. At the beginning we set $n = N$.

1. Draw uniformly at random an integer number x between 1 and $\max(1, \frac{n}{4096} - 1)$ (to ensure there are at least two applications).
2. Add to the set an application $\text{App}^{(k)}$ with parameters $w^{(k)}$ and $\text{vol}_{\text{io}}^{(k)}$ set as previously detailed and $\beta^{(k)} = 4096x$.
3. $n \leftarrow n - 4096x$.
4. Go to step 1 if $n > 0$.

We then generated 100 sets for Intrepid (using a total of 40,960 nodes) and 100 sets for Mira (using a total of 49,152 nodes) on which we run the online algorithms (either maximizing the system efficiency or minimizing the dilation) and PerSched. The results are presented on Figures 9.7a and 9.7b for simulations using the Intrepid settings and Figures 9.8a and 9.8b for simulations using the Mira settings.

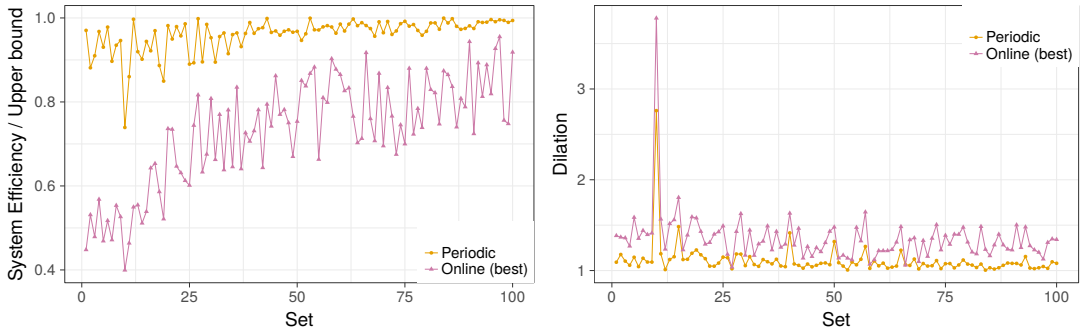


(a) SysEff/ Upper bound SysEff (b) Dilation

Figure 9.7: Comparison between online heuristics and PerSched on synthetic applications, based on Intrepid settings.

We can see that overall, our algorithm increases the system efficiency in almost every case. On average the system efficiency is improved by 16% on intrepid (32% on Mira) with peaks up to 116%! On Intrepid the dilation has overall similar values (an average of 0.6% degradation over the best online algorithm, with variation between 11% improvement and 42% degradation). However on Mira in addition to the improvement in system efficiency, PerSched improves on average by 22% the dilation!

The main difference between Intrepid and Mira is the ratio *compute* ($= N \cdot$ GFlops/node) over *I/O bandwidth* (B). In other terms, that is the speed at which data is created/used over the speed at which data is transferred. Note that as said earlier, the trend is going towards an increase of this ratio. This ratio increases a lot (and hence incurring more I/O congestion) on Mira. To see if this indeed impacts the per-



(a) SysEff/ Upper bound SysEff (b) Dilation

Figure 9.8: Comparison between online heuristics and PerSched on synthetic applications, based on Mira settings.

formance of our algorithm, we plot on Figure 9.9 the average results of running 100 synthetic scenarios on systems with different ratios of *compute* over *I/O*. Basically, the systems we simulate have identical performance to Mira (Table 9.5), and we only increase the GFlops/node by a ratio from 2 to 1024. We plot the SysEff improvement factor ($\frac{\text{SysEff(Online)}}{\text{SysEff(PerSched)}}$) and the Dilation improvement factor ($\frac{\text{Dilation(Online)}}{\text{Dilation(PerSched)}}$).

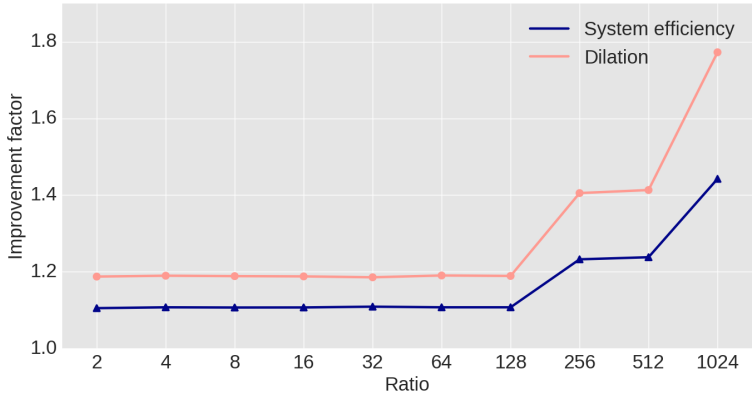


Figure 9.9: Comparison between online heuristics and PerSched on synthetic applications, for different ratios of *compute* over *I/O bandwidth*.

The trend that can be observed is that PerSched seems to be a lot more efficient on systems where congestion is even more critical, showing that this algorithm seems to be even more useful at scale. Specifically, when the ratio increases from 2 to 1024 the gain in SysEff increases on average from 1.1 to 1.5, and at the same time, the gain in Dilation increases from 1.2 to 1.8.

9.4.5 Discussion on finding the best pattern size

The core of our algorithm is a search of the best pattern size via an exponential growth of the pattern size until T_{max}. As stated in Section 9.3, the intuition of the exponential growth is that the larger the pattern size, the less precision is needed for the pattern size as it might be easier to fit many instances of each application. On the contrary, we expect that for small pattern sizes finding the right one might be a precision job.

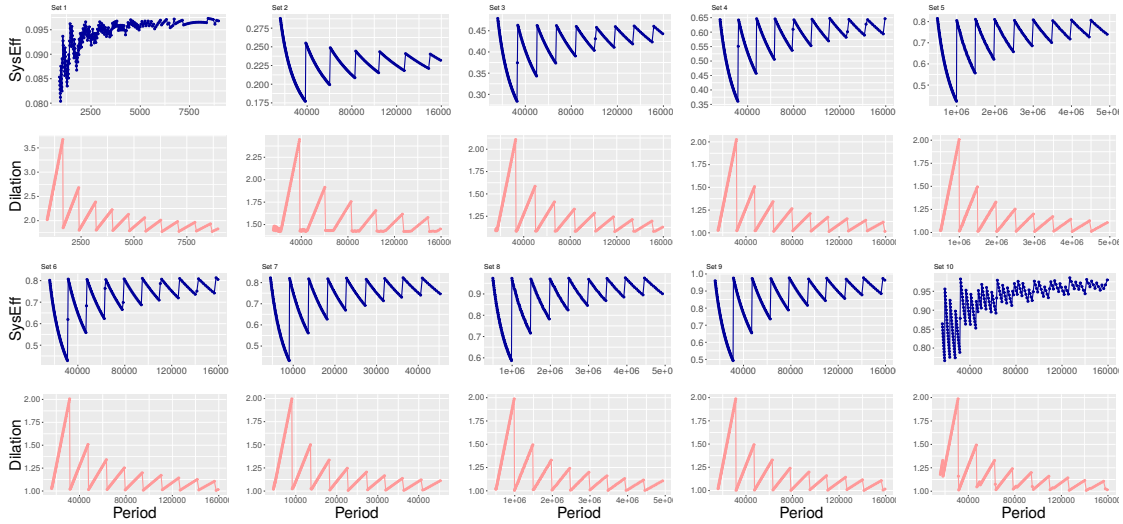


Figure 9.10: Evolution of SysEff (blue) and Dilation (pink) when the pattern size increases for all sets.

We verify this experimentally and plot on Figure 9.10 the SysEff and Dilation found by our algorithm as a function of the pattern size T for all the 10 sets. We can see that they all have very similar shape.

Finally, the last information to determine to tweak PerSched is the value of T_{max}. Remember that we denote $K' = T_{max}/T_{min}$.

Set	n_{inst}	n_{max}
1	11	1.00
2	25	35.2
3	33	35.2
4	247	35.2
5	1086	1110

Set	n_{inst}	n_{max}
6	353	35.2
7	81	10.2
8	251	31.5
9	9	1.00
10	28	3.47

Table 9.6: Maximum number of instances per application (n_{inst}) in the solution returned by PerSched, ratio between longest and shortest application (n_{max}).

To be able to get an estimate of the pattern size returned by PERSCHED, we provide in Table 9.6 (i) the maximum number of instances $n_{\text{inst}}$of any application, and (ii) the ratio $n_{\max} = \frac{\max_k\left(w^{(k)}+\text{time}_{\text{io}}^{(k)}\right)}{\min_k\left(w^{(k)}+\text{time}_{\text{io}}^{(k)}\right)}$. Together along with the fact that the DILATION (Table 9.4) is always below 2 they give a rough idea of K' ($\approx \frac{n_{\text{inst}}}{n_{\max}}$). It is sometimes close to 1, meaning that a small value of K' can be sufficient, but choosing $K' \approx 10$ is necessary in the general case.

We then want to verify the cost of under-estimating $T_{\max}$. For this evaluation all runs were done up to $K' = 100$ with $\varepsilon = 0.01$. Denote SYSEFF(K') (resp. DILATION(K')) the maximum SYSEFF (resp. corresponding DILATION) obtained when running PERSCHED with K'. We plot their normalized version that is:

$$\frac{\text{SYSEFF}(K')}{\text{SYSEFF}(100)} \left(\text{resp. } \frac{\text{DILATION}(K')}{\text{DILATION}(100)}\right)$$

on Figure 9.11. The main noticeable information is that the convergence is very fast:

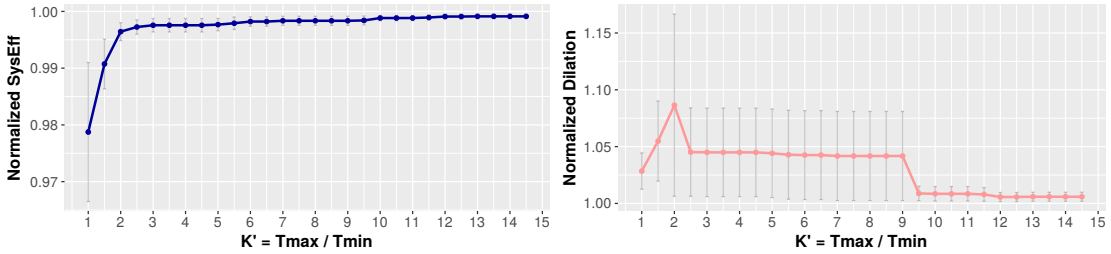


Figure 9.11: Normalized system efficiency and dilation obtained by Algorithm 6 averaged on all 10 sets as a function of K' (with Standard Error bars).

when $K' = 3$, the average SYSEFF is within 0.3% of SYSEFF(100), but the corresponding average DILATION is 5% higher than DILATION(100). If we go to $K' = 10$ then we have a SYSEFF of 0.1% of SYSEFF(100) and a DILATION within 1% of DILATION(100)! Hence validating that choosing $K' = 10$ is sufficient.

9.5 Related Work

Performance variability due to resource sharing can significantly detract from the suitability of a given architecture for a workload as well as from the overall performance realized by parallel workloads [167]. Over the last decade there have been studies to analyze the sources of performance degradation and several solutions have been proposed. In this section, we first detail some of the existing work that copes with I/O congestion and then we present some of the theoretical literature that is similar to our PERIODIC problem.

The storage I/O stack of current HPC systems has been increasingly identified as a performance bottleneck [131]. Significant improvements in both hardware and soft-

ware need to be addressed to overcome oncoming scalability challenges. The study in [111] argues for making data staging coordination driven by generic cross-layer mechanisms that enable global optimizations by enforcing local decisions at node granularity at individual stack layers.

I/O congestion While many other studies suggest that I/O congestion is one of the main problems for future scale platforms [27, 134], few papers focus on finding a solution at the platform level. Some papers consider application-side I/O management and transformation (using aggregate nodes, compression etc) [133, 203, 179]. We consider those work to be orthogonal to our work and able to work jointly. Recently, numerous works focus on using machine learning for auto tuning and performance studies [19, 122]. However these solutions also work at the application level and do not have a global view of the I/O requirements of the system and they need to be supported by a platform level I/O management for better results.

Some paper consider the use of burst buffers to reduce I/O congestion by delaying accesses to the file storage, as they found that congestion occurs on a short period of time and the bandwidth to the storage system is often underutilized [130]. Note that because the computation power increases faster than the I/O bandwidth, this assumption may not hold in the future and the bandwidth may tend to be saturated more often and thus decreasing the efficiency of burst buffers. Kougas et al. [121] present a dynamic I/O scheduling at the application level using burst buffers to stage I/O and to allow computations to continue uninterrupted. They design different strategies to mitigate I/O interference, including partitioning the PFS, which reduces the effective bandwidth non-linearly. Note that their strategy only consider two applications. Tang et al. [178] consider the use of Burst-Buffers to serve the I/O bursts of HPC applications. They prove that a basic reactive draining strategy that empties the burst buffer as soon as possible can lead to a severe degradation of the aggregate I/O throughput. On the other hand, they advocate for a proactive draining strategy, where data is divided into draining segments which are dispersed evenly over the I/O interval, and the burst buffer draining throughput is controlled through adjusting the number of I/O requests issued each time. Recently, Aupy et al. [10] have started discussing coupling of IO scheduling and buffers partitionning to improve data scheduling. They propose an optimal algorithm that determines the minimum buffer size needed to avoid congestion altogether.

The study from [160] offers ways of isolating the performance experienced by applications of one operating system from variations in the I/O request stream characteristics of applications of other operating systems. While their solution cannot be applied to HPC systems, the study offers a way of controlling the coarse grain allocation of disk time to the different operating system instances as well as determining the fine-grain interleaving of requests from the corresponding operating systems to the storage system.

Closer to this work, online schedulers for HPC systems were developed such as previous work [85], the study by Zhou et al [207], and a solution proposed by Dorier et al [65]. In [65], the authors investigate the interference of two applications and

analyze the benefits of interrupting or delaying either one in order to avoid congestion. Unfortunately their approach cannot be used for more than two applications. Another main difference with our previous work is the light-weight approach of this study where the computation is only done once.

The previous study [85] is more general by offering a range of options to schedule each I/O performed by an application. Similarly, the work from [207] also utilizes a global job scheduler to mitigate I/O congestion by monitoring and controlling jobs' I/O operations on the fly. Unlike online solutions, this paper focuses on a decentralized approach where the scheduler is integrated into the job scheduler and computes ahead of time, thus overcoming the need to monitor the I/O traffic of each application at every moment of time.

Periodic schedules As a scheduling problem, our problem is somewhat close to the cyclic scheduling problem (we refer to Hanen and Munier [101] for a survey). Namely there are given a set of activities with time dependency between consecutive tasks stored in a DAG that should be executed on p processors. The main difference is that in cyclic scheduling there is no consideration of a constant time between the end of the previous instance and the next instance. More specifically, if an instance of an application has been delayed, the next instance of the same application is not delayed by the same time. With our model this could be interpreted as not overlapping I/O and computation.

9.6 Conclusion

Performance variation due to resource sharing in HPC systems is a reality and I/O congestion is currently one of the main causes of degradation. Current storage systems are unable to keep up with the amount of data handled by all applications running on an HPC system, either during their computation or when taking checkpoints. In this document we have presented a novel I/O scheduling technique that offers a decentralized solution for minimizing the congestion due to application interference. Our method takes advantage of the periodic nature of HPC applications by allowing the job scheduler to pre-define each application's I/O behavior for their entire execution. Recent studies [66] have shown that HPC applications have predictable I/O patterns even when they are not completely periodic, thus we believe our solution is general enough to easily include the large majority of HPC applications. Furthermore, with the integration of burst buffers in HPC machines [130, 9] periodic schedules could allow to stage data from non periodic applications in Application-side burst buffers, and empty those buffers periodically to avoid congestion. This is the strategy advocated by Tang et al. [178].

We conducted simulations for different scenarios and made experiments to validate our results. Decentralized solutions are able to improve both total system efficiency and application dilation compared to dynamic state-of-the-art schedulers. Moreover, they do not require a constant monitoring of the state of all applications, nor do they require a change in the current I/O stack. One particularly interesting result is for

scenario 1 with 10 identical periodic behaviors (such as what can be observed with periodic checkpointing for fault-tolerance). In this case the periodic scheduler shows a 30% improvement in SysEff. Thus, system wide applications taking global checkpoints could benefit from such a strategy. Our scheduler performs better than existing solutions, improving the application dilation up to 16% and the maximum system efficiency up to 18%. Moreover, based on simulation results, our scheduler shows an even greater improvement for future systems with increasing ratios between the computing power and the I/O bandwidth.

Future work: we believe this work is the initialization of a new set of techniques to deal with the I/O requirements of HPC system. In particular, by showing the efficiency of the periodic technique on simple pattern, we expect this to serve as a proof of concept to open a door to multiple extensions. We give here some examples that we will consider in the future. The next natural directions is to take more complicated periodic shapes for applications (an instance could be composed of sub-instances) as well as different points of entry inside the job scheduler (multiple I/O nodes). We plan to also study the impact of non-periodic applications on this schedule and how to integrate them. This would be modifying the Insert-In-Pattern procedure and we expect that this should work well as well. Another future step would be to study how variability in the compute or I/O volumes impact a periodic schedule. This variability could be important in particular on machines where the inter-processor communication network is shared with the I/O network. Indeed, in those case, I/O would likely be delayed. Finally we plan to model burst buffers and to show how to use them conjointly with periodic schedules, specifically if they allow to implement asynchrone I/O.

Chapter 10

Conclusion

Summary of results

In this thesis, we have addressed several challenges related to exascale computing, more particularly resilience. We have presented some algorithms that allow applications to complete in a failure-prone environment. Moreover, we have been able to derive optimal solutions to minimize the execution time (or the energy consumption in some cases, as it remains one of the big challenges of exascale computing).

In Part I, we have presented an extension of Young and Daly's formula for a multi-level checkpointing scheme, which targets fail-stop errors. In particular, we have been able to derive the optimal (rational) number of checkpoints for each level and that they are equally spaced. We have also provided a dynamic programming algorithm that selects the optimal subset of levels to be used. We have compared the performance of Rigid, Moldable and GridShaped applications with checkpoint/restart (and ABFT for GridShaped applications), and showed that the yield can be improved a lot by requesting a new allocation after a number of failures corresponding to a small percentage of the total resources, even for large values of wait time. Finally, we have designed the first (for fail-stop errors) heuristic for checkpointing generic workflows while mapping tasks to processors. This heuristic achieves a trade-off between checkpointing every task (which is secure but long) and checkpointing no task (which might imply too many re-executions).

Part II deals with problems related to replication. We first proposed an optimal dynamic programming algorithm to decide which tasks are duplicated and/or checkpointed in a linear workflow. The algorithm runs in $\mathcal{O}(n^2)$ and applies to both silent and fail-stop errors. We then studied a divisible application executed on two heterogeneous platforms and we derived the optimal checkpointing period (more exactly, a first or second-order approximation) for a periodic strategy and we compared it to another strategy: on-failure checkpointing. One striking result is that when the speeds of both machines are the same, the optimal checkpointing period is proportional to $\lambda^{-\frac{2}{3}}$ instead of $\lambda^{-\frac{1}{2}}$ as it is classically. We find this result again in our study of process duplication on homogeneous processors with our *restart* strategy. All previous works used the *no-restart* strategy, and we showed that our approach can be significantly bet-

ter in terms of execution time, energy consumption, robustness as regard to the period choice or I/O utilization.

Finally, we studied two scheduling problems in Part III. We first extended approximation results on list scheduling heuristics to a context where tasks can fail and we get constant approximation ratios. We also showed that shelf-based algorithms can no longer be constant-factor approximation algorithms. Then, to tackle the problem of I/O congestion (computational power increases faster than bandwidths), we designed a nice I/O scheduling algorithm at the platform level. We relied on applications that have a periodic behavior to compute a periodic schedule, each time a new application is launched. We were able to improve two metrics: SysEff and Dilation, and we provided a additional scenario to show even more the benefits of I/O scheduling in the future.

For each theoretical result, we validated our model and solution with extensive sets of simulations. These simulations have been helpful in two major ways: assessing the accuracy of our first-order (or second-order) approximations and measuring the practical impact of the proposed solutions. When possible, we used numbers coming from real applications to further strengthen the analysis (for example, we used real logs from Mira [5] in Chapter 8) instead of using only randomly generated input data, which, in some cases, may not reflect the same exact behavior of our algorithms.

Future work and perspectives

On-going work includes two different problems. We first plan to extend the results described in Chapter 8 to moldable jobs, i.e., jobs that do not require a fixed number of processors to run. We consider several models (Amdahl's law, rooftop, communication-bound jobs, ...) and design a new algorithm for attributing the resources to each job before scheduling them. We expect to have higher approximation ratios compared to rigid jobs, as it is already the case when considering jobs that cannot fail [184].

Another study that we target is a comparison of residual checking and ABFT for correcting silent data corruptions (SDC). Residual checking is an ABFT-like technique that has one major asset: while ABFT uses embedded checksums that are transmitted through the computation (for example, for a matrix-matrix product, some rows and columns are added, then the multiplication is done as if these rows and columns are part of the orignal matrices thanks to linear properties), residual checking consists in doing this check only at the end of the computation. This is easier for a developer to implement even for complex transformations like LU and Cholesky. The challenging part in this problem is to detect where the SDCs strike and to find a way to correct them. To the best of our knowledge, using residual checking has only been used to detect errors so far, and preliminary results seem to indicate that it can have a lower overhead than ABFT for matrix-matrix multiplication.

In the long term, future work is composed of several perspectives, as most chapters can be extended with a more complex framework. In Chapter 3, we considered only

perfectly divisible applications, ABFT and checkpoint/restart separately and homogeneous platforms. It would be natural to see if similar results can be derived for jobs obeying Amdahl's law [3], heterogeneous platforms, and check what happens when both ABFT and checkpoint/restart are applied (to deal with both fail-stop and silent errors for example). As we saw in Chapter 4, it is already difficult to select which tasks should be checkpointed in a generic workflow. A challenging extension for Chapter 5 would be to find which tasks should be replicated and/or checkpointed in such a workflow. Finally, an interesting perspective found in Chapter 7 could be to study non-periodic checkpointing periods for a replicated divisible application. Indeed, we provided an example where periodic checkpointing is not optimal with one pair of processors, and we believe that this result holds for several pairs of processors.

Bibliography

[1] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids." In: *1st ACM SWEET SIGMOD*. ACM. 2012.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. "Kepler: an extensible system for design and execution of scientific workflows." In: *Proc. of 16th SSDBM*. IEEE, 2004, pp. 423–424.

[3] G. Amdahl. "The validity of the single processor approach to achieving large scale computing capabilities." In: *AFIPS Conference Proceedings*. Vol. 30. AFIPS Press, 1967, pp. 483–485.

[4] APEX. *APEX Workflows*. Research report SAND2016-2371 and LA-UR-15-29113. LANL, NERSC, SNL, 2016.

[5] Argonne Leadership Computing Facility. *Mira log traces*. `https://reports.alcf.anl.gov/data/mira.html`.

[6] R. A. Ashraf, S. Hukerikar, and C. Engelmann. "Shrink or Substitute: Handling Process Failures in HPC Systems using In-situ Recovery." In: *CoRR* abs/1801.04523 (2018). arXiv: `1801.04523`.

[7] I. Assayad, A. Girault, and H. Kalla. "A Bi-Criteria Scheduling Heuristic for Distributed Embedded Systems under Reliability and Real-Time Constraints." In: *Dependable Systems Networks (DSN)*. IEEE, 2004.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures." In: *Concur. and Comp.: Pract. and Exp.* 23.2 (2011), pp. 187–198.

[9] G. Aupy, O. Beaumont, and L. Eyraud-Dubois. "What size should your Buffers to Disk be?" In: *Proceedings of the 32nd International Parallel Processing Symposium, (IPDPS'18)*. IEEE. 2018.

[10] G. Aupy, O. Beaumont, and L. Eyraud-Dubois. "Sizing and Partitioning Strategies for Burst-Buffers to Reduce IO Contention." In: *Proceedings of the 33rd International Parallel Processing Symposium, (IPDPS'19)*. IEEE. 2019.

[11] G. Aupy, A. Benoit, H. Casanova, and Y. Robert. "Scheduling computational workflows on failure-prone platforms." In: *Int. J. of Networking and Computing* 6.1 (2016), pp. 2–26.

[12] G. Aupy, Y. Robert, and F. Vivien. "Assuming failure independence: are we right to be wrong?" In: *FTS'2017, the Workshop on Fault-Tolerant Systems, in conjunction with Cluster'2017*. IEEE Computer Society Press, 2017.

[13] B. S. Baker, E. G. Coffman, and R. L. Rivest. "Orthogonal Packings in Two Dimensions." In: *SIAM Journal on Computing* 9.4 (1980), pp. 846–855.

[14] P. Balaprakash, L. A. B. Gomez, M.-S. Bouguerra, S. M. Wild, F. Cappello, and P. D. Hovland. "Analysis of the Tradeoffs Between Energy and Run Time for Multilevel Checkpointing." In: *Proc. PMBS'14*. 2014.

[15] L. Bautista Gomez and F. Cappello. "Detecting Silent Data Corruption Through Data Dynamic Monitoring for Scientific Applications." In: *SIGPLAN Notices* 49.8 (2014), pp. 381–382.

[16] L. Bautista Gomez and F. Cappello. "Detecting and Correcting Data Corruption in Stencil Applications through Multivariate Interpolation." In: *FTS*. IEEE, 2015.

[17] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka. "FTI: High Performance Fault Tolerance Interface for Hybrid Systems." In: *Proc. SC'11*. 2011.

[18] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith. "Unprotected Computing: A Large-scale Study of DRAM Raw Error Rate on a Supercomputer." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. Salt Lake City, Utah: IEEE Press, 2016, 55:1–55:11. ISBN: 978-1-4673-8815-3.

[19] Behzad et al. "Taming parallel I/O complexity with auto-tuning." In: *SC13*. 2013.

[20] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. "Optimal Resilience Patterns to Cope with Fail-Stop and Silent Errors." In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pp. 202–211. DOI: 10.1109/IPDPS.2016.39.

[21] A. Benoit, A. Cavelan, F. Cappello, P. Raghavan, Y. Robert, and H. Sun. "Coping with silent and fail-stop errors at scale by combining replication and checkpointing." In: *J. Parallel and Distributed Computing* 122 (2018), pp. 209–225.

[22] A. Benoit, A. Cavelan, F. Ciorba, V. L. Fèvre, and Y. Robert. *Combining checkpointing and replication for reliable execution of linear workflows*. Research report RR-9152. INRIA, 2018.

[23] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. "Assessing General-Purpose Algorithms to Cope with Fail-Stop and Silent Errors." In: *ACM Trans. Parallel Comput.* 3.2 (July 2016), 13:1–13:36. ISSN: 2329-4949. DOI: 10.1145/2897189.

[24] A. Benoit, T. Herault, V. L. Fèvre, and Y. Robert. *Replication Is More Efficient Than You Think: Code and Technical Report*. https://doi.org/10.5281/zenodo.3366221. Aug. 2019.

[25] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. "Lightweight Silent Data Corruption Detection Based on Runtime Data Analysis for HPC Applications." In: *HPDC*. ACM, 2015.

[26] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. "Characterization of scientific workflows." In: *Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2008, pp. 1–10.

[27] R. Biswas, M. Aftosmis, C. Kiris, and B.-W. Shen. "Petascale computing: Impact on future NASA missions." In: *Petascale Computing: Architectures and Algorithms* (2007), pp. 29–46.

[28] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. "Post-failure recovery of MPI communication capability: Design and rationale." In: *International Journal of High Performance Computing Applications* 27.3 (2013), pp. 244–254. DOI: 10.1177/1094342013488238. eprint: http://hpc.sagepub.com/content/27/3/244.full.pdf+html.

[29] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. "An evaluation of User-Level Failure Mitigation support in MPI." In: *Computing* 95.12 (2013), pp. 1171–1184.

[30] 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." In: *Concurrency and Computation: Practice and Experience* (2013). ISSN: 1532-0634. DOI: 10.1002/cpe.3173.

[31] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. "Algorithm-based fault tolerance applied to high performance computing." In: *J. Parallel Distrib. Comput.* 69.4 (2009), pp. 410–416.

[32] S. Boyd and L. Vandenberghe. *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004. ISBN: 0521833787.

[33] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, et al. "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems." In: *Journal of Parallel and Distributed computing* 61.6 (2001), pp. 810–837.

[34] R. Brightwell, K. Ferreira, and R. Riesen. "Transparent Redundant Computing with MPI." In: *EuroMPI*. Springer, 2010.

[35] G. H. Bryan and J. M. Fritsch. "A benchmark simulation for moist nonhydrostatic numerical models." In: *Monthly Weather Review* 130.12 (2002).

[36] G. L. Bryan et al. "Enzo: An Adaptive Mesh Refinement Code for Astrophysics." In: *arXiv:1307.2265* (2013).

[37] E. S. Buneci. "Qualitative Performance Analysis for Large-Scale Scientific Workflows." PhD thesis. Duke University, 2008.

[38] S. Byna, Y. Chen, X. Sun, R. Thakur, and W. Gropp. "Parallel I/O prefetching using MPI file caching and I/O signatures." In: *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. Nov. 2008, pp. 1–12. DOI: 10.1109/SC.2008.5213604.

[39] C. Cao, T. Herault, G. Bosilca, and J. Dongarra. "Design for a Soft Error Resilient Dynamic Task-Based Runtime." In: *IPDPS*. IEEE, 2015, pp. 765–774.

[40] F. Cappello, K. Mohror, et al. *VeloC: Very Low Overhead Checkpointing System*. https://veloc.readthedocs.io/en/latest/. Mar. 2019.

[41] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. "Toward Exascale Resilience." In: *Int. J. High Performance Computing Applications* 23.4 (2009), pp. 374–388.

[42] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. "Toward Exascale Resilience: 2014 update." In: *Supercomputing frontiers and innovations* 1.1 (2014).

[43] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley. "24/7 characterization of petascale I/O workloads." In: *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on* (Jan. 2009), pp. 1–10.

[44] J. Carter, J. Borrill, and L. Oliker. "Performance characteristics of a cosmology package on leading HPC architectures." In: *HiPC*. Springer, 2005, pp. 176–188.

[45] H. Casanova, M. Bougeret, Y. Robert, F. Vivien, and D. Zaidouni. "Using group replication for resilience on exascale systems." In: *Int. Journal of High Performance Computing Applications* 28.2 (2014), pp. 210–224.

[46] H. Casanova, Y. Robert, F. Vivien, and D. Zaidouni. "On the impact of process replication on executions of large-scale parallel applications with coordinated checkpointing." In: *Future Gen. Comp. Syst.* 51 (2015), pp. 7–19.

[47] K. M. Chandy and L. Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems." In: *ACM Transactions on Computer Systems* 3.1 (1985), pp. 63–75.

[48] B. Chen and A. P. Vestjens. "Scheduling on identical machines: How good is LPT in an on-line setting." In: *Operations Research Letters* 21.4 (1997), pp. 165–169.

[49] C. Chen, G. Eisenhauer, M. Wolf, and S. Pande. "LADR: Low-cost Application-level Detector for Reducing Silent Output Corruptions." In: *HPDC*. Tempe, Arizona, 2018, pp. 156–167.

[50] Z. Chen. "Online-ABFT: An Online Algorithm Based Fault Tolerance Scheme for Soft Error Detection in Iterative Methods." In: *SIGPLAN Not.* 48.8 (2013), pp. 167–176.

[51] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. "Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines." In: *Scientific Programming* 5.3 (1996), pp. 173–184.

[52] W. Cirne and F. Berman. "Using Moldability to Improve the Performance of Supercomputer Jobs." In: *J. Parallel Distrib. Comput.* 62.10 (2002), pp. 1571–1601.

[53] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan. "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms." In: *SIAM J. Comput.* 9.4 (1980), pp. 808–826.

[54] P. Colella et al. *Chombo infrastructure for adaptive mesh refinement*. `https://seesar.lbl.gov/ANAG/chombo/`. 2005.

[55] CORAL: Collaboration of Oak Ridge, Argonne and Livermore National Laboratorie. *DRAFT CORAL-2 BUILD STATEMENT OF WORK*. Tech. rep. LLNL-TM-7390608. Lawrence Livermore National Laboratory, Mar. 2018.

[56] S. P. Crago, D. I. Kang, M. Kang, R. Kost, K. Singh, J. Suh, and J. P. Walters. "Programming Models and Development Software for a Space-Based Many-Core Processor." In: *4th Int. Conf. on Space Mission Challenges for Information Technology*. IEEE, 2011, pp. 95–102.

[57] V. Cuevas-Vicenttín, S. C. Dey, S. Köhler, S. Riddle, and B. Ludäscher. "Scientific Workflows and Provenance: Introduction and Research Opportunities." In: *Datenbank-Spektrum* 12.3 (2012), pp. 193–203.

[58] J. T. Daly. "A higher order estimate of the optimum checkpoint interval for restart dumps." In: *Future Generation Comp. Syst.* 22.3 (2006), pp. 303–312.

[59] A. Darte, Y. Robert, and F. Vivien. *Scheduling and automatic parallelization*. Birkhäuser, 2000. ISBN: 978-3-7643-4149-7.

[60] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, et al. "Pegasus: A framework for mapping complex scientific workflows onto distributed systems." In: *Scientific Programming* 13.3 (2005), pp. 219–237.

[61] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. "Pegasus, a Workflow Management System for Science Automation." In: *Future Generation Computer Systems* 46 (2015), pp. 17–35.

[62] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello. "Optimization of multi-level checkpoint model for large scale HPC applications." In: *Proc. IPDPS'14*. 2014.

[63] S. Di, Y. Robert, F. Vivien, and F. Cappello. "Toward an Optimal Online Checkpoint Solution under a Two-Level HPC Checkpoint Model." In: *IEEE Trans. Parallel & Distributed Systems* (2016).

[64] M. e. M. Diouri, O. Glück, L. Lefevre, and F. Cappello. "Energy considerations in checkpointing and fault tolerance protocols." In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. June 2012, pp. 1–6. DOI: 10.1109/DSNW.2012.6264670.

[65] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim. "CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination." In: *IPDPS'14*. 2014.

[66] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross. "Omnisc'IO: a grammar-based approach to spatial and temporal I/O patterns prediction." In: *SC*. IEEE Press. 2014, pp. 623–634.

[67] A. B. Downey. "The structural cause of file size distributions." In: *MASCOTS 2001*. IEEE. 2001, pp. 361–370.

[68] M. Drozdowski. *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer, 2009.

[69] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra. "Algorithm-based Fault Tolerance for Dense Matrix Factorizations." In: *PPoPP*. ACM. 2012, pp. 225–234.

[70] P. Du, P. Luszczek, S. Tomov, and J. Dongarra. "Soft error resilient QR factorization for hybrid system with GPGPU." In: *Journal of Computational Science* 4.6 (2013). Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011, pp. 457–464. ISSN: 1877-7503.

[71] P.-F. Dutot, G. Mounié, and D. Trystram. "Scheduling Parallel Tasks Approximation Algorithms." In: *Handbook of Scheduling - Algorithms, Models, and Performance Analysis*. Ed. by J. Y.-T. Leung. CRC Press, 2004.

[72] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. "Combining partial redundancy and checkpointing for HPC." In: *ICDCS*. IEEE, 2012.

[73] E. N. (Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. "A survey of rollback-recovery protocols in message-passing systems." In: *ACM Computing Survey* 34 (3 2002), pp. 375–408.

[74] E. Elnozahy and J. Plank. "Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery." In: *IEEE Trans. Dependable and Secure Computing* 1.2 (2004), pp. 97–108.

[75] C. Engelmann, H. H. Ong, and S. L. Scorr. "The case for modular redundancy in large-scale high performance computing systems." In: *PDCN*. IASTED, 2009.

[76] C. Engelmann and B. Swen. "Redundant execution of HPC applications with MR-MPI." In: *PDCN*. IASTED, 2011.

[77] S. Ethier, M. Adams, J. Carter, and L. Oliker. "Petascale parallelization of the gyrokinetic toroidal code." In: *VECPAR* (2012).

[78] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, et al. "Askalon: A development and grid computing environment for scientific workflows." In: *Workflows for e-Science*. Springer, 2007, pp. 450–471.

[79] A. Fang, H. Fujita, and A. A. Chien. "Towards Understanding Post-recovery Efficiency for Shrinking and Non-shrinking Recovery." In: *Euro-Par 2015: Parallel Processing Workshops*. Springer, 2015, pp. 656–668.

[80] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. "Theory and Practice in Parallel Job Scheduling." In: *JSSPP*. 1997, pp. 1–34.

[81] A. Feldmann, J. Sgall, and S.-H. Teng. "Dynamic scheduling on parallel machines." In: *Theoretical Computer Science* 130.1 (1994), pp. 49–72.

[82] K. Ferreira, J. Stearley, J. H. I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. "Evaluating the Viability of Process Replication Reliability for Exascale Systems." In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. Seattle, WA, 2011, 44:1–44:12.

[83] V. L. Fèvre. *source code*. `https://github.com/vlefevre/IO-scheduling-simu`. 2017.

[84] P. Flajolet, P. J. Grabner, P. Kirschenhofer, and H. Prodinger. "On Ramanujan's Q-Function." In: *J. Computational and Applied Mathematics* 58 (1995), pp. 103–116.

[85] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Y. Robert, and M. Snir. "Scheduling the I/O of HPC applications under congestion." In: *IPDPS*. IEEE. 2015, pp. 1013–1022.

[86] R. G. Gallager. *Information theory and reliable communication*. Vol. 2. Springer, 1968.

[87] R. G. Gallager. *Stochastic Processes: Theory for Applications*. New York, NY, USA: Cambridge University Press, 2014.

[88] M. R. Garey and R. L. Graham. "Bounds for multiprocessor scheduling with resource constraints." In: *SIAM J. Comput.* 4.2 (1975), pp. 187–200.

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

[90] E. Gaussier, J. Lelong, V. Reis, and D. Trystram. "Online Tuning of EASY-Backfilling using Queue Reordering Policies." In: *IEEE Transactions on Parallel and Distributed Systems* 29.10 (2018), pp. 2304–2316.

[91] C. George and S. S. Vadhiyar. "ADFT: An Adaptive Framework for Fault Tolerance on Large Scale Systems using Application Malleability." In: *Procedia Computer Science* 9 (2012), pp. 166–175.

[92] R. Guerraoui and A. Schiper. "Fault-tolerance by replication in distributed systems." In: *Reliable Software Technologies — Ada-Europe '96*. Ed. by A. Strohmeier. 1996, pp. 38–57.

[93] P.-L. Guhur, H. Zhang, T. Peterka, E. Constantinescu, and F. Cappello. "Lightweight and Accurate Silent Data Corruption Detection in Ordinary Differential Equation Solvers." In: *Euro-Par*. 2016.

[94] Y. Guo, W. Bland, P. Balaji, and X. Zhou. "Fault tolerant MapReduce-MPI for HPC clusters." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. 2015, 34:1–34:12.

[95] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari. "Failures in Large Scale Systems: Long-term Measurement, Analysis, and Implications." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado, 2017, 44:1–44:12. ISBN: 978-1-4503-5114-0.

[96] S. Habib et al. "The universe at extreme scale: multi-petaflop sky simulation on the BG/Q." In: *SC12*. IEEE Computer Society. 2012, p. 4.

[97] D. Hakkarinen and Z. Chen. "Multilevel Diskless Checkpointing." In: *IEEE Transactions on Computers* 62.4 (2013), pp. 772–783.

[98] D. Hakkarinen, P. Wu, and Z. Chen. "Fail-Stop Failure Algorithm-Based Fault Tolerance for Cholesky Decomposition." In: *Parallel and Distributed Systems, IEEE Transactions on* 26.5 (May 2015), pp. 1323–1335. ISSN: 1045-9219.

[99] L. Han, L.-C. Canon, H. Casanova, Y. Robert, and F. Vivien. "Checkpointing workflows for fail-stop errors." In: *IEEE Transactions on Computers* (2018).

[100] X. Han, K. Iwama, D. Ye, and G. Zhang. "Strip Packing vs. Bin Packing." In: *Algorithmic Aspects in Information and Management*. Ed. by M.-Y. Kao and X.-Y. Li. Springer, 2007, pp. 358–367.

[101] C. Hanen and A. Munier. *Cyclic scheduling on parallel processors: an overview*. Citeseer, 1993.

[102] B. Harrod. *Big data and scientific discovery*. 2014.

[103] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X.-H. Sun. "I/O Acceleration with Pattern Detection." In: *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*. HPDC '13. New York, New York, USA: ACM, 2013, pp. 25–36. ISBN: 978-1-4503-1910-2. DOI: 10.1145/2493123.2462909.

[104] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, and F. Cappello. "Modeling and tolerating heterogeneous failures in large parallel systems." In: *Proc. ACM/IEEE Supercomputing'11*. ACM Press, 2011.

[105] T. Herault, Y. Robert, A. Bouteiller, D. Arnold, K. Ferreira, G. Bosilca, and J. Dongarra. "Optimal Cooperative Checkpointing for Shared High-Performance Computing Platforms." In: *APDCM 2018, Vancouver, Canada, May 2018*. 2018.

[106] T. Hérault and Y. Robert, eds. *Fault-Tolerance Techniques for High-Performance Computing*. Computer Communications and Networks. Springer Verlag, 2015.

[107] A. Hori, K. Yoshinaga, T. Herault, A. Bouteiller, G. Bosilca, and Y. Ishikawa. "Sliding Substitution of Failed Nodes." In: *Proceedings of the 22Nd European MPI Users' Group Meeting*. EuroMPI '15. Bordeaux, France: ACM, 2015, 14:1–14:10. ISBN: 978-1-4503-3795-3. DOI: 10.1145/2802658.2802670.

[108] W. Hu, G.-m. Liu, Q. Li, Y.-h. Jiang, and G.-l. Cai. "Storage wall for exascale supercomputing." In: *Journal of Zhejiang University-SCIENCE* 2016 (2016), pp. 10–25.

[109] K.-H. Huang and J. A. Abraham. "Algorithm-Based Fault Tolerance for Matrix Operations." In: *IEEE Trans. Comput.* 33.6 (1984), pp. 518–528.

[110] Z. Hussain, T. Znati, and R. Melhem. "Partial Redundancy in HPC Systems with Non-uniform Node Reliabilities." In: *SC '18*. IEEE, 2018.

[111] F. Isaila and J. Carretero. "Making the case for data staging coordination and control for parallel applications." In: *Workshop on Exascale MPI at Supercomputing Conference*. 2015.

[112] F. Isaila, J. Carretero, and R. Ross. "Clarisse: A middleware for data-staging coordination and control on large-scale hpc platforms." In: *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE. 2016, pp. 346–355.

[113] D. B. Jackson, Q. Snell, and M. J. Clement. "Core Algorithms of the Maui Scheduler." In: *JSSPP*. 2001, pp. 87–102.

[114] K. Jansen. "A (3/2+ϵ) Approximation Algorithm for Scheduling Moldable and Non-moldable Parallel Tasks." In: *SPAA*. Pittsburgh, Pennsylvania, USA, 2012, pp. 224–235.

[115] H. Jin, X.-H. Sun, Z. Zheng, Z. Lan, and B. Xie. "Performance Under Failures of DAG-based Parallel Computing." In: *CCGRID '09*. IEEE Computer Society, 2009.

[116] B. Johannes. "Scheduling Parallel Jobs to Minimize the Makespan." In: *J. of Scheduling* 9.5 (2006), pp. 433–452.

[117] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. "Characterizing and profiling scientific workflows." In: *Future Generation Computer Systems* 29.3 (2013), pp. 682–692.

[118] E. Kail, P. fchtpen, and M. Kozlovszky. "A novel adaptive checkpointing method based on information obtained from workflow structure." In: *Computer Science* 17.3 (2016).

[119] G. Kandaswamy, A. Mandal, and D. A. Reed. "Fault Tolerance and Recovery of Scientific Workflows on Computational Grids." In: *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*. CCGRID '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 777–782. ISBN: 978-0-7695-3156-4. DOI: 10.1109/CCGRID.2008.79.

[120] D. Kondo, B. Javadi, A. Iosup, and D. Epema. "The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems." In: *Cluster Computing and the Grid, IEEE International Symposium on* (2010), pp. 398–407. DOI: http://doi.ieeecomputersociety.org/10.1109/CCGRID.2010.71.

[121] A. Kougkas, M. Dorier, R. Latham, R. Ross, and X.-H. Sun. "Leveraging Burst Buffer Coordination to Prevent I/O Interference." In: *IEEE International Conference on eScience*. IEEE. 2016.

[122] S. Kumar et al. "Characterization and modeling of PIDX parallel I/O for performance optimization." In: *SC*. ACM. 2013.

[123] S. Kumar. "Fundamental limits to Moore's law." In: *arXiv preprint arXiv:1511.05956* (2015).

[124] A. N. Lab. *The Trinity project*. http://www.lanl.gov/projects/trinity/.

[125] LANL. *Computer Failure Data Repository*. https://www.usenix.org/cfdr-data. 2006.

[126] A. Lazzarini. *Advanced LIGO Data & Computing*. 2003.

[127] T. Leblanc, R. Anand, E. Gabriel, and J. Subhlok. "VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes." In: *16th European PVM/MPI Users' Group Meeting*. Springer-Verlag, 2009, pp. 124–133.

[128] Y. LI and L. B. KISH. "HEAT, SPEED AND ERROR LIMITS OF MOORE'S LAW AT THE NANO SCALES." In: *Fluctuation and Noise Letters* 06.02 (2006), pp. L127–L131. DOI: 10.1142/S0219477506003215. eprint: https://doi.org/10.1142/S0219477506003215.

[129] D. A. Lifka. "The ANL/IBM SP Scheduling System." In: *JSSPP*. 1995, pp. 295–303.

[130] N. Liu et al. "On the Role of Burst Buffers in Leadership-Class Storage Systems." In: *MSST/SNAPI*. 2012.

[131] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. J. Wright. "A year in the life of a parallel file system." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press. 2018, p. 74.

[132] A. Lodi, S. Martello, and M. Monaci. "Two-dimensional packing problems: A survey." In: *European Journal of Operational Research* 141.2 (2002), pp. 241–252.

[133] J. Lofstead et al. "Managing variability in the IO performance of petascale storage systems." In: *SC*. IEEECS. 2010.

[134] J. Lofstead and R. Ross. "Insights for exascale IO APIs from building a petascale IO API." In: *SC13*. ACM. 2013, p. 87.

[135] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, et al. "Top ten exascale research challenges." In: *DOE ASCAC subcommittee report* (2014), pp. 1–86.

[136] R. E. Lyons and W. Vanderkulk. "The use of triple-modular redundancy to improve computer reliability." In: *IBM J. Res. Dev.* 6.2 (1962), pp. 200–209.

[137] D. P. Mehta, C. Shetters, and D. W. Bouldin. "Meta-Algorithms for Scheduling a Chain of Coarse-Grained Tasks on an Array of Reconfigurable FPGAs." In: *VLSI Design* (2013).

[138] P. Mendygral, N. Radcliffe, K. Kandalla, D. Porter, B. O'Neill, C. Nolting, P. Edmon, J. Donnert, and T. Jones. "WOMBAT: A Scalable and High-performance Astrophysical Magnetohydrodynamics Code." In: *The Astrophysical Journal Supplement Series* 228 (Feb. 2017), p. 23. DOI: 10.3847/1538-4365/aa5b9c.

[139] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[140] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System." In: *Proc. of the ACM/IEEE SC Conf.* 2010, pp. 1–11.

[141] J. E. Moreira and V. K. Naik. "Dynamic resource management on distributed systems using reconfigurable applications." In: *IBM Journal of Research and Development* 41.3 (1997), pp. 303–330.

[142] A. W. Mu'alem and D. G. Feitelson. "Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling." In: *IEEE Trans. Parallel Distrib. Syst.* 12.6 (2001), pp. 529–543.

[143] R. Nair and H. Tufo. "Petascale atmospheric general circulation models." In: *Journal of Physics: Conference Series*. Vol. 78. IOP Publishing. 2007, p. 012078.

[144] E. Naroska and U. Schwiegelshohn. "On an On-line Scheduling Problem for Parallel Jobs." In: *Inf. Process. Lett.* 81.6 (2002), pp. 297–304.

[145] X. Ni, E. Meneses, N. Jain, and L. V. Kalé. "ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection." In: *Proc. SC'13*. ACM, 2013.

[146] X. Ni, E. Meneses, and L. V. Kalé. "Hiding checkpoint overhead in HPC applications with a semi-blocking algorithm." In: *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*. IEEE Computer Society, 2012, pp. 364–372.

[147] T. O'Gorman. "The effect of cosmic rays on the soft error rate of a DRAM at ground level." In: *IEEE Trans. Electron Devices* 41.4 (1994), pp. 553–557.

[148] R. Oldfield, S. Arunagiri, P. Teller, S. Seelam, M. Varela, R. Riesen, and P. Roth. "Modeling the Impact of Checkpoints on Next-Generation Systems." In: *Proc. of IEEE MSST*. 2007, pp. 30–46. DOI: 10.1109/MSST.2007.4367962.

[149] Pegasus. *Pegasus Workflow Generator*. https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator. 2014.

[150] A. Petitet, H. Casanova, J. Dongarra, Y. Robert, and R. C. Whaley. "Parallel and Distributed Scientific Computing: A Numerical Linear Algebra Problem Solving Environment Designer's Perspective." In: *Handbook on Parallel and Distributed Processing*. Ed. by J. Blazewicz, K. Ecker, B. Plateau, and D. Trystram. Available as LAPACK Working Note 139. Springer Verlag, 1999.

[151] J. Plank, K. Li, and M. Puening. "Diskless checkpointing." In: *IEEE Trans. Parallel Dist. Systems* 9.10 (1998), pp. 972–986. ISSN: 1045-9219.

[152] A. Pothen and C. Sun. "A mapping algorithm for parallel sparse Cholesky factorization." In: *SIAM J. on Scientific Computing* 14.5 (1993), pp. 1253–1257.

[153] S. Prabhakaran. "Dynamic Resource Management and Job Scheduling for High Performance Computing." PhD thesis. Technische Universität Darmstadt, 2016.

[154] S. Prabhakaran, M. Neumann, and F. Wolf. "Efficient Fault Tolerance Through Dynamic Node Replacement." In: *18th Int. Symp. on Cluster, Cloud and Grid Computing CCGRID*. IEEE Computer Society, 2018, pp. 163–172.

[155] M. W. Rashid and M. C. Huang. "Supporting highly-decoupled thread-level redundancy for parallel programs." In: *14th Int. Conf. on High-Performance Computer Architecture (HPCA)*. IEEE, 2008, pp. 393–404.

[156] R. Riesen, K. Ferreira, and J. Stearley. "See applications run and throughput jump: The case for redundant computing in HPC." In: *Proc. of the Dependable Systems and Networks Workshops*. 2010, pp. 29–34.

[157] Sankaran et al. "Direct numerical simulations of turbulent lean premixed combustion." In: *Journal of Physics: conference series*. Vol. 46. IOP Publishing. 2006, p. 38.

[158] N. El-Sayed and B. Schroeder. "Reading between the lines of failure logs: Understanding how HPC systems fail." In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575356.

[159] B. Schroeder and G. A. Gibson. "Understanding Failures in Petascale Computers." In: *Journal of Physics: Conference Series* 78.1 (2007).

[160] S. R. Seelam and P. J. Teller. "Virtual I/O Scheduler: A Scheduler of Schedulers for Performance Virtualization." In: *Proceedings VEE*. San Diego, California, USA: ACM, 2007, pp. 105–115.

[161] H. Shan and J. Shalf. "Using IOR to Analyze the I/O Performance for HPC Platforms." In: *Cray User Group* (2007).

[162] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. "Fault Tolerant Preconditioned Conjugate Gradient for Sparse Linear System Solution." In: *ICS*. ACM, 2012.

[163] D. B. Shmoys, J. Wein, and D. P. Williamson. "Scheduling Parallel Machines On-line." In: *SIAM J. Comput.* 24.6 (1995), pp. 1313–1331.

[164] L. Silva and J. Silva. "Using two-level stable storage for efficient checkpointing." In: *IEE Proceedings - Software* 145.6 (1998), pp. 198–202.

[165] R. F. da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman. "Community resources for enabling research in distributed scientific workflows." In: *e-Science (e-Science), 2014 IEEE 10th International Conference on*. Vol. 1. IEEE. 2014, pp. 177–184.

[166] Simulation Software. *Computing the yield*. `https://zenodo.org/record/2159761#.XA51mhCnfCJ`. 2018.

[167] D. Skinner and W. Kramer. "Understanding the Causes of Performance Variability in HPC Workloads." In: *IEEE Workload Characterization Symposium* (2005), pp. 137–149.

[168] J. Skovira, W. Chan, H. Zhou, and D. A. Lifka. "The EASY - LoadLeveler API Project." In: *JSSPP*. 1996, pp. 41–47.

[169] M. Snir and et al. "Addressing Failures in Exascale Computing." In: *Int. J. High Perform. Comput. Appl.* 28.2 (2014), pp. 129–173.

[170] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. "Characterization of backfilling strategies for parallel job scheduling." In: *International Conference on Parallel Processing Workshop*. 2002.

[171] G. Staples. "TORQUE Resource Manager." In: *Proceedings of the ACM/IEEE Conference on Supercomputing*. Tampa, Florida, 2006.

[172] J. Stearley, K. B. Ferreira, D. J. Robinson, J. Laros, K. T. Pedretti, D. Arnold, P. G. Bridges, and R. Riesen. "Does partial replication pay off?" In: *FTXS*. IEEE, 2012.

[173] O. Subasi, G. Yalcin, F. Zyulkyarov, O. Unsal, and J. Labarta. "Designing and Modelling Selective Replication for Fault-Tolerant HPC Applications." In: *Proc. CCGrid'2017*. May 2017, pp. 452–457. DOI: 10.1109/CCGRID.2017.40.

[174] O. Subasi, J. Arias, O. Unsal, J. Labarta, and A. Cristal. "Programmer-directed Partial Redundancy for Resilient HPC." In: *Computing Frontiers*. ACM, 2015.

[175] R. Sudarsan and C. J. Ribbens. "Design and performance of a scheduling framework for resizable parallel applications." In: *Parallel Computing* 36.1 (2010), pp. 48–64.

[176] R. Sudarsan, C. J. Ribbens, and D. Farkas. "Dynamic Resizing of Parallel Scientific Simulations: A Case Study Using LAMMPS." In: *Int . Conf . Computational Science ICCS*. Procedia, 2009, pp. 175–184.

[177] D. Talia. "Workflow Systems for Science: Concepts and Tools." In: *ISRN Software Engineering* (2013).

[178] K. Tang, P. Huang, X. He, T. Lu, S. S. Vazhkudai, and D. Tiwari. "Toward Managing HPC Burst Buffers Effectively: Draining Strategy to Regulate Bursty I/O Behavior." In: *MASCOTS*. IEEE. 2017, pp. 87–98.

[179] F. Tessier, P. Malakar, V. Vishwanath, E. Jeannot, and F. Isaila. "Topology-aware data aggregation for intensive I/O on large-scale supercomputers." In: *First Workshop on Optimization of Communication in HPC*. IEEE Press. 2016, pp. 73–81.

[180] T. Tobita and H. Kasahara. "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms." In: *Journal of Scheduling* 5.5 (2002), pp. 379–394.

[181] Top500. *Top 500 Supercomputer Sites*. `https://www.top500.org/lists/2018/11/`. Nov. 2019.

[182] H. Topcuoglu, S. Hariri, and M.-y. Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing." In: *IEEE transactions on parallel and distributed systems* 13.3 (2002), pp. 260–274.

[183] S. Toueg and Ö. Babaoglu. "On the Optimum Checkpoint Selection Problem." In: *SIAM J. Comput.* 13.3 (1984), pp. 630–649.

[184] J. Turek, J. L. Wolf, and P. S. Yu. "Approximate Algorithms Scheduling Parallelizable Tasks." In: *SPAA*. San Diego, California, USA, 1992.

[185] N. H. Vaidya. "A Case for Two-level Distributed Recovery Schemes." In: *SIGMETRICS Perform. Eval. Rev.* 23.1 (1995), pp. 64–73.

[186] J. Valdes, R. E. Tarjan, and E. L. Lawler. "The Recognition of Series Parallel Digraphs." In: *Proc. of STOC'79*. ACM, 1979, pp. 1–12.

[187] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. "Proactive process-level live migration in HPC environments." In: *SC '08: Proc.ACM/IEEE Conference on Supercomputing*. ACM Press, 2008.

[188] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan. "Replication-Based Fault-Tolerance for Large-Scale Graph Processing." In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. June 2014, pp. 562–573.

[189] E. Weisstein. *Gauss hypergeometric function. From MathWorld–A Wolfram Web Resource*. `http://functions.wolfram.com/HypergeometricFunctions/Hypergeometric2F1/03/04/02/`.

[190] E. Weisstein. *Incomplete Beta Function. From MathWorld–A Wolfram Web Resource*. `http://mathworld.wolfram.com/IncompleteBetaFunction.html`.

[191] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. "Swift: A language for distributed parallel scripting." In: *Parallel Computing* 37.9 (2011), pp. 633–652.

[192] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher, et al. "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud." In: *Nucleic acids research* (2013), gkt328.

[193] A. K. L. Wong and A. M. Goscinski. "Evaluating the EASY-backfill Job Scheduling of Static Workloads on Clusters." In: *CLUSTER*. 2007.

[194] M. Y. Wu and D. D. Gajski. "Hypertool: a programming aid for message-passing systems." In: *IEEE Trans. Parallel Distributed Systems* 1.3 (1990), pp. 330–343.

[195] P. Wu, C. Ding, L. Chen, F. Gao, T. Davies, C. Karlsson, and Z. Chen. "Fault Tolerant Matrix-matrix Multiplication: Correcting Soft Errors On-line." In: *ScalA'11*. Seattle, Washington, USA, 2011, pp. 25–28.

[196] K. Yamamoto, A. Uno, H. Murai, T. Tsukamoto, F. Shoji, S. Matsui, R. Sekizawa, F. Sueyasu, H. Uchiyama, M. Okamoto, N. Ohgushi, K. Takashina, D. Wakabayashi, Y. Taguchi, and M. Yokokawa. "The K computer Operations: Experiences and Statistics." In: *Procedia Computer Science (ICCS)* 29 (2014), pp. 576–585.

[197] E. Yao, J. Zhang, M. Chen, G. Tan, and N. Sun. "Detection of soft errors in LU decomposition with partial pivoting using algorithm-based fault tolerance." In: *International Journal of High Performance Computing Applications* 29.4 (2015), pp. 422–436.

[198] S. Yi, D. Kondo, B. Kim, G. Park, and Y. Cho. "Using Replication and Checkpointing for Reliable Task Management in Computational Grids." In: *SC'10*. ACM, 2010.

[199] A. B. Yoo, M. A. Jette, and M. Grondona. "SLURM: Simple Linux Utility for Resource Management." In: *JSSPP*. 2003, pp. 44–60.

[200] J. W. Young. "A first order approximation to the optimum checkpoint interval." In: *Comm. of the ACM* 17.9 (1974), pp. 530–531.

[201] J. Yu, D. Jian, Z. Wu, and H. Liu. "Thread-level redundancy fault tolerant CMP based on relaxed input replication." In: *ICCIT*. IEEE, 2011.

[202] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi. "Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform." In: *Proc. 26th IEEE IPDPS*. 2012, pp. 1352–1363.

[203] X. Zhang, K. Davis, and S. Jiang. "Opportunistic data-driven execution of parallel programs for efficient I/O services." In: *IPDPS'12*. IEEE. 2012, pp. 330–341.

[204] G. Zheng, L. Shi, and L. V. Kale. "FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI." In: *Cluster Computing, 2004 IEEE International Conference on*. IEEE Computer Society, 2004, pp. 93–103.

[205] Z. Zheng and Z. Lan. "Reliability-aware scalability models for high performance computing." In: *Cluster Computing*. IEEE, 2009.

[206] Z. Zheng, L. Yu, and Z. Lan. "Reliability-Aware Speedup Models for Parallel Applications with Coordinated Checkpointing/Restart." In: *IEEE Trans. Computers* 64.5 (2015), pp. 1402–1415.

[207] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan. "I/O-Aware Batch Scheduling for Petascale Computing Systems." In: *Cluster15*. Sept. 2015, pp. 254–263. DOI: 10.1109/CLUSTER.2015.45.

[208] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. "IBM Experiments in Soft Fails in Computer Electronics." In: *IBM J. Res. Dev.* 40.1 (1996), pp. 3–18.

[209] J. Ziegler, H. Muhlfeld, C. Montrose, H. Curtis, T. O'Gorman, and J. Ross. "Accelerated testing for cosmic soft-error rate." In: *IBM J. Res. Dev.* 40.1 (1996), pp. 51–72.

[210] J. Ziegler, M. Nelson, J. Shell, R. Peterson, C. Gelderloos, H. Muhlfeld, and C. Montrose. "Cosmic ray soft error rates of 16-Mb DRAM memory chips." In: *IEEE Journal of Solid-State Circuits* 33.2 (1998), pp. 246–252.

Publications

Articles in International Refereed Journals

[J1] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. "Towards Optimal Multi-Level Checkpointing." In: *IEEE Transactions on Computers* (2016).

[J2] G. Aupy, A. Gainaru, and V. Le Fèvre. "I/O Scheduling Strategy for Periodic Applications." In: *ACM Transactions on Parallel Computing* 6.2 (July 2019). ISSN: 2329-4949. DOI: 10.1145/3338510.

[J3] A. Benoit, A. Cavelan, F. M. Ciorba, V. Le Fèvre, and Y. Robert. "Combining Checkpointing and Replication for Reliable Execution of Linear Workflows with Fail-Stop and Silent Errors." In: *International Journal of Networking and Computing* 9.1 (2019), pp. 2–27. ISSN: 2185-2847.

[J4] L. Han, V. Le Fèvre, L.-C. Canon, Y. Robert, and F. Vivien. "A generic approach to scheduling and checkpointing workflows." In: *The International Journal of High Performance Computing Applications* 33.6 (2019), pp. 1255–1274. DOI: 10.1177/1094342019866891. eprint: https://doi.org/10.1177/1094342019866891.

[J5] V. Le Fèvre, T. Herault, Y. Robert, A. Bouteiller, A. Hori, G. Bosilca, and J. Dongarra. "Comparing the performance of rigid, moldable and grid-shaped applications on failure-prone HPC platforms." In: *Parallel Computing* 85 (2019), pp. 1–12. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2019.02.002.

Articles in International Refereed Conferences

[C1] L. Han, V. Le Fèvre, L.-C. Canon, Y. Robert, and F. Vivien. "A Generic Approach to Scheduling and Checkpointing Workflows." In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. Eugene, OR, USA: Association for Computing Machinery, 2018. ISBN: 9781450365109. DOI: 10.1145/3225058.3225145.

[C2] A. Benoit, T. Herault, V. Le Fèvre, and Y. Robert. "Replication is More Efficient than You Think." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356171.

Articles in International Refereed Workshops

[W1] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. "A Different Re-execution Speed Can Help." In: *2016 45th International Conference on Parallel Processing Workshops (ICPPW)*. Aug. 2016, pp. 250–257. DOI: 10.1109/ICPPW.2016.45.

[W2] G. Aupy, A. Gainaru, and V. Le Fèvre. "Periodic I/O Scheduling for Super-Computers." In: *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*. Ed. by S. Jarvis, S. Wright, and S. Hammond. Cham: Springer International Publishing, 2017, pp. 44–66. ISBN: 978-3-319-72971-8.

[W3] A. Benoit, A. Cavelan, V. Le Fèvre, and Y. Robert. "Optimal Checkpointing Period with Replicated Execution on Heterogeneous Platforms." In: *Proceedings of the 2017 Workshop on Fault-Tolerance for HPC at Extreme Scale*. FTXS '17. Washington, DC, USA: Association for Computing Machinery, 2017, pp. 9–16. ISBN: 9781450350013. DOI: 10.1145/3086157.3086165.

[W4] A. Benoit, A. Cavelan, F. M. Ciorba, V. Le Fèvre, and Y. Robert. "Combining Checkpointing and Replication for Reliable Execution of Linear Workflows." In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2018, pp. 793–802. DOI: 10.1109/IPDPSW.2018.00126.

[W5] V. Le Fèvre, L. Bautista-Gomez, O. Unsal, and M. Casas. "Approximating a Multi-Grid Solver." In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. Nov. 2018, pp. 97–107. DOI: 10.1109/PMBS.2018.8641651.

[W6] V. Le Fèvre, G. Bosilca, A. Bouteiller, T. Herault, A. Hori, Y. Robert, and J. Dongarra. "Do Moldable Applications Perform Better on Failure-Prone HPC Platforms?" In: *Euro-Par 2018: Parallel Processing Workshops*. Ed. by G. Mencagli, D. B. Heras, V. Cardellini, E. Casalicchio, E. Jeannot, F. Wolf, A. Salis, C. Schifanella, R. R. Manumachu, L. Ricci, M. Beccuti, L. Antonelli, J. D. Garcia Sanchez, and S. L. Scott. Cham: Springer International Publishing, 2018, pp. 787–799. ISBN: 978-3-030-10549-5.

[W7] A. Benoit, V. Le Fèvre, P. Raghavan, Y. Robert, and H. Sun. "Design and Comparison of Resilient Scheduling Heuristics for Parallel Jobs." In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2020.