Future Generation Computer Systems 161 (2024) 315–328

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A survey on checkpointing strategies: Should we always checkpoint à la Young/Daly?

Leonardo Bautista-Gomez [a], Anne Benoit [b,c,*], Sheng Di [d], Thomas Herault [e], Yves Robert [b,e], Hongyang Sun [f]

[a] Barcelona Supercomputing Center, Spain
[b] Laboratoire LIP, ENS Lyon & Inria, France
[c] Institut Universitaire de France (IUF), France
[d] Argonne National Laboratory, USA
[e] University of Tennessee Knoxville, USA
[f] University of Kansas, USA

ARTICLE INFO

Keywords:
Checkpointing
Optimal period
Young/Daly formula
Resilience

ABSTRACT

The Young/Daly formula provides an approximation of the optimal checkpointing period for a parallel application executing on a supercomputing platform. It was originally designed to handle fail-stop errors for preemptible tightly-coupled applications, but has been extended to other application and resilience frameworks. We provide some background and survey various scenarios to assess the usefulness and limitations of the formula, both for preemptible applications and workflow applications represented as a graph of tasks. We also discuss scenarios with uncertainties, and extend the study to silent errors. We exhibit cases where the optimal period is of a different order than that dictated by the Young/Daly formula, and finally we explain how checkpointing can be further combined with replication.

1. Introduction

Checkpointing is the standard technique to protect applications running on HPC (High-Performance Computing) platforms. Every day, an HPC platform could experience a few fail-stop errors (or failures; we use both terms indifferently). After each failure, the application executing on the faulty processor (and likely on many other processors for a large parallel application) is interrupted and must be restarted. Without checkpointing, all the work executed for the application is lost. With checkpointing, the execution can resume from the last checkpoint, after some downtime (enroll a spare to replace the faulty processor) and a recovery (read the checkpoint).

There are too many HPC applications or even application types that rely on checkpointing to list them all in this survey. However, in order to give a few illustrative examples, we refer the interested reader to [1,2] for an in-depth description of some characteristic computational science workloads from the USA Department of Energy National Laboratories – namely LANL, SNL, LLNL – or academia — in particular the NERSC. These applications cover a large spectrum of domains, spanning from large-scale scientific simulations to data-intensive workflows. These are further divided into large-scale Uncertainty Quantification (UQ) and High Throughput Computing (HTC). Most of them (11 applications over 16, representing 97% of the overall workload of these laboratories) rely on some form of checkpointing, either for fault tolerance and/or for archiving and time-sharing of the platforms. S3D is an example of a complex simulation software that uses periodic checkpointing at scale to implement fault tolerance [3].

In a more industrial setup, checkpoints are used in the context of training deep learning recommendation models by Facebook (see [4]) to tolerate failures but also to improve the prediction accuracy with continuous learning.

Recently, application-level checkpointing has gained significant traction by combining the idea with compression to decrease the checkpoint size while maintaining a high level of accuracy. For instance, [5] goes in details over a set of scientific computational applications that rely on such a method.

Most High-Performance Computing applications rely on libraries like SCR [6], FTI [7], or VeloC [8] to implement their application-level checkpointing. In [9], three scientific applications from the Exascale Computing Project (HACC [10], LatticeQCD [11], EXAALT [12]) are identified as relying on VeloC to implement their checkpointing capabilities.

* Corresponding author at: Laboratoire LIP, ENS Lyon & Inria, France.
E-mail address: anne.benoit@ens-lyon.fr (A. Benoit).

https://doi.org/10.1016/j.future.2024.07.022
Received 18 December 2023; Received in revised form 5 July 2024; Accepted 13 July 2024
Available online 18 July 2024

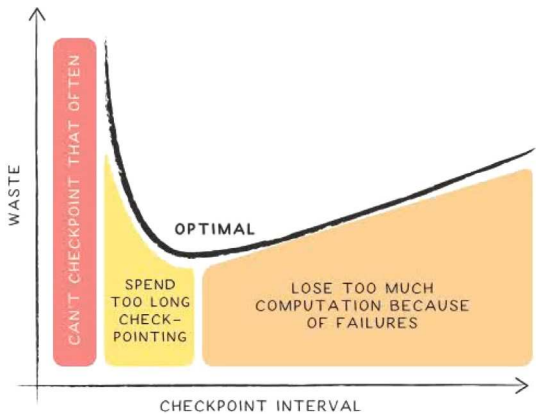

Fig. 1. Trade-off for the optimal checkpointing period.

There are many varieties of checkpointing techniques and protocols. But at a fundamental level, they behave similarly when dealing with failures, and they can be abstracted by the same model. Consider a parallel application executing on an HPC platform whose nodes are subject to fail-stop errors. The fundamental question is how frequently it should be checkpointed so that its expected execution time is minimized. There is a well-known trade-off (see Fig. 1): taking too many checkpoints leads to a high overhead, especially when there are few failures, while taking too few checkpoints leads to a large re-execution time after each failure. The optimal checkpointing period is (approximately) given by the Young/Daly formula as $W_{YD} = \sqrt{2\mu C}$ [13,14], where μ is the application MTBF (Mean Time Between Failures) and C is the checkpoint duration (or cost).

This paper provides a survey of the applicability and robustness of the Young/Daly formula for different application scenarios, and discusses several checkpointing strategies that go beyond a straightforward use of the formula. There are two main frameworks that are considered in this survey. First, we deal with preemptible applications, which may be checkpointed at any time. In this context, checkpointing is a coordinated process and involves all the processors enrolled in the execution of the application. Then, we focus on task systems, where applications are composed of a set of atomic tasks, possibly with interdependencies. In this context, checkpoints are task-based and can be taken only at the end of a task. Only the processors that execute a task are involved in its checkpoint. The problem is then to decide which tasks to checkpoint.

This paper builds on a preliminary and much shorter version [15]. We are covering several new topics such as multi-level checkpointing, checkpointing preemptible applications in practice, checkpoints that take variable times, silent error detectors, imperfect verifications, cases where the order of magnitude of the optimal checkpointing period changes, and the combination of checkpointing with replication.

The paper is organized as follows. We first survey preemptible applications in Section 2. Then, we deal with task systems in Section 3. We address questions related to uncertainty in Section 4. Section 5 is devoted to silent errors. Section 6 discusses extensions of the Young/Daly formula, and Section 7 discusses how to combine checkpointing with replication, in particular when checkpointing alone is not sufficient. Finally, we conclude with final remarks and some open questions in Section 8.

2. Preemptible applications

In this section, we deal with parallel applications that can be checkpointed at any time. In scheduling terminology, the applications are preemptible.

2.1. Background

Platform and applications. Consider a large parallel platform with m identical nodes (or processors; we use both terms indifferently). These nodes are subject to fail-stop errors, or failures. A failure interrupts the execution of the application on this node and provokes the loss of the data located in its memory.

Consider a parallel application running on $p \leq m$ nodes: when one of these nodes is struck by a failure, the state of the application is lost, and execution must restart from scratch unless a fault-tolerance mechanism has been deployed. The classical technique to deal with failures makes use of a checkpoint-restart mechanism: the state of the application is periodically checkpointed, i.e., all participating nodes take a checkpoint simultaneously. This is the standard coordinated checkpointing protocol, which is routinely used on large-scale platforms [16], where each node writes its share of application data to non-volatile (a.k.a. stable) storage, leading to a checkpoint of duration C. When a failure occurs, the platform is unavailable during a downtime D, which is the time to enroll a spare processor that will replace the faulty processor [14,17]. Then, all application nodes (including the spare) recover from the last valid checkpoint in a coordinated manner, reading the checkpoint file from non-volatile storage (recovery of duration R). Finally, the execution is resumed from that point on, rather than starting again from scratch. Note that failures can strike during checkpoint and recovery, but not during downtime (otherwise, there are no differences between downtime and recovery and we can simply include the downtime in the recovery time). When a failure hits a processor, that processor is replaced by a spare. This amounts to starting anew with a fresh processor. In the terminology of stochastic processes, the faulty processor is rejuvenated. However, all the other processors are not rejuvenated: this would be infeasible due to the multitudinous spares that would be needed.

Failures. We assume that each node experiences failures, whose inter-arrival times follow Independent and Identically Distributed (IID) random variables obeying an arbitrary probability distribution $\mathcal{D}$. We only assume that $\mathcal{D}$ is continuous and of finite expectation and variance, a condition satisfied by all standard distributions. We let μ_{ind} denote the expectation of $\mathcal{D}$, also known as the individual processor MTBF. Even if each node has an MTBF of several years, large-scale parallel platforms are composed of so many nodes that they will experience several failures per day [18,19]. Hence, a parallel application using a significant fraction of the platform will typically experience a failure every few hours. More precisely, an application executing with p processors has an MTBF $\mu = \frac{\mu_{ind}}{p}$: intuitively, the application is struck by failures at a rate that is p times higher than that of each enrolled processor. We come back to this statement in Section 2.3.

Checkpointing strategies. Given a parallel application whose length is T_{base} (base time without checkpoints nor failures), the optimization problem is to decide when and how often to take a checkpoint to minimize the expected execution time of the application. The application is divided into N_c segments of length W_i, $1 \leq i \leq N_c$, each followed by a checkpoint of length C. Of course, $\sum_{i=1}^{N_c} W_i = T_{base}$. We add a final checkpoint at the end of the last segment, e.g., to write final outputs to non-volatile storage. Symmetrically, we add an initial recovery when re-executing the first segment of an application (e.g., to read inputs from non-volatile storage) if it has been struck by a failure before completing the first checkpoint. Adding a final checkpoint and an initial recovery brings symmetry and simplifies formulas, but it is not at all mandatory: see [20] for an extension relaxing either or both assumptions. The question is then to determine the number N_c of segments and their lengths W_i.

2.2. The Young/Daly formula

We start with an intuitive (but simplified) derivation of the Young/Daly formula for the optimal checkpointing period. Owing to the addition of the final checkpoint and the initial recovery, all segments of the application have the same shape. It is thus natural (by symmetry) to assume that they have the same length W in the optimal solution. Thus, we assume that checkpoints are taken periodically, after every W unit of work. We define the waste as the fraction of time during which the application is not performing useful computations; checkpoint, recovery, downtime, and re-execution do not count as useful computations. Now, after every W unit of work, we spend C seconds to checkpoint, which corresponds to a first source of waste $S_1 = \frac{C}{W+C}$. S_1 is the *failure-free* waste. The second source of waste S_2 is due to failures: each time a failure strikes, which happens every μ seconds on average, we lose $D + R$ for downtime and recovery, and then we have to re-execute some work, namely the work performed since the last checkpoint (or from the beginning of the execution if none has been taken yet). On average again, the failure strikes in the middle of the segment: sometimes before, sometimes after, hence, on average after $\frac{W+C}{2}$ seconds. We obtain $S_2 = \frac{1}{\mu}(D+R+\frac{W+C}{2})$. S_2 is the *failure-induced* waste. Altogether, both sources of waste approximately add up, so we have to find W that minimizes S_1+S_2. We further simplify the solution by assuming that W must be an order of magnitude higher than the fault-tolerance parameters D, C, R. This is a necessary condition for the waste to remain reasonably low. This leads to $S_1 \approx \frac{C}{W}$ and $S_2 \approx \frac{W}{2\mu}$. The total waste $S_1 + S_2 \approx \frac{C}{W} + \frac{W}{2\mu}$ is minimum for

$$W_{YD} = \sqrt{2\mu C}. \tag{1}$$

This is nothing else than the famous Young/Daly formula! Finally, note that $S_1 = S_2$ for W_{YD}, which corroborates the intuition given in Fig. 1 that both sources of waste, failure-free and failure-induced, should be balanced in the optimal solution. See [17] for a more detailed derivation using the waste argument.

2.3. Accuracy of the derivation

Recall that each node experiences failures whose inter-arrival times follow IID random variables obeying a probability distribution $\mathcal{D}$. When $\mathcal{D}$ is $\text{Exp}(\lambda)$, i.e., an Exponential distribution of rate λ, the framework is well-understood. This is because the inter-arrival times of the failures that strike an application with p processors are IID random variables obeying an Exponential distribution $\text{Exp}(p\lambda)$. This is due to the memoryless property of the Exponential distribution: when a failure strikes one processor, that processor is rejuvenated, while the remaining $p-1$ processors are not. With an arbitrary distribution $\mathcal{D}$, the time to the next failure would depend upon the history of these $p-1$ processors: for each of them, the time to their next failure depends upon when their last failure struck. This is not the case for an Exponential distribution, owing to its memoryless property: after a failure on any of the p processors, the time to the next failure remains the same random variable $\text{Exp}(\lambda)$ for each of them, rejuvenated or not. Therefore, the time to the next failure for the application obeys an $\text{Exp}(p\lambda)$ distribution, as the minimum of p $\text{Exp}(\lambda)$ distributions. From the resilience point of view, the application executes on a single processor of fault rate $p\lambda$. Owing to this observation, one can formally derive that the optimal checkpointing strategy is periodic, and compute the optimal checkpointing period. The derivation is a bit technical and the optimal segment length W_{opt} is obtained using the Lambert W function. But comfortingly, a first-order approximation of W_{opt} is W_{YD}, the value given by the Young/Daly formula. See [20,21] for details on the derivation.

Now, any continuous distribution D other than Exponential is not memoryless, and the optimal checkpointing strategy is unknown in that case. The bad news is that the most accurate probability distributions modeling processor failures are LogNormal [22] and Weibull [23–26] instead of Exponential. For instance, LANL failure traces are best fit by Weibull distributions of different shapes [27]. Weibull distributions with a shape parameter smaller than one experience infant mortality: their instantaneous failure rate decreases with time, so that failures are more frequent at the beginning of the execution than at its end. For those distributions, it is known that periodic checkpointing is not optimal. Intuitively, the length of a segment between two consecutive checkpoints should increase with time, since the instantaneous failure rate decreases. However, the good news is that the MTBF can still be defined as the limit:

$$\lim_{T\to\infty} \frac{n(T)}{T} = \frac{\mu_{ind}}{p},$$

where $n(T)$ is the expected number of failures striking an application with p processors in the time interval $[0, T]$. This limit exists for any regular distribution $\mathcal{D}$. A natural heuristic is to use a periodic checkpointing strategy, with a segment length given by the Young/Daly formula and using that latter value for the MTBF. It is unknown how this approach is close to the optimal but it seems good enough in many scenarios. See [20,21] for an assessment of this heuristic, and for a comparison with other checkpointing strategies that aim at maximizing work or efficiency until the next failure.

2.4. Extensions

We now discuss extensions of the Young/Daly formula in several frameworks.

2.4.1. Overlapping checkpointing and computation

In Section 2.2, we have shown how to derive the optimal checkpointing period when the objective is to minimize the expected completion time of the application. We used a simplified model where no computation could take place while checkpointing. Modern processors could run several threads in parallel and compute while executing I/O transfers. A first extension to the framework of Section 2.2 is to extend the model with a linear slowdown factor $\alpha \in [0, 1]$, where, say, $\alpha = 0.5$ means that computations progress at half the main speed when checkpointing. The two extreme values are $\alpha = 0$ when checkpoints are blocking (no overlap), and $\alpha = 1$ when execution can progress with no penalty while a checkpoint is taken (full overlap). The Young/Daly formula becomes $W_{YD} = \sqrt{2\mu(1-\alpha)C}$. Note that $\alpha = 0$ leads to the original Young/Daly formula, while $\alpha = 1$ leads to $W_{YD} = 0$, which means that one should checkpoint all the time if checkpointing is free. Of course, in practical scenarios, we expect $\alpha < 1$. See [17] for more details.

2.4.2. Checkpointing to minimize energy consumption

Another extension to the framework of Section 2.2 is to target a different optimization objective: instead of minimizing the (expected) total execution time, one would aim at minimizing the (expected) total energy consumed to execute the application. This objective is important both for economic and environmental reasons. See [27–29] and the references therein for further details. The optimal period W_{energy} to minimize energy consumption is different from the Young/Daly formula mainly because the power spent when computing is not the same as the power spent when checkpointing. More precisely, the power consumption at each time step of the application relies on three components:

- $\mathcal{P}_{\text{Static}}$: base power consumed when the platform is switched on.
- $\mathcal{P}_{\text{Cal}}$: when the platform is computing, we have to consider the CPU overhead in addition to the static power $\mathcal{P}_{\text{Static}}$.
- $\mathcal{P}_{\text{I/O}}$: similarly, this is the power overhead due to file I/O. This supplementary power consumption is induced by checkpointing, or when recovering from a failure.

A key parameter to compare W_{energy} and W_{YD} is the ratio $\frac{\mathcal{P}_{\text{Static}}+\mathcal{P}_{\text{I/O}}}{\mathcal{P}_{\text{Static}}+\mathcal{P}_{\text{Cal}}}$. Unfortunately, there is no compact expression for the optimal period W_{energy}, which is obtained as the root of a second-degree equation [28].

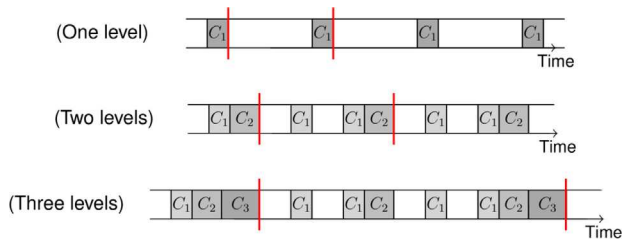

Fig. 2. Different levels of checkpointing. The vertical red lines mark the beginning and end of a periodic pattern. The higher the error level, the less frequent the checkpoints at that level (and the darker their shade of gray).

2.4.3. Multi-level checkpointing

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 (e.g., [6–8]) 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 failure type, and is associated to a storage device that is resilient to that type. For instance, a two-level checkpointing 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 non-volatile storage (remote redundant disks).

The main 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. Fig. 2 illustrates different levels of checkpointing protocols, from a single level to three levels. Another extension of the Young/Daly formula is to derive the optimal period and pattern in the presence of multi-level checkpoints.

The optimal two-level checkpointing intervals can be derived theoretically. In this case, several level-1 checkpoints (with cost C_1) could be taken before taking a level-2 checkpoint (with cost C_2, which is larger than C_1). If a level-1 failure occurs, we just need to recover from the latest level-1 checkpoint, instead of from the last level-2 checkpoint, which is more costly. The optimal period for the outer-level (i.e., level-2) checkpoints can be approximated as:

$$W = \sqrt{\frac{2(nC_1 + C_2)}{\frac{1}{n\mu_1} + \frac{1}{\mu_2}}}, \tag{2}$$

where μ_1 and μ_2 denote the MTBFs of the level-1 and level-2 failures, respectively [30]. Here, n denotes the optimal number of level-1 checkpoints between two consecutive level-2 checkpoints, and its value is also related to the failure MTBFs and the checkpoint costs of the two levels. Another optimal two-level checkpointing solution was proposed in [31], which offers two novel insights: (1) it proves that periodic patterns are optimal and derives the exact best pattern (instead of an approximate period); (2) it evaluates the overall wall-clock times based on the derived optimal checkpointing intervals for nine cases, each with different checkpoint/restart overheads and failure rates.

Identifying the optimal checkpointing intervals for the situation with more than two levels of checkpoints has also been studied. First, the formula in Eq. (2) for two-level checkpointing can be extended to an arbitrary number of levels, where both the checkpoint cost and the MTBF typically increase with the levels. Benoit et al. [30] derived a general approximate formula for this case. Furthermore, Di et al. [32] proposed a generic mathematical formulation for the problem with various types of failures, and developed an iterative method to calculate the optimal checkpointing intervals for different levels efficiently. They further extended their iterative method to the situation with uncertain execution scales [33]. Specifically, an in-depth analysis is provided on why it is non-trivial to derive the optimal checkpointing intervals for different checkpointing levels and optimize the number of cores simultaneously. Then, a fixed-point iterative method that can quickly obtain an optimized solution is proposed — the first successful attempt in multi-level checkpointing models with uncertain scales.

2.4.4. Minimizing I/O due to checkpointing

Finally, another optimization objective is to minimize the expected volume of I/O operations due to checkpointing and recovery. This objective is important because I/O resources are scarce in HPC platforms. Typical HPC applications execute on dedicated computing nodes but share the I/O bandwidth of the platform with other applications. Hence, decreasing the volume of I/O operations by each application will likely improve the global throughput of the platform. A natural question is then: given a single application that executes on the platform, can we increase the checkpointing period significantly beyond the Young/Daly formula without sacrificing too much in performance? Note that we have a bi-criteria optimization problem here because we need to trade off performance with I/O pressure. Note also that a single application running on the platform may be a *capability* workload that spans the entire platform. The answer to the question is yes: Arunagiri et al. [34] studied longer, sub-optimal periods for a single application, with the intent of reducing I/O pressure. They showed, both analytically and empirically using four real platforms, that a decrease in the I/O requirements can be achieved with only a small increase in waste.

However, *space-sharing* HPC platforms for the concurrent execution of multiple parallel applications is the prevalent usage strategy in today's HPC centers, and *capability* workloads that span the entire platform are much less common [35]. The question becomes how to avoid contention when several applications try to checkpoint at the same time: the I/O bandwidth will be shared among these applications, their checkpoint time will increase, and the Young/Daly formula that was computed for each application in isolation is no longer optimal due to these interferences. We will come back to this question in Section 4.2.

2.5. Loosely-coupled applications

The Young/Daly formula applies to a parallel application where all processors progress and cooperate continuously, e.g., by exchanging messages: the application cannot continue its execution when one processor is struck by a failure; it has to wait until a spare is up and running. In other words, the application is assumed to be tightly coupled and behaves as if it were executed on a single (very powerful) processor.

What if the application is not tightly-coupled? If the application includes several tasks that can execute concurrently and independently on different subsets of resources, how frequently should each task be checkpointed? We use the word *task* here, but not in the traditional meaning where tasks are atomic and can only be checkpointed at the end of their execution (see Section 3 for such a framework). On the contrary, we assume that each task is preemptible and can be checkpointed at any time step. It is then natural to checkpoint each task using the Young/Daly period. But is this a good strategy, given that many tasks execute in parallel, and that the failure of one task will slow down the whole application?

Consider the simple example of a fork-join application that consists of 302 tasks: an entry task, 300 identical parallel tasks, and an exit task. Each parallel task runs on $p = 30$ processors for $T_{base} = 10$ hours, and is checkpointed in $C = 6$ minutes. The platform has at least 9,000 processors so that the 300 parallel tasks can indeed execute concurrently. Such applications are typical of HPC applications that

explore a wide range of parameters or launch subproblems in parallel. Assume a short downtime $D = 1$ minute, and recovery time $R = C$. Finally, assume that each task has 0.5% chances to fail during execution; this setting corresponds to an individual MTBF μ_{ind} such that $1 - e^{-\frac{pT_{base}}{\mu_{ind}}} = 0.005$, i.e., $\mu_{ind} = 59{,}850$ hours (or 6.8 years). This is in accordance with MTBFs typically observed on large-scale platforms, which range from a few years to a few dozens of years [19].

In the following paragraphs, we refer to [36] for the details of computing the expectations of execution times. Indeed, the details are complicated, and we need the reader to trust us for all expected values below. For each task, the Young/Daly period is $W_{YD} = \sqrt{2\frac{\mu_{ind}}{p}C} \approx 20$ hours, and the expected execution time of a single task $\mathbb{E}(T_{1\text{-}task})$ is minimized when only a single checkpoint is taken at the end of the execution. Recall that we always take a checkpoint at the end of the execution for simplification, thus the optimal solution for each task is to take no additional checkpoint. Then, one can derive that $\mathbb{E}(T_{1\text{-}task}) \approx 10.4$ hours.

However, with 300 tasks executing concurrently, one can compute that the expectation of the total time required to complete all tasks is $\mathbb{E}(T_{all\text{-}tasks}) > 14$ hours. The key point here is that the expectation $\mathbb{E}(T_{all\text{-}tasks})$ of the total time required to complete all tasks is far larger than the maximum of the expectations (which in the example all have the same value $\mathbb{E}(T_{1\text{-}task})$). The intuition is the following: if each parallel task is expected to be struck by, say, 3 failures, then most tasks will experience between 0 and 6 failures, but some unlucky task may well experience 20 failures, and the total time is dictated by the slowest task. In other words, the expectation $\mathbb{E}(T_{all\text{-}tasks})$ of the maximum time over all tasks is likely to be much larger than the maximum of the expected time $\mathbb{E}(T_{1\text{-}task})$ for each time; since all tasks are identical, the latter maximum is also $\mathbb{E}(T_{1\text{-}task})$.

Because the exit task cannot start before the last parallel task is completed, the expectation of the total execution time of the fork-join application is $\mathbb{E}(T_{total}) = \mathbb{E}(T_{entry}) + \mathbb{E}(T_{all\text{-}tasks}) + \mathbb{E}(T_{exit})$, where $\mathbb{E}(T_{entry})$ and $\mathbb{E}(T_{exit})$ are the expected duration of the entry and exit tasks. Now, when adding four intermediate checkpoints to each task, we obtain $\mathbb{E}(T_{all\text{-}tasks}) < 12.75$ hours. The tasks are then slightly longer (10.5 h without failure), but the impact of a failure is dramatically reduced if a checkpoint is taken every two hours. By diminishing $\mathbb{E}(T_{all\text{-}tasks})$, we save 75 minutes (and in fact much more than that because the lower and upper bounds for $\mathbb{E}(T_{all\text{-}tasks})$ are loosely computed).

This little example shows that for loosely-coupled applications with a high degree of parallelism, checkpointing each task à la Young/Daly is not good enough. The key reason is that the expectation of the maximum number of failures across parallel tasks is much higher than the maximum of the expectations of the number of failures for each task. In our example with identical tasks, the intuition is even simpler: the expected number of failures is the same for each task taken independently, but it is very likely that some tasks will experience many more failures if many tasks execute in parallel. See [36] for a comprehensive analysis and evaluation.

2.6. Coordinated checkpointing and rollback recovery for preemptible applications in practice

Achieving exact preemptibility requires the ability to checkpoint at any point in the execution when the protocol demands it. This necessitates the application and all libraries in the software stack to be capable of checkpointing at any time and restarting from that checkpoint. In practice, parallel applications often depend on external libraries to implement coordinated checkpointing and rollback recovery at a high level. The MPI-Agnostic Network-Agnostic Transparent Checkpoint (MANA for MPI, [37]) stands out as a state-of-the-art library that provides this capability for parallel applications relying on the Message Passing Interface (MPI) for communication.

MANA is a Proxy MPI library. It introduces a split process approach, dividing the upper-half of the process (MPI application and associated libraries) from the lower-half (MPI Proxy library and MPI Native library, implementing the communication system). When a checkpoint is required, MANA utilizes DMTCP (Distributed MultiThreaded CheckPointing, [38]), a process-checkpointing library for Linux to save the current state of the upper-half and a synthetic representation of the MPI library's state. During a restart, DMTCP restores the upper-half of all processes, and the Proxy MPI library uses the Native MPI library and the synthetic representation to restore all MPI objects to their state at the time of checkpoint. MANA maintains a translation table between Proxy MPI objects and Native MPI objects for portability, ensuring valid references to MPI objects even after a restart. In the upper-half of the process, only references to the Proxy MPI objects can be saved (and restored). When a restart occurs, MANA updates its translation table to map those object references to the new Native MPI objects that are re-created using the Native MPI library.

However, a significant class of parallel applications opts for checkpoint-restart via application-level checkpointing. In this scenario, applications utilize well-distributed libraries to implement multi-level checkpointing, simplifying the serialization operation for their checkpoints. Popular libraries for this purpose include FTI [7], SCR [6], and VeloC [8], which leverage all memory hierarchy resources (local and remote memory, local and remote storage) to introduce as much asynchrony in the I/O system as possible.

Some applications targeting high performance may employ a diskless checkpointing approach, where the state is solely serialized in the memory of other processes within the same application [39–41]. However, this approach requires surviving processes to continue execution after a failure, necessitating the use of a fault-tolerant version of the MPI library implementing the User-Level Failure Mitigation (ULFM) extension to the MPI Standard [42], available in both Open MPI [43] and MPICH [44]. Such diskless checkpointing capability has been implemented over ULFM, for example in the Fault-Tolerant Programming Framework Fenix [45].

These approaches, albeit effective, lack precise preemptibility. Application-level checkpointing requires programmers to modify the application to serialize the process state at specific points, considering the application's specifics to ensure data consistency between processes. To serialize its state, an application needs to save the segments of memory (belonging to the heap and/or stack, depending on the case) that hold data needed to continue the computation, and the progress position in the execution. Typically, an application will need to save its loop counters, and any non-temporary memory that was modified since the beginning of the execution. Serializing such a state is usually easier to do at the high levels of the application call stack (when only the state of progress of a few functions is required) than deep within the computation (when the state of third party libraries might be involved). Consequently, these applications can only approximate the optimal checkpointing frequency by taking a checkpoint as close as possible to the target checkpoint time. The deviation from the theoretical framework depends on the frequency at which the application reaches a serializable state. Let p_c be the (average) period between two serializable states of a parallel application, and let $W_{YD} = \sqrt{2\mu C}$ be the optimal checkpointing period according to the Young/Daly formula. While the application might not achieve a checkpoint every W_{YD} seconds, assuming $p_c \leq W_{YD}$, it will be able to checkpoint somewhere in the interval $[W_{YD} - p_c, W_{YD} + p_c]$. If the actual checkpointing period is t, the relative efficiency, denoted as $\mathcal{R}$, is given by the formula:

$$\mathcal{R} = \frac{(S_1 + S_2)_{W=W_{YD}}}{(S_1 + S_2)_{W=t}} = \frac{2C\left(\sqrt{2}\mu + \sqrt{\mu C}\right)(t + C)}{\sqrt{\mu C}\left(3C^2 + (2\mu + 4t)C + t^2\right)}.$$

The worst efficiency (minimum value for $\mathcal{R}$) is obtained for $t_{worst} = W_{YD} - p_c$. For instance, in a system where $\mu = 8$ h, $C = 20$ min, and $p_c = 20$ min, $W_{YD} \approx 2$ h, and $t_{worst} = 1$ h 40 min leads to a relative efficiency of 99%. Thus, utilizing non-preemptible application-level checkpointing with the Young/Daly heuristic and an opportunity to checkpoint every 20 min, the efficiency remains very close to the theoretical optimum achievable via preemptible checkpointing.

3. Task graphs

In this section, we deal with non-preemptible, task-based applications. The application is structured as a Directed Acyclic Graph (DAG) of tasks (also called workflow). Each task is atomic and checkpointing is only possible right after the completion of a task. The task graph summarizes the dependencies between the tasks. The problem is then to determine which tasks should be checkpointed. It turns out that optimal, or even efficient, checkpointing strategies are much more difficult to derive than for preemptible applications.

3.1. Baseline

In task-based systems, checkpoint and rollback-recovery has been considered, but the granularity of the task system has motivated a different approach. Since each task represents an atomic application in itself, the inputs of tasks (that are usually the outputs of other tasks) are checkpointed to enable the re-execution of failed tasks.

The de-facto standard approach for workflow-based task systems is the *checkpoint every task* approach. This approach is inspired by the work done in cloud workflow systems, as is typically done in [46] for a recent example. See [47–51] for a comprehensive survey of techniques. The outputs of all tasks, which will serve as inputs to other tasks later in the execution, are saved on non-volatile storage as soon as each task completes. The non-volatile storage is typically located in a data center whose disks are accessed by the virtual machines (VMs) that support the execution of the tasks. This approach guarantees that recovering from a failure only requires the re-execution of the task(s) that were executing when the failure stroke; no rollback to previous tasks is needed since their outputs have been checkpointed previously and can be retrieved from the disks.

Of course, checkpointing (the output of) every task may induce a huge overhead, in particular when there are many small tasks and limited I/O bandwidth to non-volatile storage. In micro-task systems [52–55], the duration of a task is typically a few µs to a few hundred of ms, and there are millions to billions of tasks [56,57]. These systems make tremendous efforts to avoid creating unnecessary copies of the data, as high efficiency is only achieved by reusing data already loaded on the processors. In this context, checkpointing every input data of every task is redhibitory. The approach then consists in detecting, at runtime, which parts of the sub-DAG of tasks need to be reinstantiated, in order to restart the execution from the inputs that have been checkpointed [58,59].

In [58], the heuristic to decide to checkpoint an input data is parametric: if a data is new (has never been checkpointed), or has been updated k times locally, a new checkpoint is created. As the algorithms studied use the owner-compute strategy, this approach leads to a drastic reduction of the number of checkpoints, but it is neither optimal nor applicable to arbitrary DAGs of tasks.

In [59], the DAG of tasks is built sequentially: tasks are discovered one after another, and the runtime system builds the DAG based on how each task accesses which data. Checkpoints are introduced in this sequence of discovery, and the runtime system then computes which data needs to be checkpointed in order to create a restartable cut in the DAG of tasks. To cite [59]: “adding checkpoints to this programming model consists in introducing [explicit] checkpoint() calls within the program. They effectively cut the task graph inferred by the [...] model, between the tasks inserted before the call, and the tasks inserted after the call. [...] since this is done identically on every node, all nodes agree on exactly what will be saved in the checkpoints, without any need for synchronization at run time." The checkpointing algorithm leverages the knowledge of redundant data due to the caching approach of the runtime system, and this is used to reduce the size of the checkpoint. However, placing the checkpoints at optimal times remains an open problem.

We outline below a few cases where the optimal solution is known, before coming back to the general case of a workflow whose task graph is arbitrary.

3.2. Linear chains

The simplest case is when the task graph of the workflow is a linear chain of (parallel) tasks $T_1, T_2, \dots, T_n$. There is a dependence from T_i to T_{i+1} for $1 \leq i \leq n-1$. The optimal solution consists in determining which tasks should be checkpointed.

The execution time of T_i is w_i, its size is q_i processors, its checkpoint time is C_i, and its recovery time is R_i. Assuming that failures obey an Exponential distribution $\text{Exp}(\frac{1}{\mu_{ind}})$, where μ_{ind} is the MTBF of each individual processor, the expected execution time $\mathbb{E}(T_i)$ to execute T_i and to checkpoint it at the end of the execution is well-known; we have:

$$\mathbb{E}(T_i) = \left(\frac{q_i}{\mu_{ind}} + D\right) e^{\frac{q_i}{\mu_{ind}} R_i} \left(e^{-\frac{q_i}{\mu_{ind}}(w_i+C_i)} - 1\right),$$

where D is the downtime (see [17,20]). The expression for $\mathbb{E}(T_i)$ can be extended for a block of consecutive tasks followed by a checkpoint (simply replace w_i by the execution time of the block). This gives the baseline for a dynamic programming algorithm where one tries to place the first checkpoint at the end of task T_k for $1 \leq k \leq n$ and computes recursively the optimal solution for the remaining sub-chain $T_{k+1}, T_{k+2}, \dots, T_n$. This is the approach followed by Toueg and Babaoglu [60].

3.3. Iterative applications

The next problem after a linear chain is that of a *pipelined linear workflow*: we now consider a workflow made of a large number of iterations, each iteration being the same linear chain of parallel tasks. A typical example is an application consisting of an outer loop “`While convergence is not met, do`”, and where the loop body includes a sequence of large parallel operations. As in Section 3.2, the objective is to find which task outputs should be saved on non-volatile storage to minimize the expected duration of the whole computation. However, if the workflow consists of, say, ten thousand iterations, each with twenty tasks, one does not want to apply the dynamic programming algorithm of Toueg and Babaoglu [60] to a chain of two hundred thousand tasks.

A natural heuristic is to use the Young/Daly formula and checkpoint at the end of the current task as soon as the total work executed since the last checkpoint exceeds the quantity $\sqrt{2\mu C}$. Unfortunately, even if all tasks may well enroll the same number of processors q and, hence, have the same MTBF $\mu = \frac{\mu_{ind}}{q}$, they are not likely to have the same checkpoint duration C. One can approximate C by the minimum, maximum, or average values of the checkpoint duration of all tasks. This is the heuristic proposed in [61], and its performance is shown satisfactory for a wide range of application scenarios.

As a side note, when the number of iterations is infinite (or very large in practice), it is shown in [61] that there exists an optimal checkpointing strategy that is periodic. It consists of a pattern of task outputs to checkpoint, where this pattern spans over a set of iterations of bounded size. This pattern is repeated over and over throughout the execution: after some initialization phase, the same set of tasks (which we call the pattern) is checkpointed again and again. [61] also provides a dynamic programming algorithm, which is polynomial in the number of operations included in the outer loop to compute the optimal periodic checkpoint pattern. The complexity of the algorithm does not depend on the number of iterations of the outer loop. This pattern may well checkpoint many different tasks, across many different iterations. For a workflow with a fixed number of iterations, this periodic strategy is appealing because the checkpointing strategy can be described concisely and independently on how many times the outer loop is executed. However, the cost of computing the optimal pattern may be high, and the Young/Daly extension described above may be preferred in some frameworks.

3.4. General workflows

Another special case is that of a workflow whose dependence graph is arbitrary but whose tasks are parallel tasks that each executes on the whole platform. In other words, the tasks have to be serialized. The problem of ordering the tasks and placing checkpoints is proven NP-complete for simple join graphs in [62], which also introduces several heuristics.

For general workflows, the news is not good either. Consider the problem of scheduling an arbitrary workflow. As mentioned in Section 3.1, the common strategy used in practice is *checkpoint everything*, or CkptAll: all output data of each task is saved onto non-volatile storage. 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 have been proposed to reduce I/O overhead [63]. 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 an efficient checkpointing strategy is to achieve a desirable trade-off between these two extremes. But the complexity of this problem is steep.

The fundamental difficulty lies in the evaluation of a solution. A solution consists of an ordered list of tasks to execute for each processor, and for each task whether or not to save its output data to non-volatile storage after its execution. In a failure-free execution, the total execution time, or makespan, of a solution is simply the longest path in the DAG, accounting for serialized task executions at each processor. With failures, the makespan becomes a random variable because task execution times are probabilistic, due to failures causing task re-executions. Consider a first simple case with the CkptAll strategy and a solution in which each task is assigned to a different processor. Computing the expected makespan amounts to computing the expected length of the longest path in the schedule. Unfortunately, computing the expected length of the longest path in a DAG with probabilistic task durations is a known difficult problem [64,65]. Even in the simplified case when task durations are random variables that can take only two discrete values, the problem is #P-complete [64].[1]

Now, at the other extreme, consider a second simple example with the CkptNone strategy and a solution in which each task is assigned to a different processor. Even if each task has the unitary cost and can fail only once, computing the expected makespan is a #P-complete problem again [69]. These two examples show all the difficulty of the problem, even when an ordered list of tasks to execute is already assigned to each processor. Several heuristics to tackle the general problem are proposed in [70].

4. Dealing with uncertainty

This section briefly addresses two scenarios where it is impossible to apply the Young/Daly formula directly, even though the target application is preemptible and tightly coupled as in Section 2. Basically, in the $W_{YD} = \sqrt{2\mu C}$ formula, this is when either μ or C is unknown.

4.1. Unknown MTBF

When the MBTF μ_{ind} of an individual processor is unknown, the MTBF $\mu = \frac{\mu_{ind}}{p}$ of the application is unknown as well. There is no other solution than to learn the value of μ by trial and error. The initial guess for μ is arbitrary, say from a few hours to several weeks depending upon the size of the application. Compute W_{YD} accordingly and schedule the first checkpoint. If a failure strikes before this checkpoint, decrease the current estimate of μ. If no failure strikes before this checkpoint, keep the current value for μ and proceed for a few periods of the same length. if there is still no failure at this point, it should be safe to increase the estimate of μ. The rates for decreasing/increasing the current estimate could follow some geometric progression, e.g., the next estimate is either half or twice the current one.

An interesting heuristic is proposed in [71]. The checkpointing period is dynamically adjusted so that the aggregate checkpointing cost always equals the expected rework cost after failure recovery. The intuition follows the discussion in Section 2.2: in the optimal solution, both sources of waste (checkpoint and re-execution) should be balanced.

4.2. Unknown checkpoint time (due to contention)

This section deals with the scenario where the checkpoint cost C is unknown. In fact, this corresponds to a scenario where several applications are executing concurrently on the platform (recall *space-sharing* from Section 2.4). Each application has precise knowledge of the volume of data to be saved, but the I/O bandwidth to non-volatile storage that is granted is subject to variations over time. The main reason is contention: consider the simple case where two applications of the same size (number of processors) checkpoint simultaneously a file of the same size (volume). Each application will be assigned half the I/O bandwidth to checkpoint, therefore the commits will take twice as long as expected. In other words, the checkpoint time of each application is doubled, and the Young/Daly period $\sqrt{2\mu C}$ should have been increased by a factor $\sqrt{2}$; the checkpointing strategy is no longer optimal, and efficiency will decrease.

Several heuristics are described in [72] for this contention problem. Each application attempts to use its Young/Daly period. The I/O token is given to only one application at every time step, and I/O operations cannot be interrupted once started. If several applications post concurrent requests to checkpoint, one will be selected and the other ones will continue their execution. The selection is based upon several criteria, including the time already spent waiting for I/O and the risk incurred by all the applications (increased waste) that have not been selected. See [72] for details.

4.3. Variable checkpoint time

This section deals with the scenario where the checkpoint duration is a stochastic random variable that obeys some well-known probability distributions. In this case, the question is when to take a checkpoint towards the end of the execution, so that the expectation of the work done is maximized. This assumes that the application is executing for a fixed duration, namely the length of the reservation that it has been granted, and the goal is to complete as much work as possible during this reservation. If the checkpoint is taken too early, some time without working has been wasted, but we may well lose the whole work if the checkpoint is taken later and lasts longer than expected, thereby exceeding the length of the reservation.

This problem has been studied in two flavors [73]. If checkpoints can be taken at any time, the optimal solution can be derived for a variety of probability distributions modeling checkpoint durations, in particular for uniform, exponential, normal, and lognormal distributions. The gain that can be achieved over the pessimistic approach, which assumes the longest possible checkpoint time and ensures that there is enough time to checkpoint (hence not taking any risk but losing some work if the checkpoint was faster), has also been assessed.

The problem is also interesting when the application is a linear chain of tasks, and checkpoints can only be taken at the end of a task. A static strategy has been proposed, where the optimal number of tasks before a checkpoint is computed before the execution, when

[1] Recall that #P is the class of counting problems that correspond to NP decision problems [66–68], and that #P-complete problems are at least as hard as NP-complete problems.

tasks (in addition to checkpoints) have IID stochastic execution times. The decision might be adapted dynamically, depending on the time effectively taken by each task, and hence a dynamic strategy has also been designed. This strategy decides whether to checkpoint or to continue the execution at the end of each task. Hence, it can be used even if distributions are not IID. Please refer to [73] for details.

5. Silent errors

In this section, we consider another type of error: while all previous sections addressed fail-stop errors, we now deal with silent errors, first in isolation and then in combination with fail-stop errors. It turns out that the Young/Daly formula can be extended to deal with both types of errors.

5.1. Background

We start with some background on silent errors, a.k.a. silent data corruptions (SDCs). While fail-stop errors lead to fatal interruptions (such as a crash) and cause the loss of the entire memory of the processor, silent errors only impact a given process and lead to incorrect results. But a silent error strikes undetected and the processor can continue its execution; sometimes the silent error can be detected and corrected, and some other times it degenerates into a fatal fail-stop error.

Silent errors may be caused, for instance, by arithmetic errors in the Arithmetic and Logic Unit (ALU), soft errors in the L1 cache which is usually not well protected, or in the L2 cache which might be protected by one parity bit, or bit flips in the dynamic random-access memory (DRAM) due to cosmic radiation, overheat and other sources [6,74–76].

There are several mathematics mechanisms to detect and correct silent errors, such as parity bits, error correcting codes (ECCs), and Chip-kill technology. They have been implemented to protect the DRAM and different cache layers to some extent. However, the closer the data is to the processing unit, the more frequent the access to that data and therefore the higher the overhead of these methods. Thus, processor caches are not protected by ECC in general, but by weaker mechanisms, like simple parity, exposing a higher risk of undetectable error in case of multiple simultaneous bit flips. Buses also often are a weak link in the protecting chain, making all data transfers at higher risk. In addition, the constant need to reduce component size and voltage increases the likelihood of silent errors.

Although many silent errors caused by one or multiple bits that spontaneously flip to the opposite state are caught by the above-mentioned hardware mechanisms, in reality, some bit flips still manage to pass undetected [77,78]. In a nutshell, silent errors have become a major threat due to the increase in problem size [79]: the larger the problem, the more memory to be used to store the data, the more frequent the errors, and the higher the probability of overriding ECC protection, generating multiple errors.

Another major problem 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 solve this problem, one may envision keeping several checkpoints in memory and restoring the application from the last *valid* checkpoint, thereby rolling back to the last *correct* state of the application [80]. But even if it was at all possible to store many checkpoints (which is very demanding in memory), one would not know how to identify the last valid one. Some verification mechanism, or detector, must be enforced.

5.2. Verification mechanisms

Considerable efforts have been directed at designing such verification mechanisms to reveal silent errors because error detection is usually very costly. Hardware mechanisms, such as ECC memory, can detect and even correct a fraction of errors, but in practice, they are complemented with software techniques. The only general-purpose method is to replicate the execution of the target computational kernel on two sets of processors (i.e., duplication) and to compare the results of both executions. If they do not coincide, an error has been detected, and the application must be executed a third time. To avoid a-posteriori re-execution, triplication (i.e., using three parallel executions of the same work) can be enforced, which allows for error correction in addition to error detection, using a simple majority vote. However, triplication (originally known as triple modular redundancy and voting [81]) is even more costly than duplication, which already requires half the resources to execute redundant operations.

Application-specific information can be very useful to enable ad-hoc solutions, which dramatically decreases the cost of detection. Many techniques have been advocated. They include memory scrubbing [82] and Algorithm-Based Fault Tolerance (ABFT) techniques [83–85], such as coding for the sparse-matrix vector multiplication kernel [85], block-wise checksum calculation for error-bounded lossy compressor [86], and coupling a higher-order with a lower-order scheme for PDEs [87]. Self-stabilizing corrections after error detection in the conjugate gradient method are investigated in [88]. Fault-tolerant iterative solvers for sparse linear algebra are introduced in [89–91], using extra checks such as re-computing inner products of vectors that should be orthogonal, or even re-computing the residual.

Overall speaking, application-specific detectors are very appealing due to their low cost as compared to replication, but they suffer from some limitations. Most application-specific detectors can only detect errors, not correct them. Next, they are used to detect errors of a certain type, while many types of errors can strike. For instance, with iterative methods, orthogonality tests will detect arithmetic errors but cannot do anything if we start with corrupted data in memory. Worse, even for a given type of error, the detector will likely not detect all the errors of that type, but only a fraction of them (one says that the detector recall is strictly smaller than one). In the previous example with the orthogonality test, the detector may well estimate that a scalar product is below some threshold while an error has struck one of the vectors. Finally, ABFT is one of the few methods that enables error correction in addition to detection; however, while ABFT can in principle detect and correct an arbitrary number of errors, it is currently limited in practice to detecting and correcting a single error due to the numerical instability of state-of-the-art methods that aim at building linearly independent checksum vectors in floating-point arithmetic.

Another category of SDC detection is based on data-analysis method, which makes use of the regular smoothness feature of the simulation data in either temporal or spatial dimension to detect potential outliers caused by SDC. This type of SDC detector relies on some sort of machine-learning algorithm that monitors the data produced by the application and gradually learns the type of variations and data ranges that the application observes during execution. While different regions of the global domain can be exposed to different behaviors, the learning process is local and therefore its detection mechanism is tuned to that specific region of the domain. There has been multiple machine-learning techniques proposed to achieve this type of SDC detection. The initial idea, proposed by Bautista-Gomez et al. [92], relied on a point-wise time series analysis capable of predicting the next value for each data point. This already produced promising results and the methodology was then refined in multiple directions.

For instance, in [93], Di et al. proposed an error feedback control model that can reduce the prediction errors for different linear prediction methods in SDC detection. They also developed a spatial-data-based even-sampling method to minimize the detection overheads.

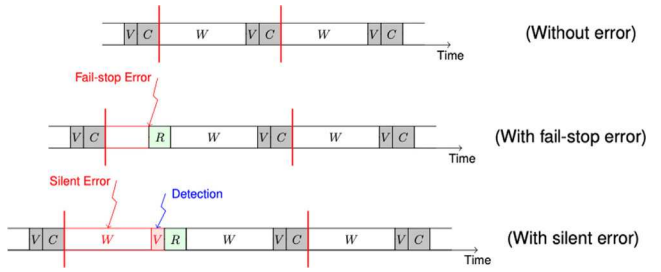

Fig. 3. Fail-stop errors versus silent errors. The vertical red lines mark the beginning and end of a periodic pattern.

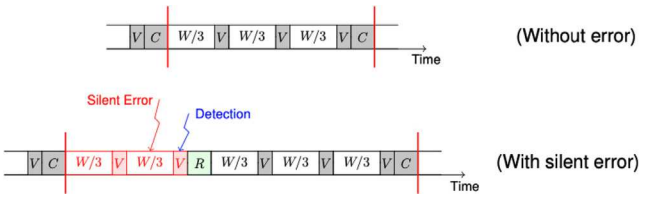

Fig. 4. Introducing two intermediate verifications in the period. The vertical red lines mark the beginning and end of a periodic pattern.

Berrocal et al. [94] analyzed the effectiveness of multiple spatial data prediction methods in detecting SDC errors, such as auto-regression (AR), autoregressive moving average (ARMA), acceleration-based predictor (ABP), linear curve fitting (LCF), and quadratic curve fitting (QCF).

Di et al. [95] further proposed an adaptive impact-driven SDC detection scheme (called AID) for HPC applications. In this work, the authors carefully characterized 18 HPC applications/benchmarks and discussed the runtime data features as well as the impact of SDCs on their execution results. They proposed an impact-driven detection model that does not blindly improve the prediction accuracy, but instead detects only influential SDCs to guarantee user-acceptable execution results. The AID method features a high runtime adaptability by allowing to select the best-fit prediction method according to the data changes for each process at runtime. These machine-learning based strategies rely mostly on detecting important anomaly variations induced by bit-flips. However, they cannot detect all types of variations, and small perturbations are likely to go undetected, although arguably those small variations might be irrelevant for the end result, in the same way rounding errors are. Nonetheless, undetected small perturbations could still produce significant changes in the final result.

Subasi et al. [96–98] explored the capabilities of machine learning techniques, such as support vector machines (SVM), in detecting SDCs, which further improves the detection capability, with a slight increase in execution cost.

An important drawback of these machine-learning based methods is that similarly to application-specific methods, they can only detect but not correct the errors.

5.3. Optimal period for silent errors

We study a generic solution that is agnostic of the nature of the verification mechanism (replication, checksum, error correcting code, coherence tests, etc.). We assume that we can rely on a fully general-purpose detector, of cost V. The idea is to perform a verification just before taking each checkpoint. If the verification succeeds, then one can safely store the checkpoint. If the verification fails, it means that a silent error has struck since the last checkpoint, which was duly verified, and one can safely recover from that checkpoint to resume the execution of the application.

See Fig. 3 for an illustration and comparison with fail-stop errors. If a silent error strikes, it is always detected only at the end of the period, when the verification V reveals the error. On the contrary, a fail-stop error strikes in the middle of the period on average and is detected immediately. Note that there is no downtime for silent errors because the processor can continue its execution after a silent error and does not need to be replaced. For simplification, we have used $D = 0$ in the second row of Fig. 3 (with fail-stop error), but recall there is a downtime after a fail-stop error in the general case.

Just as for fail-stop errors, we introduce the MTBE of individual nodes as μ_{ind}^{silent}. Here, the MTBE is the Mean Time Between Errors, the counterpart for silent errors of the MBTF for fail-stop errors. The MTBE of an application with p processors will be $\mu^{silent} = \frac{\mu_{ind}^{silent}}{p}$. Indeed, the frequency of silent errors is proportional to the number of arithmetic operations executed, and/or to the volume of the memory footprint of the application. Hence, the MTBE scales linearly with the size of the application, just as the MTBF does.

Now, consider a parallel application of MTBE μ^{silent}. At first sight, one could think that the optimal Young/Daly period for silent errors will be $W_{YD} = \sqrt{2\mu^{silent}(V+C)}$ because we have replaced each checkpoint of cost C by a verified checkpoint of cost $V + C$. However, because a silent error is always detected only at the end of the period, when the verification reveals the error, the formula will be different. With the notations of Section 2.2, the two sources of waste become $S_1 = \frac{V+C}{W+V+C}$ and $S_2 = \frac{1}{\mu^{silent}}(R + W + V)$. Altogether, both sources of waste approximately add up, so we have to find W that minimizes $S_1 + S_2$. Simplifying as before, we obtain $S_1 + S_2 \approx \frac{V+C}{W} + \frac{W}{\mu^{silent}}$, which is minimized for

$$W_{YD} = \sqrt{\mu^{silent}(V+C)}. \tag{3}$$

Eq. (3) is the Young/Daly formula for silent errors.

5.4. Extensions

The Young/Daly formula for silent errors could also be extended in several ways. The following discusses some extensions in this context.

5.4.1. Dealing with both fail-stop and silent errors

A first natural extension is to deal with both fail-stop and silent errors. Indeed, both sources of errors are likely to strike simultaneously when executing a parallel application. In that case, the failure-free waste $S_1 = \frac{V+C}{W+V+C}$ remains unchanged but the failure-induced waste S_2 should be updated to account for both error types:

$$S_2 = \frac{1}{\mu^{fail}}\left(D + R + \frac{1}{2}(W+V+C)\right) + \frac{1}{\mu^{silent}}(R+W+V).$$

Here, for clarity, we have used μ^{fail} instead of simply μ for the MTBF of the application. Simplifying again, we obtain that the total waste is minimized for

$$W_{YD} = \sqrt{\frac{V+C}{\frac{1}{2\mu^{fail}} + \frac{1}{\mu^{silent}}}}. \tag{4}$$

Eq. (4) is the Young/Daly formula for fail-stop and silent errors combined. We check that Eq. (4) reduces to Eq. (1) when $\mu^{silent} = \infty$ and $V = 0$ (only fail-stop errors) and to Eq. (3) when $\mu^{fail} = \infty$ (only silent errors). The general case uses the harmonic mean of μ^{fail} and μ^{silent} weighted by the average proportion of re-execution time in a period when struck by an error.

5.4.2. Placing intermediate verifications

The second extension applies when application-specific information enables to decrease the cost of a verification well below the cost of a checkpoint, i.e., when $V \ll C$. In that case, it is useful to insert some intermediate verifications within the period to detect silent errors early on. Assume that we deal with silent errors only and see Fig. 4 for an example of a period with two intermediate verifications (and a third one at the end of the period to verify the checkpoint). The failure-free waste S_1 is increased to $S_1 = \frac{3V+C}{W+3V+C} \approx \frac{3V+C}{W}$. However, the failure-induced waste is reduced to

$$\begin{aligned} S_2 &= \frac{1}{\mu^{silent}}\left(\frac{1}{3}(R+\frac{W}{3}+V)\right. \\ &\quad + \frac{1}{3}(R+\frac{2W}{3}+2V) \\ &\quad \left. + \frac{1}{3}(R+W+3V)\right) \\ &\approx \frac{1}{\mu^{silent}}\frac{(1+2+3)W}{9} \\ &= \frac{2W}{3\mu^{silent}}. \end{aligned}$$

To see this, with equal probability $\frac{1}{3}$, the silent error will strike either third of the pattern, and re-execution will cost either $\frac{W}{3}$ (first third), or $\frac{2W}{3}$ (second third), or W (last third). This leads to $S_1 + S_2$ minimized for $W = \sqrt{\frac{3}{2}\mu^{silent}(3V+C)}$ and we get $(S_1+S_2)_{\min} = 2\sqrt{\frac{2(3V+C)}{3\mu^{silent}}}$ for that value. In comparison, without intermediate verification, we had $(S_1+S_2)_{\min} = 2\sqrt{\frac{V+C}{\mu^{silent}}}$. We check that adding two intermediate verifications is better than none as long as $V \leq \frac{C}{3}$. This is very likely to be the case with an application-specific detector. We refer to [99] for the analysis of more general patterns including the derivation of the optimal number of intermediate verifications.

5.4.3. Embracing imperfect verifications

The results so far have assumed a perfect verification mechanism, while most real-world verifications are imperfect. In fact, many light-weight detectors (e.g., [92,94,100]) rely on data-analytics or machine-learning approaches to detect silent errors, and as a result, they typically have a limited recall (ratio of missed errors, or false negatives) and/or a limited precision (ratio of detected errors that are in fact not errors, or false positives). Also, if more than one such detectors are available to use, which one(s) should be favored? It turns out that imperfect verifications are nevertheless valuable, but their optimal placements within a pattern would not be equally spaced between two consecutive checkpoints. Further, choosing which verification (or combination of verifications) to use is an NP-hard problem, but greedy heuristics are shown to offer good practical performance by favoring those detectors with higher accuracy-to-cost ratios; see [101,102] for details on the analysis for incorporating imperfect verifications.

Finally, all the results regarding silent errors discussed in this section could also be applied to a linear chain of tasks by extending the dynamic programming framework by Toueg and Babaoglu [60] while including (imperfect) verifications; more details on the design of such algorithms can be found in [99,103].

6. When the optimal period is not $\Theta(\mu^{1/2})$

The Young/Daly formula and its many variations discussed so far all derived the optimal checkpoint period to be in the order of $\Theta(\mu^{1/2})$. However, in a few scenarios that apply redundancy to the application execution (e.g., via replication or faster re-execution), the optimal period turns out to deviate from this order. Intuitively, the application's resiliency to failures increases due to the added redundancy, making the optimal period longer than the classical result.

In one such scenario, two platforms cooperate to execute an application. Both platforms share the same periodic pattern (with length W followed by a checkpoint), and they also share the same storage system for placing the checkpoints. If a failure strikes one platform, it will recover from the previous checkpoint to re-execute the pattern (same as the single-platform case). However, if any platform successfully completes the pattern, the other platform will "jump ahead" in its execution by synchronizing through the checkpoint, so that both platforms can start executing the next pattern simultaneously. The optimal period W for this scenario turns out to be in the order of $\Theta(\mu^{2/3})$ when the two platforms are homogeneous (i.e., with the same execution speed). A thorough analysis is also provided in [104] for heterogeneous platforms.

In a similar scenario that copes with silent errors, an application is executed synchronously by three platforms that share the same periodic pattern. To detect/correct silent errors, "majority voting" is used at the end of a pattern: If at least two platforms agree on the execution results, then a checkpoint can be safely taken, and all platforms will start executing the next pattern together. However, if the results returned by all the three platforms are different from each other, suggesting that at least two platforms have been struck by silent errors, then no consensus can be reached, and all platforms will roll back to the previous checkpoint and re-execute the same pattern again. In this scenario, the optimal period turns out to be in the order of $\Theta(\mu^{2/3})$ as well. The details are derived in [105], which also shows the optimal period for the general case where more than three platforms are used to execute the application.

In another scenario, a different speed is applied when re-executing a periodic pattern when a (fail-stop) failure occurs. In particular, the first execution of the pattern uses speed s_1, and all subsequent re-executions of the same pattern (due to failures and rolling back to the last checkpoint) are executed with a faster speed $s_2 > s_1$, assuming that the platform is equipped with dynamic voltage and frequency scaling (DVFS) capabilities. This scenario was originally studied in [106] in the context of minimizing the total energy consumption subject to an execution time constraint for running an application. A side result obtained under this scenario for the special case of $s_2 = 2s_1$ shows that the optimal checkpointing period is again in the order of $\Theta(\mu^{2/3})$, even for optimizing the execution time alone. This result suggests that a faster re-execution speed can help reduce the resilience overhead with a longer checkpointing period.

7. Combining checkpointing with replication

When the checkpointing cost and/or the error rate are very high, checkpointing might not be enough since the checkpointing period might become smaller than the time required to take a checkpoint. In this case, a solution consists in replicating part of the execution, as has already been discussed in Section 6 in some particular settings.

7.1. Preemptible applications

The use of replication in order to deal with fail-stop errors, in the case of preemptible applications, enables the application to survive several errors before being interrupted. The idea is to group processors by pairs, and have two processors do the same bunch of work. Hence, this means that the checkpointing period can be significantly longer than without replication, since the execution is more reliable. The standard way of using replication in that case consists in using, once more, the Young/Daly formula, but considering the mean time to interruption, MTTI (rather than the MTBF), which accounts for the effect of replication. Furthermore, failed processors are never restarted with the usual assumptions. For example, in the *restart* strategy introduced in [107], failed processors are restarted after each checkpoint. With this strategy, the optimal checkpointing period can be computed, and it turns out to be much larger than the period dictated by Young/Daly,

hence also decreasing the I/O pressure and decreasing the overhead induced by replication.

As discussed in Section 6, previous work has also investigated replication in terms of using a whole platform as backup (see [104]), where the backup platform may execute at a different speed than the main platform. The technique has been extended to detect and correct silent errors in [105], where either the platform is partitioned into several parts, or where each process is replicated. A detailed analytical study has been conducted for all scenarios, hence guiding the user in deciding whether it is beneficial, given the parameters of the application and the target platform, to combine checkpointing with replication.

7.2. Linear chains of tasks

When the platform is subject to both fail-stop and silent errors, it might be useful to apply both checkpointing and replication for some tasks, if either resilience technique is not sufficient by itself. In particular, for a linear chain of tasks, the goal is to decide, for each task, whether to checkpoint and/or replicate it to ensure its reliable execution. In [108], an optimal dynamic programming algorithm of quadratic complexity is proposed to solve both problems. This 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. Hence, combining both techniques has a promising potential to minimize the execution time of linear workflows in error-prone environments.

8. Conclusion and open problems

Summary. This survey has dealt with checkpointing policies based upon the Young/Daly period and has assessed its usefulness and robustness together with its limitations. Originally restricted to preemptible applications and blocking coordinated checkpointing protocols to cope with fail-stop errors, the Young/Daly formula has proven very useful in a much larger applicative spectrum, as shown by the many extensions addressed in this survey. While the accuracy of the formula is only known for memoryless failures, the robustness and efficiency of the formula has been experimentally established in a wide variety of settings. In a nutshell, the Young/Daly formula is a solid tool for tightly-coupled parallel applications, and the answer to the question in the title is a plain yes. The main limitations of the formula are related to its use for workflows because inter-task dependencies dramatically complicate the problem to decide when and which tasks to checkpoint.

Open problems. We discuss some further extensions and open problems to conclude the paper.

For preemptible applications (Section 2), we have focused on coordinated checkpointing onto non-volatile storage, but most of the results hold for other methods that reduce checkpoint overhead, such as in-memory checkpointing [109–111], two-level checkpointing [31,112] and multi-level checkpointing [6,7,30,32] (see also Section 2.4.3).

For task-based applications, one could envision an extension of coordinated checkpointing designed for such systems. Using periodic coordinated checkpointing to decide which tasks to checkpoint in a distributed task system consists of finding a period between two checkpoint waves, and coordinating all the processes of the application to checkpoint their state. Applying this heuristic to a task-based system does not ensure optimal performance because the amount of data to checkpoint depends on the number and input of the ready tasks and varies over time, which is outside the assumptions of the periodic checkpointing approach. However, by continuously adapting the period to the amount of work executed (either maximal or averaged across all processors), this strategy may provide an efficient solution in scenarios where tasks are small and where failures are rare.

For both application models (preemptible and task-based), we have assumed failure independence. Indeed, the standard model assumes IID failure inter-arrival times, or IATs, on each node, with a common distribution D. As for *temporal* dependence, it has been observed many times that when a failure occurs, it may trigger other failures that will strike different system components [22,113,114]. As an example, a failing cooling system may cause a series of successive crashes of different nodes. Also, an outstanding error in the file system will likely be followed by several others [115,116]. As for *spatial* dependence, it is clear that the overheating of some node in a cabinet is quite likely to be followed by the overheating of neighbor nodes (which comes atop of a temporal dependence as well). Bautista-Gomez et al. [114] have studied nine systems, and they report periods of high failure density in all of them. They call these periods *cascade failures*. This observation has led them to revisit the temporal failure independence assumption, and to design bi-periodic checkpointing algorithms that use different periods in normal (failure-free) and degraded (with failure cascades) modes. Tiwari et al. [113] introduce a dynamic strategy called *lazy checkpointing* to adjust to changes in the failure rate. Another approach has been proposed in [117], using quantiles of consecutive IAT pairs. It is an open problem to derive an efficient checkpointing strategy that can account for temporal or spatial dependence between failures. For example, spatial dependence calls for a variant of in-memory checkpointing where the buddy of a processor (acting replica of a checkpoint) is chosen far away from that processor, while it is better to select a physical neighbor to optimize communication overhead if failures are truly independent. Complicated trade-offs must be achieved.

Finally, some parts of the application are critical (such as execution code) and must be protected from silent errors at all costs while other parts (like non-critical data) may be loosely and infrequently verified by cheap mechanisms; we speak of *selective reliability* in such a framework. More generally, *trustworthy computing* is the problem of guaranteeing, at least with some high probability, that the final results of a parallel application are correct. The higher the flop count and the larger the data footprint, the more challenging to achieve this goal.

CRediT authorship contribution statement

Leonardo Bautista-Gomez: Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **Anne Benoit:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **Sheng Di:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **Thomas Herault:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **Yves Robert:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **Hongyang Sun:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

A preliminary and much shorter version [15] of this paper has appeared in Proceedings of the 14th International Conference on Contemporary Computing, August 2022. Many new topics have been added to [15] owing to the contributions of authors from several JLESC institutions. This research was supported in part by the U.S. National Science Foundation grant #2135309, U.S. Department of Energy, Office

of Science, Advanced Scientific Computing Research (ASCR), under contract DE-AC02-06CH11357. The authors thank the reviewers for their comments and suggestions, which greatly helped improve the final version of the paper. Finally, the authors gratefully acknowledge the support of their institutions to JLESC, the *Joint Laboratory for Extreme Scale Computing* (formerly the *Joint Laboratory for Petascale Computing*), which has been a perfect mechanism to foster collaboration since its creation in 2009.

References

[1] LANL, NERSC, SNL, APEX workflows (version 1), Technical report, Sandia National Laboratories and Los Alamos National Laboratories, 2015, https://www.nersc.gov/assets/Crossroads--NERSC-9-RFP/apex-workflow-v1.pdf.

[2] LANL, NERSC, SNL, APEX workflows (version 2), Technical report, Sandia National Laboratories and Los Alamos National Laboratories, 2016, http://www.nersc.gov/assets/apex-workflows-v2.pdf.

[3] M. Gamell, D.S. Katz, H. Kolla, J. Chen, S. Klasky, M. Parashar, Exploring automatic, online failure recovery for scientific applications at extreme scales, in: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 895–906.

[4] A. Eisenman, K.K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, M. Annavaram, {Check-n-run}: A checkpointing system for training deep learning recommendation models, in: 19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 22, 2022, pp. 929–943.

[5] F. Cappello, S. Di, S. Li, X. Liang, A.M. Gok, D. Tao, C.H. Yoon, X.-C. Wu, Y. Alexeev, F.T. Chong, Use cases of lossy compression for floating-point data in scientific data sets, Int. J. High Perform. Comput. Appl. 33 (6) (2019) 1201–1220.

[6] A. Moody, G. Bronevetsky, K. Mohror, B.R. De Supinski, Design, modeling, and evaluation of a scalable multi-level checkpointing system, in: SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2010, pp. 1–11.

[7] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, S. Matsuoka, FTI: High performance fault tolerance interface for hybrid systems, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–32.

[8] B. Nicolae, A. Moody, E. Gonsiorowski, K. Mohror, F. Cappello, Veloc: Towards high performance adaptive asynchronous checkpointing at large scale, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS, IEEE, 2019, pp. 911–920.

[9] M.A. Heroux, L.C. McInnes, R. Thakur, J.S. Vetter, X.S. Li, J. Aherns, T. Munson, K. Mohror, ECP software technology capability assessment report, Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2020.

[10] S. Habib, Cosmology and computers: Haccing the universe, in: 2015 International Conference on Parallel Architecture and Compilation, PACT, IEEE Computer Society, 2015, p. 406.

[11] A.S. Kronfeld, LATTICE QCD, 1992, pp. 421–474, Perspectives in the Standard Model (TASI-91)-Proceedings of the Theoretical Study Institute in Elementary Particle Physics. Edited by ELLIS RK ET AL. Published by World Scientific Publishing Co. Pte. Ltd.

[12] S. Atchley, C. Zimmer, J.R. Lange, D.E. Bernholdt, V.G.M. Vergara, T. Beck, M.J. Brim, R. Budiardja, S. Chandrasekaran, M. Eisenbach, et al., Frontier: exploring exascale the system architecture of the first exascale supercomputer, in: SC23: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2023, pp. 1–16.

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

[14] J.T. Daly, A higher order estimate of the optimum checkpoint interval for restart dumps, FGCS 22 (3) (2006) 303–312.

[15] A. Benoit, Y. Du, T. Herault, L. Marchal, G. Pallez, L. Perotin, Y. Robert, H. Sun, F. Vivien, Checkpointing à La Young/Daly: An Overview, in: Proceedings of the 14th International Conference on Contemporary Computing (IC3), 2022, pp. 701–710.

[16] K.M. Chandy, L. Lamport, Distributed snapshots: Determining global states of distributed systems, ACM Trans. Comput. Syst. 3 (1) (1985) 63–75.

[17] T. Herault, Y. Robert (Eds.), Fault-tolerance techniques for high-performance computing, in: Computer Communications and Networks, Springer Verlag, 2015.

[18] K. Ferreira, J. Stearley, J.H.I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P.G. Bridges, D. Arnold, Evaluating the viability of process replication reliability for exascale systems, in: SC'11, ACM, 2011.

[19] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, M. Snir, Toward exascale resilience: 2014 update, Supercomput. Front. Innovat. 1 (1) (2014).

[20] A. Benoit, L. Perotin, Y. Robert, F. Vivien, Checkpointing strategies to protect parallel jobs from non-memoryless fail-stop errors, Research report RR-9465, INRIA, 2022, Available at https://hal.inria.fr/hal-03610883.

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

[22] E. Heien, D. Kondo, A. Gainaru, D. LaPine, B. Kramer, F. Cappello, Modeling and tolerating heterogeneous failures in large parallel systems, in: Proc. SC'11, 2011.

[23] B. Schroeder, G.A. Gibson, A large-scale study of failures in high-performance computing systems, in: Proc. of DSN, 2006, pp. 249–258.

[24] B. Schroeder, G.A. Gibson, Understanding failures in petascale computers, J. Phys. Conf. Ser. 78 (1) (2007).

[25] O. Subasi, G. Kestor, S. Krishnamoorthy, Toward a general theory of optimal checkpoint placement, in: CLUSTER, IEEE, 2017, pp. 464–474.

[26] O. Subasi, T. Martsinkevich, F. Zyulkyarov, O. Unsal, J. Labarta, F. Cappello, Unified fault-tolerance framework for hybrid task-parallel message-passing applications, IJHPCA 32 (5) (2018) 641–657.

[27] N. El-Sayed, B. Schroeder, To checkpoint or not to checkpoint: Understanding energy-performance-I/O tradeoffs in HPC checkpointing, in: CLUSTER, 2014, pp. 93–102.

[28] G. Aupy, A. Benoit, T. Hérault, Y. Robert, Optimal checkpointing period: time vs. energy, in: PMBS 2013, the 4th Int. Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, LNCS Springer Verlag, 2013.

[29] E. Gelenbe, P. Boryszko, M. Siavvas, J. Domanska, Optimum checkpoints for time and energy, in: 28th MASCOTS, IEEE, 2020, pp. 1–8.

[30] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, H. Sun, Towards optimal multi-level checkpointing, IEEE Trans. Comput. 66 (7) (2017) 1212–1226.

[31] S. Di, Y. Robert, F. Vivien, F. Cappello, Toward an optimal online checkpoint solution under a two-level HPC checkpoint model, IEEE Trans. Parallel Distrib. Syst. 28 (1) (2017) 244–259.

[32] S. Di, M.S. Bouguerra, L. Bautista-Gomez, F. Cappello, Optimization of multi-level checkpoint model for large scale HPC applications, in: IPDPS, IEEE, 2014.

[33] S. Di, L. Bautista-Gomez, F. Cappello, Optimization of a multilevel checkpoint model with uncertain execution scales, in: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 907–918.

[34] S. Arunagiri, J.T. Daly, P.J. Teller, Modeling and Analysis of Checkpoint I/O Operations, in: Analytical and Stochastic Modeling Techniques and Applications: 17th International Conference, Springer, 2010, pp. 387–399.

[35] O. Weidner, M. Atkinson, A. Barker, R. Filgueira Vicente, Rethinking high performance computing platforms: Challenges, opportunities and recommendations, in: Proc. Data-Intensive Distributed Computing, DIDC, ACM, 2016.

[36] A. Benoit, L. Perotin, Y. Robert, H. Sun, Checkpointing workflows à la Young/daly is not good enough, ACM Trans. Parallel Comput. 9 (4) (2022) 1–25.

[37] R. Garg, G. Price, G. Cooperman, MANA for MPI: MPI-agnostic network-agnostic transparent checkpointing, in: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 49–60.

[38] J. Ansel, K. Arya, G. Cooperman, DMTCP: Transparent checkpointing for cluster computations and the desktop, in: 2009 IEEE International Symposium on Parallel & Distributed Processing, IPDPS'09, IEEE, Rome, Italy, 2009, pp. 1–12.

[39] G. Zheng, X. Ni, L.V. Kalé, A scalable double in-memory checkpoint and restart scheme towards exascale, in: IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN 2012, IEEE, 2012, pp. 1–6.

[40] J. Dongarra, T. Herault, Y. Robert, Revisiting the double checkpointing algorithm, in: APDCM 2013, IEEE Computer Society Press, 2013, pp. 706–715.

[41] J.S. Plank, K. Li, M.A. Puening, Diskless checkpointing, IEEE Trans. Parallel Distrib. Syst. 9 (10) (1998) 972–986.

[42] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, J.J. Dongarra, An evaluation of user-level failure mitigation support in MPI, Computing 95 (12) (2013) 1171–1184.

[43] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, et al., Open MPI: Goals, concept, and design of a next generation MPI implementation, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11, Springer, 2004, pp. 97–104.

[44] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Comput. 22 (6) (1996) 789–828, see also http://www-unix.mcs.anl.gov/mpi/mpich/.

[45] M. Gamell, R.F. Van der Wijngaart, K. Teranishi, M. Parashar, Specification of Fenix MPI Fault Tolerance library, Version 1.0.1, Technical Report SAND2016-10522, Sandia National Laboratory, 2016, https://www.osti.gov/servlets/purl/1330192.

[46] X. Xu, R. Mo, F. Dai, W. Lin, S. Wan, W. Dou, Dynamic resource provisioning with fault tolerance for data-intensive meteorological workflows in cloud, IEEE Trans. Ind. Inform. 16 (9) (2019) 6172–6181.

[47] G. Kandaswamy, A. Mandal, D.A. Reed, Fault tolerance and recovery of scientific workflows on computational grids, in: 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid, CCGRID, IEEE, 2008, pp. 777–782.

[48] A. Bala, I. Chana, Fault tolerance-challenges, techniques and implementation in cloud computing, Int. J. Comput. Sci. Issues (IJCSI) 9 (1) (2012) 288.

[49] S. Prathiba, S. Sowvarnica, Survey of failures and fault tolerance in cloud, in: 2017 2nd International Conference on Computing and Communications Technologies, ICCCT, 2017, pp. 169–172.

[50] Y. Ding, G. Yao, K. Hao, Fault-tolerant elastic scheduling algorithm for workflow in cloud systems, Inform. Sci. 393 (2017) 47–65.

[51] P. Kumari, P. Kaur, A survey of fault tolerance in cloud computing, J. King Saud Univ. Comput. Inf. Sci. (2018).

[52] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, J.J. Dongarra, PaRSEC: Exploiting heterogeneity to enhance scalability, Comput. Sci. Eng. 15 (6) (2013) 36–45.

[53] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, in: Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, the Netherlands, August 25-28, 2009. Proceedings 15, Springer, 2009, pp. 863–874.

[54] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing locality and independence with logical regions, in: SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE, 2012, pp. 1–11.

[55] A. Fernández, V. Beltran, X. Martorell, R.M. Badia, E. Ayguadé, J. Labarta, Task-based programming with OmpSs and its application, in: Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II 20, Springer, 2014, pp. 601–612.

[56] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, et al., Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, IEEE, 2011, pp. 1432–1441.

[57] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, S.P. Thibault, Achieving high performance on supercomputers with a sequential task-based programming model, IEEE Trans. Parallel Distrib. Syst. (2017).

[58] C. Cao, T. Hérault, G. Bosilca, J.J. Dongarra, Design for a soft error resilient dynamic task-based runtime, in: 2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015, IEEE Computer Society, 2015, pp. 765–774.

[59] R. Lion, S. Thibault, From tasks graphs to asynchronous distributed checkpointing with local restart, in: 2020 IEEE/ACM 10th Workshop on Fault Tolerance for HPC At EXtreme Scale, FTXS, IEEE, 2020, pp. 31–40.

[60] S. Toueg, Ö. Babaoğlu, On the optimum checkpoint selection problem, SIAM J. Comput. 13 (3) (1984).

[61] Y. Du, G. Pallez, L. Marchal, Y. Robert, Optimal checkpointing strategies for iterative applications, IEEE Trans. Parallel Distrib. Syst. 33 (3) (2022) 507–522.

[62] G. Aupy, A. Benoit, H. Casanova, Y. Robert, Scheduling computational workflows on failure-prone platforms, Int. J. Network. Comput. 6 (1) (2016) 2–26.

[63] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, H. Abbasi, Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform, in: Proc. 26th IEEE Int. Parallel and Distributed Processing Symposium, 2012, pp. 1352–1363.

[64] J.N. Hagstrom, Computational complexity of PERT problems, Networks 18 (2) (1988) 139–147.

[65] M.L. Pinedo, Scheduling: Theory, Algorithms, and Systems, fifth ed., Springer, 2016.

[66] L.G. Valiant, The complexity of enumeration and reliability problems, SIAM J. Comput. 8 (3) (1979) 410–421.

[67] J.S. Provan, M.O. Ball, The complexity of counting cuts and of computing the probability that a graph is connected, SIAM J. Comp. 12 (4) (1983) 777–788.

[68] H.L. Bodlaender, T. Wolle, A note on the complexity of network reliability problems, IEEE Trans. Inform. Theory 47 (2004) 1971–1988.

[69] L. Han, L.-C. Canon, H. Casanova, Y. Robert, F. Vivien, Checkpointing workflows for fail-stop errors, IEEE Trans. Comput. 67 (8) (2018) 1105–1120.

[70] L. Han, V. Le Fèvre, L.-C. Canon, Y. Robert, F. Vivien, A generic approach to scheduling and checkpointing workflows, in: ICPP'2018, the 47th Int. Conf. on Parallel Processing, 2018.

[71] P. Sigdel, X. Yuan, N. Tzeng, Realizing best checkpointing control in computing systems, IEEE TPDS 32 (2) (2021) 315–329.

[72] T. Herault, Y. Robert, A. Bouteiller, D. Arnold, K.B. Ferreira, G. Bosilca, J. Dongarra, Checkpointing strategies for shared high-performance computing platforms, Int. J. Network. Comput. 9 (1) (2019) 28–52.

[73] Q. Barbut, A. Benoit, T. Herault, Y. Robert, F. Vivien, When to checkpoint at the end of a fixed-length reservation? in: Proc. of ACM Conference FTXS'23, 2023.

[74] T. O'Gorman, The effect of cosmic rays on the soft error rate of a DRAM at ground level, IEEE Trans. Electron Devices 41 (4) (1994) 553–557.

[75] J.F. Ziegler, H.W. Curtis, H.P. Muhlfeld, C.J. Montrose, B. Chin, IBM experiments in soft fails in computer electronics, IBM J. Res. Dev. 40 (1) (1996) 3–18.

[76] J. Ziegler, M. Nelson, J. Shell, R. Peterson, C. Gelderloos, H. Muhlfeld, C. Montrose, Cosmic ray soft error rates of 16-mb DRAM memory chips, IEEE J. Solid-State Circuits 33 (2) (1998) 246–252.

[77] V. Sridharan, N. DeBardeleben, S. Blanchard, K.B. Ferreira, J. Stearley, J. Shalf, S. Gurumurthi, Memory errors in modern systems: The good, the bad, and the ugly, in: 20th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS, ACM, 2015, pp. 297–310.

[78] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, S. McIntosh-Smith, Unprotected computing: A large-scale study of DRAM raw error rate on a supercomputer, in: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 645–655.

[79] M. Snir, et al., Addressing failures in exascale computing, Int. J. High Perform. Comput. Appl. 28 (2) (2014) 129–173.

[80] G. Lu, Z. Zheng, A.A. Chien, When is multi-version checkpointing needed? in: Proc. 3rd Workshop on Fault-Tolerance for HPC At Extreme Scale, FTXS, 2013, pp. 49–56.

[81] R.E. Lyons, W. Vanderkulk, The use of triple-modular redundancy to improve computer reliability, IBM J. Res. Dev. 6 (2) (1962) 200–209.

[82] A.A. Hwang, I.A. Stefanovici, B. Schroeder, Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design, SIGARCH Comput. Archit. News 40 (1) (2012) 111–122.

[83] K.-H. Huang, J.A. Abraham, Algorithm-based fault tolerance for matrix operations, IEEE Trans. Comput. 33 (6) (1984) 518–528.

[84] G. Bosilca, R. Delmas, J. Dongarra, J. Langou, Algorithm-based fault tolerance applied to high performance computing, J. Parallel Distrib. Comput. 69 (4) (2009) 410–416.

[85] M. Shantharam, S. Srinivasmurthy, P. Raghavan, Fault tolerant preconditioned conjugate gradient for sparse linear system solution, in: ICS, ACM, 2012.

[86] S. Li, S. Di, K. Zhao, X. Liang, Z. Chen, F. Cappello, Resilient error-bounded lossy compressor for data transfer, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '21, Association for Computing Machinery, New York, NY, USA, 2021.

[87] A.R. Benson, S. Schmit, R. Schreiber, Silent error detection in numerical time-stepping schemes., Int. J. High Perform. Comput. Appl. (2014).

[88] P. Sao, R. Vuduc, Self-stabilizing iterative solvers, in: ScalA '13, 2013.

[89] Z. Chen, Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods, in: Proc. PPoPP, 2013, pp. 167–176.

[90] M. Heroux, M. Hoemmen, Fault-tolerant iterative methods via selective reliability, Research report SAND2011-3915 C, Sandia Nat. Lab., 2011.

[91] G. Bronevetsky, B. de Supinski, Soft error vulnerability of iterative linear algebra methods, in: ICS, ACM, 2008.

[92] L. Bautista-Gomez, F. Cappello, Detecting and correcting data corruption in stencil applications through multivariate interpolation, in: 2015 IEEE International Conference on Cluster Computing, 2015, pp. 595–602.

[93] S. Di, E. Berrocal, F. Cappello, An efficient silent data corruption detection method with error-feedback control and even sampling for HPC applications, in: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), IEEE Computer Society, Los Alamitos, CA, USA, 2015, pp. 271–280.

[94] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, F. Cappello, Lightweight silent data corruption detection based on runtime data analysis for HPC applications, in: HPDC, ACM, 2015.

[95] S. Di, F. Cappello, Adaptive impact-driven detection of silent data corruption for HPC applications, IEEE Trans. Parallel Distrib. Syst. 27 (10) (2016) 2809–2823.

[96] O. Subasi, S. Di, L. Bautista-Gomez, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, S. Krishnamoorthy, F. Cappello, Exploring the capabilities of support vector machines in detecting silent data corruptions, Sustain. Comput. Informat. Syst. 19 (2018) 277–290.

[97] O. Subasi, S. Di, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, S. Krishnamoorthy, F. Cappello, MACORD: Online adaptive machine learning framework for silent error detection, in: 2017 IEEE International Conference on Cluster Computing, CLUSTER, 2017, pp. 717–724.

[98] O. Subasi, S. Di, L. Bautista-Gomez, P. Balaprakash, O. Unsal, J. Labarta, A. Cristal, F. Cappello, Spatial support vector regression to detect silent errors in the exascale era, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid, 2016, pp. 413–424.

[99] A. Benoit, A. Cavelan, Y. Robert, H. Sun, Assessing general-purpose algorithms to cope with fail-stop and silent errors, ACM Trans. Parallel Comput. 3 (2) (2016).

[100] L. Bautista Gomez, F. Cappello, Detecting silent data corruption through data dynamic monitoring for scientific applications, in: PPoPP, ACM, 2014.

[101] A. Cavelan, S.K. Raina, Y. Robert, H. Sun, Assessing the impact of partial verifications against silent data corruptions, in: Proc. ICPP, 2015.

[102] L. Bautista-Gomez, A. Benoit, A. Cavelan, S. Raina, Y. Robert, H. Sun, Coping with recall and precision of soft error detectors, J. Parallel Distrib. Comput. 98 (2016) 8–24.

[103] A. Benoit, A. Cavelan, Y. Robert, H. Sun, Multi-level checkpointing and silent error detection for linear workflows, J. Comput. Sci. 28 (2018) 398–415.

[104] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, Optimal checkpointing period with replicated execution on heterogeneous platforms, in: FTXS, 2017, pp. 9–16.
[105] A. Benoit, A. Cavelan, F. Cappello, P. Raghavan, Y. Robert, H. Sun, Coping with silent and fail-stop errors at scale by combining replication and checkpointing, J. Parallel Distrib. Comput. 122 (2018) 209–225.
[106] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, H. Sun, A different re-execution speed can help, in: ICPP Workshop, 2016, pp. 250–257.
[107] A. Benoit, T. Herault, V. Le Fèvre, 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, Association for Computing Machinery, New York, NY, USA, 2019.
[108] A. Benoit, A. Cavelan, F. Ciorba, V. Le Fèvre, Y. Robert, Combining checkpointing and replication for reliable execution of linear workflows with fail-stop and silent errors, Int. J. Network. Comput. 9 (1) (2019) 2–27.
[109] G. Zheng, L. Shi, 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.
[110] X. Ni, E. Meneses, 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.
[111] J. Dongarra, T. Hérault, Y. Robert, Performance and reliability trade-offs for the double checkpointing algorithm, Int. J. Network. Comput. 4 (1) (2014) 23–41.
[112] L. Silva, J. Silva, Using two-level stable storage for efficient checkpointing, IEE Proc. Softw. 145 (6) (1998) 198–202.
[113] D. Tiwari, S. Gupta, S.S. Vazhkudai, Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems, in: 44th Int. Conf. on Dependable Systems and Networks, IEEE, 2014, pp. 25–36.
[114] L. Bautista-Gomez, A. Gainaru, S. Perarnau, D. Tiwari, S. Gupta, C. Engelmann, F. Cappello, M. Snir, Reducing waste in extreme scale systems through introspective analysis, in: IPDPS, IEEE, 2016, pp. 212–221.
[115] K. Schroiff, P. Gemsjaeger, C. Bolik, Cascading failover of a data management application for shared disk file systems in loosely coupled node clusters, 2006, US Patent 6, 990, 606.
[116] S.Y. Ko, I. Hoque, B. Cho, I. Gupta, Making cloud intermediate data fault-tolerant, in: Proc. 1st ACM Symposium on Cloud Computing, SoCC '10, ACM, 2010.
[117] G. Aupy, Y. Robert, F. Vivien, Assuming failure independence: are we right to be wrong? in: FTS'2017, 2017.

L. Bautista Gomez (BSC) is a Senior Researcher at the Barcelona Supercomputing Center where he works on multiple European projects. He was awarded the 2016 IEEE Computer Society Technical Committee on Scalable Computing (TCSC) Award for Excellence in Scalable Computing (Early Career Researcher). He developed a scalable multilevel checkpointing library called Fault Tolerance Interface (FTI) to guarantee application resilience at extreme scale. FTI is currently one of the most popular multilevel checkpointing libraries and it is the focus of multiple ongoing European research projects. See https://www.bsc.es/es/bautista-gomez-leonardo for further information.

A. Benoit (Inria) is an Associate Professor in the Computer Science Laboratory LIP at ENS Lyon, France, and the IEEE TCPP Chair. She is a senior member of the IEEE, and she has been elected a Senior Member of Institut Universitaire de France in 2023. Her research interests include multi-criteria scheduling algorithms and resilient techniques for parallel and distributed platforms. See http://graal.ens-lyon.fr/~abenoit/ for further information.

S. Di (ANL) is a Computer Scientist in the Mathematics and Computer Science (MCS) division of Argonne National Laboratory. His current research interest includes lossy compression for scientific datasets, high performance computing, scalable computing, and fault tolerance. He is a senior member of IEEE, institute fellow of NAISE, also the scientist at Large through the Consortium for Advanced Science and Engineering (CASE) at the University of Chicago. He is the DOE 2021 Early Career Research Program Award Winner, and also the recipient of 2018 IEEE-Chicago Distinguished Mentoring Award and 2019 IEEE-Chicago Distinguished R&D Award. See https://www.mcs.anl.gov/~shdi for further information.

T. Herault (UTK) is a Research Assistant Professor at the Innovative Computing Laboratory at University of Tennessee, Knoxville. He is an expert in distributed algorithms and architectures. He was one of the main developers of the MPICH-V MPI implementation for volatile resources. He is one of the main architects of the User Level Fault Mitigation (ULFM) MPI extension and of the Parallel Runtime Scheduling and Execution Controller (PaRSEC) middleware. See https://icl.utk.edu/~herault/ for further information.

Y. Robert (Inria) is a Full Professor at ENS Lyon, a Fellow of the IEEE and a former Senior Member of Institut Universitaire de France. He received the 2014 IEEE TCSC Award for Excellence in Scalable Computing, the 2016 IEEE TCPP Outstanding Service Award, and the 2020 IEEE CS Charles Babbage Award. He holds a Visiting Scientist position at the Innovative Computing Laboratory, University of Tennessee Knoxville, since 2011. His main research interests are scheduling techniques, parallel algorithms and resilient approaches for large-scale platforms. See http://graal.ens-lyon.fr/~yrobert/ for further information.

H. Sun (KU) is an Assistant Professor in the Department of Electrical Engineering and Computer Science (EECS) at the University of Kansas, USA. His research interests are in the broad area of high-performance computing (HPC), cloud/edge computing, and computational data science, with a particular focus on enhancing the performance, reliability, resilience, and energy/thermal efficiency of large-scale computing systems and applications. See https://www.ittc.ku.edu/~sun/ for further information.