Fault-tolerant numerical iterative algorithms at scale

Alix Tremodeux, Emmanuel Agullo, Anne Benoit, Luc Giraud, Thomas Herault, Yves Robert

Fault-tolerant numerical iterative algorithms at scale

Alix Tremodeux, .Emmanuel Agullo, Anne Benoit, Luc Giraud, Thomas Herault, Yves Robert

RESEARCH REPORT

N° 9567

January 2025

Project-Team ROMA,Concace

ISSN 0249-6399 ISRN INRIA/RR--9567--FR+ENG

Fault-tolerant numerical iterative algorithms at scale

Alix Tremodeux*, .Emmanuel Agullo†, Anne Benoit‡, Luc Giraud, Thomas Herault§, Yves Robert¶

Project-Team ROMA,Concace

Research Report n° 9567 — January 2025 — 26 pages

Abstract: This work investigates how to protect numerical iterative algorithms from all types of errors that can strike at scale: fail-stop errors (a.k.a. failures) and silent errors, striking both as computation errors and memory bit-flips. We combine various techniques: detectors for computation errors, checksums for memory errors, and checkpoint/restart for failures. The objective is to minimize the expected time per iteration of the algorithm. We design a hierarchical pattern that combines and interleaves all these fault-tolerance mechanisms, and we determine the optimal periodic pattern that achieves this objective. We instantiate these results for the performance analysis of the Preconditioned Conjugate Gradient (PCG) algorithm: we report several scenarios where the optimal pattern dramatically decreases the overhead due to error mitigation.

Key-words: Numerical iterative algorithms, computation errors, memory bit-flips, fail-stop errors, Preconditioned Conjugate Gradient.

* ENS Lyon, UCB Lyon, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
† Inria Bordeaux
‡ ENS Lyon, UCB Lyon, CNRS, Inria, LIP, F-69342, LYON Cedex 07, and Institut Universitaire de France
§ University of Tennessee, Knoxville, TN, USA
¶ ENS Lyon, UCB Lyon, CNRS, Inria LIP, F-69342, LYON Cedex 07, France

RESEARCH CENTRE
Centre Inria de Lyon

Bâtiment CEI-2, Campus La Doua
56, Boulevard Niels Bohr - CS 52132
69603 Villeurbanne

Tolérance aux pannes pour les algorithmes numériques itératifs

Résumé : Ce travail s'intéresse à la protection des algorithmes numériques itératifs contre tous les types d'erreurs qui peuvent survenir lors d'une exécution sur plateforme à grande échelle.

Mots-clés : Algorithmes numériques itératifs, tolérance aux pannes, Gradient Conjugué Préconditionné.

Contents

1 Introduction

Large-scale simulation is one of the main tools of modern science. To achieve new scientific challenges, simulations are processed at larger and larger scale, on platforms that comprise a multitude of computational units, and execute during long periods of time, up to several days. As a consequence, High Performance Computing (HPC) experiments are increasingly exposed to errors [14].

Errors are usually classified in two main categories: fail-stop errors, that are immediately detected, and silent data corruptions, that can be detected through some verification mechanisms [25]. Fail-stop errors correspond to permanent failures, e.g., processor crashes. Silent errors are disruptions that strike and stay undetected until they manifest eventually through strange application behavior. Silent errors arise from two main sources: computation errors and memory bit-flips. Protecting algorithms and software libraries from all these errors is a major concern within the HPC community. Although there have been many contributions in that direction (e.g. see the survey [1]), the design of a resilient holistic methodology to protect HPC applications from all error types is far from having been achieved. In particular, most studies are specifically restricted to only one type of errors, either fail-stop or silent.

This work aims at studying how to protect numerical iterative algorithms from all types of errors that can strike during execution, may they be either fail-stop or silent errors. We rely on system-level and numerical state-of-the-art techniques designed for each error type, and we show how they can be combined efficiently at scale. Our comprehensive approach mitigates the impact of all these errors and aims at guaranteeing a correct execution, regardless of the number and nature of errors that will strike. We combine various techniques: detectors for computation errors, checksums for memory errors, and checkpoint/restart for fail-stop errors. We stress that our approach is agnostic of the type, nature and cost of all these techniques, as well as of the details of the target iterative algorithm. In essence, after each iteration of the algorithm, we decide whether or not it is worth to insert some fault-tolerant mechanisms, and if yes, which ones.

More precisely, we design and evaluate an execution *pattern* that repeats periodically, and within which each technique is applied with a specific frequency. In the scenarios that we explore for evaluation (see Section 6), computation errors strike more frequently than memory errors, which in turn strike more frequently than fail-stop errors. This motivates the following hierarchical shape for the periodic pattern:

• *Chunk*: after executing a certain number of iterations, we perform a verification for computation errors;

• *Segment*: after several chunks, we take a memory checkpoint to be able to recover from a memory error;

• *Pattern*: after several segments, we take a (costly) checkpoint on stable storage in order to be able to recover from fail-stop errors.

The size of the chunks, the number of chunks per segment, and the number of segments in the pattern are key parameters of our approach. All these parameters are to be determined according to the duration of each iteration of the algorithm, the frequency of each error type, and the cost of each fault-tolerance mechanism.

The first main contribution of this work is to introduce such a periodic execution pattern capable of coping with all error sources that can strike the application simultaneously; the second main contribution is to provide a performance analysis that enables to characterize the optimal pattern as a function of platform, application and resilience parameters. The optimal pattern is the one that minimizes the expected time per iteration, when both failure-free overhead (cost of resilience mechanisms) and failure-induced overhead (time lost by downtime, recovery and re-

execution) are taken into account. The interplay between fail-stop and silent errors dramatically complicates the derivation of the expected time per iteration, which characterizes the progress of the application.

While the design and analysis methodology of the execution pattern is agnostic of the target iterative algorithm, we focus on the Preconditioned Conjugate Gradient (PCG) algorithm for the practical evaluation. PCG is one of the most widely used algorithms for the solution of large problems that appear in many numerical simulations (ocean, atmosphere [35], incompressible fluid flows [18], plasma physics [11], to name a few). In addition, PCG is used for the TOP500 HPCG benchmark [48]. We evaluate execution patterns for several realistic scenarios of the deployment of PCG at scale, in order to assess the impact of each parameter. Our in-house toolkit is publicly available for the sake of reproducibility of the results, and to enable further tests and assessments with different execution scenarios.

We stress that this work provides neither actual experiments on a large-scale platform, nor in-house Monte Carlo simulations with error traces (or artificial error injection methods). Instead, we make a toolkit available for the community to study the impact of errors on the performance of numerical iterative algorithms at scale. Our study lays the foundations for deciding which resilience mechanism to deploy, given its cost, the frequency of the corresponding error type, and the detailed characteristics of the target algorithm.

This paper is organized as follows. Section 2 surveys related work. Section 3 outlines the generic framework, with a short presentation of each error source, together with their mitigation techniques. Section 4 is devoted to the design and theoretical analysis of the optimal pattern, which constitutes the heart of our algorithmic contribution. Section 5 details the nature and cost of each fault-tolerant mechanism for PCG. Section 6 presents the evaluation framework and reports the results obtained from an extensive set of scenarios. Finally, Section 7 provides concluding remarks and hints for future work.

2 Related work

We survey related work on fault-tolerance in Section 2.1. Section 2.2, discusses several papers that focus on our case study, namely the PCG algorithm. We conclude with a brief summary in Section 2.3.

2.1 Fault-tolerance techniques

Fault-tolerant approaches cover a wide spectrum of techniques, depending upon the type of errors that are addressed and mitigated. For fail-stop errors, checkpoint-restart is de-facto standard strategy. 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). Several variants of this policy have been studied; coordinated, hierarchical, multi-level, and in-memory, to name a few; see [25] for an overview. A natural strategy is to checkpoint periodically, and one must decide how often to checkpoint, i.e., derive the optimal checkpointing period. An optimal strategy is defined as a strategy that minimizes the expectation of the execution time of the application. For a preemptible application, where one can checkpoint at any time, the classical formula due to Young [49] and Daly [19] states that the optimal checkpointing period is $\sqrt{2\mu C}$, where μ is the application Mean Time Between Failures (MTBF) and C the checkpoint cost.

While fail-stop errors lead to fatal interruptions (such as processor crashes) and cause the loss of the entire memory of the processor, silent errors, a.k.a. silent data corruptions, 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. We classify silent errors into two categories: computation errors, which are transient and caused by arithmetic errors in the Arithmetic and Logic Unit (ALU), and memory errors, which are persistent and correspond to bit flips in memory. These bit flips can strike in the dynamic random-access memory (DRAM) due to cosmic radiation, overheat and other sources, but also in the L1 cache which is usually not well protected, or in the L2 cache which might be protected by one parity bit [39, 40, 51, 50].

There are several software and hardware 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 are also often a weak link in the chain of protection, making all data transfers at higher risk. In addition, the constant effort to reduce operational voltage to save energy 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 [47, 6]. In a nutshell, silent errors have become a major threat due to the increase in problem size [45]: 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.

A 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. Because checkpoint and rollback recovery assumes instantaneous error detection, 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. Some verification mechanisms, or detectors, must hence be enforced to guarantee that chekpoints are not corrupted. As stated above, 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 (originally known as triple modular redundancy and voting [34]) can be enforced, which allows for error correction in addition to error detection, using a simple majority vote.

Application-specific information can be useful to enable ad-hoc solutions, which dramatically decreases the cost of detection. Many techniques have been advocated. They include memory scrubbing [29] and Algorithm-Based Fault Tolerance (ABFT) techniques [28, 12, 43], such as coding for the sparse-matrix vector multiplication kernel [43], blockwise checksum calculation for error-bounded lossy compressor [32], and coupling a higher-order with a lower-order scheme for Partial Differential Equations (PDE) [10]. These methods can only detect an error but do not correct it.

2.2 Protecting the PCG algorithm

In this section, we focus on related work on the protection of the Preconditioned Conjugate Gradient (PCG) algorithm from silent errors. We refer to [1] for a more extensive survey on the resilience of numerical algorithms. The main idea from the literature is to design detection mechanisms based upon intrinsic properties of the PCG algorithm [15, 26, 13]. A first criterion is to check the orthogonality of some vectors; in finite precision arithmetic, the scalar product may not be computed exactly, hence a threshold has been introduced [16]. Another widely used criterion is to recompute the residual [16, 2]. Other properties from the matrix [2] or the preconditioner [36] have been introduced to reduce the cost of the detection and improve its recall. Many of these works do not discuss correction techniques after having detected an error; checkpoint and rollback is used in [16, 2]. The work [22] goes beyond and optimizes the ratio of the detection frequency and the checkpoint frequency in order to minimize execution time. All these techniques rely completely on the detection of error by the program. A completely different approach [42] advocates no rollbacks, but instead it aims at self-stabilizing the algorithm by recomputing the residual at regular intervals; however, this could impact the convergence rate in a non predictable way.

2.3 Synthesis

To the best of our knowledge, this work is the first attempt to protect numerical iterative applications from all types of errors: fail-stop, computation and memory. Previous works focused either on an optimal pattern independent from a particular algorithm [8, 9], or they were limited to the detection and/or correction of transient errors in PCG [36, 2, 22, 42, 16].

3 Framework

This section details the fault-tolerance framework that we envision for numerical iterative algorithms. We start with error sources and mitigation techniques in Section 3.1. Error probability distributions are specified in Section 3.2. Then in Section 3.3, we describe the periodic hierarchical pattern that is the key building block of our holistic approach. All the notations introduced in this section are summarized in Table 1.

3.1 Error sources and mitigation techniques

As discussed earlier, we aim at protecting numerical iterative algorithms from three error categories: fail-stop errors, and two types of silent errors, namely computation and memory. The techniques described in this general framework are agnostic of the specific target algorithm. We refer to Section 5 for a case study with the Preconditioned Conjugate Gradient (PCG) algorithm.

For fail-stop errors, we use checkpoint-restart, with a global checkpoint that saves the entire footprint of the algorithm on stable storage. This checkpoint includes both static and dynamic data. Typically, the matrix and the preconditioner form the static data, while all the vectors updated at each iteration correspond to the dynamic data. We let C_{fs} denote this global checkpoint, and we also use C_{fs} for its associated cost. Similarly, R_{fs} denote the recovery (cost) from a C_{fs} checkpoint.

For memory errors (e.g., bit flips), we recompute checksums to verify that the memory has not been corrupted. The cost of this memory verification is V_m. If the verification is successful, we take a local in-memory checkpoint of cost C_{cm}. The V_m verification detects any memory error, both for static or dynamic data, while the C_{cm} checkpoint stores only the dynamic data

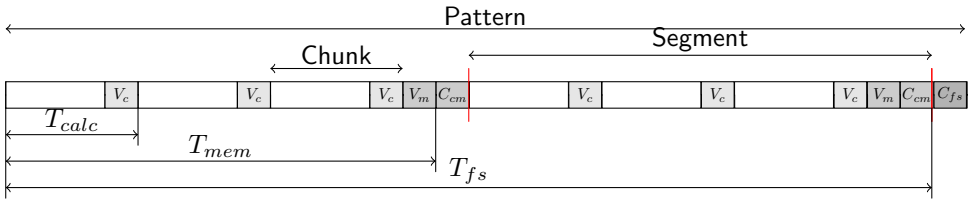

Figure 1: Example of a pattern with $n_{cm} = 3$ and $n_{fs} = 2$.

(updated vectors). This is in contrast to the global checkpoint C_{fs}, whose cost is expected much higher than that of C_{cm}. We let R_{cm} denote the recovery from a C_{cm} checkpoint.

For computation errors (e.g., incorrect arithmetic operations), we use an application-specific computation verification V_c. In addition to detecting computation errors, the V_c verification is likely to detect some (but not all) memory errors. We expect the cost of V_c to be much smaller than the cost of V_m.

In summary, the V_c verification detects computation errors, the V_m verification detects memory errors, the C_{cm} checkpoint stores only the dynamic data (vectors) in memory, and the C_{fs} checkpoint stores all data (vectors, matrix and preconditioner) on stable storage. For simplicity, we make the following assumptions:

1. fail-stop errors can strike anytime except during C_{fs} and R_{fs};
2. silent errors can strike neither during verifications V_c and V_m, nor during checkpoints C_{cm} and C_{fs}, nor during recoveries R_{cm} and R_{fs};
3. verifications V_c and V_m have perfect recall and precision 1, which means that we have neither false negative (undetected errors) nor false positive (false alarms).

These three assumptions are commonly used in the literature [25]. The third one is critical to the correctness of the approach: we always perform V_c and V_m verifications before taking a local C_{cm} checkpoint, to ensure its validity. Similarly, we only take a global C_{fs} checkpoint after a V_c, V_m, C_{cm} sequence for the same reason.

We acknowledge that assuming perfect verifications may be unrealistic in some practical settings. However, in this work, we aim at avoiding a prohibitively costly approach based on full redundancy via replication, or even triplication. Using perfect application-specific detectors is the state-of-the-art technique to guarantee that checkpoints are not corrupted [5, 21].

3.2 Probability distributions

The probability distribution of inter-arrival times for memory and fail-stop errors both obey an exponential (memoryless) distribution [25]. Let λ_{mem} and λ_{fs} be the parameter for respectively memory errors and fail-stop errors. Hence, $e^{\lambda_{mem}t}$ and $e^{\lambda_{fs}t}$ are, respectively, the memory and fail-stop probabilities of successful execution during a duration t (i.e., no error of the specified type strikes during time t). The computation errors follow a geometric distribution of parameter f_{calc}, which is the success probability per iteration (i.e., no computation error strikes during the iteration). We need a discrete probability law for computation errors because they are counted per iteration, but it remains very close to an exponential distribution of parameter λ_{calc}, with $f_{calc} = e^{-\lambda_{calc}I}$, and where I is the duration of an iteration. All three error probability distributions are considered as mutually independent.

Symbol	Definition	Symbol	Definition
I	One iteration	n_{vc}	Number of iterations in a chunk
V_c	Computation verification	n_{cm}	Number of chunks in a segment
V_m	Memory verification	n_{fs}	Number of segments in a pattern
C_{cm}	Local memory checkpoint (vectors of the algorithm)	T_{calc}	$= n_{vc}I + V_c$
C_{fs}	Checkpoint (in case of a fail-stop error)	T_{mem}	$= n_{cm}(n_{vc}I + V_c) + V_m$
R_{cm}	Local recovery process from C_{cm} checkpoint	T_{fs}	$= n_{fs}(n_{cm}(n_{vc}I + V_c) + V_m + C_{cm})$
R_{fs}	Recovery process from C_{fs} checkpoint	W	$= n_{fs}n_{cm}n_{vc}I$
P^z_{succ}	Probability of no error of type z	f_{calc}	Parameter of geometric distrib. for computation errors
P^z_{fail}	$= 1 - P^z_{succ}$	λ_{mem}	Parameter of exponential distrib. for memory errors
$P^{calc}_{fail}(i)$	Proba. that first computation error is in chunk i of segment	λ_{fs}	Parameter of exponential distrib. for fail-stop errors
P_{no_fs}	$1 - P_{no\_fs}$: the first error to strike is a fail-stop error		

Table 1: Table of notations.

3.3 Hierarchical pattern

We aim at creating a flexible and holistic pattern where verifications and checkpoints are interleaved after some iterations, and which is repeated periodically during execution. The frequency of each resilience mechanism within the pattern will depend upon many parameters: the probability of each error source, the cost of verifications and checkpoints, and the time spent per iteration.

The key hypothesis that drives the design of the pattern is that computation errors are more frequent than memory errors, which in turn are more frequent than fail-stop errors. This calls for a hierarchical pattern where computation verifications V_c are inserted more frequently than memory verifications V_m and local checkpoints C_{cm}, which in turn are inserted more frequently than global checkpoints C_{fs}.

The periodic pattern begins with $n_{vc} \geq 1$ iterations, each taking I time units. After these n_{vc} iterations, we take a V_c verification. This constitutes a *chunk*, of duration $T_{calc} = n_{vc}I + V_c$. Then, we repeat $n_{cm} \geq 1$ chunks before executing a V_m verification followed by a C_{cm} checkpoint. This constitutes a *segment*, of duration $T_{mem} + C_{cm}$, where $T_{mem} = n_{cm}(n_{vc}I + V_c) + V_m$. We repeat $n_{fs} \geq 1$ segments before terminating the pattern with a C_{fs} checkpoint. As illustrated in Figure 1, this describes the full pattern, of total length $T_{fs} + C_{fs}$, where $T_{fs} = n_{fs}(n_{cm}(n_{vc}I + V_c) + V_m + C_{cm})$. There are $n_{vc} \times n_{cm} \times n_{fs}$ iterations executed within the pattern.

3.4 Objective function

The objective is to minimize the slowdown, which is defined as the ratio of the expected time needed per iteration over the time per iteration in a failure-free execution, with no fault-tolerance mechanism. Given a pattern with parameters (n_{vc}, n_{cm}, n_{fs}), let $E(n_{vc}, n_{cm}, n_{fs})$ be the expected time to execute the pattern successfully, from its beginning up to the last C_{fs} checkpoint. We aim at minimizing the slowdown $\mathcal{S}(n_{vc}, n_{cm}, n_{fs})$, where

$$\mathcal{S}(n_{vc}, n_{cm}, n_{fs}) = \frac{E(n_{vc}, n_{cm}, n_{fs})}{n_{fs}n_{cm}n_{vc}I}.$$

We show how to derive a closed-form analytical formula for $E(n_{vc}, n_{cm}, n_{fs})$ in the next section.

4 Optimal pattern

In this section, we first detail the derivation of $E(n_{vc}, n_{cm}, n_{fs})$ in Section 4.1. Then, we explain how to derive the optimal pattern in Section 4.2.

4.1 Formula for $E(n_{vc}, n_{cm}, n_{fs})$

For simplicity, we write E for $E(n_{vc}, n_{cm}, n_{fs})$ in this section. The main difficulty to compute E is that we cannot rely on previous approaches when combining all different error sources. For fail-stop errors, we could try and dimension the pattern so that the time to execute the useful work W would correspond to the Young/Daly period [49, 19]; however, whenever a silent error strikes within the period, we need to rollback and re-execute some work (depending upon whether it is a computational or memory error), thereby leading to an actual execution time that may well largely exceed the failure-free work W. Furthermore, different error types induce different overheads in their recovery process.

An important observation is that the expected time to execute a given segment depends upon the position of that segment within the pattern. Therefore, we break down the expected time E for the whole pattern into

$$E = \sum_{k=1}^{n_{fs}} E_k + C_{fs},$$

where E_k is the mean time to complete segment k (i.e., to successfully execute its memory checkpoint). If a silent error is detected within segment k, then only this segment is restarted, while if a fail-stop error is detected, then all previous $k-1$ segments have to be completed again before the current segment k is restarted. Now, the formula for E_k is obtained by a case analysis, depending upon whether there is no error, or which error source is the first to be detected during execution. We start with some notations:

- $P_{succ}^{fs} = e^{-\lambda_{fs}(T_{mem}+C_{cm})}$ is the probability of no fail-stop error within the segment;
- $P_{succ}^{mem} = e^{-\lambda_{fs}T_{mem}}$ is the probability of no memory error within the segment;
- $P_{succ}^{calc}(i) = f_{calc}^{n_{vc}i}$ is the probability of no computation error during the first i chunks (hence during a total of $n_{vc}i$ iterations), and $P_{fail}^{calc}(i) = P_{succ}^{calc}(i-1)(1-f_{calc}^{n_{vc}})$ is the probability that the first computation error strikes during chunk i;
- Finally, $1 - P_{no\_fs}$ is the probability that the first error in the segment is a fail-stop error. Its complicated value is determined below.

We can now write down the overall formula for E_k:

$$E_k = P_{succ}^{fs} P_{succ}^{mem} P_{succ}^{calc}(n_{cm})[T_{mem} + C_{cm}] \quad \text{(Case 1)}$$

$$+ (1-P_{succ}^{mem}) P_{succ}^{fs} P_{succ}^{calc}(n_{cm})[T_{mem}+R_{cm}+E_k] \quad \text{(Case 2)}$$

$$+ \sum_{i=1}^{n_{cm}} e^{-\lambda_{fs} i T_{calc}} P_{fail}^{calc}(i)[iT_{calc}+R_{cm}+E_k] \quad \text{(Case 3)}$$

$$+ (1 - P_{no\_fs})[E_{lost} + R_{fs} + \sum_{i=0}^{k} E_i] \quad \text{(Case 4)}$$

Case 1 is when no error struck during the computation of the whole segment. It is weighted by the probability of success for all error types. Note that $P_{succ}^{calc}(n_{cm})$ is the probability of having no computation error throughout the segment.

Case 2 is when the first detected error in the segment is a memory error. This implies that no fail-stop nor computation error took place before the end of the memory verification at the end of the segment before the local in-memory checkpoint, hence the weight by the corresponding probabilities.

Case 3 is when the first detected error is a computation error. This error type can strike only during iterations. The lost time induced by this error depends upon the chunk where it

struck. If it is the i-th chunk of the segment, the lost time includes all the first i chunks, hence a time iT_{calc}. Note that in this situation, no fail-stop error has struck before the computation error, while a memory error may happen anytime before the computation error and it will not be detected. Again, we weight this case with its probability to happen.

Case 4 is when the first error to strike is a fail-stop error, which is immediately detected. When such an error strikes, the re-execution starts at the very beginning of the pattern, so all the previously completed segments have to be recomputed. Here, E_{lost} is the expected lost time during the segment, whose value is

$$\begin{aligned} E_{lost} &= \int_0^\infty x P(X = x | X < T_{mem} + C_{cm}) dx \\ &= \frac{1}{1 - \mathrm{e}^{-\lambda_{fs}(T_{mem}+C_{cm})}} \int_0^{T_{mem}+C_{cm}} x \lambda_{fs} \mathrm{e}^{-\lambda_{fs} x} \, dx \\ &= \frac{1}{\lambda_{fs}} - \frac{T_{mem} + C_{cm}}{\mathrm{e}^{\lambda_{fs}(T_{mem}+C_{cm})} - 1}, \end{aligned}$$

where X is an exponential random variable of parameter λ_{fs} (see [25] for details).

There remains to evaluate P_{no_fs}, the probability that the first error to be spotted is not a fail-stop error. But by definition, the value of P_{no_fs} is the sum of the probability weights of the first three cases, hence

$$\begin{aligned} P_{no\_fs} &= P_{succ}^{fs} P_{succ}^{mem} P_{succ}^{calc}(n_{cm}) \\ &\quad + (1 - P_{succ}^{mem}) P_{succ}^{fs} P_{succ}^{calc}(n_{cm}) \\ &\quad + \sum_{i=1}^{n_{cm}} e^{-\lambda_{fs} i T_{calc}} P_{fail}^{calc}(i) \\ &= \mathrm{e}^{-T_{mem}(\lambda_{fs}+\lambda_{mem}) - C_{cm}\lambda_{fs}} (f_{calc})^{n_{vc} n_{cm}} \\ &\quad + (1 - \mathrm{e}^{-\lambda_{mem} T_{mem}}) \, \mathrm{e}^{-\lambda_{fs} T_{mem}} (f_{calc})^{n_{vc} n_{cm}} \\ &\quad + (1 - f_{calc}^{n_{vc}}) \, \mathrm{e}^{-\lambda_{fs} T_{calc}} \left(\frac{(f_{calc}^{n_{vc}} \, \mathrm{e}^{-\lambda_{fs} T_{calc}})^{n_{cm}} - 1}{f_{calc}^{n_{vc}} \, \mathrm{e}^{-\lambda_{fs} T_{calc}} - 1} \right). \end{aligned}$$

Altogether, after some re-ordering, and double-checking the derivation with a computer algebra system, we are led to the following recurrence equation:

$$\begin{aligned} E_k = M &+ (1 - P_{no\_fs}) \textstyle\sum_{i=1}^{k-1} E_i \\ &+ \left(1 - P_{succ}^{fs} P_{succ}^{mem} P_{succ}^{calc}(n_{cm})\right) E_k, \end{aligned}$$

whose complexity is hidden into the value of M, which is the part of the formula that does not

$$
\begin{aligned}
M &= \mathrm{e}^{-T_{mem}(\lambda_{fs}+\lambda_{mem})-C_{cm}\lambda_{fs}}(f_{calc})^{n_{vc}n_{cm}}[T_{mem}+C_{cm}] \\
&+(1-\mathrm{e}^{-\lambda_{mem}T_{mem}})\,\mathrm{e}^{-\lambda_{fs}T_{mem})}(f_{calc})^{n_{vc}n_{cm}}[n_{cm}(n_{vc}I+V_c)+V_m+R_{cm}] \\
&+(1-f_{calc}^{n_{vc}})\,\mathrm{e}^{-\lambda_{fs}T_{calc}}\left(\frac{(f_{calc}^{n_{vc}}\,\mathrm{e}^{-\lambda_{fs}T_{calc}})^{n_{cm}}-1}{f_{calc}^{n_{vc}}\,\mathrm{e}^{-\lambda_{fs}T_{calc}}-1}\right)[R_{cm}] \\
&+(1-f_{calc}^{n_{vc}})\mathrm{e}^{-\lambda_{fs}T_{calc}}\left(\frac{(f_{calc}^{n_{vc}}\,\mathrm{e}^{-\lambda_{fs}T_{calc}})^{n_{cm}}(n_{cm}(f_{calc}^{n_{vc}}\,\mathrm{e}^{-\lambda_{fs}T_{calc}}-1)-1)+1}{(f_{calc}^{n_{vc}}\mathrm{e}^{-\lambda_{fs}T_{calc}}-1)^2}\right)T_{calc}
\end{aligned}
$$

depend upon k:

$$
\begin{aligned}
&+\left[1-\left[\mathrm{e}^{-(n_{cm}(n_{vc}I+V_c)+V_m)(\lambda_{fs}+\lambda_{mem})-C_{cm}\lambda_{fs}}(f_{calc})^{n_{vc}n_{cm}}\right.\right. \\
&\quad+(1-\mathrm{e}^{-\lambda_{mem}T_{mem}})\,\mathrm{e}^{-\lambda_{fs}T_{mem}}(f_{calc})^{n_{vc}n_{cm}} \\
&\quad\left.\left.+(1-f_{calc}^{n_{vc}})\,\mathrm{e}^{-\lambda_{fs}T_{calc}}\left(\frac{(f_{calc}^{n_{vc}}\,\mathrm{e}^{-\lambda_{fs}T_{calc}})^{n_{cm}}-1}{f_{calc}^{n_{vc}}\,\mathrm{e}^{-\lambda_{fs}T_{calc}}-1}\right)\right]\right] \\
&\quad\times\left[\frac{1}{\lambda_{fs}}-\frac{n_{cm}(n_{vc}I+V_c)+V_m+C_{cm}}{\mathrm{e}^{\lambda_{fs}(T_{mem}+C_{cm})}-1}+R_{fs}\right].
\end{aligned}
$$

By induction on k, we derive a compact form for E_k:

Lemma 1.

$$
E_k = \frac{M}{P_{succ}^{fs}P_{succ}^{mem}P_{succ}^{calc}(n_{cm})}\left(\frac{1-P_{no\_fs}}{P_{succ}^{fs}P_{succ}^{mem}P_{succ}^{calc}(n_{cm})}+1\right)^k \tag{4.1.1}
$$

Proof. The proof of Equation (4.1.1) is done by induction on k. Let us denote $P_{succ}^{fs,mem,calc} = P_{succ}^{fs}P_{succ}^{mem}P_{succ}^{calc}(n_{cm})$. Initially,

$$
\begin{aligned}
E_0 &= \frac{M}{P_{succ}^{fs,mem,calc}} \\
&= \frac{M}{P_{succ}^{fs,mem,calc}}\left(\frac{1-P_{no\_fs}}{P_{succ}^{fs,mem,calc}}+1\right)^0.
\end{aligned}
$$

Now, assume that the formula is true for all $i \in 0, 1, \ldots, k-1$. Then,

$$
\begin{aligned}
E_k &= \frac{M}{P_{succ}^{fs,mem,calc}} \\
&\quad+\frac{(1-P_{no\_fs})}{P_{succ}^{fs,mem,calc}}\sum_{i=0}^{k-1}\frac{M}{P_{succ}^{fs,mem,calc}}\left(\frac{1-P_{no\_fs}}{P_{succ}^{fs,mem,calc}}+1\right)^i \\
&= \frac{M}{P_{succ}^{fs,mem,calc}} \\
&\quad\times\left(1+\frac{(1-P_{no\_fs})}{P_{succ}^{fs,mem,calc}}\frac{\left(\frac{1-P_{no\_fs}}{P_{succ}^{fs,mem,calc}}+1\right)^k-1}{\frac{1-P_{no\_fs}}{P_{succ}^{fs,mem,calc}}+1-1}\right) \\
&= \frac{M}{P_{succ}^{fs,mem,calc}}\left(\frac{1-P_{no\_fs}}{P_{succ}^{fs,mem,calc}}+1\right)^k,
\end{aligned}
$$

which corresponds to Equation (4.1.1). □ □

Equation (4.1.1) is the last step before the final derivation of $E = \sum_{k=1}^{n_{fs}} E_k + C_{fs}$. We finally obtain:

$$
E = \frac{M}{1-P_{no\_fs}}\left[\left(\frac{1-P_{no\_fs}}{P_{succ}^{fs}P_{succ}^{mem}P_{succ}^{calc}(n_{cm})}+1\right)^{n_{fs}}-1\right]+C_{fs}.
$$

4.2 Optimal pattern

The formula for E is finally determined! We aim at minimizing the slowdown $\mathcal{S} = \frac{E}{n_{fs} n_{cm} n_{vc} I}$ over all possible pattern parameters n_{vc}, n_{cm} and n_{fs}. Given the complexity of the formula, it is impossible to optimize $\mathcal{S}$ analytically. However, when given all the other platform/application parameters (the duration of an iteration, the distribution of the three error types and the cost of all resilience mechanisms), it is easy to evaluate $\mathcal{S}$ numerically and to determine its minimum via an exhaustive search over the three pattern parameters. It takes only a few milliseconds to sweep over 10^7 values, letting (conservatively) n_{vc} range from 1 to 1000, and n_{cm} and n_{fs} range from 1 to 100.

It is then interesting to compare the expected cost of the optimal pattern over that of the naive approach where both verifications and checkpoints are taken after each iteration, and which corresponds to the simplest pattern with $n_{vc} = n_{cm} = n_{fs} = 1$. This naive approach is often the first approach used to design a fault-tolerant algorithm [36].

5 Case study: PCG

This section focuses on the Preconditioned Conjugate Gradient (PCG) algorithm as a case-study. First, we give a quick background on the algorithm in Section 5.1. Then, application-specific detection and correction techniques for PCG are presented in Section 5.2.

5.1 PCG algorithm

Algorithm 1: PCG algorithm

```
1  r_0 ← b − Ax_0
2  z_0 ← M^{-1} r_0
3  p_0 ← z_0
4  γ_0 ← r_0^T z_0
5  for i = 0 ... until convergence do
6      q_i ← Ap_i
7      α_i ← γ_i / p_i^T q_i
8      x_{i+1} ← x_i + α_i p_i
9      r_{i+1} ← r_i − α_i q_i
10     z_{i+1} ← M^{-1} r_{i+1}
11     γ_{i+1} ← r_{i+1}^T z_{i+1}
12     β_{i+1} ← γ_{i+1} / γ_i
13     p_{i+1} ← z_{i+1} + β_i p_i
```

The PCG algorithm is the iterative method of choice for solving a linear system $Ax = b$, with A a real symmetric positive definite matrix, and b and x real vectors. At iteration i, the algorithm uses the computed residual r_i, the direction p_i and a coefficient α_i to move x_i closer to the solution x of the system [23]. The preconditioner matrix M aims at making convergence faster. z_i is an additional vector related to the use of the preconditioner. The PCG algorithm is depicted in Algorithm 1. γ_i, α_i, β_i are scalars. The initialization of the algorithm is on lines [1,4]. The initial guess x_0 is chosen arbitrarily. The first line computes the first residual r_0. Then, the iteration is on lines [6,13]. At each iteration, the algorithm adds the approximate part

of the solution depending on the direction given by the previous iteration, to x_{i+1}. Finally, the new direction p_{i+1} is estimated and conjugated to all previous directions.

5.2 Error mitigation techniques for PCG

In exact arithmetic, the PCG algorithm is a sophisticated and elegant numerical scheme that has many properties induced by the symmetric positive definiteness of the matrix A. Unfortunately, most characteristic properties are no longer valid in finite precision calculation, such as orthogonality among the residuals r_i (i.e., $r_i^T r_j = \delta_{i,j}$), A-conjugacy of the descent direction p_i (i.e., $p_i^T A p_j = \delta_{i,j}$), or equality between the computed residual r_i and the true residual $b - Ax_i$ (i.e., $r_i = b - Ax_i$). Unfortunately, it is known that these equalities are no longer valid in finite precision arithmetic. Consequently, they cannot be used to detect silent errors. It is argued and shown through extensive experiments [2] that a robust detection mechanism can be designed by combining a check on a (first) bound on the residual gap, together with a (second) bound on α. The first bound, on the so-called residual gap, defined by $\|r_i - (b - Ax_i)\|$, is derived using classical rounding error analysis techniques. This computation is the core of the additional cost associated with numerical verification, requiring an additional matrix-vector product, vector addition (axpy), and norm computation. The second check, among the numerous relationships that exist between the quantities computed by PCG, is one that is possibly less prone to defection due to finite precision calculation. Indeed it is a bound and not a strict equality as for the orthogonality or A-orthogonality properties. It was already presented in the original paper on CG [27, Th. 5.5]:

$$\forall i \quad \frac{1}{\lambda_{\max}(A)} < \alpha_i < \frac{1}{\lambda_{\min}(A)}, \tag{5.2.1}$$

where $\lambda_{\max}(A)$ and $\lambda_{\min}(A)$ denote the largest and smallest eigenvalues of A, respectively. It was shown [2] that checking only the lower bound $\frac{1}{\lambda_{\max}(A)} < \alpha_i$ is sufficient to make the overall detection robust. This is important from a practical point of view as the calculation or the tight approximation of $\lambda_{\min}(A)$ is generally expensive while $\lambda_{\max(A)}$ can often be cheaply approximated, using for instance randomized techniques [24]. Once the approximation of $\lambda_{\max(A)}$ has been obtained (in a preliminary setup phase), the check itself $\frac{1}{\lambda_{\max}(A)} < \alpha_i$ (triggered at iteration i) comes at cost zero as α_i is a by-product of PCG. In the following, we assume the numerical verification is ensured with such a check combining a verification on the residual gap bound (at the cost of a matrix-vector product, an axpy and a norm computation) and alpha verification (at cost zero). We furthermore assume that the memory checkpoint stores only the vectors, while the fail-stop checkpoint stores the matrix and the preconditioner in addition to the vectors.

6 Evaluation

This section aims at evaluating the efficiency of the approach on a variety of realistic scenarios. We first recall the key input parameters in Section 6.1. Then, we discuss scenarios that instantiate these parameters for PCG in Section 6.2, with the detail on the memory complexity of the preconditioners. Section 6.3 deals with error sources. Finally, the results and analyses of the scenarios are provided in Section 6.4.

6.1 Input parameters

The formula for the slowdown $\mathcal{S}(n_{vc}, n_{cm}, n_{fs})$ needs the values of several parameters:

- Time I spent per iteration;

- Cost of resilience parameters: V_c and V_m for computation and memory verifications, C_{cm} and R_{cm} for local in-memory checkpoint and recovery, and C_{fs} and R_{fs} for global checkpoint and recovery;
- Frequency of each error source: f_{calc} for the geometric distribution of computation errors, and λ_{mem} and λ_{fs} for the exponential distributions of memory and fail-stop errors.

6.2 Scenarios for PCG

We consider the solution of linear systems, $Ax = b$, arising from the discretization of 3D elliptic PDE, which appear in the modeling of many complex systems, such as for atmosphere or ocean simulation [35], incompressible fluid flows [18], plasma physics [11], to name a few. For these large dimension systems, iterative solvers are the only possible solvers and PCG is the best suited one. To speed up its convergence, preconditioning techniques are used. For linear systems arising from 3D PDE discretization, multigrid approaches are probably the best option for sufficiently smooth problems because of their linear complexity of required computation and memory with respect to the problem size. Multigrid preconditioning is used in the TOP500 HPCG benchmark [48]. In the case of stiff problems, alternative preconditioning techniques that may be employed include domain decomposition methods (DDM). It should be noted that these preconditioning techniques may require greater computational and memory resources to ensure further numerical robustness, especially when using a coarse grid correction as we consider here [46, 3].

We consider both these multigrid and DDM classes of preconditioners. We first derive their memory costs on an analytical problem. We then consider large-scale real-life experiments from the literature to derive realistic timing parameters. Finally, we conduct the evaluation based on these parameters.

Assuming a 7-point stencil discretization of the PDE using a uniform Cartesian grid in the unit cube of side m, the square matrix A is of order $N = m^3$, sparse, and it contains $7N$ non-zero elements. Assuming a COOrdinates (COO) storage, it is composed of three vectors: row indices, column indices and values, inducing a storage of $7N$ real numbers plus $14N$ integers, i.e., $21 \times 8 \times N$ bytes in 64 bits integer and real arithmetic. In addition, there are six vectors (x, b, r, p, q, z) in Algorithm 1, each of size N, requiring $6 \times 8 \times N$ bytes. Hence, PCG uses at least $(21 + 6) \times 8 \times N$ bytes of memory. Some additional memory is required to store the preconditioner, as detailed below.

Memory complexity of the preconditioners

Memory complexity of the multigrid preconditioner.

The geometric multigrid algorithm is based on a series of nested 3D grids where only every second grid point is retained in each direction from one grid level (ℓ) to the next ($\ell + 1$). This means that the number of unknowns from one grid to the next is divided by eight, so that asymptotically the number of unknowns on all grids is $\sum_{\ell=0}^{\infty} N/8^{\ell} = 8/7N$. This means that the three vectors (current iterate, residual, correction) that need to be stored on each grid level have a total memory requirement of $24/7 \times 8 \times N$ bytes (assuming double precision real coefficients). In addition, the restriction matrices that allow the vectors defined on one grid to be moved to the next have to be stored. If we consider a fully weighted restriction, the corresponding matrix is rectangular with 8 times less rows than columns, with at most 27 non-zero coefficients per row. On the grid hierarchy this is $\sum_{\ell=1}^{\infty} 27 \times N/8^{\ell} = 27/7 \times N \approx 4 \times N$ coefficients, which translates to $\approx 3 \times 4 \times 8$ bytes if these matrices are stored in COO format (assuming double precision real coefficients and 64 bits integers). The prolongation is the transpose of the constraint and does not need to be stored. Finally, the discretization matrices are computed using a Galerkin approach

at each grid level, i.e., $A_\ell = R_\ell A_{\ell-1} R_\ell^T$ where R_ℓ is the restriction from level $\ell - 1$ to level ℓ. The matrix A_ℓ essentially corresponds to a 27-point grid, so the number of coefficients on the sequence of grid levels is $\sum_{\ell=1}^{\infty} 27 \times N/8^\ell = 27/7N \approx 4 \times N$. Stored in COO format, the memory footprint is $3 \times 4 \times 8 \times N$ bytes. The total memory footprint is $(24/7 + 2 \times 3 \times 4) \times 8 \times N \approx 224 \times N$.

Memory complexity of the domain decomposition preconditioner.

For the domain decomposition preconditioner, we consider a two-stage Schwarz method based on the GENEO approach [46, 3]. Formally, the preconditioner writes:

$$M = \sum_{i=1}^{P} R_i^T (R_i A R_i^T)^{-1} R_i + U_0 (U_0^T A U_0)^{-1} U_0^T,$$

where P denotes the number of subdomains, R_i is the canonical restriction that maps the entries of a vector defined on the full domain to its values in the subdomain number i, and the columns of U_0 span the coarse space defining the second level of the preconditioner. Note that R_i is required for the mathematical formalism, but is only used for the data mapping in a parallel implementation, so it has no additional memory cost within the iteration. In the GENEO approach, U_0 has $n_{ev} \times P$ columns; each subdomain contributes n_{ev} local vectors resulting from the solution of a local generalised eigenproblem. For the sake of complexity, we neglect the overlap between the subdomains, we consider a 3D decomposition of the cube and one process per subdomain, so that P also corresponds to the number of computing nodes. Each of the P subdomains lies on a subcube of side $m_{dom} = m \times P^{-1/3}$. A sparse direct solver is used to solve both the local problem defined on the subcube and the coarse space problem $A_0 = (U_0^T A U_0)$. In both cases the matrices correspond to the solution of a linear system defined by a 3D-stencil on a Cartesian grid inside a cube. The computational complexity of a sparse direct solver on a cube of side d is [20]: $d^4 \times 8$ bytes in memory. We can now calculate the memory footprint of the two-level preconditioner. For each subdomain we have to store the factors of $(R_i A R_i^T)$, which is $m_{dom}^4 \times 8$ bytes, the local column vectors contributing to U_0, which is $m_{dom}^3 \times n_{ev}$. The sum over the subdomain is $(P \times m_{dom}^4 + P \times m_{dom}^3 \times n_{ev}) \times 8 = (P \times (N/P)^{4/3} + N \times n_{ev}) \times 8$ bytes. We also need to store the Cholesky factor of A_0, which is a sparse matrix of dimension $(P \times n_e v) \times (P \times n_{ev})$, resulting from the discretisation of a template operator on a coarse cube of side $(n_{ev} \times P)^{1/3}$. Its memory footprint is $(n_{ev} \times P)^{4/3} \times 8$ bytes. In total, the domain decomposition preconditioner requires $(P \times (N/P)^{4/3} + N \times n_{ev} + (n_{ev} \times P)^{4/3}) \times 8$ bytes. Usually $n_{ev} \approx 10$ is sufficient to ensure good numerical scalability of the DDM preconditioner. We therefore consider $n_{ev} = 10$ in the evaluation.

Scenarios

We now proceed to consider four real-life scenarios based on large-scale experiments from the literature [31, 38, 30]. It should be noted that there is no exact correspondence between the parameters reported in these articles and the input parameters of Section 6.1. It is therefore necessary to extrapolate some values. While we have endeavoured to provide a rationale for our estimations, alternative extrapolations could yield equally valid results. In the first two scenarios, the use of a multigrid preconditioner is assumed; in contrast, the latter two scenarios assume the use of a DDM preconditioner.

6.2.1 Scenarios 1 and 2 using a multigrid preconditioner.

Scenario 1 is motivated by [31], where a Stokes problem of order $N = 3.6 \times 10^{12}$ unknowns is being processed on $P = 3 \times 10^3$ computing nodes. Although the original article employs

multigrid as a solver (as opposed to a preconditioner for PCG), the reported timings indicate that a PCG iteration can be estimated as follows: The application of the preconditioner requires 10 seconds, the matrix-vector product 2 seconds, and the remaining operations 1 second. This yields an estimation of $I = 13$ seconds for the duration of a complete PCG iteration. As the cost of the numerical verification is dominated by the cost of the matrix-vector product, we estimate $V_c = 2$ seconds. For the other values, we deduce them using the size N of the problem, the time of the computation and the balance that appears in Scenario 1. Hence, the cost to copy vectors in the local memory means that $C_{cm} = 0.5$ seconds. The fail-stop checkpoint is global so $C_{fs} = 180$ seconds. Finally, the checksum cost implies that $V_m = 6$ seconds. In all scenarios, we use $R_{cm} = C_{cm}$ and $R_{fs} = C_{fs}$.

Scenario 2 is motivated by [38], where a mantle convection simulation problem of order $N = 1.1 \times 10^{13}$ is processed on $P = 2 \times 10^4$ nodes. Similar extrapolations lead us to estimate a cost of $I = 110$ seconds for a PCG iteration, which can be decomposed into 85 seconds for the preconditioner, 17 seconds for the matrix-vector product, and 8 seconds for the other operations. The numerical verification being roughly the cost of one matrix-vector product, we consider $V_c = 17$ seconds. Then, although the size of the problem is 3.3 times larger than in Scenario 2, there are 6.7 times more nodes. Hence, the local checkpoint is quick, $C_{cm} = 0.25$ seconds. The memory verification takes $V_m = 3$ seconds. Finally, the fail-stop checkpoint is more expensive because of the number of nodes and the unknowns of the problem: $C_{fs} = 540$ seconds.

6.2.2 Scenarios 3 and 4 using a DDM preconditioner.

Scenarios 3 and 4 are the domain decomposition counterpart of the previous two scenarios that involved a multigrid preconditioner. The memory footprint changes as described earlier. The computation times are estimated using parallel experiments performed at scale and reported in [30, p99]. The experiments in [30, p99] essentially show that for stiff problems, the domain decomposition preconditioner executes four times fewer iterations, but is only twice as fast, which means that an iteration of PCG with domain decomposition takes twice as long as an iteration of PCG with a multigrid preconditioner.

A summary of all the parameter values for each scenario is given in Table 2.

Scenario	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Pb size (N)	3.6×10^{12}	1.1×10^{13}	3.6×10^{12}	1.1×10^{13}
#nodes (P)	3×10^3	2×10^4	3×10^3	2×10^4
I	13	110	26	220
V_c	2	17	2	17
V_m	6	3	22.8	11.4
$C_{cm} = R_{cm}$	0.5	0.25	1.9	0.95
$C_{fs} = R_{fs}$	180	540	684	2052

Table 2: Parameter values (in seconds) for the different scenarios with PCG.

6.3 Error probability distributions

There is limited information about potential values for the probability distributions of the three error sources. Let MTBF(fs), MTBF(mem) and MTBF($calc$) denote the MTBF for each error type; it is the inverse of the parameters λ_{fs}, λ_{mem} and f_{calc} of their probability distributions. On the one hand, there are good sources of data for fail-stop errors in the literature. The MTBF for fail-stop errors of recent leadership HPC machines is usually of a few hours [25, 44, 4, 41, 33]. We will typically consider a range of 1h to 8h for the MTBF(fs). On the other hand, there is no extensive study on memory and corruption errors in HPC to the best of our knowledge. A few studies have artificially modified the environment of a few nodes of HPC systems to increase the probability of memory and CPU failures due to particles of high energy interacting with the electronics (see [37, 17]). They then scale down the probability of failure linearly with the chance of normal encounter of such events, and scale it up linearly with the number of nodes. Other studies have disabled ECC mechanisms at scale on significant platforms and studied how many real failures were observed (see [7]). These studies show that the MTBF of unrecoverable and/or undetected errors vary significantly depending on the hardware used, and even on the application running on the machine.

We did not find studies on recent hardware used in exascale supercomputers, and resort to scaling the error rates observed in previous machines to provide a rough estimate of reasonable values for the MTBF($calc$) and MTBF(mem). For MTBF(mem), we could assume that silent memory corruptions scale with the amount of memory available. Using this heuristic to scale the observation of [17] on the Sequoia supercomputer to the Frontier supercomputer would lead to an MTBF(mem) of about 2.25h, and about 5h for the observations of [37] on the Roadrunner supercomputer. For MTBF($calc$), we assume that the risk of failure is proportional to the amount of computation, which leads to much lower values (as computation capability has increased much faster than the amount of memory available).

It is to be noted that we lack precise values to estimate MTBF($calc$) and MTBF(mem), as the technology has evolved. Authors of [17] also note that the computational intensity and type of operation of the application strongly impacts these values (with a factor 50 observed between the most reliable application and the least reliable one). Thus, we use these studies to give us an idea of the order of magnitude of risk for the different MTBFs, and study a large range of possible values for these different parameters.

In order to provide a unified scaling mechanism, we introduce a new parameter x, that links the different failure probabilities together. We say that:

MTBF(fs) $= x$, MTBF(mem) $= \frac{x}{2}$, and MTBF($calc$) $= \frac{x}{20}$.

It means that $\lambda_{fs} = \frac{1}{x}$, $\lambda_{mem} = \frac{2}{x}$, and $f_{calc} = \mathrm{e}^{-\frac{20}{x}}$.

We made this choice because if the majority of errors are fail-stop errors, then the system would either be really secure or not reliable at all, and our approach would not help in these cases. Moreover, even if we say that there are computation errors and memory errors, it can be difficult in the execution to differentiate between the two of them: in practice, there is a significant chance that what is originally a memory error is caught during the computation verification as it is the first verification to be encountered. In this work, we consider the case when the MTBF of computation errors is shorter than the MBTF of memory errors. This was the main rationale to design the hierarchical pattern shown in Figure 1.

6.4 Results

The code for the evaluation is publicly available at `https://doi.org/10.5281/zenodo.14572919`. This code computes the optimal values for n_{vc}, n_{cm}, n_{fs} that minimize the slowdown $\mathcal{S}$ using the exhaustive search described in Section 4.2, given the parameters of the different scenarios of Table 2.

Figures 2 to 5 show the slowdown ratio $\mathcal{S}$, with the optimal values for n_{vc}, n_{cm}, n_{fs}, and with the naive values ($n_{vc} = 1, n_{cm} = 1, n_{fs} = 1$) in some cases, as a function of the error parameter x, which represents the MTBF of fail-stop errors (MTBF(fs)), the other MTBFs (MTBF($calc$) and MTBF(mem)) being scaled accordingly, as described in Section 6.3.

When x increases, all error types are less likely to strike. The y-axis is the slowdown, which is a ratio of the expected completion time of the strategy to the execution time without faults and without fault-tolerance mechanism. We mesure the average of added costs consisting on the impact of errors on the algorithm with the recoveries and the added cost of the checkpoints and verifications that protect the algorithm. A value of one means that there is no added cost. It is safe to say that the slowdown decreases when the MTBF increases as less errors means less recoveries and there is less need to take verifications and checkpoints as often.

The triplets of numbers that appears on the plots are (n_{vc}, n_{cm}, n_{fs}) and correspond respectively to the number of iterations in a chunk, the number of chunks and the number of segments in the pattern. The product of these three variables is the number of iterations for the optimal pattern as a function of the error probabilities.

The result of Scenario 1 (Figure 2) shows a slowdown less than 2 for all values above 1h of fail-stop MTBF, and quickly drops to below 1.5 for values above 3h (so the application is at most 50% slower with faults and the optimal fault tolerance pattern than in a fully reliable machine). However, in the context of the PCG algorithm, that scenario exposes a possible limit of the approach via the number of iterations in a pattern. For $x = 4$ hours, a full pattern requires $3 \times 2 \times 22 = 132$ iterations. A simple CG algorithm in exact arithmetic takes at most the size of the problem as the number of iterations to converge. It converges even faster with the help of a preconditioner. In the case we consider, this means that in practice, only a few full patterns are expected to execute.

The naive approach consists of checkpointing at every iteration. We do not show the performance of the naive approach for this scenario, as it exhibits a slowdown systematically above 16 for the entire set of x values. This reflects the efficiency of the optimized approach.

Scenario 2 (Figure 3) shows a similar behavior, with a higher slowdown. As the number of nodes and the problem size increase, so does the duration of all other parameters, which directly impacts the overheads, resulting in a higher slowdown. The problem size, however, may require to distribute the application over more nodes. On a machine with the same levels of reliability, this leads to a minimum slowdown about twice larger (but still under a factor two for fairly reliable machines).

In this case, the optimal pattern falls back to taking verifications and in-memory checkpoints at each iteration ($n_{vc} = 1$ and $n_{cm} = 1$ for all optimal patterns), and only separates full fail-stop filesystem based checkpoints by an increasing number of iterations as failures become more rare. This is easily explained by the relative cost of the different fault tolerance techniques: compared to the high filesystem checkpoint cost of 540s, the other overheads are small and worth paying for the added benefits of avoiding full rollbacks.

This figure is also the first to show the slowdown of the naive approach. Compared to the optimal pattern, this would lead to a slowdown about three times higher than what is achievable, and above 6 times slower than the failure-free execution.

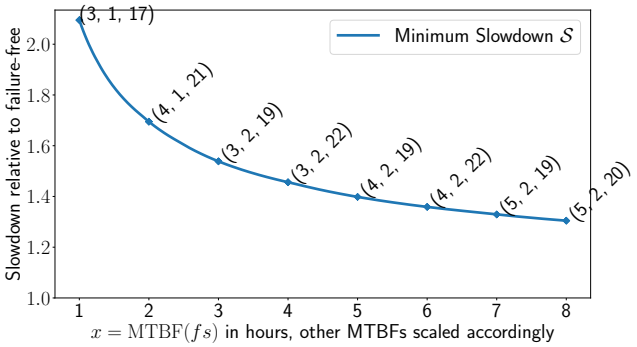

Figure 2: Results for Scenario 1.

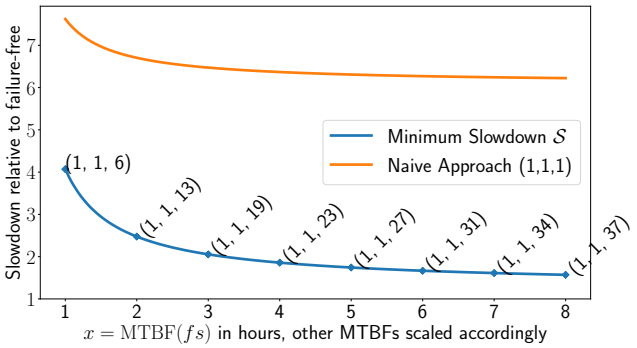

Figure 3: Results for Scenario 2.

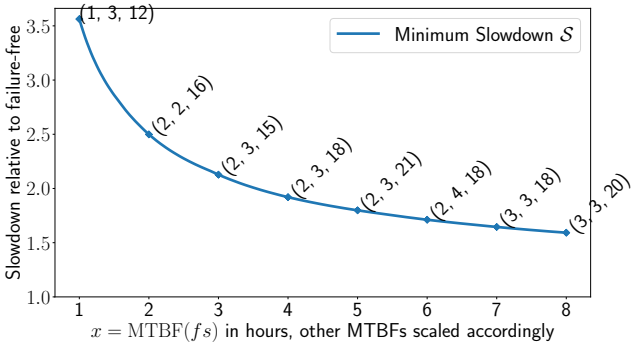

Figure 4: Results for Scenario 3.

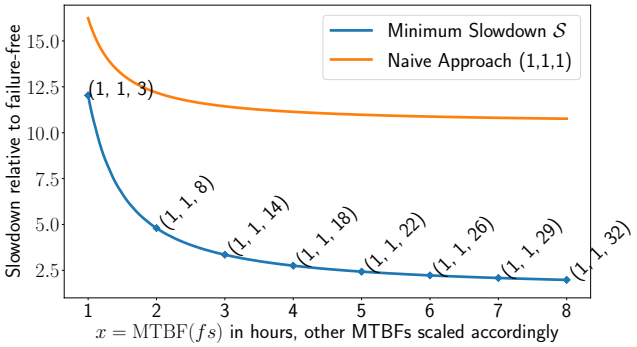

Figure 5: Results for scenario 4.

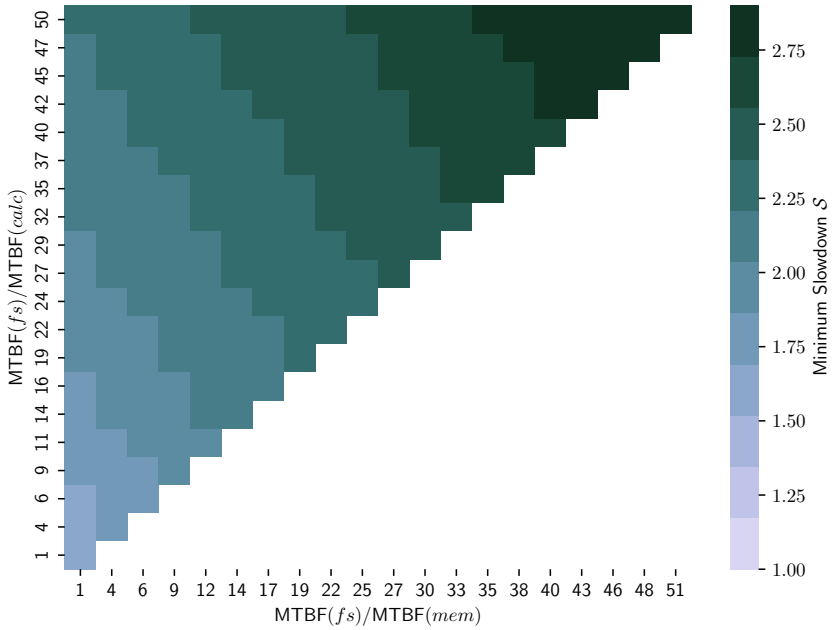

Figure 6: Scenario 2, $x = 4$ hours. Impact of the coefficient of silent error probabilities.

Scenarios 3 and 4 (Figures 4 and 5) consider DDM preconditioner, which exhibit higher overheads for the fail-stop checkpoints and memory protection. The cost of each iteration is also increased compared to the previous approach. As a consequence, the slowdowns are also increased, up to a factor 3.5 for Scenario 3, and above 10 for Scenario 4, if the system is very unreliable (1h of fail-stop MTBF). However, in more reliable setups (8h of fail-stop MTBF, which corresponds to the type of MTBF expected today on large supercomputers), these slowdown factors become more reasonable (about 1.75 for Scenario 3, and 2.5 for Scenario 4). As for Figure 2, the naive approach is too slow to appear on the figure; in this case, for all considered MTBF(fs) values, the naive approach slowdown is above 29.

At a given problem size, a lower number of nodes incurs a lower slowdown. This is mostly driven by the increase in iteration cost and in the increase in checkpointing time for the filesystem checkpoint. If a scalable approach for checkpointing can be implemented, most of these overheads could be absorbed providing better efficiency for this hierarchical approach.

All previous experiments have set a fixed relation between MTBF(fs), MTBF($calc$), and MTBF(mem). This was done to allow us representing curves in a simple space. In Figure 6, we consider a single point of Scenario 2 (Figure 3), when MTBF(fs) $= x =$ 4h, and we vary MTBF(mem) relative to MTBF(fs) on the x-axis and MTBF($calc$) relative to MTBF(fs) on the y-axis. The color shade represents the minimum Slowdown $\mathcal{S}$ for this set of parameters. We only consider the scenarios presented in this paper, i.e., when MTBF($calc$) $<$ MTBF(mem) $<$ MTBF(fs).

As expected, if either MTBF($calc$) or MTBF(mem) increases, the optimal pattern requires to take more frequent in-memory checkpoints and verifications, and the algorithm implies taking more frequent rollbacks to these checkpoints, leading to higher overheads. More interestingly, both parameters seem to impact the slowdown in a similar manner, showing that the method

can adapt to very variable systems, where either components are more or less reliable.

We have allowed in this figure MTBF(mem) and MTBF($calc$) to become highly unreliable, up to 50 times lower than MTBF(fs), meaning there would be a computational or memory failure every 5 minutes. This is to show the robustness of the approach, even though we do not expect such systems to be used in production. At values that we hope to be more realistic, the slowdown is below 1.25, making the method efficient for the benefits of obtaining a reliable result on an unreliable machine.

7 Conclusion

The main contribution of this work is the design and analysis of a new flexible approach to protect numerical iterative algorithms against three types of errors: fail-stop, memory and computation errors. We introduce a hierarchical pattern and compute the mean time per iteration within that pattern, enabling to optimize it. Finally, we instantiate the model with different scenarios based on the Preconditioned Conjugate Gradient (PCG) algorithm, demonstrating that the use of the proposed pattern allows us to significantly improve the expected execution time, compared to a naive approach where we would checkpoint at each iteration.

To the best of our knowledge, while simpler patterns have been introduced for simpler models in the literature, this is the first time that all three error types are dealt with, at the price of a very complicated derivation for the expected time per iteration. This work is the first study combining all three error types and opens several future research directions. In particular, we plan to extend the study to the case with imperfect verifications, which greatly complicates the analysis. Also, we would like to instantiate the pattern to other iterative algorithms, so as to further demonstrate the usefulness of this generic pattern-based approach.

Acknowledgements

This work was supported in part by the PEPR NumPEx (`https://numpex.org`).

References

[1] E. Agullo, M. Altenbernd, H. Anzt, L. Bautista-Gomez, T. Benacchio, L. Bonaventura, H. Bungartz, S. Chatterjee, F. M. Ciorba, N. DeBardeleben, D. Drzisga, S. Eibl, C. Engelmann, W. N. Gansterer, L. Giraud, D. Göddeke, M. Heisig, F. Jézéquel, N. Kohl, X. S. Li, R. Lion, M. Mehl, P. Mycek, M. Obersteiner, E. S. Quintana-Ortí, F. Rizzi, U. Rüde, M. Schulz, F. Fung, R. Speck, L. Stals, K. Teranishi, S. Thibault, D. Thönnes, A. Wagner, and B. I. Wohlmuth. Resiliency in numerical algorithm design for extreme scale simulations. *Int. J. High Perform. Comput. Appl.*, 36(2):251–285, 2022.

[2] E. Agullo, S. Cools, E. F. Yetkin, L. Giraud, N. Schenkels, and W. Vanroose. On soft errors in the conjugate gradient method: Sensitivity and robust numerical detection. *SIAM Journal on Scientific Computing*, 42(6):C335–C358, 2020.

[3] E. Agullo, L. Giraud, and L. Poirel. Robust preconditioners via generalized eigenproblems for hybrid sparse linear solvers. *SIAM Journal on Matrix Analysis and Applications*, 40(2):417–439, Jan. 2019.

[4] R. A. Ashraf and C. Engelmann. Analyzing the impact of system reliability events on applications in the titan supercomputer. In *2018 IEEE/ACM 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pages 39–48, 2018.

[5] L. Bautista-Gomez, A. Benoit, A. Cavelan, S. Raina, Y. Robert, and H. Sun. Coping with recall and precision of soft error detectors. *J. Parallel and Distributed Computing*, 98:8–24, 2016.

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

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

[8] A. Benoit, A. Cavelan, V. Le Fèvre, Y. Robert, and H. Sun. Towards Optimal Multi-Level Checkpointing. *IEEE Transactions on Computers*, 66(7):1212–1226, July 2017.

[9] A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Optimal resilience patterns to cope with fail-stop and silent errors. In *IPDPS'2016, the 30th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2016.

[10] A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *Int. J. High Performance Computing Applications*, 2014.

[11] J.-P. Boeuf. Tutorial: Physics and modeling of hall thrusters. *Journal of Applied Physics*, 121(1), Jan. 2017.

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

[13] G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *ICS*. ACM, 2008.

[14] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

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

[16] Z. Chen. Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. *ACM SIGPLAN Notices*, 48:167–176, 02 2013.

[17] C.-Y. Cher, M. S. Gupta, P. Bose, and K. P. Muller. Understanding soft error resiliency of blue gene/q compute chip through hardware proton irradiation and software fault injection. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 587–596, 2014.

[18] A. J. Chorin. Numerical solution of the navier-stokes equations. *Mathematics of Computation*, 22(104):745–762, 1968.

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

[20] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Jan. 2006.

[21] S. Di and F. Cappello. Adaptive Impact-Driven Detection of Silent Data Corruption for HPC Applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2809–2823, 2016.

[22] M. Fasi, J. Langou, Y. Robert, and B. Ucar. A backward/forward recovery approach for the preconditioned conjugate gradient method. *J. Computational Science*, 17:522–534, 2016.

[23] G. H. Golub and C. F. V. Loan. *Matrix computations*. Johns Hopkins, 4 edition, 2013.

[24] N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[25] T. Herault and Y. Robert, editors. *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks. Springer Verlag, 2015.

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

[27] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 46(6):409–436, Dec. 1952.

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

[29] A. A. Hwang, I. A. Stefanovici, and 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):111–122, 2012.

[30] P. Jolivet. *Méthodes de décomposition de domaine. Application au calcul haute performance.* PhD thesis, Université de Grenoble, 2006.

[31] N. Kohl and U. Rüde. Textbook efficiency: Massively parallel matrix-free multigrid for the stokes system. *SIAM J. Scientific Computing*, 44(2):C124–C155, 2022.

[32] S. Li, S. Di, K. Zhao, X. Liang, Z. Chen, and 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, New York, NY, USA, 2021. Association for Computing Machinery.

[33] R.-T. Liu and Z.-N. Chen. A large-scale study of failures on petascale supercomputers. *Journal of Computer Science and Technology*, 33(1):24–41, 2018.

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

[35] J. Marshall, C. Hill, L. Perelman, and A. Adcroft. Hydrostatic, quasi-hydrostatic, and nonhydrostatic ocean modeling. *Journal of Geophysical Research: Oceans*, 102(C3):5733–5752, Mar. 1997.

[36] G. Meurant. Detection and correction of silent errors in the conjugate gradient algorithm. *Numerical Algorithms*, 92, 07 2022.

[37] S. E. Michalak, A. J. DuBois, C. B. Storlie, H. M. Quinn, W. N. Rust, D. H. DuBois, D. G. Modl, A. Manuzzato, and S. P. Blanchard. Assessment of the impact of cosmic-ray-induced neutrons on hardware in the roadrunner supercomputer. *IEEE Transactions on Device and Materials Reliability*, 12(2):445–454, 2012.

[38] M. Mohr, U. Rüde, B. Wohlmuth, and H.-P. Bunge. Challenges for mantle convection simulations at the exa-scale: Numerics, algorithmics and software. In *Impact of Scientific Computing on Science and Society*, pages 75–92, Cham, 2023. Springer.

[39] A. Moody, G. Bronevetsky, K. Mohror, and 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*, pages 1–11. IEEE, 2010.

[40] 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):553–557, 1994.

[41] E. Rojas, E. Meneses, T. Jones, and D. Maxwell. Analyzing a five-year failure record of a leadership-class supercomputer. In *2019 31st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 196–203, 2019.

[42] P. Sao and R. Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, New York, NY, USA, 2013. Association for Computing Machinery.

[43] M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *ICS*. ACM, 2012.

[44] F. Shoji, S. Matsui, M. Okamoto, F. Sueyasu, T. Tsukamoto, A. Uno, and K. Yamamoto. Long term failure analysis of 10 peta-scale supercomputer. In *HPC in Asia Poster, ISC*, 2015.

[45] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. Debardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen. Addressing failures in exascale computing. *Int. J. High Perform. Comput. Appl.*, 28(2):129–173, 2014.

[46] N. Spillane, V. Dolean, P. Hauret, F. Nataf, C. Pechstein, and R. Scheichl. Abstract robust coarse spaces for systems of pdes via generalized eigenproblems in the overlaps. *Numerische Mathematik*, 126(4):741–770, Aug. 2013.

[47] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and 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)*, pages 297–310. ACM, 2015.

[48] The TOP500 team. HPCG, June 2024. `https://top500.org/lists/hpcg/2024/06/`.

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

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

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

RESEARCH CENTRE
Centre Inria de Lyon

Bâtiment CEI-2, Campus La Doua
56, Boulevard Niels Bohr - CS 52132
69603 Villeurbanne

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399