Coping with silent errors in HPC applications

Guillaume Aupy, Anne Benoit, Aurélien Cavelan, Massimiliano Fasi, Yves Robert, Hongyang Sun, Bora Uçar

Coping with silent errors in HPC applications

Guillaume Aupy, Anne Benoit, Aurélien Cavelan, Massimiliano Fasi, Yves Robert, Hongyang Sun and Bora Uçar

Abstract This chapter describes a unified framework for the detection and correction of silent errors, which constitute a major threat for scientific applications at extreme-scale. We first motivate the problem and explain why checkpointing must be combined with some verification mechanism. Then we introduce a general-purpose technique based upon computational patterns that periodically repeat over time. These patterns interleave verifications and checkpoints, and we show how to determine the pattern minimizing expected execution time. Then we move to application-specific techniques and review dynamic programming algorithms for linear chains of tasks, as well as ABFT-oriented algorithms for iterative methods in sparse linear algebra.

Guillaume Aupy
Penn State University, e-mail: `aupy@cse.psu.edu`

Anne Benoit
ENS Lyon, e-mail: `anne.benoit@ens-lyon.fr`

Aurélien Cavelan
ENS Lyon, e-mail: `aurelien.cavelan@ens-lyon.fr`

Massimiliano Fasi
The University of Manchester, e-mail: `massimiliano.fasi@manchester.ac.uk`

Hongyang Sun
ENS Lyon & INRIA, e-mail: `hongyang.sun@ens-lyon.fr`

Bora Uçar
CNRS & ENS Lyon, e-mail: `bora.ucar@ens-lyon.fr`

Yves Robert
ENS Lyon & Univ. Tennessee Knoxville, e-mail: `yves.robert@ens-lyon.fr`

1 Introduction

For High-Performance Computing (HPC) applications, scale is a major opportunity. Massive parallelism with 100,000+ nodes is the most viable path to achieving sustained petascale performance. Future platforms will exploit even more computing resources to enter the exascale era.

Unfortunately, scale is also a major threat, because resilience becomes a key challenge. Even if each node provides an individual MTBF (Mean Time Between Failures) of, say, one century, a machine with 100,000 such nodes encounters on average a failure every 9 hours, an interval much shorter than the execution time of many HPC applications. Note that (i) a one-century MTBF per node is an optimistic figure, given that each node features several hundreds of cores; and (ii) in some scenarios for the path to exascale computing [15], one envisions platforms including up to one million such nodes, whose MTBF will decrease to 52 minutes.

Several kinds of errors need to be considered when computing at scale. In the recent years, the HPC community has traditionally focused on fail-stop errors, such as hardware failures. The de facto general-purpose technique to recover from fail-stop errors is checkpoint/restart [11, 17]. This technique employs checkpoints to periodically save the state of a parallel application, so that when an error strikes some process, the application can be restored into one of its former states. There are several families of checkpointing protocols, but they share a common feature: each checkpoint forms a consistent recovery line, i.e., when an error is detected, one can rollback to the last checkpoint and resume execution, after a downtime and a recovery time. Many models are available to understand the behavior of the checkpointing and restarting techniques [8, 14, 31, 37].

While the picture is quite clear for fail-stop errors, the community has yet to devise an efficient approach to cope with silent errors, primary source of silent data corruptions. Such errors must also be accounted for when executing HPC applications [28, 30, 39, 40, 41]. They may be caused, for instance, by soft errors in L1 cache, arithmetic errors in the ALU (Arithmetic and Logic Unit), or bit flips due to cosmic radiation. The main issue is that the impact of silent errors is not immediate, since they do not manifest themselves until the corrupted data impact the result of the computation (see Figure 1), leading to a failure. If an error striking before the last checkpoint is detected after that checkpoint, then the checkpoint is corrupted, and cannot be used to restore the application. If only fail-stop failures are considered, a checkpoint cannot contain a corrupted state, because a process subject to failure cannot create a checkpoint or participate to the application: failures are naturally contained to failed processes. When dealing with silent errors, however, faults can propagate to other processes and checkpoints, because processes continue to participate and follow the protocol during the interval that separates the occurrence of the error from its detection.

In Figure 1, X_s and X_d are random variables that represent the time until the next silent error and its detection latency, respectively. We usually assume that silent errors strike according to a Poisson process of parameter λ, so that X_s has the distribution of an exponential law of parameter λ and mean $1/\lambda$. On the contrary, it

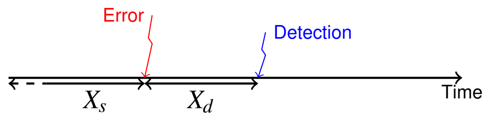

Fig. 1 Error and detection latency.

is very hard to make assumptions on the distribution of X_d. To alleviate the problem of detection latency, one may envision to keep several checkpoints in memory, and to restore the application from the last *valid* checkpoint, thereby rolling back to the last *correct* state of the application [25]. This multiple-checkpoint approach has three major drawbacks. First, it is demanding in terms of storage: each checkpoint typically represents a copy of the entire memory footprint of the application, which may well correspond to several terabytes. The second drawback is the possibility of fatal failures. Indeed, if we keep k checkpoints in memory, the approach requires that the last checkpoint still kept in memory to precede the instant when the error currently detected struck. Otherwise, all live checkpoints would be corrupted, and one would have to re-execute the entire application from scratch. The probability of a fatal failure for various error distribution laws and values of k can be evaluated [1]. The third and most serious drawback of this approach applies even without memory constraints, i.e., if we could store an infinite number of checkpoints in memory. The critical point is to determine which checkpoint is the last valid one, information which is necessary to recover from a valid application state. However, because of the detection latency (which is unknown), we do not know when the silent error has indeed occurred, hence we cannot identify the last valid checkpoint, unless some verification mechanism is enforced.

We introduce such verification mechanisms in this chapter. In Section 2, we discuss several approaches to validation (recomputation, checksums, coherence tests, orthogonalization checks, etc). Then in Section 3 we adopt a general-purpose approach, which is agnostic of the nature of the verification mechanism. We consider a divisible-load application (which means that we can take checkpoints at any instant), and we partition the execution into computational patterns that repeat over time. The simplest pattern is represented by a work chunk followed by a verified checkpoint, which corresponds to performing a verification just before taking each checkpoint. If the verification succeeds, then one can safely store the checkpoint. If the verification fails, then a silent error has struck since the last checkpoint, and one can safely recover from it to resume the execution of the application. We compute the optimal length of the work chunk in the simplest pattern in Section 3.1, which amounts to revisiting Young and Daly's formula [37, 14] for silent errors. While taking a checkpoint without verification seems a bad idea (because of the memory cost, and of the risk of saving corrupted data), a validation step not immediately followed by a checkpoint may be interesting. Indeed, if silent errors are frequent enough, verifying the data in between two (verified) checkpoints, will reduce in expectation the detection latency and thus the amount of work to be re-executed due to possible silent errors. The major goal of Section 3 is to determine the best pattern composed of *m* work chunks, where each chunk is followed by a verification and the last chunk is followed by a verified checkpoint. We show how to determine *m* and the length of each chunk so as to minimize the *makespan*, that is the total execution time.

Then we move to application workflows. In Section 4, we consider application workflows that consist of a number of parallel tasks that execute on a platform, and that exchange data at the end of their execution. In other words, the task graph is a linear chain, and each task (except maybe the first and the last one) reads data from its predecessor and produces data for its successor. This scenario corresponds to a high-performance computing application whose workflow is partitioned into a succession of (typically large) tightly-coupled computational kernels, each of them being identified as a task by the model. At the end of each task, we can either perform a verification on the task output, or perform a verification followed by a checkpoint. We provide dynamic programming algorithms to determine the optimal locations of checkpoints and verifications.

The last technique that we illustrate is application-specific. In Section 5, we deal with sparse linear algebra kernels, and we show how to combine ABFT (Algorithm Based Fault Tolerance) with checkpointing. In a nutshell, ABFT consists in adding checksums to application data, and to view them as extended data items. The application performs the same computational updates on the original data and on the checksums, thereby avoiding the need to recompute the checksums after each update. The salient feature of this approach is *forward recovery*: ABFT is used both as an error verification and error correction mechanism: whenever a single error strikes, it can be corrected via ABFT and there is no need to rollback for recovery. Finally, we outline main conclusions and directions for future work in Section 6.

2 Verification mechanisms

Considerable efforts have been directed at error-checking to reveal silent errors. Error detection is usually very costly. Hardware mechanisms, such as ECC (Error Correcting Code) memory, can detect and even correct a fraction of errors, but in practice they are complemented with software techniques. General-purpose techniques are based on replication [18, 21, 34, 38]. Indeed, performing the operation twice and comparing the results of the replicas makes it possible to detect a single silent error. With Triple Modular Redundancy [26] (TMR) , errors can also be corrected by means of a voting scheme. Another approach, proposed by Moody et al. [29], is based on checkpointing and replication and enables detection and fast recovery of applications from both silent errors and hard errors.

Coming back to verification mechanisms, application-specific information can be helpful in designing ad hoc solutions, which can dramatically decrease the cost of detection. Many techniques have been advocated. They include memory scrubbing [24], but also ABFT techniques [7, 23, 35], such as coding for the SpMxV

(Sparse Matrix-Vector multiplication) kernel [35], and coupling a higher-order with a lower-order scheme for Ordinary Differential Equations [6]. These methods can only detect an error but not correct it. Self-stabilizing corrections after error detection in the conjugate gradient method are investigated by Sao and Vuduc [33]. Also, Heroux and Hoemmen [22] design a fault-tolerant GMRES algorithm capable of converging despite silent errors, and Bronevetsky and de Supinski [9] provide a comparative study of detection cost for iterative methods. Elliot et al. [16] combine partial redundancy and checkpointing, and confirm the benefit of dual and triple redundancy. The drawback is that twice the number of processing resources is required (for dual redundancy).

A nice instantiation of the checkpoint and verification mechanism that we study in this chapter is provided by Chen [12], who deals with sparse iterative solvers. Consider a simple method such as the Preconditioned Conjugate Gradient (PCG) method: Chen's approach performs a periodic verification every d iterations, and a periodic checkpoint every $d \times c$ iterations, which is a particular case, with equi-spaced validations, of the approach presented later in Section 3.2. For PCG, the verification amounts to checking the orthogonality of two vectors and to recomputing and checking the residual. The cost of the verification is small if compared to the cost of an iteration, especially when the preconditioner requires many more flops than a SpMxV. As already mentioned, the approach presented in Section 3 is agnostic of the underlying error-detection technique and takes the cost of verification as an input parameter to the model.

3 Patterns for divisible load applications

In this section we explain how to derive the optimal pattern of interleaving checkpoints and verifications. An extended presentation of the results is available in [2, 4, 10].

3.1 Revisiting Young and Daly's formula

Consider a divisible-load application, i.e., a (parallel) job that can be interrupted at any time for checkpointing, for a nominal cost C. To deal with fail-stop failures, the execution is partitioned into same-size chunks followed by a checkpoint, and there exist well-known formulae by Young [37] and Daly [14] to determine the optimal checkpointing period.

To deal with silent errors, the simplest protocol (see Figure 2) would be to perform a verification (at a cost V) just before taking each checkpoint. If the verification succeeds, then one can safely store the checkpoint and mark it as *valid*. If the verification fails, then an error has struck since the last checkpoint, which is correct having been verified, and one can safely recover (which takes a time R) from that checkpoint to resume the execution of the application. This protocol with verifications zeroes out the risk of fatal errors that would force to restart the execution from scratch.

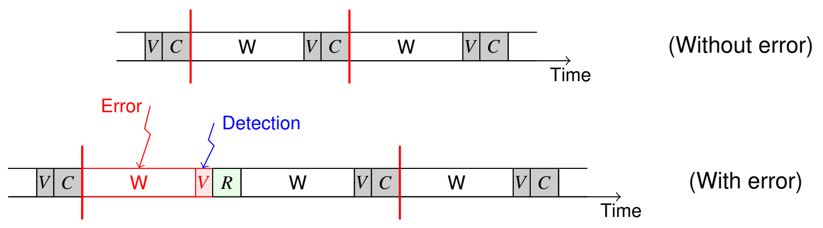

Fig. 2 The simplest pattern: a work chunk W followed by a verification V and a checkpoint C.

To compute the optimal length of the work chunk W^*, we first have to define the objective function. The aim is to find a pattern P (with a work chunk of length W followed by a verification of length V and a checkpoint of length C) that minimizes the expected execution time of the application. Let W_{base} denote the base execution time of an application without any overhead due to resilience techniques (without loss of generality, we assume unit-speed execution). The execution is divided into periodic patterns, as shown in Figure 2. Let $\mathbb{E}(\mathrm{P})$ be the expected execution time of the pattern. For large jobs, the expected makespan W_{final} of the application when taking failures into account can then be approximated by

$$W_{\text{final}} \approx \frac{\mathbb{E}(\mathrm{P})}{W} \times W_{\text{base}} = W_{\text{base}} + H(\mathrm{P}) \cdot W_{\text{base}}$$

where

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

is the expected *overhead* of the pattern. Thus, minimizing the expected makespan is equivalent to minimizing the pattern overhead $H(\mathrm{P})$. Hence, we focus on minimizing the pattern overhead. We assume that silent errors are independent and follow a *Poisson process* with arrival rate λ. The probability of having at least a silent error during a computation of length w is given by $p = 1 - e^{-\lambda w}$. We assume that errors cannot strike during recovery and verification. The following proposition shows the expected execution time of a pattern with a fixed work length W.

Proposition 1. *The expected execution time of a pattern* P *with work length* W *is*

$$\mathbb{E}(\mathrm{P}) = W + V + C + \lambda W^2 + \lambda W(V + R) + O(\lambda^2 W^3) \ . \quad (1)$$

Proof. Let $p = 1 - e^{-\lambda W}$ denote the probability of having at least one silent error in the pattern. The expected execution time obeys the recursive formula

$$\mathbb{E}(\mathrm{P}) = W + V + p(R + \mathbb{E}(\mathrm{P})) + (1 - p)C \ . \quad (2)$$

Equation (2) can be interpreted as follows: we always execute the work chunk and run the verification to detect silent errors, whose occurrence requires not only a recovery but also a re-execution of the whole pattern. Otherwise, if no silent error strikes, we can proceed with the checkpoint. Solving the recursion in Equation (2), we obtain

$$\mathbb{E}(\mathrm{P}) = e^{\lambda W}(W+V) + \left(e^{\lambda W} - 1\right)R + C \ .$$

By approximating $e^{\lambda x} = 1 + \lambda x + \frac{\lambda^2 x^2}{2}$ up to the second-order term, we can further simplify the expected execution time and obtain Equation (1). $\square$

The following theorem gives a first-order approximation to the optimal work length of a pattern.

Theorem 1. *A first-order approximation to the optimal work length W^* is given by*

$$W^* = \sqrt{\frac{V+C}{\lambda}} \ . \tag{3}$$

The optimal expected overhead is

$$H^*(\mathrm{P}) = 2\sqrt{\lambda(V+C)} + O(\lambda) \ . \tag{4}$$

Proof. From the result of Proposition 1, the expected overhead of the pattern can be computed as

$$H(\mathrm{P}) = \frac{V+C}{W} + \lambda W + \lambda(V+R) + O(\lambda^2 W^2) \ . \tag{5}$$

Assume that the MTBF of the platform $\mu = 1/\lambda$ is large if compared to the resilience parameters. Then consider the first two terms of $H(\mathrm{P})$ in Equation (5): the overhead is minimal when the pattern has length $W = \Theta(\lambda^{-1/2})$, and in that case both terms are in the order of $\lambda^{1/2}$, so that we have

$$H(\mathrm{P}) = \Theta(\lambda^{1/2}) + O(\lambda).$$

Indeed, the last term $O(\lambda)$ becomes also negligible when compared to $\Theta(\lambda^{1/2})$. Hence, the optimal pattern length W^* can be obtained by balancing the first two terms in Equation (5), which gives Equation (3). Then, by substituting W^* back into $H(\mathrm{P})$, we get the optimal expected overhead in Equation (4). $\square$

We observe from Theorem 1 that the optimal work length W^* of a pattern is in $\Theta\left(\lambda^{-1/2}\right)$, and the optimal overhead $H^*(\mathrm{P})$ is in $\Theta(\lambda^{1/2})$. This allows us to express the expected execution overhead of a pattern as $H(\mathrm{P}) = \frac{o_{\mathrm{ef}}}{W} + o_{\mathrm{rw}} W + O(\lambda)$, where o_{ef} and o_{rw} are two key parameters that characterize two different types of overheads in the execution, and they are defined below.

Definition 1. For a given pattern, o_{ef} denotes the *error-free* overhead due to the resilience operations (e.g., verification, checkpointing), and o_{rw} denotes the *re-executed work* overhead, in terms of the fraction of re-executed work due to errors.

In the simple pattern we analyze above, these two overheads are given by $o_{\text{ef}} = V + C$ and $o_{\text{rw}} = \lambda$, respectively. The optimal pattern length and the optimal expected overhead can thus be expressed as

$$W^* = \sqrt{\frac{o_{\text{ef}}}{o_{\text{rw}}}} ,$$
$$H^*(\mathrm{P}) = 2\sqrt{o_{\text{ef}} \cdot o_{\text{rw}}} + O(\lambda) .$$

We see that minimizing the expected execution overhead $H(\mathrm{P})$ of a pattern becomes equivalent to minimizing the product $o_{\text{ef}} \times o_{\text{rw}}$ up to the dominating term. Intuitively, including more resilient operations reduces the re-executed work overhead but adversely increases the error-free overhead, and vice versa. This requires a resilience protocol that finds the optimal tradeoff between o_{ef} and o_{rw}. We make use of this observation in the next section to derive the optimal pattern in a more complicated protocol where patterns are allowed to include several chunks.

3.2 Optimal pattern

If the verification cost is small when compared to the checkpoint cost, there is room for optimization. Consider the pattern illustrated in Figure 3 with three verifications per checkpoint. There are three chunks of size w_1, w_2, and w_3, each followed by a verification. Every third verification is followed by a checkpoint.

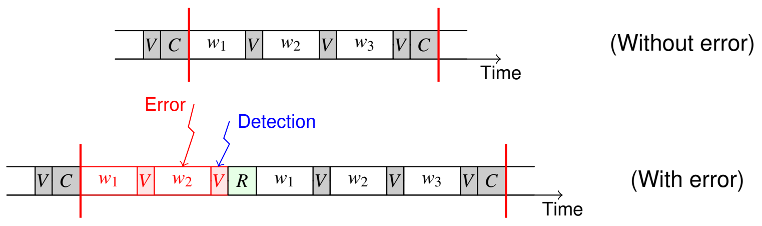

Fig. 3 Pattern with three verifications per checkpoint.

To understand the advantages of such a pattern, assume $w_1 = w_2 = w_3 = W/3$ for now, so that the total amount of work is the same as in the simplest pattern. As before, a single checkpoint needs to be kept in memory, and each error leads to re-executing the work since the last checkpoint. But detection occurs much more rapidly in the new pattern, because of the intermediate verifications. If the error strikes during the first of the three chunks, it is detected by the first verification, and only the first chunk is re-executed. Similarly, if the error strikes the execution of the second chunk (as illustrated in the figure), it is detected by the second verification, and the first two chunks are re-executed. The entire frame of work needs to be re-executed only if the error strikes during the third chunk. Under the first-order approximation as in the analysis of Theorem 1, the average amount of work to re-execute is $(1+2+3)w/3 = 2w = 2W/3$, that is, the re-executed work overhead becomes $o_{\text{rw}} = 2\lambda/3$. On the contrary, in the first pattern of Figure 2, the amount of work to re-execute is always W, because the error is never detected before the end of the pattern. Hence, the second pattern leads to a 33% gain in the re-execution time. However, this comes at the price of three times as many verifications, that is, the error-free overhead becomes $o_{\text{ef}} = 3V + C$. This overhead is paid in every error-free execution, and may be an overkill if the verification mechanism is too costly.

This example shows that finding the best trade-off between error-free overhead (what is paid due to the resilience method, when there is no failure during execution) and execution time (when errors strike) is not a trivial task. The optimization problem can be stated as follows: given the cost of checkpointing C, recovery R, and verification V, what is the optimal pattern to minimize the (expectation of the) execution time? A pattern is composed of several work chunks, each followed by a verification, and the last chunk is always followed by both a verification and a checkpoint. Let m denote the number of chunks in the pattern, and let w_j denote the length of the j-th chunk for $1 \le j \le m$. Let $W = \sum_{j=1}^{m} w_j$. We define $\beta_j = w_j/W$ be the relative length of the j-th chunk so that $\beta_j \ge 0$ and $\sum_{j=1}^{m} \beta_j = 1$. We let $\boldsymbol{\beta} = [\beta_1, \beta_2, \dots, \beta_m]$. The goal is to determine the pattern work length W, the number of chunks m as well as the relative length vector $\boldsymbol{\beta}$.

Proposition 2. *The expected execution time of the above pattern is*

$$\mathbb{E}(\mathrm{P}) = W + mV + C + \left(\lambda \boldsymbol{\beta}^T \mathbf{A} \boldsymbol{\beta}\right) W^2 + O(\sqrt{\lambda}) \;, \tag{6}$$

where $\mathbf{A}$ *is an* $m \times m$ *matrix whose diagonal coefficients are equal to 1 and whose other coefficients are all equal to* $\frac{1}{2}$.

Proof. Let $p_j = 1 - e^{-\lambda w_j}$ denote the probability of having at least one silent error in chunk j. To derive the expected execution time of the pattern, we need to know the probability q_j that the chunk j actually gets executed in the current attempt.

The first chunk is always executed, so we have $q_1 = 1$. Consider the second chunk, which is executed if no silent error strikes the first chunk, hence $q_2 = 1 - p_1$. In general, the probability that the j-th chunk gets executed is

$$q_j = \prod_{k=1}^{j-1} (1 - p_k) \;.$$

Now, we are ready to compute the expected execution time of the pattern. The following gives the recursive expression:

$$\begin{aligned}\mathbb{E}(\mathrm{P}) = &\left(\prod_{k=1}^{m}(1-p_k)\right)C \\ &+\left(1-\prod_{k=1}^{m}(1-p_k)\right)(R+\mathbb{E}(\mathrm{P})) \\ &+\sum_{j=1}^{m} q_j(w_j+V) \ . \end{aligned} \tag{7}$$

Specifically, line 1 of Equation (7) shows that the checkpoint at the end of the pattern is performed only when there has been no silent error in any of the chunks. Otherwise, we need to re-execute the pattern, after a recovery, as shown in line 2. Finally, line 3 shows the condition for each chunk j to be executed. By simplifying Equation (7) and approximating the expression up to the second-order term, as in the proof of Proposition 1, we obtain

$$\mathbb{E}(\mathrm{P}) = W + mV + C + \lambda f W^2 + O(\sqrt{\lambda}) \ ,$$

where $f = \sum_{j=1}^{m} \beta_j \left(\sum_{k=j}^{m} \beta_k\right)$, and it can be concisely written as $f = \boldsymbol{\beta}^T \mathbf{M} \boldsymbol{\beta}$, where $\mathbf{M}$ is the $m \times m$ matrix given by

$$m_{i,j} = \begin{cases} 1 & \text{for } i \leq j \\ 0 & \text{for } i > j \end{cases} .$$

By replacing $\mathbf{M}$ by its symmetric part $\mathbf{A} = \frac{\mathbf{M}+\mathbf{M}^T}{2}$, which does not affect the value of f, we obtain the matrix $\mathbf{A}$ whose diagonal coefficients are equal to 1 and whose other coefficients are all equal to $\frac{1}{2}$, and the expected execution time in Equation (6). □

Theorem 2. *The optimal pattern has m^* equal-length chunks, total length W^* and is such that:*

$$W^* = \sqrt{\frac{m^*V + C}{\frac{1}{2}\left(1+\frac{1}{m^*}\right)\lambda}} \ , \tag{8}$$

$$\beta_j^* = \frac{1}{m^*} \ \textit{for} \ 1 \leq j \leq m^* \ , \tag{9}$$

where m^* is either $\max(1, \lfloor \bar{m}^* \rfloor)$ *or* $\lceil \bar{m}^* \rceil$ *with*

$$\bar{m}^* = \sqrt{\frac{C}{V}} \ . \tag{10}$$

The optimal expected overhead is

$$H^*(\mathrm{P}) = \sqrt{2\lambda C} + \sqrt{2\lambda V} + O(\lambda) \ . \tag{11}$$

Proof. Given the number of chunks m with $\sum_{j=1}^{m} \beta_j = 1$, the function $f = \boldsymbol{\beta}^T \mathbf{A} \boldsymbol{\beta}$ is shown to be minimized [10, Theorem 1 with $r = 1$] when $\boldsymbol{\beta}$ follows Equation (9), and its minimum value is given by $f^* = \frac{1}{2}\left(1 + \frac{1}{m}\right)$. We derive the two types of overheads as follows:

$$o_{\text{ef}} = mV + C \,,$$
$$o_{\text{rw}} = \frac{1}{2}\left(1 + \frac{1}{m}\right)\lambda \,.$$

The optimal work length $W^* = \sqrt{\frac{o_{\text{ef}}}{o_{\text{rw}}}}$ for any fixed m is thus given by Equation (8). The optimal number of chunks $\bar{m}^*$ shown in Equation (10) is obtained by minimizing $F(m) = o_{\text{ef}} \times o_{\text{rw}}$. The number of chunks in a pattern can only be a positive integer, so m^* is either $\max(1, \lfloor \bar{m}^* \rfloor)$ or $\lceil \bar{m}^* \rceil$, since $F(m)$ is a convex function of m. Finally, substituting Equation (10) back into $H^*(\mathrm{P}) = 2\sqrt{o_{\text{ef}} \times o_{\text{rw}}} + O(\lambda)$ gives rise to the optimal expected overhead as shown in Equation (11). □

4 Linear workflows

For an application composed of a chain of tasks, the problem of finding the optimal checkpoint strategy, i.e., of determining which tasks to checkpoint, in order to minimize the expected execution time when subject to fail-stop failures, has been solved by Toueg and Babaoglu [36], using a dynamic programming algorithm. We revisit the problem for silent errors by exploiting verification in addition to checkpoints. An extended presentation of the results is available in [3, 5].

4.1 Setup

To deal with silent errors, resilience is provided through the use of checkpointing coupled with an error detection (or verification) mechanism. When a silent error is detected, we roll back to the nearest checkpoint and recover from there. As in Section 3.1, let C denote the cost of checkpointing, R the cost of recovery, and V the cost of a verification.

We consider a chain of tasks $T_1, T_2, \dots, T_n$, where each task T_i has a weight w_i corresponding to the computational load. For notational convenience, we also define $W_{i,j} = \sum_{k=i+1}^{j} w_k$ to be the time to execute tasks T_{i+1} to T_j for any $i \le j$. Once again we assume that silent errors occur following a *Poisson process* with arrival rate λ and that the probability of having at least one error during the execution of $W_{i,j}$ is given by $p_{i,j} = 1 - e^{-\lambda W_{i,j}}$.

We enforce that a verification is always taken immediately before each checkpoint, so that all checkpoints are valid, and hence only one checkpoint needs to be maintained at any time during the execution of the application. Furthermore, we assume that errors only strike the computations, while verifications, checkpoints, and recoveries are failure-free.

The goal is to find which task to verify and which task to checkpoint in order to minimize the expected execution time of the task chain. To solve this problem, we derive a two-level dynamic programming algorithm. For convenience, we add a virtual task T_0, which is always checkpointed, and whose recovery cost is zero. This accounts for the fact that it is always possible to restart the application from scratch at no extra cost. In the following, we describe the general scheme when considering both verifications and checkpoints.

4.2 Dynamic programming

Figures 4 and 5 illustrate the idea of the algorithm, which contains two dynamic programming levels, responsible for placing checkpoints and verifications, respectively, as well as an additional step to compute the expected execution time between two verifications. The following describes each step of the algorithm in detail.

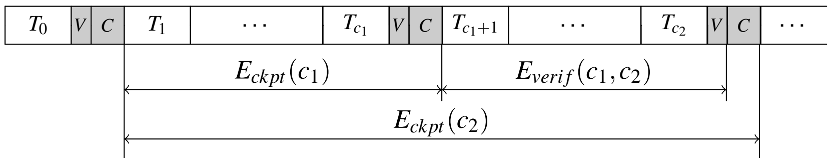

Fig. 4 First level of dynamic programming (E_{ckpt}).

Placing checkpoints. The first level focuses on the placement of verified checkpoints, i.e., checkpoints preceded immediately by a verification. Let $E_{ckpt}(c_2)$ denote the expected time to successfully execute all the tasks from T_1 to T_{c_2}, where T_{c_2} is verified and checkpointed. Now, to find the last verified checkpoint before T_{c_2}, we try all possible locations from T_0 to T_{c_2-1}. For each location, say c_1, we call the function recursively with $E_{ckpt}(c_1)$ (for placing checkpoints before T_{c_1}), and compute the expected time to execute the tasks from T_{c_1+1} to T_{c_2}. The latter is done through $E_{verif}(c_1,c_2)$, which also decides where to place additional verifications between T_{c_1+1} and T_{c_2}. Finally, we add the checkpointing cost C (after T_{c_2}) to $E_{ckpt}(c_2)$. Overall, we can express $E_{ckpt}(c_2)$ as follows:

$$E_{ckpt}(c_2) = \min_{0 \le c_1 < c_2} \{E_{ckpt}(c_1) + E_{verif}(c_1,c_2) + C\} .$$

Note that a location $c_1 = 0$ means that no further checkpoints are added. In this case, we simply set $E_{ckpt}(0) = 0$, which initializes the dynamic program. The total expected time to execute all the tasks from T_1 to T_n is thus given by $E_{ckpt}(n)$.

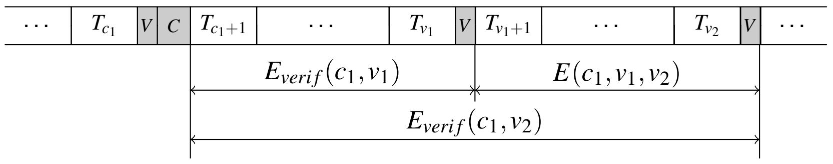

Fig. 5 Second level of dynamic programming (E_{verif}) and computation of expected execution time between two verifications (E).

Placing additional verifications. The second level decides where to insert additional verifications between two tasks with verified checkpoints. The function is initially called from the first level between two checkpointed tasks T_{c_1} and T_{c_2}, each of which also comes with a verification. Therefore, we define $E_{verif}(c_1, v_2)$ as the expected time to successfully execute all the tasks from T_{c_1+1} to T_{v_2}, knowing that the last checkpoint is right after task T_{c_1}, and there is no additional checkpoint between T_{c_1+1} and T_{v_2}. Note that $E_{verif}(c_1, v_2)$ accounts only for the time required to execute and verify these tasks. As before, we try all possible locations for the last verification between T_{c_1} and T_{v_2} and, for each location v_1, we call the function recursively with $E_{verif}(c_1, v_1)$. Furthermore, we add the expected time needed to successfully execute the tasks T_{v_1+1} to T_{v_2}, denoted by $E(c_1, v_1, v_2)$, given the position c_1 of the last checkpoint. Overall, we can express $E_{verif}(c_1, v_2)$ as follows:

$$E_{verif}(c_1, v_2) = \min_{c_1 \le v_1 < v_2} \{E_{verif}(c_1, v_1) + E(c_1, v_1, v_2)\} . \tag{12}$$

Again, the case $v_1 = c_1$ means that no further verification is added, so we initialize the dynamic program with $E_{verif}(c_1, c_1) = 0$. Note that the verification cost V at the end of task T_{v_2} will be accounted for in the function $E(c_1, v_1, v_2)$.

Computing expected execution time between two verifications. Finally, to compute the expected time to successfully execute several tasks between two verifications, we need the position of the last checkpoint c_1, as well as the positions of the two verifications v_1 and v_2.

First, we pay W_{v_1,v_2} by executing all the tasks from T_{v_1+1} to T_{v_2}, followed by the cost of verification V after T_{v_2}. During the execution, there is a probability $p_{v_1,v_2} = 1 - e^{-\lambda W_{v_1,v_2}}$ of having a silent error, which will be detected by the verification after T_{v_2}. In this case, we need to perform a recovery from the last checkpoint after T_{c_1} with a cost R (set to 0 if $c_1 = 0$), and re-execute the tasks from there by calling the function $E_{verif}(c_1, v_1)$ followed by $E(c_1, v_1, v_2)$. Therefore, we can express $E(c_1, v_1, v_2)$ as follows:

$$E(c_1,v_1,v_2) = W_{v_1,v_2} + V + p_{v_1,v_2}\left(R + E_{verif}(c_1,v_1) + E(c_1,v_1,v_2)\right) . \qquad (13)$$

Simplifying Equation (13), we get

$$E(c_1,v_1,v_2) = e^{\lambda W_{v_1,v_2}}\left(W_{v_1,v_2} + V\right) + \left(e^{\lambda W_{v_1,v_2}} - 1\right)\left(R + E_{verif}(c_1,v_1)\right) .$$

Complexity. The complexity is dominated by the computation of the expected completion time table $E_{verif}(c_1,v_2)$, which contains $O(n^2)$ entries, and each entry depends on at most n other entries that are already computed. All tables are computed in a bottom-up fashion, from the left to the right of the task chain. Hence, the overall complexity of the algorithm is $O(n^3)$.

5 ABFT and checkpointing for linear algebra kernels

In this section we introduce ABFT (Algorithm Based Fault Tolerance) as an application-specific technique which allows for both error detection and correction. We streamline our discussion on the CG method, however, the techniques that we describe are applicable to any iterative solver that uses sparse matrix vector multiplies and vector operations. This list includes many of the non-stationary iterative solvers such as CGNE (Conjugate Gradient on Normal Equations), BiCG (Bi-Conjugate Gradient), BiCGstab (Bi-Conjugate Gradient Stabilized), where sparse matrix transpose vector multiply operations also take place. Preconditioned variants of these solvers with an approximate inverse preconditioner (applied as an SpMxV, or two SpMxVs) can also be made fault-tolerant with the proposed scheme. The extension to PCG is described in [19].

In Section 5.1, we first provide a background on the CG method and give an overview of both Chen's stability tests [12] and ABFT protection schemes. Then we detail ABFT techniques for the SpMxV kernel.

5.1 CG and fault tolerance mechanisms

The code for the CG method is shown in Algorithm 1. The main loop features a sparse matrix-vector multiply, two inner products (for $\mathbf{p}_i^\mathsf{T}\mathbf{q}$ and $\|\mathbf{r}_{i+1}\|^2$), and three vector operations of the form *axpy*.

Chen's stability tests [12] amount to checking the orthogonality of vectors $\mathbf{p}_{i+1}$ and $\mathbf{q}$, at the price of computing $(\mathbf{p}_{i+1}^\mathsf{T}\mathbf{q})/(\|\mathbf{p}_{i+1}\|\,\|\mathbf{q}_i\|)$, and to checking the residual at the price of an additional SpMxV operation $\mathbf{A}\mathbf{x}_i - \mathbf{b}$. The dominant cost of these verifications is the additional SpMxV operation.

We investigate three fault tolerance mechanisms. The first one is ONLINE-DETECTION; this is Chen's original approach modified to save the matrix $\mathbf{A}$ in addition to the current iteration vectors. This is needed when a silent error is de-

Algorithm 1 The Conjugate Gradient algorithm for a positive definite matrix $\mathbf{A}$.

```
Input: $\mathbf{A} \in \mathbb{R}^{n \times n}, \mathbf{b}, \mathbf{x}_0 \in \mathbb{R}^n, \varepsilon \in \mathbb{R}$
Output: $\mathbf{x} \in \mathbb{R}^n \ : \ \|\mathbf{Ax} - \mathbf{b}\| \le \varepsilon$
 1: $\mathbf{r}_0 \leftarrow \mathbf{b} - \mathbf{Ax}_0$;
 2: $\mathbf{p}_0 \leftarrow \mathbf{r}_0$;
 3: $i \leftarrow 0$;
 4: while $\|\mathbf{r}_i\| > \varepsilon\left(\|\mathbf{A}\| \cdot \|\mathbf{r}_0\| + \|\mathbf{b}\|\right)$ do
 5:     $\mathbf{q} \leftarrow \mathbf{Ap}_i$;
 6:     $\alpha_i \leftarrow \|\mathbf{r}_i\|^2 / \mathbf{p}_i^{\mathsf{T}}\mathbf{q}$;
 7:     $\mathbf{x}_{i+1} \leftarrow \mathbf{x}_i + \alpha\,\mathbf{p}_i$;
 8:     $\mathbf{r}_{i+1} \leftarrow \mathbf{r}_i - \alpha\,\mathbf{q}$;
 9:     $\beta \leftarrow \|\mathbf{r}_{i+1}\|^2 / \|\mathbf{r}_i\|^2$;
10:     $\mathbf{p}_{i+1} \leftarrow \mathbf{r}_{i+1} + \beta\,\mathbf{p}_i$;
11:     $i \leftarrow i+1$;
12: end while
13: return $\mathbf{x}_i$;
```

tected: if this error comes for a corruption in data memory, we need to recover with a valid copy of the data matrix **A**. The second one is ABFT-DETECTION, which detects errors and restarts from the most recent checkpoint. The thirds one is ABFT-CORRECTION, which detects errors and corrects if there was only one, otherwise restarts from the last checkpoint. The three methods under the study keep a valid copy of **A** and have exactly the same checkpoint cost.

We now introduce the ingredients of our own protection and verification mechanisms ABFT-DETECTION and ABFT-CORRECTION. We use ABFT techniques to protect the SpMxV, its result (hence the vector **q**), the matrix **A** and the input vector $\mathbf{p}_i$. As ABFT methods for vector operations is as costly as a repeated computation, we use TMR for them for simplicity. That is we do not protect $\mathbf{p}_i$, $\mathbf{q}$, $\mathbf{r}_i$, and $\mathbf{x}_i$ of the ith loop beyond the SpMxV at line 5 with ABFT, but we compute the dots, norms and *axpy* operations in resilient mode.

Although theoretically possible, constructing ABFT mechanism to detect up to k errors is practically not feasible for $k > 2$. The same mechanism can be used to correct up to $\lfloor k/2 \rfloor$ errors. Therefore, we focus on detecting up to two errors and correcting single errors. That is, we detect up to two errors in the computation $\mathbf{q} \leftarrow \mathbf{Ap}_i$ (two entries in **q** are faulty), or in $\mathbf{p}_i$, or in the sparse representation of the matrix **A**. With TMR, we assume that the errors in the computation are not overly frequent so that two results out of three are correct (we assume errors do not strike the vector data here). Our fault-tolerant CG versions thus have the following ingredients: ABFT to detect up to two errors in the SpMxV and correct up to one; TMR for vector operations; and checkpoint and roll-back in case errors are not corrected. In the rest of this section, we discuss the proposed ABFT method for the SpMxV (combining ABFT with checkpointing is later in Section 5.3).

5.2 ABFT-SpMxV

The overhead of the standard single error correcting ABFT technique is too high for the sparse matrix-vector product case. Shantaram et al. [35] propose a cheaper ABFT SpMxV algorithm that guarantees detection of single errors striking either the computation or the memory representation of the two input operands (matrix and vector). As their results depend on the sparse storage format adopted, throughout this section we assume that sparse matrices are stored in the compressed storage format by rows (CSR) format [32, Sec. 3.4], that is by means of three distinct arrays, namely $\mathit{Colid} \in \mathbb{N}^{\mathrm{nnz}(\mathbf{A})}$, $\mathit{Val} \in \mathbb{R}^{\mathrm{nnz}(\mathbf{A})}$ and $\mathit{Rowidx} \in \mathbb{N}^{n+1}$.

Shantaram et al. can protect $\mathbf{y} \leftarrow \mathbf{Ax}$, where $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$. To perform error detection, they rely on a column checksum vector $\mathbf{c}$ defined by

$$c_j = \sum_{i=1}^{n} a_{i,j} \tag{14}$$

and an auxiliary copy $\mathbf{x}'$ of the $\mathbf{x}$ vector. After having performed the actual SpMxV, to validate the result it suffices to compute $\sum_{i=1}^{n} y_i$, $\mathbf{c}^\mathsf{T}\mathbf{x}$ and $\mathbf{c}^\mathsf{T}\mathbf{x}'$, and to compare their values. It can be shown [35] that in the case of no errors, these three quantities carry the same value, whereas if a single error strikes either the memory or the computation, one of them must differ from the other two. Nevertheless, this method requires $\mathbf{A}$ to be strictly diagonally dominant, that seems to restrict too much the practical applicability of their ABFT scheme. Shantaram et al. need this condition to ensure the detection of errors striking an entry of $\mathbf{x}$ corresponding to a zero checksum column of $\mathbf{A}$. We further analyze that case and show how to overcome the issue without imposing any restriction on $\mathbf{A}$.

A nice way to characterize the problem is expressing it in geometrical terms. Let us consider the computation of a single entry of the checksum as

$$(\mathbf{w}^\mathsf{T}\mathbf{A})_j = \sum_{i=1}^{n} w_i a_{i,j} = \mathbf{w}^\mathsf{T}\mathbf{A}^j,$$

where $\mathbf{w} \in \mathbb{R}^n$ denotes the weight vector and $\mathbf{A}^j$ the j-th column of $\mathbf{A}$. Let us now interpret such an operation as the result of the scalar product $\langle\cdot,\cdot\rangle : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ defined by $\langle \mathbf{u}, \mathbf{v}\rangle \mapsto \mathbf{u}^\mathsf{T}\mathbf{v}$. It is clear that a checksum entry is zero if and only if the corresponding column of the matrix is orthogonal to the weight vector. In (14), we have chosen $\mathbf{w}$ to be such that $w_i = 1$ for $1 \le i \le n$, in order to make the computation easier. Let us see now what happens without this restriction.

The problem reduces to finding a vector $\mathbf{w} \in \mathbb{R}^n$ that is not orthogonal to any vector out of a basis $\mathscr{B} = \{\mathbf{b}_1, \dots, \mathbf{b}_n\}$ of $\mathbb{R}^n$ – the rows of the input matrix. Each one of these n vectors is perpendicular to a hyperplane h_i of $\mathbb{R}^n$, and $\mathbf{w}$ does not verify the condition

$$\langle \mathbf{w}, \mathbf{b}_i \rangle \neq 0, \tag{15}$$

Algorithm 2 ABFT-protected SpMxV, detection of 2 errors, correction of 1 error

```
Input: $\mathbf{A} \in \mathbb{R}^{n\times n} (\text{as} Val \in \mathbb{R}^{\mathrm{nnz}(\mathbf{A})}, Colid \in \mathbb{N}^{\mathrm{nnz}(\mathbf{A})}, Rowidx \in \mathbb{R}^{n}), \mathbf{x} \in \mathbb{R}^{n}$
Output: $\mathbf{y} = \mathbf{Ax}$, correction of single error or detection of double error
 1: global $\mathbf{W}^{\intercal} \leftarrow \left[\begin{smallmatrix}1 & 1 & \cdots & 1\\ 1 & 2 & \cdots & n\end{smallmatrix}\right] \in \mathbb{R}^{2\times n}$;
 2: global $\underline{\mathbf{W}}^{\intercal} \leftarrow \left[\mathbf{W}^{\intercal} \begin{smallmatrix}1\\ n+1\end{smallmatrix}\right] \in \mathbb{R}^{2\times n+1}$;
 3: $\mathbf{x}' \leftarrow \mathbf{x}$;
 4: $[\mathbf{C}, \mathbf{M}, \mathbf{c}_r, \mathbf{c}_x] =$ COMPUTECHECKSUMS($Val$, $Colid$, $Rowidx$);
 5: return SPMXV($Val$, $Colid$, $Rowidx$, $\mathbf{x}$, $\mathbf{x}'$, $\mathbf{M}$, $\mathbf{c}_r$, $\mathbf{c}_x$);

 6: function COMPUTECHECKSUMS($Val$, $Colid$, $Rowidx$)
 7:     $\mathbf{C}^{\intercal} \leftarrow \mathbf{W}^{\intercal}\mathbf{A}$;
 8:     $\mathbf{M} \leftarrow \mathbf{W} - \mathbf{C}$;
 9:     $\mathbf{c}_r \leftarrow \underline{\mathbf{W}}^{\intercal} Rowidx$;
10:     $\mathbf{c}_x \leftarrow \mathbf{W}^{\intercal}\mathbf{x}$;
11:     return $\mathbf{C}, \mathbf{M}, \mathbf{c}_r, \mathbf{c}_x$;

12: function SPMXV($Val$, $Colid$, $Rowidx$, $\mathbf{x}$, $\mathbf{x}'$, $\mathbf{C}$, $\mathbf{M}$, $\mathbf{c}_r$, $\mathbf{c}_x$)
13:     $\mathbf{s}_r \leftarrow 0 \in \mathbb{R}^{2\times 1}$;
14:     for $i \leftarrow 1$ to $n$ do
15:         $y_i \leftarrow 0$;
16:         $\mathbf{s}_r \leftarrow \mathbf{s}_r + \left[\begin{smallmatrix}w_{1,i}\\ w_{2,i}\end{smallmatrix}\right] Rowidx_i$;
17:         for $j \leftarrow Rowidx_i$ to $Rowidx_{i+1} - 1$ do
18:             $ind \leftarrow Colid_j$;
19:             $y_i \leftarrow y_i + Val_j \cdot x_{ind}$;
20:     $\mathbf{d}_r = \mathbf{c}_r - \mathbf{s}_r$;
21:     $\mathbf{d}_x = \mathbf{W}^{\intercal}\mathbf{y} - \mathbf{C}^{\intercal}\mathbf{x}$;
22:     $\mathbf{d}_{x'} = \mathbf{W}^{\intercal}(\mathbf{x}' - \mathbf{y}) - \mathbf{M}^{\intercal}\mathbf{x}$;
23:     if $\mathbf{d}_r = 0 \wedge \mathbf{d}_x = 0 \wedge \mathbf{d}_{x'} = 0$ then
24:         return $\mathbf{y}$;
25:     else
26:         CORRECTERRORS($Val$, $Colid$, $Rowidx$, $\mathbf{x}$, $\mathbf{x}'$, $\mathbf{C}$, $\mathbf{M}$, $\mathbf{d}_r$, $\mathbf{d}_x$, $\mathbf{d}_{x'}$, $\mathbf{c}_r$, $\mathbf{c}_x$);
```

for any i, if and only if it lies on h_i. As the Lebesgue measure in $\mathbb{R}^n$ of an hyperplane of $\mathbb{R}^n$ itself is zero, the union of these hyperplanes is measurable with $m_n\left(\bigcup_{i=1}^{n} h_i\right) = 0$, where m_n denotes the Lebesgue measure of $\mathbb{R}^n$. Therefore, the probability that a vector $\mathbf{w}$ randomly picked in $\mathbb{R}^n$ does not satisfy condition (15) for any i is zero.

Nevertheless, there are many reasons to consider zero checksum columns. First of all, when working with finite precision, the number of elements in $\mathbb{R}^n$ one can have is finite, and the probability of randomly picking a vector that is orthogonal to a given one could be bigger than zero. Moreover, a coefficient matrix usually comes from the discretization of a physical problem, and the distribution of its columns cannot be considered as random. Finally, using a randomly chosen vector instead of $(1,\ldots,1)^{\intercal}$ increases the number of required floating point operations, causing a growth of both execution time and rounding errors. Therefore, we would like to keep $\mathbf{w} = (1,\ldots,1)^{\intercal}$ as the vector of choice, in which case we need to protect SpMxV with matrices having zero column sums. There are many matrices with this property, for example the Laplacian matrices of graphs [13, Chapter 1].

In Algorithm 2, we propose an ABFT SpMxV method that uses weighted checksums and does not require the matrix to be strictly diagonally dominant. The idea is to compute the checksum vector and then shift it by adding to all of its entries a constant value chosen so that all of the elements of the new vector are different from zero. We give the result in Theorem 3 for the simpler case of single error detection without correction, in which case Algorithm 2 has $\mathbf{W} = (1, \dots, 1)^\intercal$ at line 1 and raises an error at line 26 (instead of correcting the error) if the tests at line 23 are not passed. The cases of multiple error detection and single error correction are proved in a technical report [20, Section 3.2].

Theorem 3 (Correctness of Algorithm 2 for error detection). *Let $\mathbf{A} \in \mathbb{R}^{n\times n}$ be a square matrix, let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ be the input and output vector respectively, and let $\mathbf{x}' = \mathbf{x}$. Let us assume that the algorithm performs the computation*

$$\widetilde{\mathbf{y}} \leftarrow \widetilde{\mathbf{A}}\widetilde{\mathbf{x}}, \tag{16}$$

where $\widetilde{\mathbf{A}} \in \mathbb{R}^{n\times n}$ and $\widetilde{\mathbf{x}} \in \mathbb{R}^n$ are the possibly faulty representations of $\mathbf{A}$ and $\mathbf{x}$ respectively, while $\widetilde{\mathbf{y}} \in \mathbb{R}^n$ is the possibly erroneous result of the sparse matrix-vector product. Let us also assume that the encoding scheme relies on

1. *an auxiliary checksum vector $\mathbf{c} = [\sum_{i=1}^n a_{i,1} + k, \dots, \sum_{i=1}^n a_{i,n} + k]$, where k is such that $\sum_{i=1}^n a_{i,j} + k \neq 0$ for $1 \le j \le n$,*
2. *an auxiliary checksum $y_{n+1} = k\sum_{i=i}^n \widetilde{x}_i$,*
3. *an auxiliary counter $\mathbf{s}_r$ initialized to 0 and updated at runtime by adding the value of the hit element each time the Rowidx array is accessed,*
4. *an auxiliary checksum $\mathbf{c}_r = \sum_{i=1}^n Rowidx_i \in \mathbb{N}$.*

Then, a single error in the computation of the SpMxV causes one of the following conditions to fail:

i. $\mathbf{c}^\intercal\widetilde{\mathbf{x}} = \sum_{i=1}^{n+1} \widetilde{y}_i$, difference is in $\mathbf{d}_x$ at line 21,
ii. $\mathbf{c}^\intercal\mathbf{x}' = \sum_{i=1}^{n+1} \widetilde{y}_i$, difference is in $\mathbf{d}_{x'}$ at line 22;
iii. $\mathbf{s}_r = \mathbf{c}_r$, difference is in $\mathbf{d}_r$ at line 20.

The proof of this theorem is technical and is available elsewhere [20, Theorem 1].

The function COMPUTECHECKSUM in Algorithm 2 requires just the knowledge of the matrix. Hence in the common scenario of many SpMxVs with the same matrix, it is enough to invoke it once to protect several matrix-vector multiplications. This observation will be crucial when discussing the performance of the checksumming techniques.

Extensions to $k \ge 2$ errors are discussed elsewhere [20, Section 3.2], where the following are detailed. The method just described can be extended to detect up to a total of k errors anywhere in the computation, in the representation of $\mathbf{A}$, or in the vector $\mathbf{x}$. Building up the necessary structures requires $\mathcal{O}(k\,\mathrm{nnz}(\mathbf{A}))$ time, and the overhead per SpMxV is $O(kn)$. For the particular case of $k = 2$ a result similar to that in Theorem 3 is also shown.

We now discuss error correction. If at least one of the tests at line 23 of Algorithm 2 fails, the algorithm invokes CORRECTERRORS in order to determine whether just one error struck either the computation or the memory and, in that case, to correct it. Indeed, whenever a single error is detected, disregarding its location (i.e., computation or memory), it can be corrected by means of a succession of various steps, as explained below; if need be, partial recomputations of the result are performed.

Specifically, we proceed as follows. To detect errors striking *Rowidx*, we compute the ratio d of the second component of $\mathbf{d}_r$ to the first one, and check whether its distance from an integer is smaller than a certain threshold parameter ε. If it is so, the algorithm concludes that the d-th element of *Rowidx* is faulty, performs the correction by subtracting the first component of $\mathbf{d}_r$ to $Rowidx_d$, and recomputes y_d and y_{d-1}, if the error in $Rowidx_d$ is a decrement; or y_{d+1} if it was an increment. Otherwise, it just emits an error.

The correction of errors striking *Val*, *Colid* and the computation of y are corrected together. Let now d be the ratio of the second component of $\mathbf{d}_x$ to the first one. If d is near enough to an integer, the algorithm computes the checksum matrix $\mathbf{C}' = \mathbf{W}^\intercal \mathbf{A}$ and considers the number $z_{\widetilde{\mathbf{C}}}$ of non-zero columns of the difference matrix $\widetilde{\mathbf{C}} = \mid \mathbf{C} - \mathbf{C}' \mid$. At this stage, three cases are possible:

- If $z_{\widetilde{\mathbf{C}}} = 0$, then the error is in the computation of y_d, and can be corrected by simply recomputing this value.
- If $z_{\widetilde{\mathbf{C}}} = 1$, then the error concerns an element of *Val*. Let us call f the index of the non-zero column of $\widetilde{\mathbf{C}}$. The algorithm finds the element of *Val* corresponding to the entry at row d and column f of A and corrects it by using the column checksums much like as described for *Rowidx*. Afterwards, y_d is recomputed to fix the result.
- If $z_{\widetilde{\mathbf{C}}} = 2$, then the error concerns an element of *Colid*. Let us call f_1 and f_2 the index of the two non-zero columns and m_1, m_2 the first and last elements of *Colid* corresponding to non-zeros in row d. It is clear that there exists exactly one index m^* between m_1 and m_2 such that either $Colid_{m^*} = f_1$ or $Colid_{m^*} = f_2$. To correct the error it suffices to switch the current value of $Colid_{m^*}$, i.e., putting $Colid_{m^*} = f_2$ in the former case and $Colid_{m^*} = f_1$ in the latter. Again, y_d has to be recomputed.
- if $z_{\widetilde{\mathbf{C}}} > 2$, then errors can be detected but not corrected, and an error is emitted.

To correct errors striking $\mathbf{x}$, the algorithm computes d, that is the ratio of the second component of $\mathbf{d}_{x'}$ to the first one, and checks that the distance between d and the nearest integer is smaller than ε. Provided that this condition is verified, the algorithm computes the value of the error $\tau = \sum_{i=1}^{n} x_i - cx_1$ and corrects $x_d = x_d - \tau$. The result is updated by subtracting from $\mathbf{y}$ the vector $\mathbf{y}^\tau = \mathbf{A}\mathbf{x}^\tau$, where $\mathbf{x}^\tau \in \mathbb{R}^{n \times n}$ is such that $x_d^\tau = \tau$ and $x_i^\tau = 0$ otherwise.

Finally, note that double errors could be shadowed when using Algorithm 2, but the probability of such an event is negligible. Still, there exists an improved version which avoids this issue by adding a third checksum [20, Section 3.2].

5.3 Performance model

The performance model is a simplified instance of the one discussed in Section 4, and we instantiate it for the three methods that we are considering, namely ONLINE-DETECTION, ABFT-DETECTION and ABFT-CORRECTION. We have a linear chain of identical tasks, where each task corresponds to one or several CG iterations. We execute T units of work followed by a verification, which we call a *chunk*, and we repeat this scheme s times, i.e., we compute s chunks, before taking a checkpoint. We say that the s chunks constitute a *frame*. The whole execution is then partitioned into frames. We assume that the checkpoint, recovery and verification operations are error-free. For each method below, we let C, R and V be the respective cost of these operations. Finally, and as before, assume a Poisson process for errors and let q be the probability of successful execution for each chunk: $q = e^{-\lambda T}$, where λ is the fault rate.

5.3.1 ONLINE-DETECTION

For Chen's method [12], we have the following parameters:

- We have chunks of d iterations, hence $T = dT_{iter}$, where T_{iter} is the raw cost of a CG iteration without any resilience method.
- The verification time V is the cost of the operations described in Section 5.1.
- As for silent errors, the application is protected from arithmetic errors in the ALU, as in Chen's original method, but also for corruption in data memory (because we also checkpoint the matrix $\mathbf{A}$). Let λ_a be the rate of arithmetic errors, and λ_m be the rate of memory errors. For the latter, we have $\lambda_m = M\lambda_{word}$ if the data memory consists of M words, each susceptible to be corrupted with rate λ_{word}. Altogether, since the two error sources are independent, they have a cumulated rate of $\lambda = \lambda_a + \lambda_m$, and the success probability for a chunk is $q = e^{-\lambda T}$. The optimal values of d and s can be computed by the same method as in Section 4.

5.3.2 ABFT-DETECTION

When using ABFT techniques, we detect possible errors every iteration, so a chunk is a single iteration, and $T = T_{iter}$. For ABFT-DETECTION, V is the overhead due to the checksums and redundant operations to detect a single error in the method.

ABFT-DETECTION can protect the application from the same silent errors as ONLINE-DETECTION, and just as before the success probability for a chunk (a single iteration here) is $q = e^{-\lambda T}$.

5.3.3 ABFT-CORRECTION

In addition to detection, we now correct single errors at every iteration. Just as for ABFT-DETECTION, a chunk is a single iteration, and $T = T_{iter}$, but V corresponds to a larger overhead, mainly due to the extra checksums needed to detect two errors and correct a single one.

The main difference lies in the error rate. An iteration with ABFT-CORRECTION is successful if zero or one error has struck during that iteration, so that the success probability is much higher than for ONLINE-DETECTION and ABFT-DETECTION. We compute that value of the success probability as follows. We have a Poisson process of rate λ, where $\lambda = \lambda_a + \lambda_m$ as for ONLINE-DETECTION and ABFT-DETECTION. The probability of exactly k errors in time T is $\frac{(\lambda T)^k}{k!} e^{-\lambda T}$ [27], hence the probability of no error is $e^{-\lambda T}$ and the probability of exactly one error is $\lambda T e^{-\lambda T}$, so that $q = e^{-\lambda T} + \lambda T e^{-\lambda T}$.

5.4 *Experiments*

Comprehensive tests were performed and reported in the technical report [20]. The main observation is that ABFT-CORRECTION outperforms both ONLINE-DETECTION and ABFT-DETECTION for a wide range of fault rates, thereby demonstrating that combining checkpointing with ABFT correcting techniques is more efficient than pure checkpointing for most practical situations.

6 Conclusion

Both fail-stop errors and silent data corruptions are major threats to executing HPC applications at scale. While many techniques have been advocated to deal with fail-stop errors, the lack of an efficient solution to handle silent errors is a real issue.

We have presented both a general-purpose solution and application-specific techniques to deal with silent data corruptions, with a focus on minimizing the overhead. For a divisible load application, we have extended the classical bound of Young/Daly to handle silent errors by combining checkpointing and verification mechanisms. For linear workflows, we have devised a polynomial-time dynamic programming algorithm that decides the optimal checkpointing and verification positions. Then, we have introduced ABFT as an application-specific technique to both detect and correct silent errors in iterative solvers that use sparse matrix vector multiplies and vector operations.

Our approach only addresses silent data corruptions. While several techniques have been developed to cope with either type of errors, few approaches are devoted to addressing both of them simultaneously. Hence, the next step is to extend our study to encompass both fail-stop and silent data corruptions in order to propose a comprehensive solution for executing applications on large scale platforms.

References

1. G. Aupy, A. Benoit, T. Hérault, Y. Robert, F. Vivien, and D. Zaidouni. On the combination of silent error detection and checkpointing. In *Proceedings of the 2013 International Symposium on Dependable Computing*, pages 11–20, 2013.
2. L. Bautista-Gomez, A. Benoit, A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. Which verification for soft error detection? In *Proceedings of the 2015 International Conference on High Performance Computing (HiPC'2015)*. IEEE Computer Society Press, 2015.
3. A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Assessing general-purpose algorithms to cope with fail-stop and silent errors. In *Proceedings of the 5th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2014.
4. A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Optimal resilience patterns to cope with fail-stop and silent errors. Research report RR-8786, INRIA, 2015. Available at `graal.ens-lyon.fr/~yrobert/rr8786.pdf`.
5. A. Benoit, A. Cavelan, Y. Robert, and H. Sun. Two-level checkpointing and partial verifications for linear task graphs. In *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2015.
6. A. R. Benson, S. Schmit, and R. Schreiber. Silent error detection in numerical time-stepping schemes. *The International Journal of High Performance Computing Applications*, DOI: 10.1177/1094342014532297, 2014.
7. G. Bosilca, R. Delmas, J. Dongarra, and J. Langou. Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4):410–416, 2009.
8. M. Bougeret, H. Casanova, M. Rabie, Y. Robert, and F. Vivien. Checkpointing strategies for parallel jobs. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2011.
9. G. Bronevetsky and B. de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 2008 International Conference on Supercomputing (ICS)*, pages 155–164, 2008.
10. A. Cavelan, S. K. Raina, Y. Robert, and H. Sun. Assessing the impact of partial verifications against silent data corruptions. In *Proceedings of the 44th International Conference on Parallel Processing (ICPP)*, 2015.
11. K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
12. Z. Chen. Online-ABFT: An online algorithm based fault tolerance scheme for soft error detection in iterative methods. In *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming*, pages 167–176, 2013.
13. F. R. K. Chung. *Spectral Graph Theory*. American Mathematical Society, 1997.
14. J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.
15. J. Dongarra et al. The international exascale software project: a call to cooperative action by the global high-performance community. *International Journal of High Performance Computing Applications*, 23(4):309–322, 2009.
16. J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for HPC. In *Proceedings of the 2012 IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 615–626, 2012.

17. E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Survey*, 34:375–408, 2002.
18. C. Engelmann, H. H. Ong, and S. L. Scorr. The case for modular redundancy in large-scale highh performance computing systems. In *Proceeding of the 8th IASTED Infernational Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 189–194, 2009.
19. M. Fasi, J. Langou, Y. Robert, and B. Uçar. A backward/forward recovery approach for the preconditioned conjugate gradient method. Research report RR-8826, INRIA, 2015. Available at `graal.ens-lyon.fr/~yrobert/rr8826.pdf`.
20. M. Fasi, Y. Robert, and B. Uçar. Combining Algorithm-based Fault Tolerance and Checkpointing for Iterative Solvers. Research Report RR-8675, INRIA, 2015. Short version appears in the proceedings of PDSEC'2015.
21. K. Ferreira, J. Stearley, J. H. I. Laros, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the Viability of Process Replication Reliability for Exascale Systems. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 44:1–44:12, 2011.
22. M. Heroux and M. Hoemmen. Fault-tolerant iterative methods via selective reliability. Research report SAND2011-3915 C, Sandia National Laboratories, 2011.
23. K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528, 1984.
24. 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. *ACM SIGARCH Computer Architecture News*, 40(1):111–122, 2012.
25. G. Lu, Z. Zheng, and A. A. Chien. When is multi-version checkpointing needed? In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at extreme scale (FTXS)*, pages 49–56, 2013.
26. R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.
27. M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.
28. A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, 2010.
29. X. Ni, E. Meneses, N. Jain, and L. V. Kalé. ACR: Automatic checkpoint/restart for soft and hard error protection. In *Proceedings of the 2013 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, 2013.
30. T. O'Gorman. The effect of cosmic rays on the soft error rate of a DRAM at ground level. *IEEE Transactions on Electron Devices*, 41(4):553–557, 1994.
31. T. Ozaki, T. Dohi, H. Okamura, and N. Kaio. Distribution-free checkpoint placement algorithms based on min-max principle. *IEEE Transactions on Dependable and Secure Computing*, 3(2):130–140, 2006.
32. Y. Saad. *Iterative methods for sparse linear systems*. SIAM Press, 2nd edition, 2003.
33. P. Sao and R. Vuduc. Self-stabilizing iterative solvers. In *Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, 2013.
34. B. Schroeder and G. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1), 2007.
35. M. Shantharam, S. Srinivasmurthy, and P. Raghavan. Fault tolerant preconditioned conjugate gradient for sparse linear system solution. In *Proceedings of the 2012 International Conference on Supercomputing*, pages 69–78, 2012.
36. S. Toueg and Ö. Babaoglu. On the optimum checkpoint selection problem. *SIAM Journal on Computing*, 13(3):630–649, 1984.
37. J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.
38. Z. Zheng and Z. Lan. Reliability-aware scalability models for high performance computing. In *Proceedings of the 2009 IEEE Conference on Cluster Computing*, 2009.

39. J. Ziegler, H. Muhlfeld, C. Montrose, H. Curtis, T. O'Gorman, and J. Ross. Accelerated testing for cosmic soft-error rate. *IBM Journal of Research and Development*, 40(1):51–72, 1996.
40. 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.
41. J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. IBM experiments in soft fails in computer electronics. *IBM Journal of Research and Development*, 40(1):3–18, 1996.